Lightning コンポーネント開発者ガイド

バージョン 41.0, Winter '18

@salesforcedocs

最終更新日: 2017/08/18

本書の英語版と翻訳版で相違がある場合は英語版を優先するものとします。

© Copyright 2000–2017 salesforce.com, inc. All rights reserved. Salesforce およびその他の名称や商標は、salesforce.com, inc. の登録商標です。本ドキュメントに記載されたその他の商標は、各社に所有権があります。

目次

第 1 章: Lightning コンポーネントフレームワークとは? . . . 1
Salesforce Lightning とは? . . . 2
Lightning コンポーネントフレームワークを使用する理由 . . . 2
コンポーネント . . . 3
イベント . . . 4
オープンソースの Aura フレームワーク . . . 4
Lightning コンポーネントのブラウザサポートの考慮事項 . . . 5
開発者コンソールの使用 . . . 7

第 2 章: クイックスタート . . . 8
ご利用になる前に . . . 9
Trailhead: Lightning コンポーネントリソースの探索 . . . 9
Lightning Experience および Salesforce1 のコンポーネントの作成 . . . 10
取引先責任者の読み込み . . . 13
イベントの起動 . . . 18

第 3 章: コンポーネントの作成 . . . 20
開発者コンソールで Lightning コンポーネントを作成する . . . 22
開発者コンソールで使用できる Lightning バンドル設定 . . . 23
コンポーネントのマークアップ . . . 25
コンポーネントの名前空間 . . . 26
名前空間プレフィックスが設定されていない組織でのデフォルトの名前空間の使用 . . . 26
組織の名前空間の使用 . . . 27
管理パッケージでのまたは管理パッケージからの名前空間の使用 . . . 27
組織の名前空間の作成 . . . 27
名前空間の使用例および参照 . . . 28
コンポーネントのバンドル . . . 31
コンポーネントの ID . . . 32
コンポーネント内の HTML . . . 33
コンポーネント内の CSS . . . 34
コンポーネントの属性 . . . 35
コンポーネントのコンポジション . . . 36
コンポーネントのボディ . . . 39
コンポーネントのファセット . . . 40
条件付きマークアップのベストプラクティス . . . 41
コンポーネントのバージョン設定 . . . 42
最小 API バージョン要件のあるコンポーネント . . . 44
式の使用 . . . 45

式の動的出力 . . . 47
条件式 . . . 47
コンポーネント間のデータバインド . . . 48
値プロバイダ . . . 52
式の評価 . . . 59
式の演算子のリファレンス . . . 60
式の関数のリファレンス . . . 63
表示ラベルの使用 . . . 67
Using Custom Labels . . . 68
入力コンポーネントの表示ラベル . . . 69
表示ラベルパラメータの動的な入力 . . . 69
JavaScript での表示ラベルの取得 . . . 70
Apex での表示ラベルの取得 . . . 71
親属性による表示ラベル値の設定 . . . 73
ローカライズ . . . 73
コンポーネントのドキュメントの提供 . . . 75
Lightning 基本コンポーネントの使用 . . . 77
Lightning 基本コンポーネントの考慮事項 . . . 83
Lightning 基本コンポーネントでのイベント処理 . . . 85
Lightning Design System の考慮事項 . . . 87
UI コンポーネントの操作 . . . 103
UI コンポーネントのイベント処理 . . . 106
UI コンポーネントの使用 . . . 107
フロー Lightning コンポーネントの操作 . . . 108
Lightning コンポーネントからのフロー変数値の設定 . . . 109
Lightning コンポーネントへのフロー変数値の取得 . . . 112
Lightning コンポーネントでのフローの完了動作の制御 . . . 113
Lightning コンポーネントからのフローインタビューの再開 . . . 114
アクセシビリティのサポート . . . 115
ボタンの表示ラベル . . . 116
音声メッセージ . . . 116
フォーム、項目、および表示ラベル . . . 116
イベント . . . 117
メニュー . . . 117

第 4 章: コンポーネントの使用 . . . 119

Lightning Experience および Salesforce1 での Lightning コンポーネントの使用 . . . 120
カスタムタブのコンポーネントの設定 . . . 120
Lightning Experience のカスタムタブとしての Lightning コンポーネントの追加 . . . 121
Salesforce1 のカスタムタブとしての Lightning コンポーネントの追加 . . . 123
Lightning コンポーネントアクション . . . 124
Lightning コンポーネントでの標準アクションの上書き . . . 132
Lightning コンポーネントを Lightning ページで使用できるようにするための準備 . . . 137
Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定 . . . 139

Lightning コンポーネントバンドルのデザインリソース 141
Lightning Experience のレコードホームページのコンポーネントの設定 143
Lightning for Outlook および Lightning for Gmail のコンポーネントの作成 145
カスタムコンポーネントの動的選択リストの作成 150
カスタム Lightning ページテンプレートコンポーネントの作成 151
Lightning ページテンプレートコンポーネントのベストプラクティス 154
lightning:flexipageRegionInfo による Lightning ページコンポーネントでの幅の認識 155
Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項 157
コミュニティビルダーでの Lightning コンポーネントの使用 158
コミュニティのコンポーネントの設定 158
コミュニティ用のカスタムテーマレイアウトコンポーネントの作成 159
コミュニティ用のカスタムの検索およびプロファイルメニューコンポーネントの作成 162
コミュニティのカスタムコンテンツレイアウトコンポーネントの作成 163
アプリケーションへのコンポーネントの追加 165
Chatter パブリッシャーへのカスタムアプリケーションの統合 166
Visualforce ページでの Lightning コンポーネントの使用 171
Lightning Out を使用した任意のアプリケーションへの Lightning コンポーネントの追加 (ベータ) 173
Lightning Out の要件 175
Lightning Out の連動関係 175
Lightning Out のマークアップ 177
Lightning Out からの認証 179
未認証ユーザとの Lightning Out アプリケーションの共有 179
Lightning Out の考慮事項と制限 181

第 5 章: イベントとの通信 183

アクションとイベント 184
クライアント側コントローラを使用したイベントの処理 185
コンポーネントイベント 187
コンポーネントイベント伝達 188
カスタムコンポーネントイベントの作成 189
コンポーネントイベントの起動 190
コンポーネントイベントの処理 190
コンポーネントイベントの例 197
アプリケーションイベント 199
アプリケーションイベントの伝達 200
カスタムアプリケーションイベントの作成 202
アプリケーションイベントの起動 202
アプリケーションイベントの処理 203
アプリケーションイベントの例 206
イベント処理のライフサイクル 209

高度なイベントの例 . . . 210
非 Lightning コードからの Lightning イベントの起動 . . . 215
イベントのベストプラクティス . . . 215
イベントのアンチパターン . . . 217
表示ライフサイクル中に起動されたイベント . . . 217
Salesforce1 と Lightning Experience で処理されるイベント . . . 219
システムイベント . . . 221

第 6 章: アプリケーションの作成 . . . 223

アプリケーションの概要 . . . 224
アプリケーションの UI の設計 . . . 224
アプリケーションテンプレートの作成 . . . 225
セキュアなコードの開発 . . . 225
LockerService とは? . . . 226
コンテンツセキュリティポリシーの概要 . . . 236
Salesforce Lightning CLI . . . 238
アプリケーションのスタイル設定 . . . 248
アプリケーションでの Salesforce Lightning Design System の使用 . . . 249
外部 CSS の使用 . . . 250
join 式を使用したスタイル設定マークアップの可読性の向上 . . . 251
コンポーネントの CSS のヒント . . . 252
ベンダープレフィックス . . . 253
設計トークンを使用したスタイル設定 . . . 253
JavaScript の使用 . . . 269
コンポーネントの初期化時のアクションの呼び出し . . . 271
コンポーネントのバンドル内の JavaScript コードの共有 . . . 271
コンポーネント間の JavaScript コードの共有 . . . 274
外部 JavaScript ライブラリの使用 . . . 276
JavaScript での属性値の操作 . . . 277
JavaScript でのコンポーネントのボディの操作 . . . 278
JavaScript でのイベントの操作 . . . 279
DOM の変更 . . . 282
コンポーネントの有効性の確認 . . . 288
フレームワークのライフサイクル外のコンポーネントの変更 . . . 290
項目の検証 . . . 290
エラーの発生および処理 . . . 293
コンポーネントメソッドのコール . . . 295
JavaScript Promise の使用 . . . 301
コンポーネントからの API コールの実行 . . . 303
サードパーティ API にアクセスするための CSP 信頼済みサイトの作成 . . . 304
JavaScript Cookbook . . . 305
コンポーネントの動的な作成 . . . 306
変更ハンドラを使用したデータ変更の検出 . . . 309
ID によるコンポーネントの検索 . . . 310

コンポーネントへのイベントハンドラの動的な追加 311
マークアップの動的な表示または非表示 313
スタイルの追加と削除 314
押下されたボタンの確認 315
JavaScript での日付の書式設定 317
Apex の使用 318
コントローラのサーバ側ロジックの作成 319
Salesforce レコードの操作 335
Apex コードのテスト 343
Apex からの API コールの実行 344
Apex でのコンポーネントの作成 345
Lightning データサービス 345
レコードの読み込み 346
レコードの保存 348
レコードの作成 351
レコードの削除 355
レコードの変更 357
エラー 358
考慮事項 359
Lightning データサービスの例 362
SaveRecordResult 366
Lightning コンテナ 367
サードパーティフレームワークの使用 367
Lightning コンテナコンポーネントの制限 376
Lightning Realty アプリケーション 377
lightning-container NPM モジュールリファレンス 380
アクセスの制御 385
アプリケーションのアクセス制御 388
インターフェースのアクセス制御 389
コンポーネントのアクセス制御 389
属性のアクセス制御 389
イベントのアクセス制御 390
オブジェクト指向開発の使用 390
継承とは? 390
継承されるコンポーネントの属性 391
抽象コンポーネント 393
インターフェース 394
継承ルール 395
AppCache の使用 395
アプリケーションとコンポーネントの配布 395

第 7 章: デバッグ 397

Lightning コンポーネントのデバッグモードの有効化 398
Salesforce Lightning Inspector Chrome 拡張機能 398

Salesforce Lightning Inspector のインストール . . . 399
Salesforce Lightning Inspector . . . 399
ログメッセージ . . . 414

第 8 章: パフォーマンスの警告の修正 . . . 415

<aura:if> — 表示されない内容のクリーンアップ . . . 416
<aura:iteration> — 複数の items 設定 . . . 417

第 9 章: リファレンス . . . 420

リファレンスドキュメントアプリケーション . . . 421
サポートされる aura:attribute の型 . . . 421
基本の型 . . . 422
オブジェクト型 . . . 424
標準オブジェクト型とカスタムオブジェクト型 . . . 424
コレクション型 . . . 425
カスタム Apex クラス型 . . . 426
フレームワーク固有の型 . . . 427
aura:application . . . 428
aura:component . . . 429
aura:dependency . . . 431
aura:event . . . 432
aura:interface . . . 432
aura:method . . . 433
aura:set . . . 435
スーパーコンポーネントから継承される属性の設定 . . . 435
コンポーネント参照での属性の設定 . . . 436
インターフェースから継承される属性の設定 . . . 437
コンポーネントの参照 . . . 437
aura:expression . . . 438
aura:html . . . 438
aura:if . . . 439
aura:iteration . . . 439
aura:renderIf . . . 440
aura:template . . . 441
aura:text . . . 442
aura:unescapedHtml . . . 442
auraStorage:init . . . 442
force:canvasApp . . . 444
force:inputField . . . 446
force:outputField . . . 447
force:recordData . . . 449
force:recordEdit . . . 450
force:utilityBarAPI . . . 451
force:workspaceAPI . . . 453

force:recordPreview 455
force:recordView 456
forceChatter:feed 457
forceChatter:fullFeed 459
forceChatter:publisher 460
forceCommunity:appLauncher 461
forceCommunity:navigationMenuBase 463
forceCommunity:notifications 464
forceCommunity:routeLink 465
forceCommunity:waveDashboard 467
lightning:accordion 469
lightning:accordionSection 470
lightning:avatar 471
lightning:badge 473
lightning:breadcrumb 473
lightning:breadcrumbs 475
lightning:button 476
lightning:buttonGroup 478
lightning:buttonIcon 479
lightning:buttonIconStateful 481
lightning:buttonMenu (ベータ) 483
lightning:buttonStateful 486
lightning:card 488
lightning:checkboxGroup 489
lightning:clickToDial 492
lightning:combobox (ベータ) 493
lightning:container 496
lightning:datatable 498
lightning:dualListbox 503
lightning:dynamicIcon 506
lightning:fileUpload (ベータ) 507
lightning:flexipageRegionInfo 509
lightning:flow 510
lightning:formattedDateTime (ベータ) 511
lightning:formattedEmail 513
lightning:formattedLocation 514
lightning:formattedNumber (ベータ) 515
lightning:formattedPhone 517
lightning:formattedRichText 518
lightning:formattedText 520
lightning:formattedUrl 521
lightning:helptext (ベータ) 522
lightning:icon 523
lightning:input (ベータ) 525

lightning:inputLocation . . . 532
lightning:inputRichText (ベータ) . . . 534
lightning:layout . . . 537
lightning:layoutItem . . . 539
lightning:menuItem (ベータ) . . . 540
lightning:pill . . . 542
lightning:progressBar . . . 544
lightning:progressIndicator . . . 545
lightning:radioGroup . . . 546
lightning:relativeDateTime . . . 549
lightning:select . . . 550
lightning:slider . . . 554
lightning:spinner . . . 557
lightning:tab (ベータ) . . . 558
lightning:tabset (ベータ) . . . 559
lightning:textarea . . . 561
lightning:tile . . . 564
lightning:tree . . . 565
lightning:verticalNavigation . . . 568
lightning:verticalNavigationItem . . . 571
lightning:verticalNavigationItemBadge . . . 572
lightning:verticalNavigationItemIcon . . . 573
lightning:verticalNavigationOverflow . . . 574
lightning:verticalNavigationSection . . . 574
ltng:require . . . 575
ui:actionMenuItem . . . 576
ui:button . . . 578
ui:checkboxMenuItem . . . 580
ui:inputCheckbox . . . 582
ui:inputCurrency . . . 585
ui:inputDate . . . 587
ui:inputDateTime . . . 591
ui:inputDefaultError . . . 594
ui:inputEmail . . . 597
ui:inputNumber . . . 600
ui:inputPhone . . . 603
ui:inputRadio . . . 605
ui:inputRichText . . . 608
ui:inputSecret . . . 611
ui:inputSelect . . . 613
ui:inputSelectOption . . . 618
ui:inputText . . . 620
ui:inputTextArea . . . 623
ui:inputURL . . . 626

ui:menu . . . 629
ui:menuItem . . . 632
ui:menuItemSeparator . . . 634
ui:menuList . . . 635
ui:menuTrigger . . . 637
ui:menuTriggerLink . . . 638
ui:message . . . 639
ui:outputCheckbox . . . 641
ui:outputCurrency . . . 643
ui:outputDate . . . 644
ui:outputDateTime . . . 646
ui:outputEmail . . . 648
ui:outputNumber . . . 650
ui:outputPhone . . . 651
ui:outputRichText . . . 653
ui:outputText . . . 655
ui:outputTextArea . . . 656
ui:outputURL . . . 658
ui:radioMenuItem . . . 660
ui:scrollerWrapper . . . 662
ui:spinner . . . 663
wave:waveDashboard . . . 664
メッセージングコンポーネントの参照 . . . 666
lightning:notificationsLibrary . . . 666
lightning:overlayLibrary . . . 669
インターフェースの参照 . . . 673
force:hasRecordId . . . 676
force:hasSObjectName . . . 677
lightning:actionOverride . . . 677
lightning:appHomeTemplate . . . 679
lightning:availableForChatterExtensionComposer . . . 679
lightning:availableForChatterExtensionRenderer . . . 679
lightning:homeTemplate . . . 680
lightning:recordHomeTemplate . . . 680
イベントの参照 . . . 681
force:closeQuickAction . . . 681
force:createRecord . . . 682
force:editRecord . . . 684
force:navigateToList . . . 684
force:navigateToObjectHome . . . 685
force:navigateToRelatedList . . . 686
force:navigateToSObject . . . 686
force:navigateToURL . . . 687
force:recordSave . . . 688

force:recordSaveSuccess 689
force:refreshView 689
force:showToast 690
forceCommunity:analyticsInteraction 692
forceCommunity:routeChange 692
lightning:openFiles 693
lightning:sendChatterExtensionPayload 694
ltng:selectSObject 694
ltng:sendMessage 695
ui:clearErrors 695
ui:collapse 696
ui:expand 696
ui:menuFocusChange 697
ui:menuSelect 698
ui:menuTriggerPress 698
ui:validationError 699
wave:discoverDashboard 700
wave:discoverResponse 701
wave:selectionChanged 701
wave:update 702
システムイベントの参照 703
aura:doneRendering 704
aura:doneWaiting 705
aura:locationChange 705
aura:systemError 706
aura:valueChange 707
aura:valueDestroy 708
aura:valueInit 709
aura:valueRender 710
aura:waiting 710
サポートされる HTML タグ 711

第 1 章　Lightning コンポーネントフレームワークとは?

トピック:

- Salesforce Lightning とは?
- Lightning コンポーネントフレームワークを使用する理由
- コンポーネント
- イベント
- オープンソースの Aura フレームワーク
- Lightning コンポーネントのブラウザサポートの考慮事項
- 開発者コンソールの使用

Lightning コンポーネントフレームワークは、モバイルデバイス用およびデスクトップデバイス用の動的な Web アプリケーションを開発する UI フレームワークです。これは、拡張性に優れた単一ページアプリケーションを構築する最新のフレームワークです。

このフレームワークでは、クライアントとサーバの橋渡しをする、分離された多層コンポーネント開発がサポートされています。クライアント側では JavaScript、サーバ側では Apex が使用されます。

Salesforce Lightning とは?

Lightning には、Lightning コンポーネントフレームワークおよび開発者向けの魅力的なツールがいくつか用意されています。Lightningを使用すると、あらゆるデバイスに対応するアプリケーションを簡単に構築できます。

Lightning には次のテクノロジがあります。

- Lightning コンポーネント。迅速な開発とアプリケーションパフォーマンスの向上を実現するクライアント-サーバフレームワークが提供されます。このフレームワークは、Salesforce1モバイルアプリケーションおよび Salesforce Lightning Experience での使用に最適です。
- Lightningアプリケーションビルダー。標準およびカスタムLightningコンポーネントを使用することで、コードを作成することなく、これまでにないほど迅速にアプリケーションを視覚的に構築できます。システム管理者がコードを使用せずにカスタムユーザインターフェースを構築できるように、Lightning アプリケーションビルダーで Lightning コンポーネントを使用できます。

これらのテクノロジを使用すれば、新しいアプリケーションをシームレスにカスタマイズして、Salesforce1 を実行しているモバイルデバイスに簡単にリリースできます。実際、Salesforce1 モバイルアプリケーションおよび Salesforce Lightning Experience は Lightning コンポーネントで構築されています。

このガイドには、Salesforce1モバイルアプリケーションで使用できるカスタムLightningコンポーネントのほか、独自のスタンドアロンLightningアプリケーションを作成するための詳細な説明が記載されています。また、アプリケーションおよびコンポーネントをパッケージ化して、AppExchange で配布する方法についても学習します。

Lightning コンポーネントフレームワークを使用する理由

標準のコンポーネントセット、イベント駆動型アーキテクチャ、パフォーマンスが最適化されたフレームワークなどの利点があります。

標準のコンポーネントセット

コンポーネントセットが標準装備されているため、アプリケーションの構築にすぐに着手できます。アプリケーションのデバイス別の最適化はコンポーネントが行うため、最適化に時間を取られることがありません。

リッチコンポーネントエコシステム

業務に対応したコンポーネントを作成し、Salesforce1、Lightning Experience、およびコミュニティでそれらを使用できるようにします。Salesforce1ユーザは、ナビゲーションメニューからコンポーネントにアクセスします。Lightning Experienceまたはコミュニティをカスタマイズするには、Lightningアプリケーションビルダーの Lightning ページでドラッグアンドドロップコンポーネントを使用するか、コミュニティビルダーを使用します。AppExchangeには、組織で使用できる追加のコンポーネントがあります。同様に、独自のコンポーネントを公開して他のユーザと共有できます。

パフォーマンス

クライアント側で JavaScript に依存するステートフルクライアントとステートレスサーバアーキテクチャを使用して、UIコンポーネントのメタデータおよびアプリケーションデータを管理します。クライアントは、不可欠な場合 (追加メタデータまたはデータを取得する場合など) にのみサーバをコールします。効率性を最大にするために、サーバはユーザが必要なデータのみを送信します。このフレームワークでは、JSON を使用して、サーバとクライアント間のデータをやりとりします。サーバやブラウザ、デバイス、ネットワー

クがインテリジェントに活用されるため、開発者はアプリケーションのロジックやインタラクションに集中できます。

イベント駆動型アーキテクチャ
イベント駆動型アーキテクチャを使用して、個々のコンポーネントを適切に切り離します。どのコンポーネントも、アプリケーションイベント、または表示可能なコンポーネントイベントを登録できます。

短時間で開発
デスクトップやモバイルデバイスとシームレスに連動する標準コンポーネントにより、チームの取り組みが迅速化します。アプリケーションをコンポーネントベースで構築するため、並列設計が可能になり、開発全般の効率性が向上します。

コンポーネントはカプセル化され、内部は非公開に保たれますが、公開形状はコンポーネントのコンシューマから参照できます。この強固な分離により、コンポーネント作成者は自由に内部実装の詳細を変更することができ、コンポーネントのコンシューマはこうした変更から隔離されます。

デバイス対応およびブラウザ間の互換性
アプリケーションには応答性が高い設計が使用され、快適なユーザ環境を実現します。Lightning コンポーネントフレームワークは、HTML5、CSS3、タッチイベントなど、最新のブラウザテクノロジをサポートしています。

コンポーネント

コンポーネントは、アプリケーションの自己完結型の再利用可能なユニットで、UIの再利用可能なセクションを表します。粒度の面では、1 行のテキストからアプリケーション全体に至るものまで、さまざまです。

フレームワークには、事前構築された一連のコンポーネントが含まれます。たとえば、Lightning Design System スタイル設定に付属するコンポーネントは、`lightning` 名前空間で使用できます。これらのコンポーネントは、基本 Lightning コンポーネントとも呼ばれます。コンポーネントを組み合わせて設定すれば、アプリケーションに新しいコンポーネントを作成できます。コンポーネントが表示されると、ブラウザ内に HTML DOM 要素が生成されます。

コンポーネントには、他のコンポーネントのほか、HTML、CSS、JavaScript、その他の Web 対応コードを含めることができます。そのため、洗練された UI を備えたアプリケーションを構築できます。

コンポーネントの実装の細部はカプセル化されています。そのため、コンポーネントのコンシューマがアプリケーションの構築に集中する一方で、コンポーネントの作成者はイノベーションに取り組み、コンシューマの作業を遮ることなく変更を実行できます。コンポーネントの設定では、定義に公開する指定の属性を設定します。コンポーネントは、イベントをリスンまたは公開して、それぞれの環境とやりとりします。

関連トピック:
コンポーネントの作成
コンポーネントの参照
Lightning 基本コンポーネントの使用

イベント

イベント駆動型プログラミングは、JavaScript や Java Swing など、多くの言語およびフレームワークで使用されています。この概念は、インターフェースイベントの発生時にそのイベントに対応するハンドラを作成するというものです。

コンポーネントは、そのマークアップでイベントを起動できるように登録します。イベントは JavaScript コントローラアクションから起動されます。このアクションは、一般的にユーザがユーザインターフェースを操作することでトリガされます。

このフレームワークには次の 2 つのタイプのイベントがあります。

- コンポーネントイベントは、そのコンポーネント自体、そのコンポーネントをインスタンス化するコンポーネント、そのコンポーネントを含むコンポーネントによって処理されます。
- アプリケーションイベントは、イベントをリスンしているすべてのコンポーネントによって処理されます。これらのイベントは、基本的には、従来の publish-subscribe モデルです。

ハンドラは JavaScript コントローラアクションに記述します。

関連トピック:

イベントとの通信

クライアント側コントローラを使用したイベントの処理

オープンソースの Aura フレームワーク

Lightning コンポーネントフレームワークは、オープンソースの Aura フレームワーク上に構築されています。Aura フレームワークでは、Salesforce のデータから完全に独立したアプリケーションを構築できます。

Aura フレームワークは、https://github.com/forcedotcom/aura から入手できます。オープンソースの Aura フレームワークには、Lightning コンポーネントフレームワークで現在使用できない機能およびコンポーネントがあります。弊社は、これらの機能およびコンポーネントを Salesforce 開発者が使用できるように作業を進めています。

このガイド内のサンプルコードは、`aura:iteration` や `ui:button` など、Aura フレームワークの標準のコンポーネントを使用しています。`aura` 名前空間にはアプリケーションロジックを簡略化するコンポーネントが含まれ、`ui` 名前空間にはボタンや入力項目などユーザインターフェース要素のコンポーネントが含まれます。`force` 名前空間には、Salesforce 固有のコンポーネントが含まれます。

Lightning コンポーネントのブラウザサポートの考慮事項

ブラウザサポートは、Salesforce 製品と環境によって異なります。Lightning コンポーネントを使用して構築するときに、このブラウザサポート情報を使用します。

次の表に、さまざまな Salesforce 機能の最小ブラウザバージョンの概要を示します。すべてのブラウザには追加要件と推奨設定があり、ブラウザ固有の考慮事項がいくつかあります。Salesforce ヘルプでブラウザの互換性の詳細を参照してください。

エディション

Salesforce Classic を使用可能なエディション: すべてのエディション

エディション

Lightning Experience を使用可能なエディション: **Group** Edition、**Professional** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、および **Developer** Edition

Lightning Experience および Lightning ベースの機能

次の表では、次世代のユーザインターフェースプラットフォームで構築されたさまざまな機能内で Lightning コンポーネントを使用するための、最小ブラウザバージョン要件について説明します。

	Microsoft® Internet Explorer®	Microsoft® Edge	Google Chrome™	Mozilla® Firefox®	Apple® Safari®
Lightning Experience	IE 11 (2020 年 12 月 21 日にサポート終了)[1]	Windows 10	最新	最新	10.x+
Lightning Experience の Salesforce コンソール	N/A	Windows 10	最新	最新	10.x+
Lightning コミュニティ	IE 11	Windows 10	最新	最新	10.x+
Lightning for Outlook (クライアント)	IE 11	N/A	N/A	N/A	N/A
Lightning for Outlook (Web)	IE 11	Windows 10	最新	最新	10.x+
スタンドアロン Lightning アプリケーション (my.app)	IE 11	Windows 10	最新	最新	10.x+
Lightning Out	IE 9+	Windows 10	最新	最新	10.x+

[1] LockerService は IE11 では無効です。セキュリティを強化するため、IE11 以外のサポートされているブラウザを使用することをお勧めします。

重要: Internet Explorer 11 による Lightning Experience へのアクセスは、Summer '16 以降サポートされなくなります。

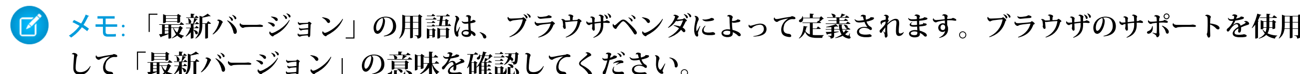
- 2017 年 12 月 16 日まで引き続き IE11 を使用して Lightning Experience にアクセスできます。
- IE11 の拡張サポートを選択すると、2020 年 12 月 31 日まで引き続き IE11 を使用して Lightning Experience にアクセスできます。
- この変更は、Salesforce Classic または Salesforce Communities を使用した組織のユーザには影響しません。

この変更についての詳細は、「Retirement of Support for Accessing Lightning Experience Using Microsoft Internet Explorer version 11 (Microsoft Internet Explorer バージョン 11 を使用した Lightning Experience へのアクセスのサポート終了)」を参照してください。

Salesforce Classic での Visualforce の Lightning コンポーネント

次の表では、従来のユーザインターフェースプラットフォームで構築されたさまざまな機能内で Lightning コンポーネントを使用するための、最小ブラウザバージョン要件について説明します。

	Microsoft® Internet Explorer®	Microsoft® Edge	Google Chrome™	Mozilla® Firefox®	Apple® Safari®
Salesforce Classic	IE 9+	Windows 10	最新	最新	10.x+
Salesforce Classic の Salesforce コンソール	IE 9+	Windows 10	最新	最新	N/A
Classic コミュニティ	IE 9+	Windows 10	最新	最新	10.x+
Force.com サイト	IE 9+	Windows 10	最新	最新	10.x+



メモ: 「最新バージョン」の用語は、ブラウザベンダによって定義されます。ブラウザのサポートを使用して「最新バージョン」の意味を確認してください。

関連トピック:

Salesforce ヘルプ: サポートされるブラウザ

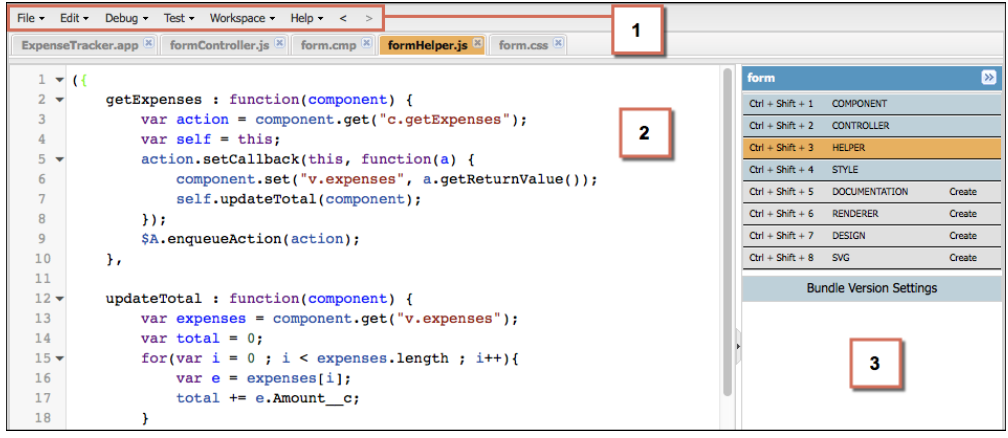
Salesforce ヘルプ: すべてのブラウザに関する推奨事項と要件

サポートされていないブラウザで無効化された LockService

コンテンツセキュリティポリシーの概要

開発者コンソールの使用

開発者コンソールには、コンポーネントおよびアプリケーションを開発するためのツールが用意されています。



開発者コンソールでは、次の機能を実行できます。

- メニューバー (1) を使用して、次の Lightning リソースを作成したり、開いたりする。
  - アプリケーション
  - コンポーネント
  - インターフェース
  - 行動
  - Tokens
- ワークスペース (2) を使用して、Lightning リソースを操作する。
- サイドバー (3) を使用して、特定のコンポーネントのバンドルに含まれるクライアント側のリソースを作成したり、開いたりする。
  - コントローラ
  - ヘルパー
  - スタイル
  - ドキュメント
  - レンダラ
  - 設計
  - SVG

開発者コンソールについての詳細は、「開発者コンソールユーザインターフェース」を参照してください。

関連トピック:

Salesforce ヘルプ: 開発者コンソールを開く

開発者コンソールで Lightning コンポーネントを作成する

コンポーネントのバンドル

第 2 章　クイックスタート

トピック:

- ご利用になる前に
- Trailhead: Lightning コンポーネントリソースの探索
- Lightning Experience および Salesforce1 のコンポーネントの作成

クイックスタートには、Lightning コアコンポーネントの概念を説明する Trailhead リソース、および Salesforce1 と Lightning Experience で選択した取引先責任者を管理する Lightning コンポーネントを作成する短いチュートリアルがあります。開発者コンソールからすべてのコンポーネントを作成します。チュートリアルでは、いくつかのイベントを使用して、取引先責任者レコードの作成や編集、および関連ケースの表示を行います。

ご利用になる前に

Lightning のアプリケーションやコンポーネントを操作するには、次の前提条件に従います。

1. Developer Edition 組織を作成する
2. カスタム Salesforce ドメイン名を定義する

メモ: このクイックスタートチュートリアルでは、Developer Edition 組織を作成したり、名前空間プレフィックスを登録したりする必要はありません。ただし、管理パッケージを提供する予定の場合は、上記の作業が必要になります。UI を使用して Lightning コンポーネントを作成可能なエディションは、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、**Developer** Edition、または Sandbox です。Developer Edition 組織を使用する予定がない場合は、直接「カスタム Salesforce ドメイン名を定義する」に進むことができます。

Developer Edition 組織を作成する

このクイックスタートチュートリアルを組織が行う必要があり、本番組織を使用しないことをお勧めします。まだ Developer Edition 組織がない場合、作成するだけで十分です。

1. ブラウザで `https://developer.salesforce.com/signup?d=70130000000td6N` にアクセスします。
2. 各項目にユーザ情報と会社情報を入力します。
3. [メール] 項目には、Web ブラウザから簡単に確認できる公開アドレスを使用してください。
4. 一意の [ユーザ名] を入力します。ユーザ名もメールアドレスの形式にする必要がありますが、メールアドレスと同じにする必要はなく、通常は違うものを入力することをお勧めします。ユーザ名は `developer.salesforce.com` でのログイン情報および ID であるため、自分自身を表す `firstname@lastname.com` などのユーザ名を選ぶことで、より有益に使用できます。
5. [マスターサブスクリプション契約] を読み、チェックボックスをオンにしてから [サインアップ] をクリックします。
6. その後まもなく、ログインリンクを記載したメールが届きます。リンクをクリックし、パスワードを変更します。

カスタム Salesforce ドメイン名を定義する

カスタムドメイン名を使用すると、アクセスセキュリティを強化し、組織のログインおよび認証をより適切に管理できます。カスタムドメインが *`universalcontainers`* の場合、ログイン URL は `https://universalcontainers.lightning.force.com` になります。詳細は、Salesforce ヘルプの「私のドメイン」を参照してください。

Trailhead: Lightning コンポーネントリソースの探索

Trailhead リソースで Lightning コンポーネントの基本を習得してください。

Salesforce の開発に初めて携わる方も、経験豊富な Visualforce 開発者も、次の Trailhead リソースから始めることをお勧めします。

[Lightning コンポーネントの基本](#)
Lightning コンポーネントを使用し、再利用可能な UI コンポーネントによって最新 Web アプリケーションを構築します。Lightning コアコンポーネントの概念について学び、スタンドアロンアプリケーション、Salesforce1、または Lightning Experience で実行できる単純な経費追跡アプリケーションを作成します。

[クイックスタート: Lightning コンポーネント](#)
組織の取引先責任者のリストを表示する最初のコンポーネントを作成します。

[取引先の地理位置情報アプリケーションの作成](#)
Lightning コンポーネントを使用して取引先をマップに表示するアプリケーションを作成します。

[Lightning Design System を使用した Lightning アプリケーションの作成](#)
取引先リストを表示する Lightning コンポーネントを設計します。

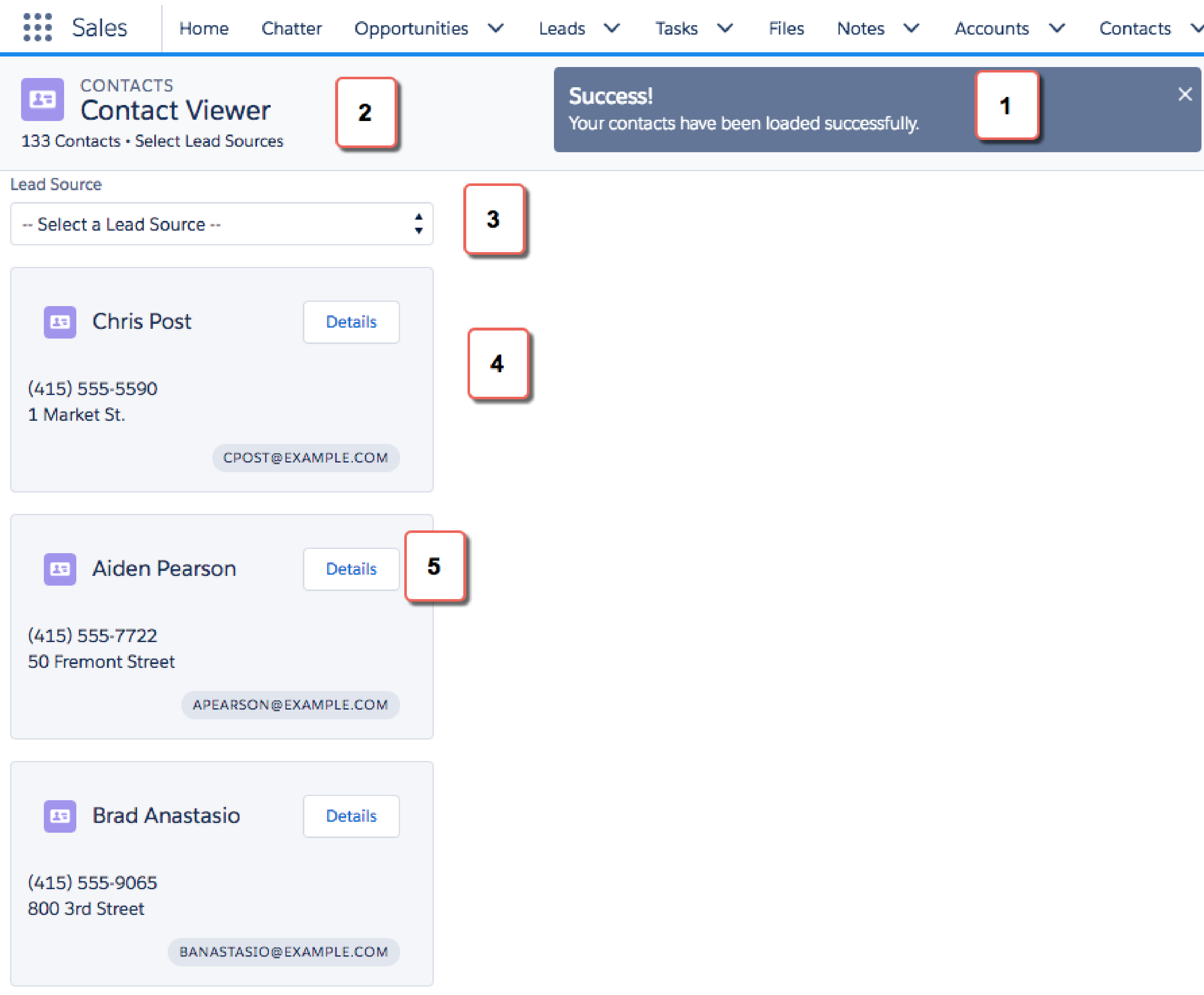
[レストランロケータ Lightning コンポーネントの作成](#)
特定の場所付近にあるビジネスのリストを表示する Yelp の Search API を使用して Lightning コンポーネントを作成します。

Lightning Experience および Salesforce1 のコンポーネントの作成

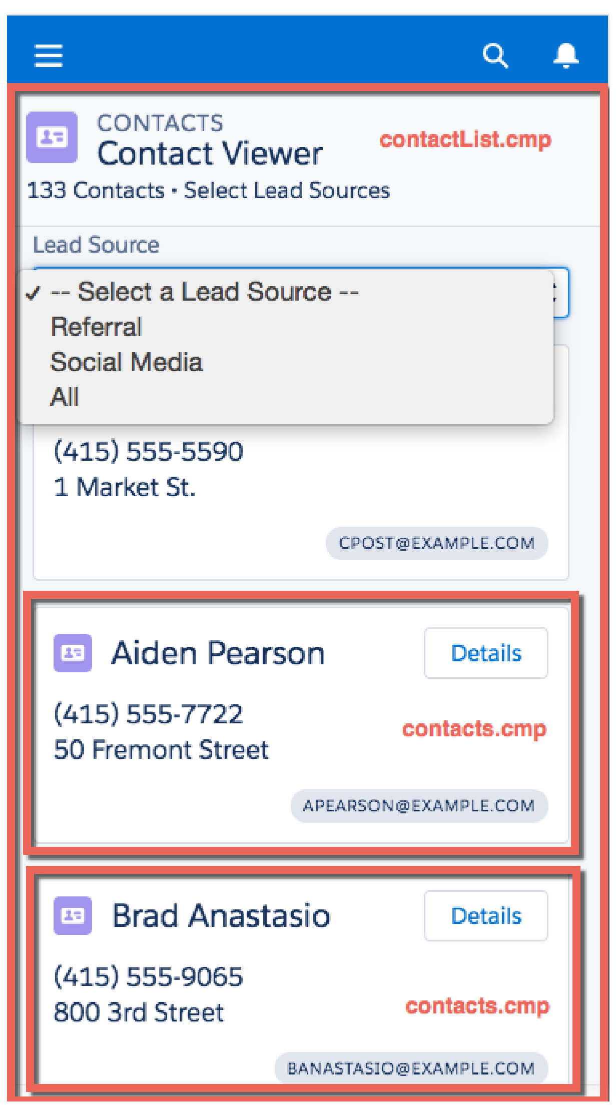
取引先責任者データを読み込んで Lightning Experience および Salesforce1 と連動するカスタム UI を作成する方法を確認します。

このチュートリアルでは、以下を行うコンポーネントの作成について順を追って説明します。

- すべての取引先責任者が正常に読み込まれたら、`force:showToast` イベントを使用してトーストメッセージを表示する (1)。
- 選択したリードソースに基づいて取引先責任者の数を更新する (2)。
- リードソース (紹介またはソーシャルメディア) が選択されている場合、`lightning:select` コンポーネントを使用して取引先責任者を絞り込む (3)。
- `lightning:card` コンポーネントを使用して取引先責任者データを表示する (4)。
- [詳細] ボタンがクリックされたらレコードに移動する (5)。



以下に、Salesforce1 にコンポーネントがどのように表示されるかを示します。`contactList` と `contacts` の 2 つのコンポーネントを作成しています。`contactList` は反復するコンテナコンポーネントで `contacts` コンポーネントを表示します。すべての取引先責任者は `contactList` に表示されますが、異なるリードソースを選択してそのリードソースに関連付けられている取引先責任者のサブセットを表示できます。



次のいくつかのトピックで、次のリソースを作成します。

リソース	説明
取引先責任者バンドル	
`contacts.cmp`	個々の取引先責任者データを表示するコンポーネント
`contactsController.js`	`force:navigateToSObject` イベントを使用して取引先責任者レコードに移動する、クライアント側コントローラアクション
contactList バンドル	
`contactList.cmp`	取引先責任者のリストを読み込むコンポーネント
`contactListController.js`	ヘルパーリソースをコールして取引先責任者データを読み込み、リードソースの選択を処理する、クライアント側コントローラアクション

リソース	説明
contactListHelper.js	取引先責任者データを取得し、取引先責任者データを正常に読み込んだときにトーストメッセージを表示し、リードソースに基づいて取引先責任者データを表示し、取引先責任者の合計数を更新するヘルパー関数
Apex コントローラ	
ContactController.apxc	すべての取引先責任者レコードをクエリし、リードソース別に取引先責任者レコードをクエリする Apex コントローラ

取引先責任者の読み込み

Apexコントローラを作成して、取引先責任者を読み込みます。Apexコントローラはコンポーネントと Salesforce データを接続するブリッジです。

組織に、このチュートリアルで使用できる既存の取引先責任者レコードが必要です。

1. **[File (ファイル)]** > **[New (新規)]** > **[Apex Class (Apex クラス)]** をクリックして、**[New Class (新規クラス)]** ウィンドウに「*ContactController*」と入力します。ContactController.apxc という新しい Apex クラスが作成されます。次のコードを入力して保存します。

```
public with sharing class ContactController {
@AuraEnabled
    public static List<Contact> getContacts() {
        List<Contact> contacts =
               [SELECT Id, Name, MailingStreet, Phone, Email, LeadSource FROM Contact];


        //Add isAccessible() check
        return contacts;
    }
}
```

ContactController には、SOQL ステートメントを使用して取引先責任者データを返すメソッドが含まれます。この Apex コントローラは、後のステップでコンポーネントに結び付けられます。getContacts() は選択した項目のすべての取引先責任者を返します。

2. **[File (ファイル)]** > **[New (新規)]** > **[Lightning Component (Lightning コンポーネント)]** をクリックして、[New Lightning Bundle (新規 Lightning バンドル)] ポップアップウィンドウの [Name (名前)] 項目に「*contacts*」と入力します。これにより、contacts.cmp というコンポーネントが作成されます。次のコードを入力して保存します。

```
<aura:component>
    <aura:attribute name="contact" type="Contact" />

        <lightning:card variant="Narrow" title="{!v.contact.Name}"
                        iconName="standard:contact">
            <aura:set attribute="actions">
              <lightning:button name="details" label="Details" onclick="{!c.goToRecord}"
 />
            </aura:set>
```

```
            <aura:set attribute="footer">
                <lightning:badge label="{!v.contact.Email}"/>
            </aura:set>
            <p class="slds-p-horizontal--small">
                {!v.contact.Phone}
            </p>
            <p class="slds-p-horizontal--small">
                {!v.contact.MailingStreet}
            </p>
        </lightning:card>

</aura:component>
```

このコンポーネントは、lightning:card コンポーネントを使用して取引先責任者データのテンプレートを作成します。情報グループの周囲にビジュアルコンテナを作成します。このテンプレートは所有する取引先責任者ごとに表示されるため、ビューに、異なるデータを持つ複数のコンポーネントのインスタンスを持つことができます。ボタンがクリックされると、lightning:button コンポーネントの onclick イベントハンドラが goToRecord クライアント側コントローラをコールします。式 {!v.contact.Name} に気づきましたか? v はコンポーネントの一連の属性からなるビューを表し、contact はタイプ Contact の属性です。次のステップで Apex コントローラをコンポーネントに結び付けた後に、このドット表記を使用して、Name や Email などの取引先責任者オブジェクトの項目にアクセスできます。

3. **[File (ファイル)]** > **[New (新規)]** > **[Lightning Component (Lightning コンポーネント)]** をクリックして、[New Lightning Bundle (新規 Lightning バンドル)] ポップアップウィンドウの [Name (名前)] 項目に「*contactList*」と入力します。これにより、contactList.cmp コンポーネントが作成されます。次のコードを入力して保存します。組織で名前空間を使用している場合は、ContactController を myNamespace.ContactController に置き換えます。controller="ContactController" を使用して、 Apex コントローラをコンポーネントに結び付けます。

```
<aura:component implements="force:appHostable" controller="ContactController">
    <!-- Handle component initialization in a client-side controller -->
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <!-- Dynamically load the list of contacts -->
    <aura:attribute name="contacts" type="Contact[]"/>
    <aura:attribute name="contactList" type="Contact[]"/>
    <aura:attribute name="totalContacts" type="Integer"/>

    <!-- Page header with a counter that displays total number of contacts -->
    <div class="slds-page-header slds-page-header--object-home">
        <lightning:layout>
            <lightning:layoutItem>
                <lightning:icon iconName="standard:contact" />
            </lightning:layoutItem>
            <lightning:layoutItem class="slds-m-left--small">
                <p class="slds-text-title--caps slds-line-height--reset">Contacts</p>
                <h1 class="slds-page-header__title slds-p-right--x-small">Contact
Viewer</h1>
            </lightning:layoutItem>
        </lightning:layout>

        <lightning:layout>
```

```
            <lightning:layoutItem>
                <p class="slds-text-body--small">{!v.totalContacts} Contacts • View
Contacts Based on Lead Sources</p>
            </lightning:layoutItem>
        </lightning:layout>
    </div>

    <!-- Body with dropdown menu and list of contacts -->
    <lightning:layout>
        <lightning:layoutItem padding="horizontal-medium" >
            <!-- Create a dropdown menu with options -->
            <lightning:select aura:id="select" label="Lead Source" name="source"
                              onchange="{!c.handleSelect}" class="slds-m-bottom--medium">

                <option value="">-- Select a Lead Source --</option>
                <option value="Referral" text="Referral"/>
                <option value="Social Media" text="Social Media"/>
                <option value="All" text="All"/>
            </lightning:select>

            <!-- Iterate over the list of contacts and display them -->
            <aura:iteration var="contact" items="{!v.contacts}">
                <!-- If you're using a namespace, replace with myNamespace:contacts-->
                <c:contacts contact="{!contact}"/>
            </aura:iteration>
        </lightning:layoutItem>
    </lightning:layout>
</aura:component>
```

では、コードを詳しく見ていきましょう。初期化中に取引先責任者データを読み込むために `init` ハンドラを追加しました。次のステップで、このハンドラはクライアント側コントローラコードをコールします。さらに、取引先責任者のリストとカウンタを格納して取引先責任者の合計数を表示する、`contacts` と `totalContacts` という 2 つの属性を追加しました。また、`contactList` コンポーネントは、リードソースのドロップダウンメニューでオプションが選択されている場合に、取引先責任者の条件設定済みリストを格納するのに使用される属性です。`lightning:layout` コンポーネントはグリッドを作成し、Lightning Design System CSS クラスを使用してビューにコンテンツを配置します。

ページヘッダーには、選択したリードソースに基づいて取引先責任者の数を動的に表示する `{!v.totalContacts}` 式が含まれます。たとえば、**[Referral (紹介)]** を選択し、[リードソース] 項目が [Referral (紹介)] に設定されている 30 の取引先責任者がいる場合、式は 30 に評価されます。

次に、`lightning:select` コンポーネントでドロップダウンメニューを作成します。ドロップダウンメニューでオプションを選択すると、`onchange` イベントハンドラがクライアント側コントローラをコールして、取引先責任者のサブセットでビューを更新します。次のいくつかのステップでは、クライアント側のロジックを作成します。

`force:appHostable` インターフェースを使用して、コンポーネントを Lightning Experience および Salesforce1 にタブとして表示できるか疑問に思っているかもしれませんが、これについては後で説明します。

4. **contactList** サイドバーで、**[CONTROLLER]** をクリックして、`contactListController.js` というリソースを作成します。プレースホルダコードを次のコードに置き換えて保存します。

```
({
    doInit : function(component, event, helper) {
        // Retrieve contacts during component initialization
        helper.loadContacts(component);
    },

    handleSelect : function(component, event, helper) {
        var contacts = component.get("v.contacts");
        var contactList = component.get("v.contactList");

        //Get the selected option: "Referral", "Social Media", or "All"
        var selected = event.getSource().get("v.value");

        var filter = [];
        var k = 0;
        for (var i=0; i<contactList.length; i++){
            var c = contactList[i];
            if (selected != "All"){
                if(c.LeadSource == selected) {
                    filter[k] = c;
                    k++;
                }
            }
            else {
                   filter = contactList;
            }
        }
        //Set the filtered list of contacts based on the selected option
        component.set("v.contacts", filter);
        helper.updateTotal(component);
    }
})
```

クライアント側コントローラは、ヘルパー関数をコールして複雑な作業のほとんどを行います。これはコードの再利用を促すための推奨されるパターンです。ヘルパー関数により、データの処理やサーバ側のアクションの起動などのタスクを特化することもできます。これについては次で説明します。`lightning:select` コンポーネントで `onchange` イベントハンドラを取り消すと、`handleSelect` クライアント側コントローラアクションがコールされます。これは、ドロップダウンメニューでオプションを選択するとトリガされます。`handleSelect` は `event.getSource().get("v.value")` を使用して渡されるオプション値を確認します。各取引先責任者のリードソース項目が選択したリードソースと一致することを確認して、取引先責任者の条件設定済みリストを作成します。最後に、選択したリードソースに基づいて、ビューと取引先責任者の合計数を更新します。

5. **contactList** サイドバーで、**[HELPER]** をクリックして、`contactListHelper.js` というリソースを作成します。プレースホルダコードを次のコードに置き換えて保存します。

```
({
   loadContacts : function(cmp) {
        // Load all contact data
        var action = cmp.get("c.getContacts");
```

```
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                cmp.set("v.contacts", response.getReturnValue());
                cmp.set("v.contactList", response.getReturnValue());
                this.updateTotal(cmp);
            }

            // Display toast message to indicate load status
            var toastEvent = $A.get("e.force:showToast");
            if (state === 'SUCCESS'){
                toastEvent.setParams({
                    "title": "Success!",
                    "message": " Your contacts have been loaded successfully."
                });
            }
            else {
                toastEvent.setParams({
                        "title": "Error!",
                        "message": " Something has gone wrong."
                });
            }
            toastEvent.fire();
        });
         $A.enqueueAction(action);
    },

    updateTotal: function(cmp) {
      var contacts = cmp.get("v.contacts");
      cmp.set("v.totalContacts", contacts.length);
    }
})
```

初期化中、`contactList` コンポーネントは以下によって取引先責任者データを読み込みます。

- Apex コントローラメソッド `getContacts` をコールします。SOQL ステートメントを介して取引先責任者データが返されます。
- アクションのコールバックで `cmp.set("v.contacts", response.getReturnValue())` を介して戻り値を設定します。これにより、取引先責任者データでビューが更新されます。
- ビューで取引先責任者の合計数を更新します。これは `updateTotal` で評価されます。

コンポーネントが Lightning Experience と Salesforce1 でどのように機能するのか疑問に思っているでしょう。次に、これについて調べてみましょう。

6. `contactList` コンポーネントを Lightning Experience と Salesforce1 のカスタムタブを介して使用できるようにします。
    - Lightning Experience のカスタムタブとしての Lightning コンポーネントの追加
    - Salesforce1 のカスタムタブとしての Lightning コンポーネントの追加

このチュートリアルでは、コンポーネントを Lightning Experience でカスタムタブとして追加することをおすすめします。

コンポーネントが Lightning Experience または Salesforce1 に読み込まれると、取引先責任者が正常に読み込まれたことを示すトーストメッセージが表示されます。ドロップダウンメニューからリードソースを選択し、ビューで取引先責任者リストと取引先責任者数の更新を確認します。

次に、取引先責任者リストでボタンをクリックしたときに取引先責任者レコードに移動するイベントを結び付けます。

イベントの起動

クライアント側のコントローラまたはヘルパー関数でイベントを起動します。`force` イベントは Lightning Experience と Salesforce1 で処理されますが、簡潔にするために Lightning Experience でコンポーネントを表示およびテストしてみましょう。

このデモは、「取引先責任者の読み込み」(ページ 13)で作成した取引先責任者コンポーネントを基に作成されています。

1. **contacts** サイドバーで、**[CONTROLLER]** をクリックして、`contactsController.js` というリソースを作成します。プレースホルダコードを次のコードに置き換えて保存します。

```
({
    goToRecord : function(component, event, helper) {
        // Fire the event to navigate to the contact record
        var sObjectEvent = $A.get("e.force:navigateToSObject");
        sObjectEvent.setParams({
            "recordId": component.get("v.contact.Id")
        })
        sObjectEvent.fire();
    }
})
```

   ボタンがクリックされると、次のボタンコンポーネントの `onclick` イベントハンドラが `goToRecord` クライアント側コントローラをトリガします。

```
<lightning:button name="details" label="Details" onclick="{!c.goToRecord}" />
```

   `event.setParams()` 構文を使用して、イベントに渡すパラメータを設定します。この場合、移動する取引先責任者レコードの ID を渡します。`force:navigateToSObject` のほかにも、Lightning Experience と Salesforce1 内のナビゲーションを簡易化するためのその他のイベントがあります。詳細は、「Salesforce1 と Lightning Experience で処理されるイベント」を参照してください。

2. イベントをテストするには、Lightning Experience でカスタムタブを更新して、**[詳細]** ボタンをクリックします。

   `force:navigateToSObject` が起動され、取引先責任者レコードページを表示するビューが更新されます。

クライアント側コントローラと Apex コントローラメソッドの組み合わせを使用して取引先責任者データを読み込むコンポーネントを作成し、Salesforce データでカスタム UI を作成する手順について説明しました。Lightning コンポーネントで実行可能な操作は無限にあります。Lightning Experience と Salesforce1 でタブを介してコンポーネントを表示する方法を紹介しましたが、Lightning アプリケーションビルダーおよびコミュニティを介してレコードページ上にコンポーネントを表示することで、更に上級の操作を実行できます。「Trailhead: Lightning コ

ンポーネントリソースの調査」にあるリソースを参照して実行可能な操作について探索し、新たな道を切り開いてください。

第 3 章　コンポーネントの作成

トピック:

- 開発者コンソールで Lightning コンポーネントを作成する
- コンポーネントのマークアップ
- コンポーネントの名前空間
- コンポーネントのバンドル
- コンポーネントの ID
- コンポーネント内の HTML
- コンポーネント内の CSS
- コンポーネントの属性
- コンポーネントのコンポジション
- コンポーネントのボディ
- コンポーネントのファセット
- 条件付きマークアップのベストプラクティス
- コンポーネントのバージョン設定
- 最小 API バージョン要件のあるコンポーネント
- 式の使用
- 表示ラベルの使用

コンポーネントは、Lightning コンポーネントフレームワークの機能単位です。

コンポーネントは、モジュール形式で再利用可能な UI のセクションをカプセル化します。テキスト 1 行からアプリケーション全体までさまざまな粒度に対応できます。

- ローカライズ
- コンポーネントのドキュメントの提供
- Lightning 基本コンポーネントの使用
- UI コンポーネントの操作
- フロー Lightning コンポーネントの操作
- アクセシビリティのサポート

開発者コンソールで Lightning コンポーネントを作成する

開発者コンソールは、Lightningコンポーネントやその他のバンドルを新規作成したり既存のものを編集したりできる便利な組み込みツールです。

1. 開発者コンソールを開きます。

   *あなたの名前の*下にある [開発者コンソール]、またはクイックアクセスメニュー ( ) を選択します。

2. Lightning コンポーネントの [New Lightning Bundle (新規 Lightning バンドル)] パネルを開きます。

   [File (ファイル)] > **[New (新規)]** > **[Lightning Component (Lightning コンポーネント)]** を選択します。

3. コンポーネントに名前を付けます。

   たとえば、[Name (名前)] 項目に「`helloWorld`」と入力します。

4. オプション: コンポーネントについて説明します。

   [Description (説明)] 項目を使用してコンポーネントの詳細を追加します。

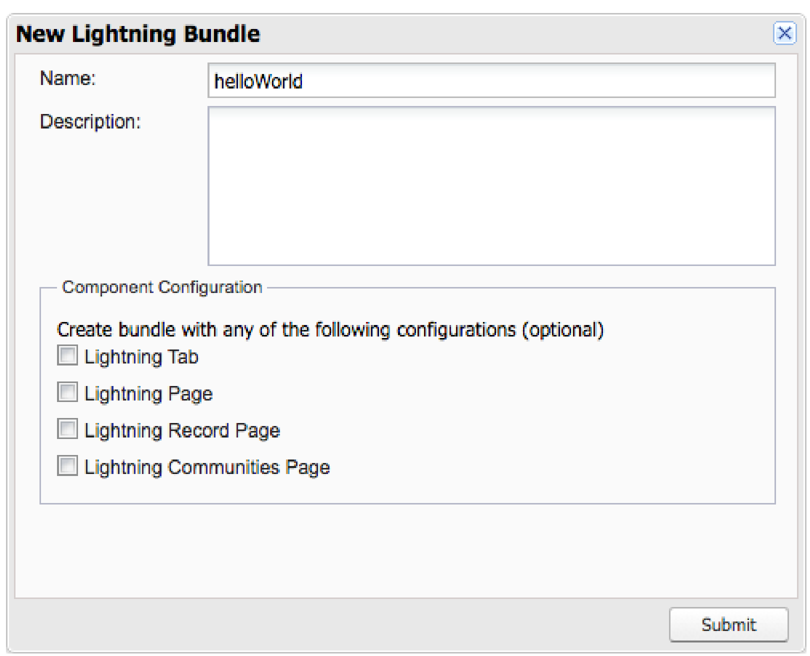
5. オプション: 新しいコンポーネントにコンポーネント設定を追加します。

   [Component Configuration (コンポーネント設定)] セクションでオプションを好きなだけ選ぶことも、設定を一切選択しないこともできます。

6. [送信] をクリックして、コンポーネントを作成します。

   コンポーネントの作成をキャンセルする場合は、右上にあるパネルの閉じるボックスをクリックします。

例:



このセクションの内容:

開発者コンソールで使用できる Lightning バンドル設定

この設定により、Lightning ページや Lightning コミュニティページ、または Lightning Experience や Salesforce1 のクイックアクションやナビゲーション項目など、特定の用途のコンポーネントまたはアプリケーションを簡単に作成できます。開発者コンソールの [New Lightning Bundle (新規 Lightning バンドル)] パネルでは、Lightning コンポーネントまたはアプリケーションバンドルを作成するときにコンポーネントの設定を選択できます。

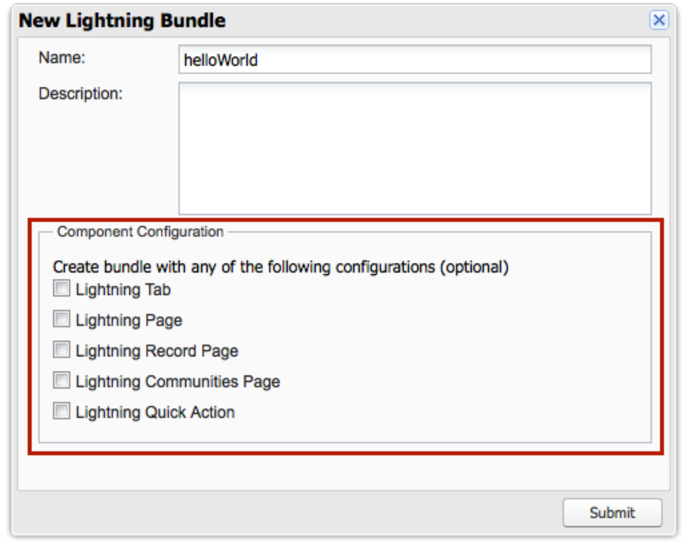
関連トピック:

開発者コンソールの使用
開発者コンソールで使用できる Lightning バンドル設定

開発者コンソールで使用できる Lightning バンドル設定

この設定により、Lightning ページや Lightning コミュニティページ、または Lightning Experience や Salesforce1 のクイックアクションやナビゲーション項目など、特定の用途のコンポーネントまたはアプリケーションを簡単に作成できます。開発者コンソールの [New Lightning Bundle (新規 Lightning バンドル)] パネルでは、Lightning コンポーネントまたはアプリケーションバンドルを作成するときにコンポーネントの設定を選択できます。

設定により、目的の状況でのコンポーネントの使用をサポートするために必要なインターフェースが追加されます。たとえば、**[Lightning タブ]** 設定を選択すると、新しいコンポーネントの `<aura:component>` タグに `implements="force:appHostable"` が挿入されます。



設定の使用は任意です。すべて使用することから一切使用しないことまで、任意の組み合わせで使用できます。

[New Lightning Bundle (新規 Lightning バンドル)] パネルでは、次の設定を使用できます。

設定	マークアップ	説明
Lightning コンポーネントバンドル		

設定	マークアップ	説明
Lightning タブ	`implements="force:appHostable"`	Lightning Experience または Salesforce1 のナビゲーション要素として使用するコンポーネントを作成します。
Lightning ページ	`implements="flexipage:availableForAllPageTypes"` および `access="global"`	Lightning ページまたは Lightning アプリケーションビルダーで使用するコンポーネントを作成します。
Lightning レコードページ	`implements="flexipage:availableForRecordHome, force:hasRecordId"` および `access="global"`	Lightning Experience のレコードホームページで使用するコンポーネントを作成します。
Lightning コミュニティページ	`implements="forceCommunity:availableForAllPageTypes"` および `access="global"`	コミュニティビルダーでドラッグアンドドロップできるコンポーネントを作成します。
Lightning クイックアクション	`implements="force:lightningQuickAction"`	Lightning クイックアクションと使用できるコンポーネントを作成します。
Lightning アプリケーションバンドル		
Lightning Out 連動関係アプリケーション	`extends="ltng:outApp"`	空の Lightning Out 連動関係アプリケーションを作成します。

メモ: 各設定によって追加されるマークアップについての詳細は、各機能のドキュメントを参照してください。

関連トピック:

開発者コンソールで Lightning コンポーネントを作成する
インターフェースの参照
カスタムタブのコンポーネントの設定
カスタムアクション用のコンポーネントの設定
Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定
Lightning Experience のレコードホームページのコンポーネントの設定
コミュニティのコンポーネントの設定

コンポーネントのマークアップ

コンポーネントリソースにはマークアップが含まれ、.cmp サフィックスが付いています。マークアップには、テキストまたは他のコンポーネントへの参照を含めることができます。また、マークアップはコンポーネントに関するメタデータの宣言も行います。

まず、helloWorld.cmp コンポーネントのシンプルな「Hello, world!」の例から始めましょう。

```
<aura:component>
    Hello, world!
</aura:component>
```

この例では、コンポーネントを可能な限りシンプルにしています。テキスト「Hello, world!」は、<aura:component> タグでラップされています。このタグは、すべてのコンポーネント定義の最初と最後にあります。

コンポーネントには、ほとんどの HTML タグを含めることができるため、<div> や <span> などのマークアップを使用できます。HTML5 タグもサポートされています。

```
<aura:component>
    <div class="container">
        <!--Other HTML tags or components here-->
    </div>
</aura:component>
```

メモ: マークアップは JavaScript、CSS、および Apex と連動するため、大文字と小文字を区別する必要があります。

コンポーネントを作成するには、開発者コンソールを使用します。

コンポーネントの命名規則

コンポーネント名は、次の命名規則に従う必要があります。

- 文字で始まる
- 英数字とアンダースコアのみで構成される。
- 名前空間内で一意である
- 空白が含まれていない
- 末尾がアンダースコアではない
- アンダースコアが 2 つ続けて使用されていない

関連トピック:

aura:component

開発者コンソールの使用

コンポーネントのアクセス制御

カスタムレンダラの作成

コンポーネントの動的な作成

コンポーネントの名前空間

各コンポーネントは1つの名前空間に属しています。名前空間は関連するコンポーネントをまとめてグループ化するために使用されます。組織に名前空間プレフィックスが設定されている場合は、その名前空間を使用してコンポーネントにアクセスします。設定されていない場合は、デフォルトの名前空間を使用してコンポーネントにアクセスします。

`<myNamespace:myComponent>` をマークアップに追加することで、別のコンポーネントまたはアプリケーションがコンポーネントを参照できます。たとえば、`helloWorld` コンポーネントは `docsample` 名前空間内にあります。別のコンポーネントは、`<docsample:helloWorld />` をマークアップに追加することで、このコンポーネントを参照できます。

Salesforce が提供する Lightning コンポーネントは、`aura`、`ui`、`force` などいくつかの名前空間にグループ化されています。サードパーティの管理パッケージのコンポーネントには、提供元組織が定めた名前空間があります。

組織で、名前空間プレフィックスの設定を選択できます。設定する場合は、すべてのLightningコンポーネントにその名前空間が使用されます。AppExchangeで管理パッケージを提供する予定の場合は、名前空間プレフィックスが必須です。

組織に名前空間プレフィックスを設定していない場合は、作成したコンポーネントを参照するときにデフォルトの名前空間 `c` を使用します。

コードサンプルの名前空間

このガイド全体を通じて、コードサンプルでは、デフォルトの名前空間 `c` を使用します。名前空間プレフィックスを設定している場合は、`c` をご使用の名前空間に置き換えます。

名前空間プレフィックスが設定されていない組織でのデフォルトの名前空間の使用

組織に名前空間プレフィックスが設定されていない場合は、作成したLightningコンポーネントを参照するときにデフォルトの名前空間 `c` を使用します。

次の項目については、組織に名前空間プレフィックスが設定されていない場合に、`c` 名前空間を使用する必要があります。

- 作成したコンポーネントへの参照
- 定義したイベントへの参照

次の項目については、組織の黙示的な名前空間を使用し、名前空間を指定する必要はありません。

- カスタムオブジェクトへの参照
- 標準オブジェクトおよびカスタムオブジェクトのカスタム項目への参照
- Apex コントローラへの参照

上記のすべての項目の例については、「名前空間の使用例および参照」 (ページ 28) を参照してください。

組織の名前空間の使用

組織に名前空間プレフィックスが設定されている場合は、その名前空間を使用して Lightning コンポーネント、イベント、カスタムオブジェクト、カスタム項目、および Lightning マークアップのその他の項目を参照します。

次の項目は、組織に名前空間プレフィックスが設定されている場合、組織の名前空間を使用します。

- 作成したコンポーネントへの参照
- 定義したイベントへの参照
- カスタムオブジェクトへの参照
- 標準オブジェクトおよびカスタムオブジェクトのカスタム項目への参照
- Apex コントローラへの参照
- 静的リソースへの参照

メモ: 名前空間プレフィックスが設定されている組織の c 名前空間のサポートは完全ではありません。次の項目では、ショートカットを使用する場合に c 名前空間を使用できますが、現在は推奨されていません。

- 作成したコンポーネントを Lightning マークアップで使用する場合のそのコンポーネントへの参照 (式または JavaScript で使用する場合を除く)
- 定義したイベントを Lightning マークアップで使用する場合のそのイベントへの参照 (式または JavaScript で使用する場合を除く)
- カスタムオブジェクトをコンポーネントやイベントの `type` および `default` システム属性で使用する場合のそのオブジェクトへの参照 (式または JavaScript で使用する場合を除く)

上記のすべての項目の例については、「名前空間の使用例および参照」 (ページ 28) を参照してください。

管理パッケージでのまたは管理パッケージからの名前空間の使用

管理パッケージから項目を参照する場合や、自分の管理パッケージでの配布を目的とするコードを作成する場合は、常に完全な名前空間を使用します。

組織の名前空間の作成

名前空間プレフィックスを登録して、組織の名前空間を作成します。

配布用の管理パッケージを作成しない場合は、名前空間プレフィックスを登録する必要はありませんが、ごく小規模な組織を除き、どの組織にとっても登録することがベストプラクティスです。

名前空間プレフィックスは、以下の条件で指定します。

- 1 文字目が英字である。
- 1 ～ 15 文字までの英数字を含む。
- アンダースコア (_) を 2 つ続けて入力しない。

たとえば、`myNp123` と `my_np` は有効な名前空間ですが、`123Company` と `my__np` は無効です。

名前空間プレフィックスを登録する手順は、次のとおりです。

1. [設定]から、[クイック検索]ボックスに「パッケージ」と入力します。[作成]で[パッケージ]を選択します。

   メモ: この項目は、Salesforce Classic でのみ使用できます。

2. [開発者設定] パネルで、[編集] をクリックします。

   メモ: すでに開発者設定が定義されている場合は、このボタンは表示されません。

3. 開発者設定に必要な選択項目を確認し、[続行] をクリックします。
4. 登録する名前空間プレフィックスを入力します。
5. [使用可能か調べる] をクリックして、名前空間プレフィックスが使用済みかどうかを確認します。
6. 入力した名前空間プレフィックスを使用できない場合は、上記の 2 つの手順を繰り返します。
7. [選択内容の確認] をクリックします。
8. [保存] をクリックします。

名前空間の使用例および参照

このトピックでは、Lightningコンポーネントのコードでコンポーネント、オブジェクト、項目などを参照する例を示します。

次の例が含まれています。

- 組織のコンポーネント、イベント、およびインターフェース
- 組織のカスタムオブジェクト
- 組織の標準オブジェクトおよびカスタムオブジェクトのカスタム項目
- 組織のサーバ側の Apex コントローラ
- JavaScript のコンポーネントの動的作成
- 組織内の静的リソース

名前空間プレフィックスが設定されていない組織

組織に名前空間プレフィックスが設定されていない場合の組織の要素への参照を、次に示します。参照は必要に応じて、デフォルトの名前空間である c を使用します。

参照される項目	例
マークアップで使用されるコンポーネント	`<c:myComponent />`
システム属性で使用されるコンポーネント	`<aura:component extends="c:myComponent">` `<aura:component implements="c:myInterface">`
Apex コントローラ	`<aura:component controller="ExpenseController">`
属性データ型のカスタムオブジェクト	`<aura:attribute name="expense" type="Expense__c" />`

参照される項目	例
属性のデフォルトのカスタムオブジェクトまたはカスタム項目	`<aura:attribute name="newExpense" type="Expense__c" default="{ 'sobjectType': 'Expense__c', 'Name': '', 'Amount__c': 0, … }" />`
式のカスタム項目	`<ui:inputNumber value="{!v.newExpense.Amount__c}" label=… />`
JavaScript 関数のカスタム項目	`updateTotal: function(component) { … for(var i = 0 ; i < expenses.length ; i++){ var exp = expenses[i]; total += exp.Amount__c; } … }`
JavaScript 関数で動的に作成されたコンポーネント	`var myCmp = $A.createComponent("c:myComponent", {}, function(myCmp) { } );`
JavaScript 関数のインターフェース比較	`aCmp.isInstanceOf("c:myInterface")`
イベントの登録	`<aura:registerEvent type="c:updateExpenseItem" name=… />`
イベントハンドラ	`<aura:handler event="c:updateExpenseItem" action=… />`
明示的な連動関係	`<aura:dependency resource="markup://c:myComponent" />`
JavaScript 関数のアプリケーションイベント	`var updateEvent = $A.get("e.c:updateExpenseItem");`
静的リソース	`<ltng:require scripts="{!$Resource.resourceName}" styles="{!$Resource.resourceName}" />`

名前空間プレフィックスのある組織

組織に名前空間プレフィックスが設定されている場合の組織の要素への参照を、次に示します。参照は、サンプルの名前空間 `yournamespace` を使用します。

参照される項目	例
マークアップで使用されるコンポーネント	`<yournamespace:myComponent />`

参照される項目	例
システム属性で使用されるコンポーネント	<aura:component extends="**yournamespace:myComponent**"> <aura:component implements="**yournamespace:myInterface**">
Apex コントローラ	<aura:component controller="**yournamespace.ExpenseController**">
属性データ型のカスタムオブジェクト	<aura:attribute name="expenses" type="**yournamespace__Expense__c[]**" />

属性のデフォルトのカスタムオブジェクトまたはカスタム項目	<aura:attribute name="newExpense"
	type="yournamespace__Expense__c"
	    default="{ 'sobjectType': '**yournamespace__Expense__c**',
	                'Name': '',
	                '**yournamespace__Amount__c**': 0,
	                …
	    }" />

式のカスタム項目	<ui:inputNumber value="{!v.newExpense.**yournamespace__Amount__c**}" label=… />

JavaScript 関数のカスタム項目	updateTotal: function(component) {
	    …
	    for(var i = 0 ; i < expenses.length ; i++){
	        var exp = expenses[i];
	        total += exp.**yournamespace__Amount__c**;
	    }
	    …
	}

JavaScript 関数で動的に作成されたコンポーネント	var myCmp = $A.createComponent("**yournamespace:myComponent**",
	    {},
	    function(myCmp) { }
	);

JavaScript 関数のインターフェース比較	aCmp.isInstanceOf("**yournamespace:myInterface**")
イベントの登録	<aura:registerEvent type="**yournamespace:updateExpenseItem**" name=… />
イベントハンドラ	<aura:handler event="**yournamespace:updateExpenseItem**" action=… />
明示的な連動関係	<aura:dependency resource="markup://**yournamespace:myComponent**" />
JavaScript 関数のアプリケーションイベント	var updateEvent = $A.get("e.**yournamespace:updateExpenseItem**");

参照される項目	例
静的リソース	`<ltng:require scripts="{!$Resource.yournamespace__resourceName}" styles="{!$Resource.yournamespace__resourceName}" />`

コンポーネントのバンドル

コンポーネントのバンドルには、コンポーネントまたはアプリケーションとそれに関連するすべてのリソースが含まれます。

リソース	リソース名	使用方法	関連トピック
コンポーネントまたはアプリケーション	sample.cmp または sample.app	バンドル内の唯一の必須リソース。コンポーネントまたはアプリケーションのマークアップが含まれます。各バンドルに含まれるコンポーネントまたはアプリケーションリソースは 1 つのみです。	コンポーネントの作成 (ページ 20) aura:application (ページ 428)
CSS スタイル	sample.css	コンポーネントのスタイルが含まれます。	コンポーネント内の CSS (ページ 34)
コントローラ	sampleController.js	コンポーネント内のイベントを処理するクライアント側コントローラのメソッドが含まれます。	クライアント側コントローラを使用したイベントの処理 (ページ 185)
設計	sample.design	Lightning アプリケーションビルダー、Lightning ページ、またはコミュニティビルダーで使用されるコンポーネントに必要なファイル。	Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定
ドキュメント	sample.auradoc	説明、サンプルコード、およびコンポーネント例への 1 つ以上の参照	コンポーネントのドキュメントの提供 (ページ 75)
レンダラ	sampleRenderer.js	コンポーネントのデフォルトの表示を上書きするクライアント側レンダラ。	カスタムレンダラの作成 (ページ 284)

リソース	リソース名	使用方法	関連トピック
ヘルパー	sampleHelper.js	コンポーネントのバンドル内の JavaScript コードからコール可能な JavaScript 関数	コンポーネントのバンドル内の JavaScript コードの共有 (ページ 271)
SVG ファイル	sample.svg	Lightning アプリケーションビルダーまたはコミュニティビルダーで使用されるコンポーネントのカスタムアイコンのリソース。	Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定 (ページ 139)

コンポーネントのバンドル内のすべてのリソースは命名規則に従い、自動的に結び付けられます。たとえば、コントローラ `<componentName>Controller.js` は、そのコンポーネントに自動的に結び付けられます。つまり、コンポーネントの範囲内で使用できます。

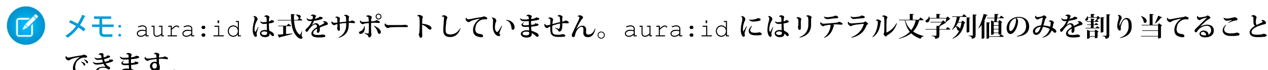
コンポーネントの ID

コンポーネントには、ローカル ID とグローバル ID という 2 種類の ID があります。JavaScript コードにローカル ID を使用して、コンポーネントを取得できます。グローバル ID は、コンポーネントの複数のインスタンスを区別する場合や、デバッグする場合に便利です。

ローカル ID

ローカル ID は、範囲がそのコンポーネント内のみの ID です。多くの場合ローカル ID は一意ですが、一意である必要はありません。

ローカル ID を作成するには、`aura:id` 属性を使用します。次に例を示します。



```
<lightning:button aura:id="button1" label="button1"/>
```

メモ: `aura:id` は式をサポートしていません。`aura:id` にはリテラル文字列値のみを割り当てることができます。

このボタンコンポーネントを検索するには、クライアント側コントローラで `cmp.find("button1")` をコールします。ここで `cmp` は、ボタンが含まれるコンポーネントへの参照です。

`find()` は、結果によって異なる種別を返します。

- ローカル ID が一意である場合、`find()` はコンポーネントを返します。
- 同じローカル ID のコンポーネントが複数ある場合、`find()` はコンポーネントの配列を返します。
- 一致するローカル ID がない場合、`find()` は `undefined` を返します。

JavaScript でコンポーネントのローカル ID を検索するには、`cmp.getLocalId()` を使用します。

グローバル ID

すべてのコンポーネントには一意の globalId があります。これはコンポーネントインスタンスに対して生成される実行時に一意の ID です。グローバル ID (1) は、コンポーネントの有効期間以外では同じである保証はないため、利用しないでください。グローバル ID は、コンポーネントの複数のインスタンスを区別する場合や、デバッグする場合に便利です。

```
▼<div class="slds-select_container" data-aura-rendered-by="1624:0">
  ::before
 ▼<select class="slds-select" id="1611:0" data-aura-rendered-by="1625:0" name="select" required aria-describedby="1611:0-desc">
    <option data-aura-rendered-by="1613:0">Red</option>
    <option data-aura-rendered-by="1614:0">Green</option>
    <option data-aura-rendered-by="1615:0">Blue</option>
  </select>
  ::after
</div>
```

1

HTML 要素に一意の ID を作成するために、globalId を要素のプレフィックスまたはサフィックスとして使用できます。次に例を示します。

```
<div id="{!globalId + '_footer'}"></div>
```

ブラウザの開発者コンソールで、document.getElementById("<globalId>_footer") を使用して要素を取得します。<globalId> は、生成される実行時に一意の ID です。

JavaScript でコンポーネントのグローバル ID を取得するには、getGlobalId() 関数を使用します。

```
var globalId = cmp.getGlobalId();
```

関連トピック:

ID によるコンポーネントの検索

押下されたボタンの確認

コンポーネント内の HTML

HTML タグは、フレームワークで第一級のコンポーネントとして処理されます。各HTML タグは、<aura:html> コンポーネントに変換され、他のコンポーネントと同様の権限を使用できます。

たとえば、フレームワークは、標準の HTML <div> タグを次のコンポーネントに自動的に変換します。

```
<aura:html tag="div" />
```

コンポーネントに HTML マークアップを追加できます。厳密な XHTML を使用する必要がある点に注意してください。たとえば、
 ではなく
 を使用します。HTML 属性と、onclick などの DOM イベントも使用できます。

警告: <applet> や <font> など、一部のタグはサポートされていません。サポートされていないタグの全リストは、「サポートされる HTML タグ」 (ページ 711)を参照してください。

HTML のエスケープ解除

書式設定済みの HTML を出力するには、`aura:unescapedHTML` を使用します。これは、たとえば、サーバで生成された HTML を表示し、DOM に追加する場合に便利です。必要に応じて HTML をエスケープする必要があります。これを行わないと、アプリケーションにセキュリティの脆弱性が生じるおそれがあります。

`<aura:unescapedHtml value="{!v.note.body}"/>` のように、式から値を渡すことができます。

`{!expression}` はフレームワークの式の構文です。詳細は、「式の使用」 (ページ 45)を参照してください。

関連トピック:

サポートされる HTML タグ

コンポーネント内の CSS

コンポーネント内の CSS

CSS を使用してコンポーネントのスタイルを設定します。

CSSをコンポーネントのバンドルに追加するには、開発者コンソールのサイドバーで **[STYLE]** ボタンをクリックします。

外部 CSS リソースの場合は、「アプリケーションのスタイル設定」 (ページ 248)を参照してください。

コンポーネントのすべての最上位要素には、特殊な `THIS` CSS クラスが追加されています。これにより、事実上 CSS に名前空間設定が追加されます。これは、コンポーネントの CSS が別のコンポーネントのスタイル設定を上書きすることを回避するのに役立ちます。CSS ファイルがこの規則に従わない場合、フレームワークはエラーを発生させます。

サンプルの `helloHTML.cmp` コンポーネントを見てみましょう。CSS は `helloHTML.css` 内にあります。

コンポーネントのソース

```
<aura:component>
  <div class="white">
    Hello, HTML!
  </div>

  <h2>Check out the style in this list.</h2>

  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
    <li class="green">I'm green.</li>
  </ul>
</aura:component>
```

CSS ソース

```
.THIS {
    background-color: grey;
}
```

```
.THIS.white {
    background-color: white;
}

.THIS .red {
    background-color: red;
}

.THIS .blue {
    background-color: blue;
}

.THIS .green {
    background-color: green;
}
```

出力

Hello, HTML!
Check out the style in this list.

最上位要素の `h2` と `ul` は `THIS` クラスと一致し、灰色の背景で表示されます。最上位要素は HTML `body` タグでラップされるタグで、他のタグでラップされることはありません。この例では、`li` タグは `ul` タグにネストされているため、最上位要素ではありません。

`<div class="white">` 要素は `.THIS.white` セレクタと一致し、白の背景で表示されます。このルールは最上位要素用であるため、セレクタにはスペースがありません。

`<li class="red">` 要素は `.THIS .red` セレクタと一致し、赤の背景で表示されます。これは下位のセレクタであり、`<li>` は最上位要素ではないため、スペースが含まれます。

関連トピック:

スタイルの追加と削除

コンポーネント内の HTML

コンポーネントの属性

コンポーネントの属性は、Apex のクラスのメンバー変数に似ています。これらは型付けされた項目で、コンポーネントの特定のインスタンスに設定されており、式の構文を使用したコンポーネントのマークアップ内から参照できます。属性を使用すると、コンポーネントをより動的に扱うことができます。

属性をコンポーネントまたはアプリケーションに追加するには、`<aura:attribute>` タグを使用します。次のサンプル `helloAttributes.app` を見てみましょう。

```
<aura:application>
    <aura:attribute name="whom" type="String" default="world"/>
    Hello {!v.whom}!
</aura:application>
```

すべての属性には名前と型があります。属性には、required="true" を指定して必須としてマークできます。デフォルト値を指定することもできます。

このサンプルには、whom という名前の文字列型の属性があります。値が指定されない場合は、デフォルトの「world」になります。

厳格な要件ではありませんが、<aura:attribute> タグは通常、コンポーネントのマークアップの先頭に置きます。こうすることで、コンポーネントの形状を一目で簡単に参照できるためです。

属性の命名規則

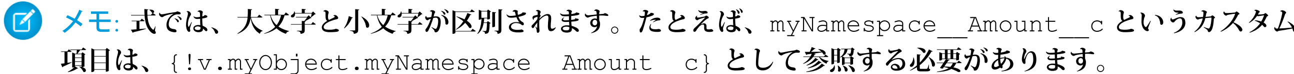
属性名は、次の命名規則に従う必要があります。

- 先頭文字が英字またはアンダースコアである。
- 英数字とアンダースコアのみで構成される。

式

helloAttributes.app には、コンポーネントの動的出力を担う式 {!v.whom} が含まれます。

{!***expression***} はフレームワークの式の構文です。この場合、評価する式は v.whom です。定義した属性の名前が whom で、v が、ビューを表すコンポーネントの属性セットに値を提供します。



メモ: 式では、大文字と小文字が区別されます。たとえば、myNamespace__Amount__c というカスタム項目は、{!v.myObject.myNamespace__Amount__c} として参照する必要があります。

関連トピック:

サポートされる aura:attribute の型

式の使用

コンポーネントのコンポジション

コンポーネントを、細分化された複数のコンポーネントで構成することで、さまざまなコンポーネントとアプリケーションを作成できます。

コンポーネントをどのようにまとめられるか見てみましょう。まず、簡単なコンポーネント c:helloHTML と c:helloAttributes を作成しましょう。その後で、ラッパーコンポーネントの c:nestedComponents を作成し、簡単なコンポーネントを囲みます。

helloHTML.cmp のソースは次のようになります。

```
<!--c:helloHTML-->
<aura:component>
  <div class="white">
    Hello, HTML!
  </div>

  <h2>Check out the style in this list.</h2>
```

```
  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
    <li class="green">I'm green.</li>
  </ul>
</aura:component>
```

CSS ソース

```
.THIS {
    background-color: grey;
}

.THIS.white {
    background-color: white;
}

.THIS .red {
    background-color: red;
}

.THIS .blue {
    background-color: blue;
}

.THIS .green {
    background-color: green;
}
```

出力

Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

`helloAttributes.cmp` のソースは次のようになります。

```
<!--c:helloAttributes-->
<aura:component>
    <aura:attribute name="whom" type="String" default="world"/>
    Hello {!v.whom}!
</aura:component>
```

`nestedComponents.cmp` では、他のコンポーネントをマークアップ内に追加するコンポジションを使用します。

```
<!--c:nestedComponents-->
<aura:component>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="component composition"/>
</aura:component>
```

出力

Observe! Components within components!
Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

Hello component composition!

既存のコンポーネントを追加するのは、HTML タグの挿入に似ています。コンポーネントをその *namespace:component* 形式の「記述子」で参照します。`nestedComponents.cmp` は、`c` 名前空間内に存在する `helloHTML.cmp` コンポーネントを参照します。したがって、その記述子は `c:helloHTML` です。

`nestedComponents.cmp` が `c:helloAttributes` をどのように参照しているかについても注目してください。HTML タグに属性を追加するのと同様に、コンポーネント内の属性値をコンポーネントタグの一部として設定できます。`nestedComponents.cmp` では、`helloAttributes.cmp` の `whom` 属性を「component composition」に設定しています。

属性の渡し方

属性をネストされたコンポーネントに渡すこともできます。`nestedComponents2.cmp` は `nestedComponents.cmp` と似ていますが、`passthrough` 属性が含まれている点が異なります。この値は `c:helloAttributes` の属性値として渡されます。

```
<!--c:nestedComponents2-->
<aura:component>
    <aura:attribute name="passthrough" type="String" default="passed attribute"/>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="{#v.passthrough}"/>
</aura:component>
```

出力

Observe! Components within components!
Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

Hello passed attribute!

`helloAttributes` が渡された属性値を使用しています。

メモ: `{#v.passthrough}` は非バインド式です。つまり、`c:helloAttributes` の `whom` 属性値を変更しても、逆伝播して `c:nestedComponents2` の `passthrough` 属性値に影響が及ぶことはありません。詳細は、「コンポーネント間のデータバインド」(ページ 48)を参照してください。

定義とインスタンス

オブジェクト指向プログラミングでは、クラスとそのクラスのインスタンスには違いがあります。コンポーネントにも同じような概念があります。`.cmp` リソースを作成することで、そのコンポーネントの定義 (クラス) を指定します。`.cmp` にコンポーネントタグを追加することで、そのコンポーネント (のインスタンス) への参照を作成します。

もちろん、異なる属性を持つ同じコンポーネントのインスタンスを複数追加することもできます。nestedComponents3.cmp では、異なる属性値を持つ別のインスタンスの c:helloAttributes を追加します。c:helloAttributes コンポーネントの 2 つのインスタンスでは、それぞれの whom 属性の値が異なっています。

```
<!--c:nestedComponents3-->
<aura:component>
    <aura:attribute name="passthrough" type="String" default="passed attribute"/>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="{#v.passthrough}"/>

    <c:helloAttributes whom="separate instance"/>
</aura:component>
```

出力

Observe! Components within components!
Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

Hello passed attribute! Hello separate instance!

コンポーネントのボディ

すべてのコンポーネントのルートレベルタグは <aura:component> です。すべてのコンポーネントは <aura:component> から body 属性を継承します。

<aura:component> タグには、<aura:attribute>、<aura:registerEvent>、<aura:handler>、<aura:set> などのタグを含めることができます。コンポーネント内で許可されるタグのいずれかで囲まれていない独立したマークアップは、ボディの一部とみなされ、body 属性内に設定されます。

body 属性の型は Aura.Component[] です。1 つのコンポーネントの配列にすることも、空の配列にすることもできますが、常に配列です。

コンポーネントでは、属性のコレクションにアクセスするには「v」を使用します。たとえば、{!v.body} はコンポーネントのボディを出力します。

ボディコンテンツの設定

コンポーネントに body 属性を設定するには、独立したマークアップを <aura:component> タグ内に追加します。次に例を示します。

```
<aura:component>
    <!--START BODY-->
    <div>Body part</div>
    <lightning:button label="Push Me" onclick="{!c.doSomething}"/>
    <!--END BODY-->
</aura:component>
```

継承された属性の値を設定するには、<aura:set> タグを使用します。ボディコンテンツを設定することは、その独立したマークアップを <aura:set attribute="body"> 内にラップすることと同じです。body 属性にはこの特殊な動作があるため、<aura:set attribute="body"> を省略できます。

上記のサンプルは、次のマークアップを簡易にしたものです。より簡易な上記のサンプルの構文を使用することをお勧めします。

```
<aura:component>
    <aura:set attribute="body">
        <!--START BODY-->
        <div>Body part</div>
        <lightning:button label="Push Me" onclick="{!c.doSomething}"/>
        <!--END BODY-->
    </aura:set>
</aura:component>
```

<aura:component> だけでなく、body 属性があるどのコンポーネントを使用する場合も同様です。次に例を示します。

```
<lightning:tabset>
    <lightning:tab label="Tab 1">
        Hello world!
    </lightning:tab>
</lightning:tabset>
```

これは次の指定を簡易にしたものです。

```
<lightning:tabset>
    <lightning:tab label="Tab 1">
        <aura:set attribute="body">
            Hello World!
        </aura:set>
    </lightning:tab>
</lightning:tabset>
```

コンポーネントのボディへのアクセス

JavaScript のコンポーネントのボディにアクセスするには、component.get("v.body") を使用します。

関連トピック:

aura:set

JavaScript でのコンポーネントのボディの操作

コンポーネントのファセット

ファセットは、Aura.Component[] 型の属性です。body 属性は、ファセットの一例です。

独自のファセットを定義するには、Aura.Component[] 型の aura:attribute タグをコンポーネントに追加します。たとえば、facetHeader.cmp という新しいコンポーネントを作成するとします。

```
<!--c:facetHeader-->
<aura:component>
    <aura:attribute name="header" type="Aura.Component[]"/>

    <div>
        <span class="header">{!v.header}</span><br/>
        <span class="body">{!v.body}</span>
    </div>
</aura:component>
```

このコンポーネントにはヘッダーファセットがあります。ヘッダーの出力は、v.header 式を使用して配置されています。

header および body 属性が設定されていないため、このコンポーネントに直接アクセスしたとき、コンポーネントからの出力はありません。これらの属性を設定する別のコンポーネント helloFacets.cmp を作成しましょう。

```
<!--c:helloFacets-->
<aura:component>
    See how we set the header facet.<br/>

    <c:facetHeader>

        Nice body!

        <aura:set attribute="header">
            Hello Header!
        </aura:set>
    </c:facetHeader>

</aura:component>
```

aura:set は、facetHeader.cmp の header 属性の値を設定しますが、body 属性を設定する場合は、aura:set を使用する必要はありません。

関連トピック:

コンポーネントのボディ

条件付きマークアップのベストプラクティス

マークアップを条件に応じて表示する方法として、<aura:if> タグの使用をお勧めしますが、代替方法もあります。コンポーネントを設計するときは、パフォーマンスコストおよびコードの保守性を考慮します。設計上の最適な選択は、使用事例によって異なります。

<aura:if> を使用して条件に応じて要素を作成する

エラーの発生時にエラーメッセージを表示する簡単な例を見てみましょう。

```
<aura:if isTrue="{!v.isError}">
    <div>{!v.errorMessage}</div>
</aura:if>
```

`isTrue` 式の値が `true` に評価された場合、`<div>` コンポーネントとそのコンテンツのみが作成されて表示されます。`isTrue` 式の値が変更され、`false` に評価された場合、`<aura:if>` タグ内のすべてのコンポーネントが破棄されます。`isTrue` 式が再度変更され、`true` に評価された場合、コンポーネントが再度作成されます。

一般的なガイドラインとして、`<aura:if>` を使用することをお勧めします。囲まれた要素ツリーの作成と表示を条件が満たされるまで延期することで、コンポーネントを最初に迅速に読み込むことができるためです。

CSS を使用して表示を切り替える

CSS を使用して、JavaScript コードで `$A.util.toggleClass(cmp, 'class')` をコールすることで、マークアップの表示を切り替えることができます。

マークアップ内の要素は事前に作成されてレンダリングされますが、非表示です。例については、「マークアップの動的な表示または非表示」を参照してください。

条件付きマークアップは、使用されない場合も作成されレンダリングされます。このため、`<aura:if>` をお勧めします。

JavaScript でコンポーネントを動的に作成する

コンポーネントは JavaScript コードで動的に作成できます。ただし、通常、マークアップを使用するよりも、コード作成の維持とデバッグが困難になります。`<aura:if>` の使用を再度お勧めしますが、使用事例に合わせて最適なデザインを選択してください。

関連トピック:

aura:if

条件式

コンポーネントの動的な作成

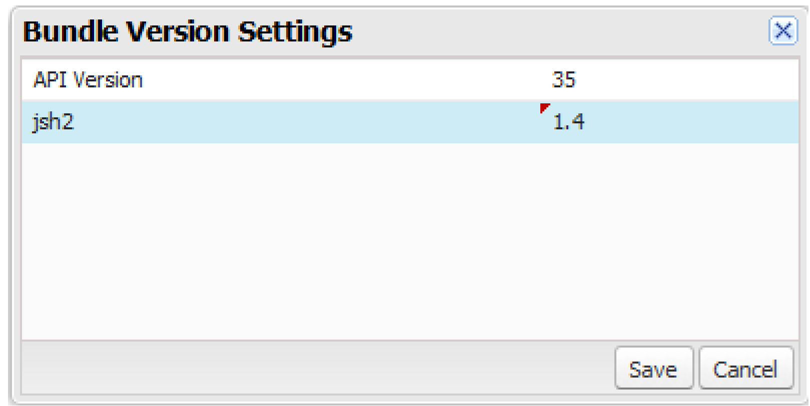
コンポーネントのバージョン設定

コンポーネントにバージョン設定すると、インストール済みの管理パッケージの特定のリビジョンに対して連動関係を宣言できます。

コンポーネントにバージョンを割り当てることで、管理パッケージの新しいバージョンがリリースされたときにコンポーネントがどのように機能するかを詳細に制御できます。たとえば、***<packageNamespace>***`:button` がバージョン 2.0 のパッケージに固定されているとします。バージョン 3.0 をインストールしても、ボタンにはバージョン 2.0 の機能が残ります。

メモ: パッケージの開発者は、コンポーネントの更新時にバージョン設定ロジックをマークアップに挿入する必要があります。更新でコンポーネントが変更されていないか、マークアップでバージョンが考慮されていない場合、コンポーネントは最新バージョンのコンテキストで動作します。

バージョンは、開発者コンソールで宣言的に割り当てられます。コンポーネントを操作するときに、右パネルの [Bundle Version Settings (バージョン設定を対応付ける)] をクリックしてバージョンを定義します。コンポーネントのバージョン設定はパッケージをインストールした場合にのみ可能で、コンポーネントの有効なバージョンがそのパッケージで使用できるバージョンになります。バージョンの形式は *<major>.<minor>* です。したがって、コンポーネントバージョン 1.4 を割り当てると、その動作は、関連パッケージの最初のメジャーリリースおよび 4 番目のマイナーリリースによって決まります。



コンポーネントを使用する場合、次のバージョン設定が可能です。

- Apex コントローラ
- JavaScript コントローラ
- JavaScript ヘルパー
- JavaScript レンダラ
- バンドルマークアップ
  - アプリケーション (`.app`)
  - コンポーネント (`.cmp`)
  - インターフェース (`.intf`)
  - イベント (`.evt`)

バンドル内の他の種類のリソースはバージョン設定できません。サポート対象外の種類は、次のとおりです。

- スタイル (`.css`)
- ドキュメント (`.doc`)
- デザイン (`.design`)
- SVG (`.svg`)

コンポーネントにバージョンを割り当てるか、パッケージのコンポーネントを開発している場合は、いくつかのコンテキストでバージョンを取得できます。

リソース	戻り値	式
Apex	Version	`System.requestVersion()`
JavaScript	String	`cmp.getVersion()`
Lightning コンポーネントのマークアップ	String	`{!Version}`

取得されたバージョンを使用して、ロジックをコードまたはマークアップに追加し、異なる機能を異なるバージョンに割り当てることができます。次に、`<aura:if>` ステートメントでのバージョン設定の使用例を示します。

```
<aura:component>
 <aura:if isTrue="{!Version > 1.0}">
  <c:newVersionFunctionality/>
 </aura:if>
 <c:oldVersionFunctionality/>
 ...
</aura:component>
```

関連トピック:

最小 API バージョン要件のあるコンポーネント

コンポーネントの API バージョンを混在させせない

最小 API バージョン要件のあるコンポーネント

一部の組み込みコンポーネントでは、それらを使用するカスタムコンポーネントが最小 API バージョンに設定されていることが必要です。カスタムコンポーネントは、使用するコンポーネントで求められる最新のAPIバージョン以降であることが必要です。

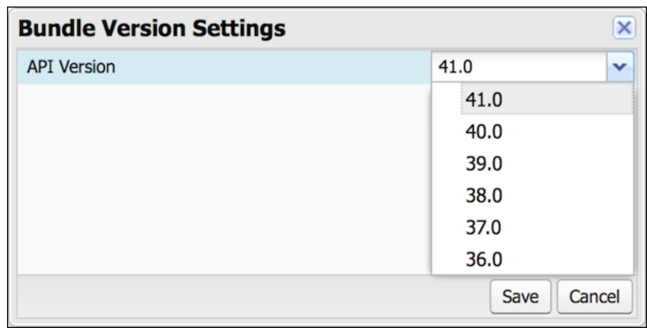
カスタムコンポーネントは、最小バージョン要件があり、その要件の適用対象となる別のコンポーネントをいくつかの異なる方法で使用できます。

- カスタムコンポーネントは、最小バージョン要件のあるコンポーネントから拡張できる。
- カスタムコンポーネントは、マークアップで子コンポーネントとして別のコンポーネントを追加できる。
- カスタムコンポーネントは、JavaScript で動的に子コンポーネントを作成および追加できる。

静的分析でコンポーネント間のリレーションを判断できる場合、コンポーネントの保存時にバージョン連動関係がチェックされます。カスタムコンポーネントの API バージョンが、使用されるコンポーネントで求められる最小バージョンに満たない場合、エラーが報告され、コンポーネントは保存されません。使用しているツールによって、このエラーの表示方法は異なります。

コンポーネントが動的に作成される場合、保存時にそのコンポーネントと親コンポーネント間のリレーションを判断できません。代わりに実行時に最小バージョン要件がチェックされ、失敗するとランタイムエラーが現在のユーザに報告されます。

コンポーネントのAPIバージョンは、開発者コンソールまたはForce.com IDEで設定するか、APIを介して設定できます。



組み込みコンポーネントのバージョン設定

組み込みコンポーネントを使用するために必要な最小のAPIバージョンは、*『Lightning コンポーネント開発者ガイド』*のコンポーネントの参照ページに記載されています。最小APIバージョンが指定されていないコンポーネントは、LightningコンポーネントでサポートされているすべてのAPIバージョンで使用できます。

正式リリース(GA)されている組み込みコンポーネントの最小バージョンは、今後のリリースで増加することはありません。ただし、Visualforceコンポーネントと同様に、それらの動作は含まれるコンポーネントのAPIバージョンによって変わる可能性があります。

関連トピック:

コンポーネントのバージョン設定
コンポーネントのAPIバージョンを混在させない

式の使用

式を使用すると、コンポーネントのマークアップ内で計算することや、プロパティ値やその他のデータにアクセスすることができます。式は、動的出力や、値を属性に割り当ててコンポーネントに渡す場合に使用します。

式はリテラル値、変数、サブ式、演算子などで構成され、1つの値に解決されます。メソッドコールは式に使用できません。

式の構文は、`{!expression}`です。

expressionは、式のプレースホルダです。

コンポーネントが表示されるとき、またはコンポーネントが値を使用するときに、`{! }`区切り文字内にあるすべてが評価され、動的に置換されます。空白文字は無視されます。

結果は、整数、文字列、booleanなどのプリミティブ値になります。また、JavaScriptオブジェクト、コンポーネントまたはコレクション、コントローラメソッド(アクションメソッドなど)、その他の有益な値のこともあります。

メモ: 他の言語に慣れている場合、! を「感嘆符」演算子と混同することがあります (これは、多くのプログラミング言語で boolean 値を否定する演算子です)。Lightning コンポーネントフレームワークでは、{! は式の先頭に使用する単なる区切り文字です。

Visualforce を使い慣れている場合は、この構文も目にしています。

式の構文にはもう 1 つ {#**expression**} もあります。式の構文の 2 つの形式の違いについての詳細は、「コンポーネント間のデータバインド」を参照してください。

ビュー、コントローラの値、または表示レベルからアクセスする属性名など、式の識別子は、先頭を文字または下線にする必要があります。2 文字目以降には数字やハイフンも使用できます。たとえば、{!v.2count} は有効ではありませんが、{!v.count} は有効です。

重要: {! } 構文は、.app または .cmp ファイルのマークアップのみで使用します。JavaScript では、文字列構文を使用して式を評価します。次に例を示します。

```
var theLabel = cmp.get("v.label");
```

{! をエスケープする場合は、次の構文を使用します。

```
<aura:text value="{!"/>
```

aura:text コンポーネントは {! を式の先頭と解釈しないため、プレーンテキストではこの構文が {! と表示されます。

このセクションの内容:

式の動的出力
式を使用する最も簡単な方法は、動的な値を出力することです。

条件式
3 項演算子と <aura:if> タグを使用した条件式の例を示します。

コンポーネント間のデータバインド
マークアップにコンポーネントを追加すると、式を使用して、コンテナコンポーネントの属性値を基にコンポーネントの属性値を初期化できます。式の構文には 2 つの形式があり、それぞれコンポーネント間のデータバインドの動作が異なります。

値プロバイダ
値プロバイダは、データにアクセスする 1 つの方法で、オブジェクトがプロパティやメソッドをカプセル化する場合と同じようなやり方で関連する値をまとめてカプセル化します。

式の評価
式は、JavaScript やその他のプログラミング言語の式が評価される方法とほぼ同じ方法で評価されます。

式の演算子のリファレンス
式言語では演算子がサポートされ、より複雑な式を作成できます。

式の関数のリファレンス
式言語には、算術、文字列、配列、比較、boolean、条件などの関数が含まれています。すべての関数で大文字と小文字が区別されます。

式の動的出力

式を使用する最も簡単な方法は、動的な値を出力することです。

式には、コンポーネントの属性、リテラル値、boolean などの値を使用できます。次に例を示します。

```
{!v.desc}
```

この式の v はコンポーネントの一連の属性からなるビューを表し、desc はコンポーネントの属性です。この式は、単にこのマークアップを含むコンポーネントの desc 属性値を出力します。

式にリテラル値を含める場合は、テキスト値を単一引用符で囲みます (例: {!'Some text'})。

数字は引用符で囲みません (例: {!123})。

boolean の場合、true には {!true}、false には {!false} を使用します。

関連トピック:

コンポーネントの属性

値プロバイダ

条件式

3 項演算子と <aura:if> タグを使用した条件式の例を示します。

3 項演算子

次の式は、3 項演算子を使用して、2 つの値のいずれかを条件に応じて出力します。

```
<a class="{!v.location == '/active' ? 'selected' : ''}" href="#/active">Active</a>
```

{!v.location == '/active' ? 'selected' : ''} 式は、location 属性が /active に設定されているかどうかを確認して、HTML <a> タグの class 属性を条件に応じて設定します。true の場合は、式が class を selected に設定します。

条件付きマークアップでの <aura:if> の使用

マークアップの次のスニペットは、<aura:if> タグを使用して、編集ボタンを条件に応じて表示します。

```
<aura:attribute name="edit" type="Boolean" default="true"/>
<aura:if isTrue="{!v.edit}">
    <ui:button label="Edit"/>
    <aura:set attribute="else">
        You can’t edit this.
    </aura:set>
</aura:if>
```

edit 属性が true に設定されている場合は、ui:button が表示されます。それ以外の場合は、else 属性のテキストが表示されます。

関連トピック:

条件付きマークアップのベストプラクティス

コンポーネント間のデータバインド

マークアップにコンポーネントを追加すると、式を使用して、コンテナコンポーネントの属性値を基にコンポーネントの属性値を初期化できます。式の構文には2つの形式があり、それぞれコンポーネント間のデータバインドの動作が異なります。

この概念はやや複雑ですが、わかりやすく例を使用して説明します。parentAttr 属性のある c:parent コンポーネントがあるとします。c:parent には、parentAttr 属性の値に初期設定された childAttr のある c:child コンポーネントが含まれます。parentAttr 属性値を c:parent から c:child コンポーネントに渡します。これは、2 つのコンポーネント間のデータバインド (値バインドとも呼ばれる) になります。

```
<!--c:parent-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>

    <!-- Instantiate the child component -->
    <c:child childAttr="{!v.parentAttr}" />
</aura:component>
```

{!v.parentAttr} はバインド式です。c:child の childAttr 属性値を変更すると、c:parent の parentAttr 属性が影響を受けます。その逆も同様です。

では、次のマークアップを変更してみましょう。

```
<c:child childAttr="{!v.parentAttr}" />
```

変更後

```
<c:child childAttr="{#v.parentAttr}" />
```

{#v.parentAttr} は非バインド式です。c:child の childAttr 属性値を変更しても、c:parent の parentAttr 属性は影響を受けません。その逆も同様です。

式の構文形式間の違いについて概要を次に示します。

{#expression} (非バインド式)

データの更新は、JavaScript で期待どおりに動作します。String などのプリミティブが値によって渡され、親と子の式でのデータ更新は分離しています。

Array や Map などのオブジェクトは参照として渡されるため、子のデータへの変更は親に伝搬されます。ただし、親の変更ハンドラには通知されません。子に伝搬される親の変更も同様に動作します。

{!expression} (バインド式)

どちらのコンポーネントでのデータの更新も、双方向データバインドによって両方のコンポーネントに反映されます。同様に、変更ハンドラは親と子両方のコンポーネントでトリガされます。

ヒント: 双方向データバインドの場合はパフォーマンスが損なわれ、データの変更がネストされたコンポーネントまで伝搬するため、エラーのデバッグも困難になります。式を親コンポーネントから子コンポーネントに渡すときは、双方向データバインドが必要な場合を除き、代わりに `{#expression}` 構文を使用することをお勧めします。

非バインド式

別のコンポーネント `c:childExpr` を含む `c:parentExpr` コンポーネントのもう 1 つの例を見てみましょう。

`c:childExpr` のマークアップは次のようになります。

```
<!--c:childExpr-->
<aura:component>
    <aura:attribute name="childAttr" type="String" />

    <p>childExpr childAttr: {!v.childAttr}</p>
    <p><lightning:button label="Update childAttr"
          onclick="{!c.updateChildAttr}"/></p>
</aura:component>
```

`c:parentExpr` のマークアップは次のようになります。

```
<!--c:parentExpr-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>

    <!-- Instantiate the child component -->
    <c:childExpr childAttr="{#v.parentAttr}" />

    <p>parentExpr parentAttr: {!v.parentAttr}</p>
    <p><lightning:button label="Update parentAttr"
          onclick="{!c.updateParentAttr}"/></p>
</aura:component>
```

`c:parentExpr` コンポーネントは、非バウンド式を使用して `c:childExpr` コンポーネントの属性を設定します。

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

`childExpr` をインスタンス化するときに、`childAttr` 属性を `c:parentExpr` の `parentAttr` 属性の値に設定します。`{#v.parentAttr}` 構文が使用されているため、`v.parentAttr` 式は `childAttr` 属性の値にバインドされません。

`c:exprApp` アプリケーションは `c:parentExpr` を囲むラッパーです。

```
<!--c:exprApp-->
<aura:application >
    <c:parentExpr />
</aura:application>
```

開発者コンソールで、`c:exprApp` のサイドバーにある **[Preview (プレビュー)]** をクリックして、ブラウザにアプリケーションを表示します。

`parentAttr` と `childAttr` の両方が「親属性」に設定されます。これは `parentAttr` のデフォルト値です。

次に、c:childExpr のクライアント側コントローラを作成して、コンポーネントを動的に更新できるようにします。childExprController.js のソースは次のようになります。

```
/* childExprController.js */
({
    updateChildAttr: function(cmp) {
        cmp.set("v.childAttr", "updated child attribute");
    }
})
```

開発者コンソールで、c:exprApp の **[Update Preview (プレビューを更新)]** をクリックします。

[Update childAttr (childAttr を更新)] ボタンを押します。この結果、childAttr が「更新された子属性」に更新されます。非バインド式を使用したため、parentAttr の値は変わりません。

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

では、c:parentExpr のクライアント側コントローラを追加してみましょう。parentExprController.js のソースは次のようになります。

```
/* parentExprController.js */
({
    updateParentAttr: function(cmp) {
        cmp.set("v.parentAttr", "updated parent attribute");
    }
})
```

開発者コンソールで、c:exprApp の **[Update Preview (プレビューを更新)]** をクリックします。

[Update parentAttr (parentAttr を更新)] ボタンを押します。今回は、parentAttr が「更新された親属性」に設定されますが、非バインド式のため childAttr は変わりません。

警告: コンポーネントの init イベントとクライアント側コントローラを使用して、非バインド式で使用される属性を初期化しないでください。この属性は初期化されません。代わりにバインド式を使用します。コンポーネントの init イベントについての詳細は、「コンポーネントの初期化時のアクションの呼び出し」 (ページ 271)を参照してください。

または、コンポーネントを別のコンポーネントでラップすることもできます。ラッパーコンポーネントでラップされたコンポーネントをインスタンス化するときは、ラップされたコンポーネントのクライアント側コントローラで属性を初期化するのではなく、属性値を初期化します。

バインド式

次に、代わりにバインド式を使うようにコードを更新してみましょう。c:parentExpr の次の行を変更します。

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

変更後

```
<c:childExpr childAttr="{!v.parentAttr}" />
```

開発者コンソールで、c:exprApp の **[Update Preview (プレビューを更新)]** をクリックします。

[Update childAttr (childAttr を更新)] ボタンを押します。この結果、childExpr のクライアント側コントローラの v.childAttr しか設定しなくても、childAttr と parentAttr の両方が「更新された子属性」に更新されます。バインド式を使用して childAttr 属性を設定したため、両方の属性が更新されました。

変更ハンドラおよびデータバインド

コンポーネントのいずれかの属性の値が変更されたときに、変更ハンドラを自動的に呼び出す (クライアント側コントローラのアクション) ようにコンポーネントを設定できます。

バインド式を使用する場合、親または子コンポーネントの属性を変更すると、両方のコンポーネントの変更ハンドラがトリガされます。非バインド式を使用する場合は、変更がコンポーネント間で伝播されないため、変更された属性を含むコンポーネントのみで変更ハンドラがトリガされます。

では、前述の例に変更ハンドラを追加して、バインド式と非バインド式によってどのような影響を受けるか見てみましょう。

以下は、c:childExpr の更新されたマークアップです。

```
<!--c:childExpr-->
<aura:component>
    <aura:attribute name="childAttr" type="String" />

    <aura:handler name="change" value="{!v.childAttr}" action="{!c.onChildAttrChange}"/>

    <p>childExpr childAttr: {!v.childAttr}</p>
    <p><lightning:button label="Update childAttr"
          onclick="{!c.updateChildAttr}"/></p>
</aura:component>
```

<aura:handler> タグに、変更ハンドラを表す name="change" が設定されています。value="{!v.childAttr}" は、変更ハンドラに childAttr 属性を追跡するよう指示します。childAttr が変更されると、onChildAttrChange というクライアント側コントローラのアクションが呼び出されます。

以下は、c:childExpr のクライアント側コントローラです。

```
/* childExprController.js */
({
    updateChildAttr: function(cmp) {
        cmp.set("v.childAttr", "updated child attribute");
    },

    onChildAttrChange: function(cmp, evt) {
        console.log("childAttr has changed");
        console.log("old value: " + evt.getParam("oldValue"));
        console.log("current value: " + evt.getParam("value"));
    }
})
```

以下は、変更ハンドラが設定された c:parentExpr の更新されたマークアップです。

```
<!--c:parentExpr-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>
```

```
    <aura:handler name="change" value="{!v.parentAttr}" action="{!c.onParentAttrChange}"/>


    <!-- Instantiate the child component -->
    <c:childExpr childAttr="{!v.parentAttr}" />

    <p>parentExpr parentAttr: {!v.parentAttr}</p>
    <p><lightning:button label="Update parentAttr"
          onclick="{!c.updateParentAttr}"/></p>
</aura:component>
```

以下は、c:parentExpr のクライアント側コントローラです。

```
/* parentExprController.js */
({
    updateParentAttr: function(cmp) {
        cmp.set("v.parentAttr", "updated parent attribute");
    },

    onParentAttrChange: function(cmp, evt) {
        console.log("parentAttr has changed");
        console.log("old value: " + evt.getParam("oldValue"));ui
        console.log("current value: " + evt.getParam("value"));
    }
})
```

開発者コンソールで、c:exprApp の [Update Preview (プレビューを更新)] をクリックします。

ブラウザのコンソールを開きます (Chrome の [その他のツール] > [デベロッパー ツール])。

[Update parentAttr (parentAttr を更新)] ボタンを押します。バインド式を使用しているため、c:parentExpr と c:childExpr の両方の変更ハンドラがトリガされます。

```
<c:childExpr childAttr="{!v.parentAttr}" />
```

代わりに非バインド式を使用するように c:parentExpr を変更します。

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

開発者コンソールで、c:exprApp の [Update Preview (プレビューを更新)] をクリックします。

[Update childAttr (childAttr を更新)] ボタンを押します。この場合は、非バインド式を使用しているため、c:childExpr の変更ハンドラのみがトリガされます。

関連トピック:

変更ハンドラを使用したデータ変更の検出
式の動的出力
コンポーネントのコンポジション

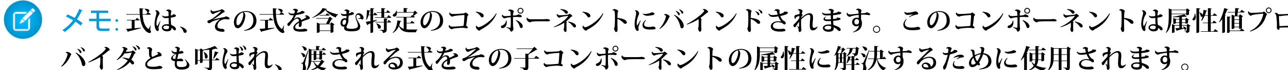
値プロバイダ

値プロバイダは、データにアクセスする1つの方法で、オブジェクトがプロパティやメソッドをカプセル化する場合と同じようなやり方で関連する値をまとめてカプセル化します。

コンポーネントの値プロバイダは、v (ビュー) および c (コントローラ) です。

値プロバイダ	説明	関連トピック
v	コンポーネントの属性セット。この値プロバイダを使用すると、コンポーネントのマークアップ内のコンポーネントの属性の値にアクセスできます。	コンポーネントの属性
c	コンポーネントのコントローラ。コンポーネントのイベントハンドラとアクションを結び付けることができます。	クライアント側コントローラを使用したイベントの処理

どのコンポーネントにも v 値プロバイダがありますが、コントローラの設定は必須ではありません。どちらの値プロバイダも、コンポーネントの定義時に自動的に作成されます。


メモ: 式は、その式を含む特定のコンポーネントにバインドされます。このコンポーネントは属性値プロバイダとも呼ばれ、渡される式をその子コンポーネントの属性に解決するために使用されます。

グローバル値プロバイダ

グローバル値プロバイダは、コンポーネントが式で使用できるグローバルな値およびメソッドです。

グローバル値プロバイダ	説明	関連トピック
globalID	globalId グローバル値プロバイダは、コンポーネントのグローバル ID を返します。すべてのコンポーネントには一意の globalId があります。これはコンポーネントインスタンスに対して生成される実行時に一意の ID です。	コンポーネントの ID
$Browser	$Browser グローバル値プロバイダは、アプリケーションにアクセスしているブラウザのハードウェアおよびオペレーティングシステムに関する情報を返します。	$Browser
$Label	$Label グローバル値プロバイダを使用すると、コードの外部に保存されたラベルにアクセスできます。	カスタムラベルの使用
$Locale	$Locale グローバル値プロバイダは、現在のユーザが選択しているロケールに関する情報を返します。	$Locale

グローバル値プロバイダ	説明	関連トピック
`$Resource`	`$Resource` グローバル値プロバイダにより、静的リソースにアップロードした画像、スタイルシート、JavaScript コードを参照できます。	$Resource

項目および関連オブジェクトへのアクセス

値プロバイダの値には、指定したプロパティとしてアクセスします。値を使用するには、値プロバイダとプロパティ名をドット (ピリオド) で区切ります。たとえば、`v.body` です。値プロバイダにはマークアップまたは JavaScript コード内でアクセスできます。

コンポーネントの属性がオブジェクトやその他の構造化されたデータ (プリミティブ値ではない) の場合は、同じドット表記を使用してその属性の値にアクセスします。

たとえば、`{!v.accounts.id}` は、取引先レコードの ID 項目にアクセスします。

ネストが深いオブジェクトまたは属性については、ドットを繰り返し追加して構造をトラバースし、ネストされた値にアクセスします。

関連トピック:

式の動的出力

$Browser

`$Browser` グローバル値プロバイダは、アプリケーションにアクセスしているブラウザのハードウェアおよびオペレーティングシステムに関する情報を返します。

属性	説明
`formFactor`	ブラウザを実行しているハードウェアの種類に基づいて `FormFactor` Enum 値を返します。 • DESKTOP: デスクトップクライアント • PHONE: 電話 (ブラウザ対応の携帯電話やスマートフォンを含む) • TABLET: タブレットクライアント (`isTablet` が `true` を返します)
`isAndroid`	ブラウザが Android デバイス上で実行されているか (`true`)、否か (`false`) を示します。
`isIOS`	すべての実装で使用できるわけではありません。ブラウザが iOS デバイス上で実行されているか (`true`)、否か (`false`) を示します。
`isIPad`	すべての実装で使用できるわけではありません。ブラウザが iPad 上で実行されているか (`true`)、否か (`false`) を示します。
`isIPhone`	すべての実装で使用できるわけではありません。ブラウザが iPhone 上で実行されているか (`true`)、否か (`false`) を示します。

属性	説明
isPhone	ブラウザが電話 (ブラウザ対応の携帯電話やスマートフォンを含む) 上で実行されているか (true)、否か (false) を示します。
isTablet	ブラウザが iPad 上または Android 2.2 以降を搭載したタブレット上で実行されているか (true)、否か (false) を示します。
isWindowsPhone	ブラウザが Windows Phone 上で実行されているか (true)、否か (false) を示します。Windows Phone のみが検出され、タブレットやその他のタッチ対応の Windows 8 デバイスは検出されません。

例: 次の例に、$Browser グローバル値プロバイダの使用法を示します。

```
<aura:component>
    {!$Browser.isTablet}
    {!$Browser.isPhone}
    {!$Browser.isAndroid}
    {!$Browser.formFactor}
</aura:component>
```

同様に、$A.get() を使用して、クライアント側コントローラのブラウザ情報を確認できます。

```
({
    checkBrowser: function(component) {
        var device = $A.get("$Browser.formFactor");
        alert("You are using a " + device);
    }
})
```

$Locale

$Locale グローバル値プロバイダは、現在のユーザが選択しているロケールに関する情報を返します。

これらの属性は、Java の Calendar、Locale、および TimeZone クラスに基づきます。

属性	説明	サンプル値
country	言語ロケールに基づく ISO 3166 に従った国コード	「US」、「DE」、「GB」
currency	通貨記号	「$」
currencyCode	ISO 4217 に従った国コード	「USD」
decimal	小数点	「.」
firstDayOfWeek	週の開始曜日 (1 は日曜日)	1
grouping	桁区切り記号	「,」

属性	説明	サンプル値
isEasternNameStyle	名前が東洋式のスタイルに基づくかどうか (last name first name [middle] [suffix] など)	false
labelForToday	日付ピッカーのTodayのリンクの表示ラベル	「Today」
language	言語ロケールに基づく言語コード	「en」、「de」、「zh」
langLocale	ロケール ID	「en_US」、「en_GB」
nameOfMonths	カレンダー月の完全名と短縮名	{ fullName: "January", shortName: "Jan" }
nameOfWeekdays	カレンダー週の完全名と短縮名	{ fullName: "Sunday", shortName: "SUN" }
timezone	タイムゾーン ID	「America/Los_Angeles」
userLocaleCountry	現在のユーザのロケールに基づく国	「US」
userLocaleLang	現在のユーザのロケールに基づく言語	「en」
variant	ベンダおよびブラウザ固有のコード	「WIN」、「MAC」、「POSIX」

数値と日付の書式設定

フレームワークの数値と日付の書式設定は、Java の DecimalFormat および DateFormat クラスに基づきます。

属性	説明	サンプル値
currencyformat	通貨形式	「¤#,##0.00;(¤#,##0.00)」 ¤は通貨記号を表し、通貨のマークに置換されます。
dateFormat	日付形式	「MMM d, yyyy」
datetimeFormat	日時形式	「MMM d, yyyy h:mm:ss a」
numberformat	数値形式	「#,##0.###」 # は数字、カンマは 3 桁区切り文字のプレースホルダ、ピリオドは小数点区切り文字のプレースホルダを表します。末尾のゼロを表示する場合は、# をゼロ (0) に置換します。
percentformat	パーセント形式	「#,##0%」
timeFormat	時間形式	「h:mm:ss a」

属性	説明	サンプル値
zero	ゼロ桁を表す文字	"0"

例: 次の例は、さまざまな $Locale 属性を取得する方法を示します。

コンポーネントのソース

```
<aura:component>
    {!$Locale.language}
    {!$Locale.timezone}
    {!$Locale.numberFormat}
    {!$Locale.currencyFormat}
</aura:component>
```

同様に、$A.get() を使用して、クライアント側コントローラのロケール情報を確認できます。

```
({
    checkDevice: function(component) {
        var locale = $A.get("$Locale.language");
        alert("You are using " + locale);
    }
})
```

関連トピック:

ローカライズ

$Resource

$Resource グローバル値プロバイダにより、静的リソースにアップロードした画像、スタイルシート、JavaScript コードを参照できます。

$Resource を使用すると、正確な URL やファイルパスがわからなくても、名前でアセットを参照できます。$Resource は、Lightning コンポーネントのマークアップおよび JavaScript コントローラやヘルパーコード内で使用できます。

コンポーネントのマークアップでの $Resource の使用

コンポーネントのマークアップで特定のリソースを参照するには、式内に $Resource.***resourceName*** を使用します。*resourceName* は、静的リソースの [名前] です。管理パッケージでは、リソース名にパッケージ名前空間プレフィックス($Resource.yourNamespace__resourceName など)を含める必要があります。個々のグラフィックやスクリプトなど、スタンドアロンの静的リソースの場合、必要なのはそれだけです。アーカイブ静的リソース内の項目を参照するには、項目へのパスの残りを文字列の連結を使用して含めます。次は、その例です。

```
<aura:component>
    <!-- Stand-alone static resources -->
    <img src="{!$Resource.generic_profile_svg}"/>
    <img src="{!$Resource.yourNamespace__generic_profile_svg}"/>
```

```
    <!-- Asset from an archive static resource -->
    <img src="{!$Resource.SLDSv2 + '/assets/images/avatar1.jpg'}"/>
    <img src="{!$Resource.yourNamespace__SLDSv2 + '/assets/images/avatar1.jpg'}"/>
</aura:component>
```

<ltng:require> タグを使用して、CSS スタイルシートまたは JavaScript ライブラリをコンポーネントに含めます。以下に例を示します。

```
<aura:component>
  <ltng:require
    styles="{!$Resource.SLDSv2 + '/assets/styles/lightning-design-system-ltng.css'}"
    scripts="{!$Resource.jsLibraries + '/jsLibOne.js'}"
    afterScriptsLoaded="{!c.scriptsLoaded}" />
</aura:component>
```

メモ: 式で $Resource が解析される方法に予測できない動作があるため、複数の $Resource 参照を 1 つの属性に含めるには join 演算子を使用します。たとえば、コンポーネントに含める JavaScript ライブラリが複数ある場合、scripts 属性は次のようになります。

```
scripts="{!join(',',
    $Resource.jsLibraries + '/jsLibOne.js',
    $Resource.jsLibraries + '/jsLibTwo.js')}"
```

JavaScript での $Resource の使用

JavaScript コードで静的リソースへの参照を取得するには、$A.get('$Resource.***resourceName***') を使用します。

resourceName は、静的リソースの [名前] です。管理パッケージでは、リソース名にパッケージ名前空間プレフィックス ($Resource.yourNamespace__resourceName など) を含める必要があります。個々のグラフィックやスクリプトなど、スタンドアロンの静的リソースの場合、必要なのはそれだけです。アーカイブ静的リソース内の項目を参照するには、項目へのパスの残りを文字列の連結を使用して含めます。以下に例を示します。

```
({
    profileUrl: function(component) {
        var profUrl = $A.get('$Resource.SLDSv2') + '/assets/images/avatar1.jpg';
        alert("Profile URL: " + profUrl);
    }
})
```

メモ: JavaScript で参照される静的リソースは、パッケージに自動的に追加されません。コンポーネントのマークアップで参照されていないリソースに依存している JavaScript は、JavaScript コードが含まれているパッケージに手動で追加します。

$Resource の考慮事項

Lightning コンポーネントフレームワークのグローバル値プロバイダは、Salesforce のグローバル変数とは大きく異なる形でバックグラウンドで実装されます。$Resource は、Visualforce や数式項目などで使用できる同じ名

前のグローバル変数に似ていますが、重要な違いがあります。他のドキュメントを、その使用や動作のガイドラインとして使用しないでください。

Lightning コンポーネントフレームワークの `$Resource` に特有の、留意すべき事項が 2 つあります。

1 つ目は、`$Resource` は Lightning コンポーネントフレームワークがクライアントに読み込まれるまで使用できないということです。マークアップのみで構成される一部の非常に単純なコンポーネントは、`$Resource` を使用できないサーバ側で表示できます。これを回避するには、新しいアプリケーションの作成時にコンポーネントがクライアントで表示されるようにクライアント側コントローラをスタブアウトします。

2 つ目は、Visualforce などの `$Resource` グローバル変数を操作する場合、特定のリソースの完全な URL を構築するために `URLFOR()` 数式関数も使用するということです。Lightning コンポーネントフレームワーク の `URLFOR()` とは大きく異なります。こちらでは、前の例のように代わりに単純な文字列の連結を使用します。

関連トピック:

Salesforce ヘルプ: 静的リソース

式の評価

式は、JavaScript やその他のプログラミング言語の式が評価される方法とほぼ同じ方法で評価されます。

演算子は、JavaScript で使用可能なものの一部で、評価順序や優先順位は概ね JavaScript と同じです。特定の評価順序は、括弧を使用して指定します。式に関して意外に思われる点は、評価が行われる頻度です。変更が行われるとフレームワークで検出され、影響を受けるコンポーネントの再表示がトリガされます。連動関係は自動的に処理されます。この点は、フレームワークの基本的な利点の 1 つです。フレームワークは、ページで何らかの内容を再表示する時点を検出します。コンポーネントが再表示されると、そのコンポーネントが使用する式が再評価されます。

action メソッド

式は、`onclick`、`onhover`、その他の「`on`」で始まるコンポーネントの属性など、ユーザインターフェースイベントのアクションメソッドの指定にも使用されます。

アクションメソッドは、`{!c.theAction}` のような式を使用して属性に割り当てる必要があります。この式は、アクションを処理するコントローラ関数への参照である `Aura.Action` を割り当てます。

式を使用してアクションメソッドを割り当てると、アプリケーションやユーザインターフェースの状態に基づく条件付きの割り当てを行うことができます。詳細は、「条件式」 (ページ 47) を参照してください。

```
<aura:component>
    <aura:attribute name="liked" type="Boolean" default="true"/>
    <lightning:button aura:id="likeBtn"
     label="{!(v.liked) ? 'Like It' : 'Unlike It'}"
     onclick="{!(v.liked) ? c.likeIt  : c.unlikeIt}"
    />
</aura:component>
```

いいね! とまだ言っていない項目に対してはこのボタンに「いいね!」と表示され、ボタンをクリックすると `likeIt` アクションメソッドがコールされます。その後でコンポーネントが再表示され、反対のユーザイン

ターフェースの表示とメソッドの割り当てが行われます。もう 1 回クリックすると、項目のいいね! が取り消されます。

メモ: 例は、属性がボタンの状態を制御するのにどのように役立つかを示しています。状態間を切り替えるボタンを作成するには、`lightning:buttonStateful` コンポーネントを使用することをおすすめします。

式の演算子のリファレンス

式言語では演算子がサポートされ、より複雑な式を作成できます。

算術演算子

算術演算子に基づく式では、数値が返されます。

演算子	使用方法	説明
+	`1 + 1`	2 つの数字を加算します。
-	`2 - 1`	一方の数字からもう一方の数字を減算します。
*	`2 * 2`	2 つの数字を乗算します。
/	`4 / 2`	一方の数字をもう一方の数字で除算します。
%	`5 % 2`	最初の数字を 2 つ目の数字で除算した残りの整数を返します。
-	`-v.exp`	単項演算子。後続の数字の正負記号を逆にします。たとえば、`expenses` の値が `100` の場合、`-expenses` は `-100` になります。

数値リテラル

リテラル	使用方法	説明
Integer	`2`	整数は小数点や指数のない数字です。
Float	`3.14` `-1.1e10`	小数点のある数字、または指数のある数字です。
Null	`null`	リテラルの null 数。明示的な null 値と未定義値のある数字を一致させます。

文字列演算子

文字列演算子に基づく式では、文字列値が返されます。

演算子	使用方法	説明
+	`'Title: ' + v.note.title`	2 つの文字列を連結します。

文字列リテラル

文字列リテラルは単一引用付で囲む必要があります (例: `'like this'`)。

リテラル	使用方法	説明
string	`'hello world'`	文字列リテラルは単一引用付で囲む必要があります。二重引用符は属性値を囲む場合にのみ使用し、文字列ではエスケープする必要があります。
\\<escape>	`'\n'`	空白文字: • `\t` (タブ) • `\n` (改行) • `\r` (行頭復帰) エスケープ文字: • `\"` (リテラル ") • `\'` (リテラル ') • `\\` (リテラル \\)
Unicode	`'\u####'`	Unicode のコードポイント。# 記号は 16 進数です。Unicode リテラルは 4 桁にする必要があります。
null	`null`	リテラルの null 文字列。明示的な null 値と未定義値のある文字列を一致させます。

比較演算子

比較演算子に基づく式では、`true` または `false` の値が返されます。比較の目的で、数字は同じ型として処理されます。他のすべての比較では、値と型の両方がチェックされます。

演算子	代替方法	使用方法	説明

==	eq	`1 == 1`
		`1 == 1.0`
		`1 eq 1`
		メモ: `undefined==null` の評価は `true` になります。
			オペランドが等しい場合に、`true` が返されます。この比較は、すべてのデータ型で有効です。
			警告: String や Integer など基本のデータ型の代わりに、オブジェクトに == 演算子を使用しないでください。たとえば、`object1==object2` は、一貫性なくクライアントあるい

演算子	代替方法	使用方法	説明
			はサーバで評価するため、信頼できません。
!=	ne	1 != 2 1 != true 1 != '1' null != false 1 ne 2	オペランドが等しくない場合に、true が返されます。この比較は、すべてのデータ型で有効です。
<	lt	1 < 2 1 lt 2	最初のオペランドの数値が 2 つ目のオペランドより小さい場合に、true を返します。< 演算子を < にエスケープして、コンポーネントのマークアップで使用できるようにする必要があります。または、lt 演算子を使用できます。
>	gt	42 > 2 42 gt 2	最初のオペランドの数値が 2 つ目のオペランドより大きい場合に、true を返します。
<=	le	2 <= 42 2 le 42	最初のオペランドの数値が 2 つ目のオペランド以下の場合に、true を返します。<= 演算子を <= にエスケープして、コンポーネントのマークアップで使用できるようにする必要があります。または、le 演算子を使用できます。
>=	ge	42 >= 42 42 ge 42	最初のオペランドの数値が 2 つ目のオペランド以上の場合に、true を返します。

論理演算子

論理演算子に基づく式では、true または false の値が返されます。

<table>
<tr><th>演算子</th><th>使用方法</th><th>説明</th></tr>
<tr><td>&&</td><td>isEnabled && hasPermission</td><td>両方のオペランドが true の場合に、true を返します。&& 演算子を && にエスケープして、コンポーネントのマークアップで使用できるようにする必要があります。または、and() 関数を使用して、2つの引数を渡すこともできます。たとえば、and(isEnabled, hasPermission) です。</td></tr>
<tr><td>||</td><td>hasPermission || isRequired</td><td>いずれかのオペランドが true の場合に、true を返します。</td></tr>
</table>

演算子	使用方法	説明
!	!isRequired	単項演算子。オペランドが false の場合に、true を返します。この演算子を、{! の形式で式の先頭に使用する ! 区切り文字と混同しないようにします。式区切り文字をこの否定演算子と組み合わせて、値の論理否定を返すことができます。たとえば、{!!true} は false を返します。

論理リテラル

論理値が非論理値と等しくなることはありません。つまり、true == true のみ、false == false のみ、1 != true および 0 != false、null != false です。

リテラル	使用方法	説明
true	true	boolean の true 値。
false	false	boolean の false 値。

条件演算子

条件演算子は、従来の 3 項演算子のみです。

演算子	使用方法	説明
? :	(1 != 2) ? "Obviously" : "Black is White"	? 演算子の前のオペランドは、boolean として評価されます。true の場合は、2 つ目のオペランドが返されます。false の場合は、3 つ目のオペランドが返されます。

関連トピック:

式の関数のリファレンス

式の関数のリファレンス

式言語には、算術、文字列、配列、比較、boolean、条件などの関数が含まれています。すべての関数で大文字と小文字が区別されます。

算術関数

算術関数は、数値の算術処理を行います。この関数は数値の引数を取ります。「対応する演算子」列に、同じ機能の演算子 (ある場合) を記載します。

関数	代替方法	使用方法	説明	対応する演算子
add	concat	add(1,2)	最初の引数を 2 つ目の引数に加算します。	+
sub	subtract	sub(10,2)	最初の引数から 2 つ目の引数を減算します。	-
mult	multiply	mult(2,10)	最初の引数を 2 つ目の引数で乗算します。	*
div	divide	div(4,2)	最初の引数を 2 つ目の引数で除算します。	/
mod	modulus	mod(5,2)	最初の引数を 2 つ目の引数で除算した残りの整数を返します。	%
abs		abs(-5)	引数の絶対値を返します。つまり、引数が正の場合はそのままの数値、負の場合はマイナス記号を除いた数値を返します。たとえば、abs(-5) は 5 です。	なし
neg	negate	neg(100)	引数の正負記号を逆にします。たとえば、neg(100) は -100 です。	- (単項)

文字列関数

関数	代替方法	使用方法	説明	対応する演算子
concat	add	concat('Hello ', 'world') add('Walk ', 'the dog')	2 つの引数を連結します。	+

関数	代替方法	使用方法	説明	対応する演算子
format		format($Label.ns.labelName, v.myVal) メモ: この関数は、String、Decimal、Double、Integer、Long、Array、String[]、List、および Set の型の引数で機能します。	パラメータプレースホルダをカンマ区切りの属性値で置き換えます。	
join		join(separator, subStr1, subStr2, subStrN) join(' ','class1', 'class2', v.class)	後続の各引数間の区切り文字列 (第 1 引数) を追加してサブ文字列を結合します。	

表示ラベル関数

関数	使用方法	説明
format	format($Label.np.labelName, v.attribute1 , v.attribute2) format($Label.np.hello, v.name)	表示ラベルを出力して更新します。パラメータプレースホルダをカンマ区切りの属性値で置き換えます。表示ラベルと属性には 3 項演算子を使用できます。

情報関数

関数	使用方法	説明
length	myArray.length	配列または文字列の長さを返します。
empty	empty(v.attributeName) メモ: この関数は、String、Array、Object、List、Map、Set のいずれかの型の引数で機能します。	引数が空の場合に true を返します。空の引数とは、undefined、null、空の配列、空の文字列などです。プロパティのないオブジェクトは空とはみなされません。 ヒント: {! !empty(v.myArray)} は {!v.myArray && v.myArray.length > 0} よりも評価が速いため、パフォーマ

関数	使用方法	説明
		ンスを向上させるためには empty() の使用をお勧めします。 JavaScript の $A.util.isEmpty() メソッドは、マークアップの empty() 式と同じです。

比較関数

比較関数は 2 つの数値引数を取り、比較の結果に応じて true または false のいずれかを返します。eq および ne 関数は、引数に文字列などの他のデータ型を取ることもできます。

関数	使用方法	説明	対応する演算子
equals	equals(1,1)	指定した引数が等しい場合に true を返します。この引数には、任意のデータ型を使用できます。	== または eq
notequals	notequals(1,2)	指定した引数が等しくない場合に true を返します。この引数には、任意のデータ型を使用できます。	!= または ne
lessthan	lessthan(1,5)	最初の引数の数値が2つ目の引数より小さい場合に true を返します。	< または lt
greaterthan	greaterthan(5,1)	最初の引数の数値が2つ目の引数より大きい場合に true を返します。	> または gt
lessthanorequal	lessthanorequal(1,2)	最初の引数の数値が2つ目の引数以下の場合に true を返します。	<= または le
greaterthanorequal	greaterthanorequal(2,1)	最初の引数の数値が2つ目の引数以上の場合に true を返します。	>= または ge

boolean 関数

boolean 関数は、boolean 引数を処理します。論理演算子と同じ機能です。

関数	使用方法	説明	対応する演算子
and	and(isEnabled, hasPermission)	両方の引数がtrueの場合に true を返します。	&&

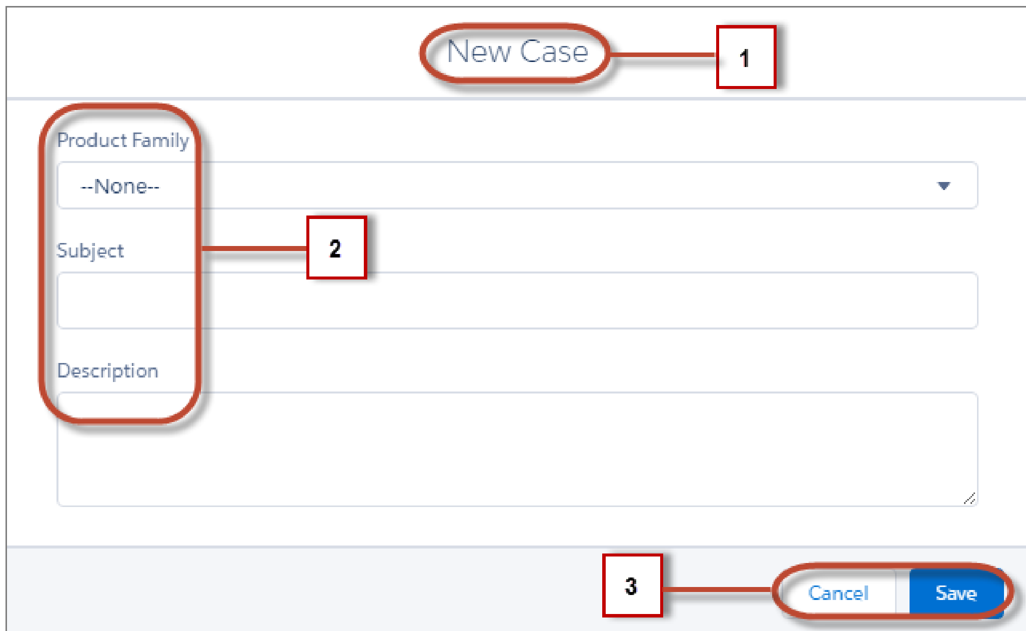
関数	使用方法	説明	対応する演算子
or	or(hasPermission, hasVIPPass)	いずれかの引数がtrueの場合に true を返します。	||
not	not(isNew)	引数がfalseの場合に true を返します。	!

条件関数

関数	使用方法	説明	対応する演算子
if	if(isEnabled, 'Enabled', 'Not enabled')	最初の引数を boolean として評価します。trueの場合は、2つ目の引数を返します。それ以外の場合は、3つ目の引数を返します。	?: (3 項)

表示ラベルの使用

表示ラベルは、ヘッダー (1)、入力項目 (2)、ボタン (3) など、ユーザインターフェースに関する情報を表すテキストです。コンポーネントのマークアップでテキスト値を指定して表示ラベルを指定できますが、式の構文で $Label グローバル値プロバイダを使用して、コードの外部に保存された表示ラベルにアクセスすることもできます。



このセクションでは、次のコンテキストで $Label グローバル値プロバイダを使用する方法について説明します。

- 入力コンポーネントの label 属性
- 表示ラベルに動的に入力されるプレースホルダ値の format() 式の関数

このセクションの内容:

Using Custom Labels
カスタム表示ラベルは、Salesforceでサポートされている言語に翻訳できるカスタムテキスト値です。Lightning コンポーネントでカスタム表示ラベルにアクセスするには、$Label グローバル値プロバイダを使用します。

入力コンポーネントの表示ラベル
表示ラベルで、入力コンポーネントの目的を説明します。入力コンポーネントの表示ラベルを設定するには、label 属性を使用します。

表示ラベルパラメータの動的な入力
format() 式の関数を使用して、表示ラベルを出力および更新します。

JavaScript での表示ラベルの取得
表示ラベルは JavaScript コードで取得できます。表示ラベルを静的に定義して、コンポーネントの読み込み時にクライアントに送信すると、コードは最適に実行されます。

Apex での表示ラベルの取得
Apexコードで表示ラベルを取得し、JavaScriptを使用してその表示ラベルをコンポーネントに設定できます。

親属性による表示ラベル値の設定
親属性による表示ラベル値の設定は、子コンポーネントの表示ラベルを制御する場合に便利です。

Using Custom Labels

カスタム表示ラベルは、Salesforce でサポートされている言語に翻訳できるカスタムテキスト値です。Lightning コンポーネントでカスタム表示ラベルにアクセスするには、$Label グローバル値プロバイダを使用します。

カスタム表示ラベルを使用することで、開発者は、情報(ヘルプテキストやエラーメッセージなど)を自動的にユーザの母国語に表示する多言語アプリケーションを作成できます。

カスタム表示ラベルを作成するには、[設定] から、[クイック検索] ボックスに「カスタム表示ラベル」と入力し、[カスタム表示ラベル] を選択します。

Lightning コンポーネントでカスタム表示ラベルにアクセスするには、次の構文を使用します。

- `$Label.c.`***`labelName`*** (デフォルトの名前空間の場合)
- `$Label.`***`namespace.labelName`*** (組織に名前空間があるか、管理パッケージの表示ラベルにアクセスする場合)

次に、いくつか例を示します。

デフォルトの名前空間を使用するマークアップ式の表示ラベル

```
{!$Label.c.labelName}
```

組織に名前空間がある場合の JavaScript コードの表示ラベル

```
$A.get("$Label.namespace.labelName")
```

関連トピック:

値プロバイダ

入力コンポーネントの表示ラベル

表示ラベルで、入力コンポーネントの目的を説明します。入力コンポーネントの表示ラベルを設定するには、`label` 属性を使用します。

次の例に、入力コンポーネントの `label` 属性で表示ラベルを使用する方法を示します。

```
<lightning:input type="number" name="myNumber" label="Pick a Number:" value="54" />
```

表示ラベルは入力項目の左側に配置されますが、`variant="label-hidden"` を設定して非表示にすることもできます。`slds-assistive-text` クラスを表示ラベルに適用してアクセシビリティをサポートします。

`$Label` の使用

`$Label` グローバル値プロバイダを使用して、外部ソースに保存された表示ラベルにアクセスします。以下に例を示します。

```
<lightning:input type="number" name="myNumber" label="{!$Label.Number.PickOne}" />
```

表示ラベルを出力して動的に更新するには、`format()` 式の関数を使用します。たとえば、`np.labelName` を `Hello {0}` に設定し、`v.name` を `World` に設定した場合、次の式で `Hello World` が返されます。

```
{!format($Label.np.labelName, v.name)}
```

関連トピック:

アクセシビリティのサポート

表示ラベルパラメータの動的な入力

`format()` 式の関数を使用して、表示ラベルを出力および更新します。

実行時に代入値で置き換えられるプレースホルダの文字列を指定できます。

パラメータは必要なだけ追加できます。パラメータは、ゼロから順に番号が付けられます。たとえば、3 つのパラメータがある場合、`{0}`、`{1}`、`{2}` という名前が付けられ、指定された順に置換されます。

`Hello {0} and {1}` の値を持つカスタム表示ラベル `$Label.mySection.myLabel` を見てみましょう。ここで、`$Label` は表示ラベルにアクセスするグローバル値プロバイダです。

次の式により、プレースホルダパラメータに指定した属性の値が動的に入力されます。

```
{!format($Label.mySection.myLabel, v.attribute1, v.attribute2)}
```

属性値の 1 つが変更されると表示ラベルが自動的に更新されます。

メモ: プレースホルダパラメータで表示ラベルを参照するには、常に `$Label` グローバル値プロバイダを使用します。プレースホルダパラメータの文字列を `format()` の最初の引数として設定することはできません。たとえば、次の構文は機能しません。

```
{!format('Hello {0}', v.name)}
```

代わりに次の式を使用します。

```
{!format($Label.mySection.salutation, v.name)}
```

この `$Label.mySection.salutation` は `Hello {0}` に設定されています。

JavaScript での表示ラベルの取得

表示ラベルはJavaScriptコードで取得できます。表示ラベルを静的に定義して、コンポーネントの読み込み時にクライアントに送信すると、コードは最適に実行されます。

静的表示ラベル

静的表示ラベルは、`"$Label.c.task_mode_today"` のように、1つの文字列で定義します。フレームワークは静的表示ラベルをマークアップかJavaScriptコードで解析し、コンポーネントの読み込み時にクライアントに表示ラベルを送信します。表示ラベルを解決するためのサーバとの往復は必要ありません。静的表示ラベルをJavaScript コードで取得するには、`$A.get()` を使用します。次に例を示します。

```
var staticLabel = $A.get("$Label.c.task_mode_today");
component.set("v.mylabel", staticLabel);
```

Apexコードで表示ラベルを定義し、JavaScriptを使用してその表示ラベルをコンポーネントに送信することもできます。詳細は、「Apex での表示ラベルの取得」を参照してください。

動的表示ラベル

表示ラベルはJavaScriptコードで動的に作成できます。この技法が便利なのは、実行時に動的に作成されるまで不明な表示ラベルを使用する必要があるときです。

```
// Assume the day variable is dynamically generated
// earlier in the code
// THIS CODE WON'T WORK
var dynamicLabel = $A.get("$Label.c." + day);
```

クライアントが表示ラベルをすでに把握している場合は、`$A.get()` によって表示ラベルが表示されます。この値が把握されていない場合、本番モードでは空の文字列が表示され、デバッグモードでは表示ラベルキーを示すプレースホルダ値が表示されます。

実行時に特定できない表示ラベルで `$A.get()` を使用すると、`dynamicLabel` は空の文字列になり、取得された値には更新されません。表示ラベル `"$Label.c." + day` は動的に作成されるため、コンポーネントが要求されるとき、フレームワークは表示ラベルを解析したりクライアントに送信したりできません。

別のアプローチによって `$A.get()` を使用し、動的に生成される表示ラベルを操作できます。

動的に生成された既知の表示ラベルをコンポーネントで使用する場合は、JavaScript リソースで表示ラベルの参照を追加して、表示ラベルのサーバの往復を避けることができます。コンポーネントが要求されると、フレームワークはこれらの表示ラベルをクライアントに送信します。たとえば、`$Label.c.task_mode_today` と `$Label.c.task_mode_tomorrow` の表示ラベルキーがコンポーネントによって動的に生成される場合は、クライアント側コントローラまたはヘルパーなど、JavaScript リソースのコメントで表示ラベルの参照を追加します。

```
// hints to ensure labels are preloaded
// $Label.Related_Lists.task_mode_today
// $Label.Related_Lists.task_mode_tomorrow
```

コードによって多くの表示ラベルが生成される場合、このアプローチでは適切にスケールされません。

潜在的な表示ラベルのコメントヒントのすべては追加しない場合、`$A.getReference()` を使用することもできます。このアプローチでは、サーバとの往復というコストをかけて表示ラベルの値を取得することになります。

次の例では、`$A.getReference()` をコールして表示ラベルの値を動的に作成し、取得した表示ラベルで `tempLabelAttr` コンポーネント属性を更新しています。

```
var labelSubStr = "task_mode_today";
var labelReference = $A.getReference("$Label.c." + labelSubStr);
cmp.set("v.tempLabelAttr", labelReference);
var dynamicLabel = cmp.get("v.tempLabelAttr");
```

`$A.getReference()` は表示ラベルの参照を返します。これは文字列ではないため、文字列のように扱わないでください。文字列の表示ラベルを `$A.getReference()` から直接取得することはありません。

その代わりに、返された参照を使用して、コンポーネントの属性値を設定します。このコードでは、`cmp.set("v.tempLabelAttr", labelReference);` でこれを行っています。

表示ラベルの値がサーバから非同期に返されるとき、属性値は参照であるために自動的に更新されます。コンポーネントは再表示されて、表示ラベルの値が表示されます。

メモ: このコードでは、参照の取得直後に `dynamicLabel = cmp.get("v.tempLabelAttr")` を設定しています。このコードでは、表示ラベルの値がサーバから返されるまで、空の文字列が表示されます。このような動作が望ましくない場合は、コメントヒントを使用して、後でサーバとの往復を必要とせずに表示ラベルをクライアントに送信できるようにします。

関連トピック:

JavaScript の使用
入力コンポーネントの表示ラベル
表示ラベルパラメータの動的な入力

Apex での表示ラベルの取得

Apex コードで表示ラベルを取得し、JavaScript を使用してその表示ラベルをコンポーネントに設定できます。

カスタム表示ラベル

カスタム表示ラベルには1,000文字の制限があり、Apexクラスからアクセスできます。構文 `Label.MyLabelName` を使用して、静的表示ラベルを定義できます。コンポーネントは、初期化中などに要求してサーバから表示ラベルを読み込みます。次に例を示します。

```
public with sharing class LabelController {
    @AuraEnabled
    public static String getLabel() {
        String s1 = 'Hello from Apex Controller, ' ;
        String s2 = Label.MyLabelName;
        return s1 + s2;
    }
}
```

JavaScript コードで表示ラベルが取得されます。

```
({
    doInit : function(component, event, helper) {
        var action = component.get("c.getLabel");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                component.set("v.mylabel", response.getReturnValue());
            }
            // error handling when state is "INCOMPLETE" or "ERROR"
         });
         $A.enqueueAction(action);
    }
})
```

最後に、Apexクラスがコンポーネントに結び付けられていることを確認します。表示ラベルは、初期化中にコンポーネントで設定されます。

```
<aura:component controller="LabelController">
    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />
    <aura:attribute name="mylabel" type="String"/>
    {!v.mylabel}
</aura:component>
```

表示ラベルの値を渡す

式の構文 `!{v.mylabel}` を使用して、表示ラベルの値を渡します。String属性にデフォルト値を指定する必要があります。使用事例に応じて、デフォルト値はデフォルト言語の表示ラベルであったり、実行時まで特定の表示ラベルが不明な場合は 1 文字の空白であったりします。

```
<aura:component controller="LabelController">
    <aura:attribute name="mylabel" type="String" default=" "/>
    <lightning:input name="mytext" label="{!v.mylabel}"/>
</aura:component>
```

JavaScriptコードで、実行時に生成される表示ラベルの動的な作成を含み、表示ラベルを取得することもできます。詳細は、「JavaScript での表示ラベルの取得」を参照してください。

親属性による表示ラベル値の設定

親属性による表示ラベル値の設定は、子コンポーネントの表示ラベルを制御する場合に便利です。

コンテナコンポーネントに inner.cmp という別のコンポーネントが含まれているとします。コンテナコンポーネントの属性で inner.cmp の表示ラベル値を設定します。これを行うには、属性型とデフォルト値を指定します。内部コンポーネントの表示ラベルを設定する場合、次の例のように親属性でデフォルト値を設定する必要があります。

次のコンポーネントは、_label 属性のデフォルト値 My Label を含むコンテナコンポーネントです。

```
<aura:component>
    <aura:attribute name="_label"
                    type="String"
                    default="My Label"/>
    <lightning:button label="Set Label" aura:id="button1" onclick="{!c.setLabel}"/>
    <auradocs:inner aura:id="inner" label="{!v._label}"/>
</aura:component>
```

次の inner コンポーネントには、テキストエリアコンポーネントおよびコンテナコンポーネントで設定された label 属性が含まれます。

```
<aura:component>
    <aura:attribute name="label" type="String"/>
    <lightning:textarea aura:id="textarea"
                        name="myTextarea"
                        label="{!v.label}"/>
</aura:component>
```

次のクライアント側のコントローラアクションで表示ラベル値を更新します。

```
({
    setLabel:function(cmp) {
        cmp.set("v._label", 'new label');
    }
})
```

コンポーネントが初期化されると、My Label という表示ラベルのボタンおよびテキストエリアが表示されます。コンテナコンポーネントのボタンがクリックされると、setLabel アクションによって、inner コンポーネントの表示ラベル値が更新されます。このアクションによって label 属性が検索され、その値が new label に設定されます。

関連トピック:

入力コンポーネントの表示ラベル

コンポーネントの属性

ローカライズ

このフレームワークでは、入力および出力コンポーネントでクライアント側ローカライズのサポートを提供します。

次の例では、デフォルトの timezone 属性を上書きする方法を示します。出力には、デフォルトで hh:mm 形式の時刻が表示されます。

```
<aura:component>
    <ui:outputDateTime value="2013-10-07T00:17:08.997Z"  timezone="Europe/Berlin" />
</aura:component>
```

このコンポーネントは、Oct 7, 2013 2:17:08 AM と表示されます。

日時形式をカスタマイズするには、lightning:formattedDateTime を使用することをお勧めします。次の例では、init ハンドラを使用して、日時を設定します。

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <aura:attribute name="datetime" type="DateTime"/>
    <lightning:formattedDateTime value="{!v.datetime}" timeZone="Europe/Berlin"
                               year="numeric" month="short" day="2-digit" hour="2-digit"

                                minute="2-digit" second="2-digit"/>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var date = new Date();
        component.set("v.datetime", date)
    }
})
```

次の例では、MMM DD, YYYY HH:MM:SS AM の形式で表示される JavaScript の日付インスタンスを作成します。

この例の出力は <ui:outputDateTime value="{!v.datetime}" timezone="Europe/Berlin" /> と似ていますが、lightning:formattedDateTime の属性では、詳細なレベルで形式を制御できます。たとえば、MM/DD/YYYY 形式を使用して日付を表示できます。

```
<lightning:formattedDateTime value="{!v.datetime}" timeZone="Europe/Berlin" year="numeric"
 month="numeric" day="numeric"/>
```

メモ: 詳細は、「lightning:formattedDateTime (ベータ)」および「ui:outputDateTime」を参照してください。

さらに、グローバル値プロバイダ $Locale を使用してロケール情報を取得できます。組織のロケール設定は、ブラウザのロケール情報より優先されます。

ロケール情報の操作

単一通貨の組織の場合、Salesforce システム管理者が組織の通貨のロケール、デフォルトの言語、デフォルトのロケール、デフォルトのタイムゾーンを設定します。ユーザは個人設定ページで各自の言語、ロケール、タイムゾーンを設定できます。

メモ: 単一言語の組織は言語は変更できませんが、ロケールは変更できます。

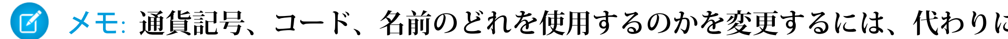
たとえば、[言語とタイムゾーン] ページでタイムゾーンを (GMT+02:00) に設定して次のコードを実行すると、28.09.2015 09:00:00 が返されます。

```
<ui:outputDateTime value="09/28/2015" />
```

`$A.get("$Locale.timezone")` を実行すると、タイムゾーン名 (`Europe/Paris` など) が返されます。詳細は、Salesforce ヘルプの「サポートされているタイムゾーン」を参照してください。

[組織情報] ページで通貨のロケールを `[Japanese (Japan) - JPY]` に設定して次のコードを実行すると、`¥100,000` が返されます。

```
<ui:outputCurrency value="100000" />
```



メモ: 通貨記号、コード、名前のどれを使用するのかを変更するには、代わりに `lightning:formattedNumber` を使用します。詳細は、「lightning:formattedNumber (ベータ)」を参照してください。

同様に、組織の通貨のロケールが `[Japanese (Japan) - JPY]` に設定されているときに `$A.get("$Locale.currency")` を実行すると、`"¥"` が返されます。詳細は、Salesforce ヘルプの「サポートされている通貨」を参照してください。

関連トピック:

JavaScript での日付の書式設定

コンポーネントのドキュメントの提供

コンポーネントのドキュメントは、作成したコンポーネントを他のユーザが理解し、使用するのに役立ちます。

次の 2 種類のコンポーネント参照ドキュメントを提供できます。

- ドキュメント定義 (DocDef): 説明、サンプルコード、例への参照などを含む、コンポーネントの詳細なドキュメント。DocDef は、幅広い HTML マークアップをサポートし、コンポーネントの概要と機能を説明するのに役立ちます。
- インライン説明: テキストのみの説明。通常は 1 文か 2 文で、タグ内の `description` 属性で設定します。

DocDef を入力するには、開発者コンソールのコンポーネントサイドバーにある **[DOCUMENTATION]** をクリックします。次の例では、`np:myComponent` の DocDef を示します。

メモ: DocDef は現在コンポーネントとアプリケーションでサポートされています。イベントとインターフェースでは、インライン説明のみがサポートされます。

```
<aura:documentation>
    <aura:description>
        <p>An <code>np:myComponent</code> component represents an element that executes
an action defined by a controller.</p>
        <!--More markup here, such as <pre> for code samples-->
    </aura:description>
    <aura:example name="myComponentExample" ref="np:myComponentExample" label="Using the
np:myComponent Component">
 <p>This example shows a simple setup of <code>myComponent</code>.</p>
    </aura:example>
    <aura:example name="mySecondExample" ref="np:mySecondExample" label="Customizing the
np:myComponent Component">
       <p>This example shows how you can customize <code>myComponent</code>.</p>
```

```
    </aura:example>
</aura:documentation>
```

ドキュメント定義には次のタグが含まれます。

タグ	説明
<aura:documentation>	DocDef の最上位定義
<aura:description>	幅広いHTMLマークアップを使用してコンポーネントを記述します。説明にコードサンプルを含めるには、コードブロックとして表示される <pre> タグを使用します。<pre> タグに入力するコードはエスケープされる必要があります。たとえば、<aura:component> と入力して <aura:component> をエスケープします。
<aura:example>	コンポーネントの使用方法を示す例を参照します。幅広い HTML マークアップをサポートし、視覚的な出力とコンポーネントのソース例の前にテキストとして表示されます。例は、インタラクティブな出力として表示されます。例は複数作成できます。例は、個々の <aura:example> タグでラップする必要があります。 • name: 例の API 名 • ref: <namespace:exampleComponent> 形式のコンポーネント例への参照 • label: タイトルの表示ラベル

コンポーネント例の提供

DocDefにはコンポーネント例への参照が含まれます。コンポーネント例は、aura:example を使用して結び付けられると、コンポーネント参照ドキュメント内にインタラクティブなデモとして表示されます。

```
 <aura:example name="myComponentExample" ref="np:myComponentExample" label="Using the
np:myComponent Component">
```

np:myComponent の使用方法を示すコンポーネント例を次に示します。

```
<!--The np:myComponentExample example component-->
<aura:component>
    <np:myComponent>
        <aura:set attribute=”myAttribute”>This sets the attribute on the np:myComponent
component.</aura:set>
        <!--More markup that demonstrates the usage of np:myComponent-->
    </np:myComponent>
</aura:component>
```

インライン説明の提供

インライン説明は、要素の短い概要を提供します。インライン説明では HTML マークアップはサポートされていません。description 属性を介するインライン説明が、次のタグでサポートされています。

タグ	例
<aura:component>	<aura:component description="Represents a button element">
<aura:attribute>	<aura:attribute name="label" type="String" description="The text to be displayed inside the button."/>
<aura:event>	<aura:event type="COMPONENT" description="Indicates that a keyboard key has been pressed and released"/>
<aura:interface>	<aura:interface description="A common interface for date components"/>
<aura:registerEvent>	<aura:registerEvent name="keydown" type="ui:keydown" description="Indicates that a key is pressed"/>

ドキュメントの表示

作成したドキュメントは、https://<myDomain>.lightning.force.com/auradocs/reference.app (<myDomain> はカスタム Salesforce ドメインの名前) で使用できます。

関連トピック:

リファレンス

Lightning 基本コンポーネントの使用

Lightning 基本コンポーネントは、最新の Lightning Experience、Salesforce1、および Lightning Communities ユーザインターフェースを構成するビルディングブロックです。

Lightning 基本コンポーネントには Lightning Design System のマークアップとクラスが組み込まれており、最小のフットプリントでパフォーマンスとアクセシビリティの向上を提供します。

これらの基本コンポーネントは、HTML と CSS の詳細を処理します。各コンポーネントには、さまざまなスタイル設定を可能にする簡単な属性が用意されています。つまり、通常は CSS を使用する必要はまったくありません。Lightning基本コンポーネント属性の簡素化と、その属性の無駄がなく一貫性のある定義によって属性が使いやすくなり、ビジネスロジックに集中できます。

Lightning 基本コンポーネントを lightning 名前空間で検索し、既存の ui 名前空間コンポーネントを補完できます。一致する ui と lightning 名前空間コンポーネントがあるインスタンスでは、lightning 名前空間コンポーネントを使用することをお勧めします。lightning 名前空間コンポーネントは、一般的な使用事例に合うように最適化されています。Lightning Design System のスタイル設定が装備されているコンポーネントの範囲を超えると、アクセシビリティ、リアルタイム操作、および拡張エラーメッセージが処理されます。

後続のリリースでは、追加の Lightning 基本コンポーネントを提供していく予定です。今後 lightning 名前空間が ui 名前空間と同等になり、やがてそれを超えることが想定されています。

また、Lightning 基本コンポーネントは時間と共に Lightning Design System に合わせて進化していきます。そのため、カスタマイズは引き続き Lightning Experience と Salesforce1 に整合します。

すべての使用できるコンポーネントについては、https://<myDomain>.lightning.force.com/auradocs/reference.app (<myDomain> は、Salesforce カスタムドメインの名前) のコンポーネントの参照、または「コンポーネントの参照」セクションを参照してください。

コンテナコンポーネント

次のコンポーネントは、関連情報をまとめてグルーピングします。

型	Lightning コンポーネント名	説明	Lightning Design System
アコーディオン	lightning:accordion	複数のコンテンツ領域を含むセクションを縦に積み上げたコレクション。同時に展開されるのは 1 つのみです。	アコーディオン
	lightning:accordionSection	lightning:accordion コンポーネント内にネストされた 1 つのセクション。	
カード	lightning:card	情報の関連グルーピングを囲むコンテナを適用します。	カード
レイアウト	lightning:layout	ページにコンテナを配置するための反応型グリッドシステム。	グリッド
	lightning:layoutItem	lightning:layout コンポーネント内のコンテナ。	
タブ	lightning:tab	lightning:tabset コンポーネント内にネストされた 1 つのタブ。	タブ
	lightning:tabset	タブのリストを表します。	
タイル	lightning:tile	レコードに関連付けられた関連情報のグループ。	タイル

入力コントロールコンポーネント

次のコンポーネントは、ボタンに基づきます。

型	Lightning コンポーネント名	説明	Lightning Design System
ボタン	lightning:button	ボタン要素を表します。	ボタン

型	Lightning コンポーネント名	説明	Lightning Design System
ボタンアイコン	lightning:buttonIcon	アイコンのみの HTML ボタン。	ボタンアイコン
ボタンアイコン (ステートフル)	lightning:buttonIconStateful	状態を保持するアイコンのみのボタンです。	ボタンアイコン
ボタングループ	lightning:buttonGroup	ボタンのグループを表します。	ボタングループ
ボタンメニュー	lightning:buttonMenu	アクションまたは関数のリストを含むドロップダウンメニュー。	メニュー
	lightning:menuItem	lightning:buttonMenu のリスト項目。	
ボタンステートフル	lightning:buttonStateful	状態を切り替えるボタン。	ボタンステートフル

ナビゲーションコンポーネント

次のコンポーネントは、ボタンに基づきます。

型	Lightning コンポーネント名	説明	Lightning Design System
ブレッドクラム	lightning:breadcrumb	ユーザが表示しているページの階層パス内の項目。	Breadcrumbs
	lightning:breadcrumbs	Web サイトまたはアプリケーション内で現在アクセスしているページの階層パス。	
ツリー	lightning:tree	ネストされた項目の階層構造を表示します。	ツリー
ボタンメニュー	lightning:buttonMenu	アクションまたは関数のリストを含むドロップダウンメニュー。	メニュー
	lightning:menuItem	lightning:buttonMenu のリスト項目。	
垂直ナビゲーション	lightning:verticalNavigation	別のページまたは現在のページの別の部分に移動するための垂直型のリンクリスト。	垂直ナビゲーション
	lightning:verticalNavigationItem	lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のテキストのみのリンク。	

型	Lightning コンポーネント名	説明	Lightning Design System
	lightning:verticalNavigationItemBadge	lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のリンクとバッジ。	
	lightning:verticalNavigationItemIcon	lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のリンクとアイコン。	
	lightning:verticalNavigationOverflow	lightning:verticalNavigation の項目のオーバーフロー。	
	lightning:verticalNavigationSection	lightning:verticalNavigation 内のセクション。	

ビジュアルコンポーネント

次のコンポーネントは、たとえばアイコンや読み込みスピナーなどの情報キューを提供します。

型	Lightning コンポーネント名	説明	Lightning Design System
アバター	lightning:avatar	オブジェクトのビジュアル表現。	アバター
バッジ	lightning:badge	少量の情報を保持する表示ラベル。	バッジ
データテーブル	lightning:datatable	型に応じて書式設定されたデータの列を表示するテーブルです。	
動的アイコン	lightning:dynamicIcon	さまざまなアニメーションアイコン。	動的アイコン
ヘルプテキスト (ツールチップ)	lightning:helptext	少量のテキストのポップオーバーコンテナ付きアイコン。	ツールチップ
アイコン	lightning:icon	コンテキストを提供するビジュアル要素。	アイコン
ピル	lightning:pill	ピルとは、ユーザが生成する自由形式のテキストではない、データベース内の既存の項目を表します。	ピル
進行状況バー	lightning:progressBar	操作の進行状況を示す、左から右の水平方向の進行状況バー。	進行状況バー

型	Lightning コンポーネント名	説明	Lightning Design System
進行状況インジケータとパス	lightning:progressIndicator	完了している内容を示す処理のステップを表示します。	進行状況インジケータ パス
スピナー	lightning:spinner	アニメーションスピナーを表示します。	スピナー

項目コンポーネント

次のコンポーネントは、値の入力を有効にします。

型	Lightning コンポーネント名	説明	Lightning Design System
チェックボックスグループ	lightning:checkboxGroup	オプションのグループの単一または複数の選択を有効にします。	チェックボックス
コンボボックス	lightning:combobox	オプションのリストから単一の選択を有効にする入力要素。	コンボボックス
デュアルリストボックス	lightning:dualListbox	選択可能なオプションのリストボックスを含む入力リストボックスを提供します。2 つのリスト間でオプションを移動できます。	デュエル選択リスト
ファイルアップローダおよびプレビュー	lightning:fileUpload	レコードへのファイルのアップロードを有効にします。	ファイルセレクタ
入力	lightning:input	type 属性に応じてユーザ入力を受け入れる対話型コントロールを表します。	入力
入力位置 (地理位置情報)	lightning:inputLocation	緯度値と経度値を受け入れる地理位置情報の複合項目。	N/A
ラジオグループ	lightning:radioGroup	オプションのグループの単一の選択を有効にします。	ラジオボタン ラジオボタングループ
選択	lightning:select	HTML select 要素を作成します。	選択
スライダ	lightning:slider	指定した 2 つの数字間の値を指定するための入力範囲スライダ。	スライダ

型	Lightning コンポーネント名	説明	Lightning Design System
リッチテキストエリア	lightning:inputRichText	カスタマイズ可能なツールバーを備えた、リッチテキスト入力用の WYSIWYG エディタ。	リッチテキストエディタ
テキストエリア	lightning:textArea	複数行のテキスト入力。	TextArea

書式設定済みコンポーネント

次のコンポーネントは、参照のみの書式設定済みの値を表示できるようにします。

型	Lightning コンポーネント名	説明	Lightning Design System
日付/時間	lightning:formattedDateTime	書式設定された日付と時刻を表示します。	なし
メール	lightning:formattedEmail	mailto: URL スキームが付いているハイパーリンクとしてメールを表示します。	
地理位置情報	lightning:formattedLocation	緯度，経度の形式を使用して地理位置情報を表示します。	
数値	lightning:formattedNumber	書式設定された数値を表示します。	
電話	lightning:formattedPhone	tel: URL スキームが付いているハイパーリンクとして電話番号を表示します。	
リッチテキスト	lightning:formattedRichText	ホワイトリストに登録されているタグと属性を使用して書式設定されたリッチテキストを表示します。	
テキスト	lightning:formattedText	テキストを表示し、改行を禁則処理で置き換え、要求された場合は linkify で表示します。	
URL	lightning:formattedUrl	URL をハイパーリンクとして表示します。	
相対日付/時間	lightning:relativeDateTime	ソースの日時と指定された日時の相対的な時差を表示します。	

Lightning 基本コンポーネントの考慮事項

Lightning 基本コンポーネントの使用に関するガイドラインを確認してください。

Lightningコンポーネントのマークアップに依存しないことをお勧めします。その内部は今後、変更されることがあります。たとえば、`cmp.get("v.body")` を使用してDOM要素を検討すると、コンポーネントマークアップが今後変更された場合、問題を引き起こす可能性があります。LockerService を適用すると、所有していないコンポーネントの DOM をトラバースできなくなります。DOM ツリーにアクセスする代わりに、コンポーネント属性とバインドしている値を活用し、利用可能なコンポーネントのメソッドを使用できます。たとえば、コンポーネントの属性を取得するには、`cmp.find("myInput").getElement().name` の代わりに `cmp.find("myInput").get("v.name")` を使用します。後者は、別の名前空間のコンポーネントなど、コンポーネントへのアクセス権がない場合は機能しません。

多くのLightning基本コンポーネントは発展中であり、アプリケーションの構築中には次の考慮事項が役立つことがあります。

lightning:buttonMenu (ベータ)

このコンポーネントには、ボタンがトリガされた場合に限って作成されるメニュー項目が含まれます。初期化中、またはボタンがトリガされていない場合は、メニュー項目を参照できません。

lightning:formattedDateTime (ベータ)

このコンポーネントは、Apple Safari 10以下でフォールバック動作を提供します。次のフォーマッティングオプションには、古いブラウザでフォールバック動作を使用するときに例外があります。

- `era` はサポートされていません。
- `timeZoneName` は、短縮形では `GMT`、長い形式では `GMT-h:mm` または `GMT+h:mm` を追加します。
- `timeZone` は `UTC` をサポートします。別のタイムゾーン値を使用する場合、`lightning:formattedDateTime` はブラウザのタイムゾーンを使用します。

lightning:formattedNumber (ベータ)

このコンポーネントは、Apple Safari 10 以下で次のフォールバック動作を提供します。

- `style` を `currency` に設定した場合、ロケールとは異なる `currencyCode` 値を指定すると、記号の代わりに通貨コードが表示されます。次の例では、フォールバックモードの場合に `EUR12.34` が表示され、その他の場合に `€12.34` が表示されます。

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="EUR"/>
```

- `currencyDisplayAs` では、`symbol` のみがサポートされます。次の例では、`currencyCode` がユーザのロケール通貨と一致する場合に限ってフォールバックモードで `$12.34` が表示され、その他の場合は `USD12.34` が表示されます。

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="USD" currencyDisplayAs="symbol"/>
```

lightning:input (ベータ)

次のコンポーネントでは日付ピッカーを使用できますが、Lightning Design System のスタイル設定は継承されません。

- `<lightning:input type="date" />`
- `<lightning:input type="datetime-local" />`

パーセント入力と通貨入力の項目では、ネイティブ実装によって要求されるように、段階的な増分を 0.01 に指定する必要があります。

```
<lightning:input type="number" name="percentVal" label="Enter a percentage value"
formatter="percent" step="0.01" />
<lightning:input type="number" name="currencyVal" label="Enter a dollar amount"
formatter="currency" step="0.01" />
```

チェックボックス、ラジオボタン、トグルスイッチを操作するときは、aura:idを使用し、コンポーネントの配列をグループ化してトラバースします。グループ化すると、get("v.checked") を使用して、どの要素がオンまたはオフになっているかを判断でき、DOM にアクセスする必要はありません。name 属性と value 属性を使用し、反復中に各コンポーネントを識別することもできます。次の例では、aura:id を使用して 3 個のチェックボックスをグループ化しています。

```
<aura:component>
    <form>
      <fieldset>
        <legend>Select your favorite color:</legend>
        <lightning:input type="checkbox" label="Red"
            name="color1" value="1" aura:id="colors"/>
        <lightning:input type="checkbox" label="Blue"
            name="color2" value="2" aura:id="colors"/>
        <lightning:input type="checkbox" label="Green"
            name="color3" value="3" aura:id="colors"/>
      </fieldset>
    <lightning:button label="Submit" onclick="{!c.submitForm}"/>
    </form>
</aura:component>
```

クライアント側のコントローラでは、cmp.find("colors") を使用して配列を取得し、checked の値を調べることができます。

type="file" を使用する場合、Salesforceにファイルをアップロードする独自のサーバ側ロジックを提供する必要があります。FileReader HTML オブジェクトを使用してファイルを読み取り、Apex コントローラに送信する前にファイルの内容をエンコードします。Apex コントローラでは、EncodingUtil メソッドを使用してファイルデータをデコードできます。たとえば、Attachmentオブジェクトを使用して、親オブジェクトにフィルをアップロードできます。この場合、base64 エンコードファイルを Body 項目に渡し、Apex コントローラでファイルを添付ファイルとして保存できます。

このコンポーネントを使用したファイルのアップロードには、1 MB という通常の Apex コントローラの制限が適用されます。base64 エンコードによるファイルサイズの増加に対応するために、最大ファイルサイズを 750 KB に設定することをお勧めします。ファイルサイズが 1 MB を超える場合は、チャンクを実装する必要があります。チャンクを使用してアップロードするファイルには、4 MB というサイズ制限が適用されます。詳細は、『*Apex コード開発者ガイド*』を参照してください。

lightning:tab (ベータ)

このコンポーネントでは、実行中に本文が作成されます。初期化中にはコンポーネントを参照できません。aura:id を使用してコンポーネントを参照すると、cmp.find("myComponent") の実装時にコンポーネントが未定義の値を返すというように、予期しない結果が返されることがあります。

lightning:tabset (ベータ)

viewportの幅に適合するよりも多くのタブを読み込むと、タブセットはナビゲーションボタンを提供し、横方向にスクロールして、オーバーフローしたタブを表示します。

関連トピック:

コンポーネントの参照

Lightning 基本コンポーネントでのイベント処理

基本コンポーネントは軽量で、HTMLマークアップによく似ています。標準的なHTMLの手法に従って、onfocus などのイベントハンドラを属性として提供します。ui 名前空間のコンポーネントのように、Lightning コンポーネントイベントを登録して起動するのではありません。

マークアップを見て、event.target または event.currentTarget を使用して DOM 要素にアクセスするのではないかと思うかもしれません。けれども、この方法によるアクセスでは、変更される可能性のある別のコンポーネントの DOM 要素にアクセス可能になるため、カプセル化が壊れます。

LockerService (Summer '17 ですべての組織で有効化になる予定) は、カプセル化を適用するものです。ここで説明するメソッドを使用して、コードを LockerService に対応させます。

イベントを起動したコンポーネントを取得するには、event.getSource() を使用します。

```
<aura:component>
    <lightning:button name="myButton" onclick="{!c.doSomething}"/>
</aura:component>
```

```
({
    doSomething: function(cmp, event, helper) {
        var button = event.getSource();

        //The following patterns are not supported
        //when you're trying to access another component's
        //DOM elements.
        var el = event.target;
        var currentEl = event.currentTarget;
    }
})
```

この構文を使用して、イベントに渡されたコンポーネント属性を取得します。

```
event.getSource().get("v.name")
```

イベントハンドラの再利用

event.getSource() は、どのコンポーネントがイベントを起動したかを判断するうえで役立ちます。たとえば、同じ onclick ハンドラを再利用するボタンがいくつかあるとします。イベントを起動したボタンの名前を取得するには、event.getSource().get("v.name") を使用します。

```
<aura:component>
    <lightning:button label="New Record" name="new" onclick="{!c.handleClick}"/>
    <lightning:button label="Edit" name="edit" onclick="{!c.handleClick}"/>
```

```
    <lightning:button label="Delete" name="delete" onclick="{!c.handleClick}"/>
</aura:component>
```

```
({
    handleClick: function(cmp, event, helper) {
        //returns "new", "edit", or "delete"
        var buttonName = event.getSource().get("v.name");
    }
})
```

`onactive` ハンドラを使用した有効なコンポーネントの取得

タブを使用する場合、どのタブが有効かを知る必要があります。`lightning:tab` コンポーネントを使用すると、`onactive` ハンドラを使用してターゲットコンポーネントが有効になったときに、ターゲットコンポーネントへの参照を取得できます。コンポーネントを数回クリックすると、ハンドラが1回だけ呼び出されます。

```
<aura:component>
    <lightning:tabset>
      <lightning:tab onactive="{! c.handleActive }" label="Tab 1" id="tab1" />
      <lightning:tab onactive="{! c.handleActive }" label="Tab 2" id="tab2" />
    </lightning:tabset>
</aura:component>
```

```
({
      handleActive: function (cmp, event) {
          var tab = event.getSource();
          switch (tab.get('v.id')) {
              case 'tab1':
                  //do something when tab1 is clicked
                  break;
              case 'tab2':
                  //do something when tab2 is clicked
                  break;
          }
      }
})
```

`onselect` ハンドラを使用した ID および値の取得

一部のコンポーネントは、イベントを子コンポーネントに渡すイベントハンドラを提供します。次のコンポーネントの `onselect` イベントハンドラなどです。

- `lightning:buttonMenu`
- `lightning:tabset`

`event.detail` 構文は引き続きサポートされますが、今後のリリースで `event.detail` は廃止される予定であるため、`onselect` ハンドラで次のパターンを使用するように JavaScript コードを更新することをお勧めします。

- `event.getParam("id")`
- `event.getParam("value")`

たとえば、クライアント側コントローラから lightning:buttonMenu コンポーネントで選択されたメニュー項目の値を取得できます。

```
//Before
var menuItem = event.detail.menuItem;
var itemValue = menuItem.get("v.value");
//After
var itemValue = event.getParam("value");
```

同様に、lightning:tabset コンポーネントで選択されたタブの ID を取得するには、次のコードを使用します。

```
//Before
var tab = event.detail.selectedTab;
var tabId = tab.get("v.id");
//After
var tabId = event.getParam("id");
```

メモ: ターゲットコンポーネントへの参照が必要な場合は、代わりに onactive イベントハンドラを使用します。

Lightning Design System の考慮事項

Lightning 基本コンポーネントには、Salesforce Lightning Design System のスタイル設定が標準で用意されていますが、要件に応じていくつかの CSS を記述することもできます。

スタンドアロンアプリケーションおよび Lightning Out など、Salesforce1 および Lightning Experience 外部のコンポーネントを使用している場合、force:slds を拡張して Lightning Design System のスタイル設定をコンポーネントに適用します。基本コンポーネントで Lightning Design System を使用するためのガイドラインをいくつか示します。

基本コンポーネントでのユーティリティクラスの使用

Lightning Design System ユーティリティクラスは、コンポーネントのビジュアルデザインの基本であり、位置揃え、グリッド、間隔、タイポグラフィなどの再利用を促進します。ほとんどの基本コンポーネントには class 属性が備えられているため、ユーティリティクラスまたはカスタムクラスをコンポーネントの外部要素に追加できます。たとえば、lightning:button にスペーシングのユーティリティクラスを適用できます。

```
<lightning:button name="submit" label="Submit" class="slds-m-around_medium"/>
```

追加するクラスはコンポーネントに適用される他の基本クラスに追加され、以下のマークアップが生成されます。

```
<button class="slds-button slds-button_neutral slds-m-around_medium"
        type="button" name="submit">Submit</button>
```

同様に、カスタムクラスを作成して class 属性に渡すことができます。

```
<lightning:badge label="My badge" class="myCustomClass"/>
```

提供されている CSS のスキャフォールディングを超えて、詳細レベルで柔軟にコンポーネントをカスタマイズできます。独自の本文マークアップを作成できる lightning:card を見ていきましょう。本文マークアップ

で slds-p-horizontal_small または slds-card__body_inner クラスを適用して、本文の前後にパディングを追加できます。

```
<!-- lightning:card example using slds-p-horizontal_small class -->
<lightning:card>
  <aura:set attribute="title">My Account</aura:set>
  <aura:set attribute="footer">Footer</aura:set>
  <aura:set attribute="actions">
    <lightning:button label="New"/>
  </aura:set>
  <p class="slds-p-horizontal_small">
    Card Body
  </p>
  </lightning:card>
```

```
<!-- lightning:card example using slds-card__body_inner -->
<lightning:card>
    <aura:set attribute="title">My Account</aura:set>
    <aura:set attribute="footer">Footer</aura:set>
    <aura:set attribute="actions">
      <lightning:button label="New"/>
    </aura:set>
    <div class="slds-card__body_inner">
      Card Body
    </div>
</lightning:card>
```

カスタムコンポーネントのスタイル設定の適用

ときとして、ユーティリティクラスでは不十分で、コンポーネントバンドルにカスタムのスタイル設定を追加する必要がある場合があります。前に、カスタムクラスを作成して class 属性に渡すことができることを説明しました。所有していないクラスはいつでも変更される可能性があるため、所有していないクラス名をターゲットとするのではなく、クラスを作成することをおすすめします。たとえば、.slds-input や .lightningInput はデフォルトで、基本コンポーネントで使用可能な CSS クラスであるため、ターゲットとしないでください。コンポーネント間で設計が一致するように、トークンを使用することも検討できます。トークンバンドルで値を指定し、コンポーネントの CSS リソースで再利用します。

レイアウトにグリッドを使用

lightning:layout は柔軟なグリッドシステムへの答えです。lightning:layoutItem コンポーネントを lightning:layout で囲むとシンプルなレイアウトが完成し、slds-grid クラスで div コンテナが作成されます。追加の Lightning Design System グリッドクラスを適用するには、lightning:layout 属性の任意の組み合わせを指定します。たとえば、vertical-align="stretch" を指定して slds-grid_vertical-stretch クラスを追加します。horizontalAlign、verticalAlign、および pullToBoundary 属性を使用して、Lightning Design System グリッドクラスをコンポーネントに適用できます。ただし、これらの属性を介して使用できないグリッドクラスもあります。追加のグリッドクラスを指定するには、class 属性を使用します。次のグリッドクラスは class 属性を使用して追加できます。

- .slds-grid_frame
- .slds-grid_vertical

- `.slds-grid_reverse`
- `.slds-grid_vertical-reverse`
- `.slds-grid_pull-padded-x-small`
- `.slds-grid_pull-padded-xx-small`
- `.slds-grid_pull-padded-xxx-small`

次の例では、`slds-grid_reverse` クラスを `slds-grid` クラスに追加します。

```
<lightning:layout horizontalAlign="space" class="slds-grid_reverse">
   <lightning:layoutItem padding="around-small">
     <!-- more markup here -->
   </lightning:layoutItem>
   <!-- more lightning:layoutItem components here -->
</lightning:layout>
```

詳細は、「lightning:layout」と「Grid utility (グリッドユーティリティ)」を参照してください。

基本コンポーネントへのバリエーションの適用

コンポーネントのバリエーションはそのコンポーネントの設計バリエーションを参照して、ユーザがコンポーネントの外観を容易に変更できるようにします。基本コンポーネントのバリエーションを各々のLightning Design System バリエーションと一致させようとしますが、1 対 1 の対応ではありません。ほとんどの基本コンポーネントには `variant` 属性が用意されています。たとえば、`lightning:button` はさまざまなテキストや背景色をボタンに適用するために、base、neutral、brand、destructive、inverse などの多くのバリエーションをサポートしています。

```
<lightning:button variant="brand" label="Brand" onclick="{! c.handleClick }" />
```

`success` バリエーションはサポートされていません。ただし、`slds-button_success` クラスを追加して同じ結果を得ることができます。

```
<lightning:button name="submit" label="Submit" class="slds-button_success"/>
```

別の例を見てみましょう。`lightning:tile` コンポーネントを使用して、関連する情報のグループを作成できます。このコンポーネントには `variant` 属性は用意されていませんが、`slds-tile_board` クラスを渡すことで Lightning Design System ボードバリエーションを実現できます。

```
<aura:component>
  <ul class="slds-has-dividers_around-space">
    <li class="slds-item">
      <lightning:tile label="Anypoint Connectors" href="/path/to/somewhere"
class="slds-tile_board">
          <p class="slds-text-heading_medium">$500,000</p>
          <p class="slds-truncate" title="Company One"><a href="#">Company One</a></p>
          <p class="slds-truncate">Closing 9/30/2015</p>
       </lightning:tile>
    </li>
  </ul>
</aura:component>
```

必要なバリエーションがない場合は、独自のカスタム CSS クラスを作成する前に、Lightning Design System クラスを基本コンポーネントに渡すことができるかどうかを確認します。恐れずに基本コンポーネントの Lightning

Design System クラスやバリエーションを試してみてください。詳細は、「Lightning Design System」を参照してください。

関連トピック:

アプリケーションのスタイル設定

設計トークンを使用したスタイル設定

Lightning Design System のバリエーションの使用

基本コンポーネントのバリエーションは、Lightning Design System のバリエーションに対応しています。バリエーションは variant 属性で制御され、これによってコンポーネントの外観が変化します。

サポートされていないバリエーションを渡すと、代わりにデフォルトのバリエーションが使用されます。次の例では、base バリエーションを使用してボタンを作成します。

```
<lightning:button variant="base" label="Base" onclick="{! c.handleClick }"/>
```

次のリファレンスでは、基本コンポーネントのバリエーションが Lightning Design System のバリエーションにどのように対応しているのかについて説明します。lightning:formattedDateTime など、ビジュアルスタイル設定のない基本コンポーネントはここに記載していません。これらのコンポーネントについての詳細は、「コンポーネントの参照」を参照してください。

アコーディオン

lightning:accordion コンポーネントは、コンテンツ領域を含むセクションを縦に積み上げたコレクションです。同時に展開されるのは 1 つのみです。このコンポーネントでは variant 属性はサポートされません。lightning:accordion は、Lightning Design System のアコーディオンのスタイル設定を使用します。

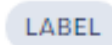
アバター

lightning:avatar コンポーネントは、取引先やユーザなどのオブジェクトを表す画像です。アバターはさまざまなサイズで作成できます。lightning:avatar は、Lightning Design System のアバターのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
square (デフォルト)	slds-avatar	角丸正方形のアバター
circle	slds-avatar_circle	円形のアバター

バッジ

lightning:badge コンポーネントは、少量の情報を保持する表示ラベルです。このコンポーネントでは variant 属性はサポートされません。lightning:badge は、Lightning Design System のバッジのスタイル設定を使用します。



ブレッドクラム

lightning:breadcrumbs コンポーネントは、少量の情報を保持する表示ラベルです。このコンポーネントでは variant 属性はサポートされません。lightning:breadcrumb は、Lightning Design System のブレッドクラムのスタイル設定を使用します。

PARENT ENTITY > PARENT RECORD NAME

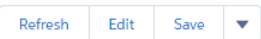
ボタン

lightning:button コンポーネントは、クライアント側コントローラでアクションを実行するボタンです。ボタンでは、テキスト表示ラベルの左側または右側のアイコンがサポートされます。lightning:button は、Lightning Design System のボタンのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
neutral (デフォルト)	slds-button_neutral	灰色の境界線と白い背景を使用するボタン
base	slds-button	境界線がなく、テキストリンクのように見えるボタン
brand	slds-button_brand	白いテキストを使用した青いボタン
destructive	slds-button_desctructive	白いテキストを使用した赤いボタン
inverse	slds-button_inverse	白いテキストを使用した紺色のボタン

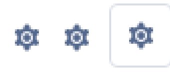
ボタングループ

lightning:buttonGroup コンポーネントは、ナビゲーションバーを作成するために一緒に表示できるボタンのグループです。グループ内に lightning:button コンポーネントと lightning:buttonMenu コンポーネントをネストできます。ボタングループ自体では variant 属性はサポートされませんが、ボタンおよびボタンメニューコンポーネントではバリエーションがサポートされます。たとえば、ボタングループ内に <lightning:button variant="inverse" label="Refresh" /> をネストできます。lightning:buttonMenu を含める場合は、ボタンの後に配置し、slds-button_last クラスを渡して境界線を調整します。lightning:buttonGroup は、Lightning Design System のボタングループのスタイル設定を使用します。



ボタンアイコン

lightning:buttonIcon コンポーネントは、クライアント側コントローラでアクションを実行する、アイコンのみのボタンです。ボタンアイコンはさまざまなサイズで作成できます。Lightning Design System のユーティリティアイコンのみがサポートされます。lightning:buttonIcon は、Lightning Design System のボタンアイコンのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
border (デフォルト)	slds-button_icon-border	灰色の境界線を使用するアイコンを含むボタン
bare	slds-button_icon-bare	境界線のないアイコンを含むボタン
container	slds-button_icon-container	境界線のないアイコンを含む 32 x 32 ピクセルのボタン
border-filled	slds-button_icon-border-filled	灰色の境界線と白の背景を使用するアイコンを含むボタン
bare-inverse	slds-button_icon-inverse	暗色の背景用の境界線のない白いアイコンを含むボタン
border-inverse	slds-button_icon-border-inverse	暗色の背景を使用する白いアイコンを含むボタン

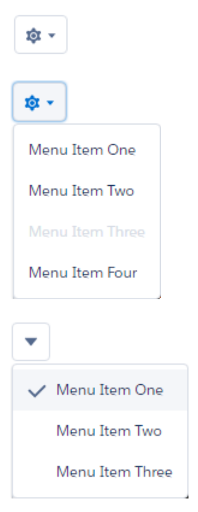
ボタンアイコン (ステートフル)

lightning:buttonIconStateful コンポーネントは、状態を保持するアイコンのみのボタンです。ボタンを押すと、状態を切り替えることができます。ボタンアイコンはさまざまなサイズで作成できます。Lightning Design System のユーティリティアイコンのみがサポートされます。selected 属性は、true に設定されている場合、slds-is-selected クラスを追加します。lightning:buttonIconStateful は、Lightning Design System のボタンアイコンのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
border (デフォルト)	slds-button_icon-border	灰色の境界線を使用するアイコンを含むボタン
border-inverse	slds-button_icon-border-inverse	暗色の背景を使用する白いアイコンを含むボタン

ボタンメニュー

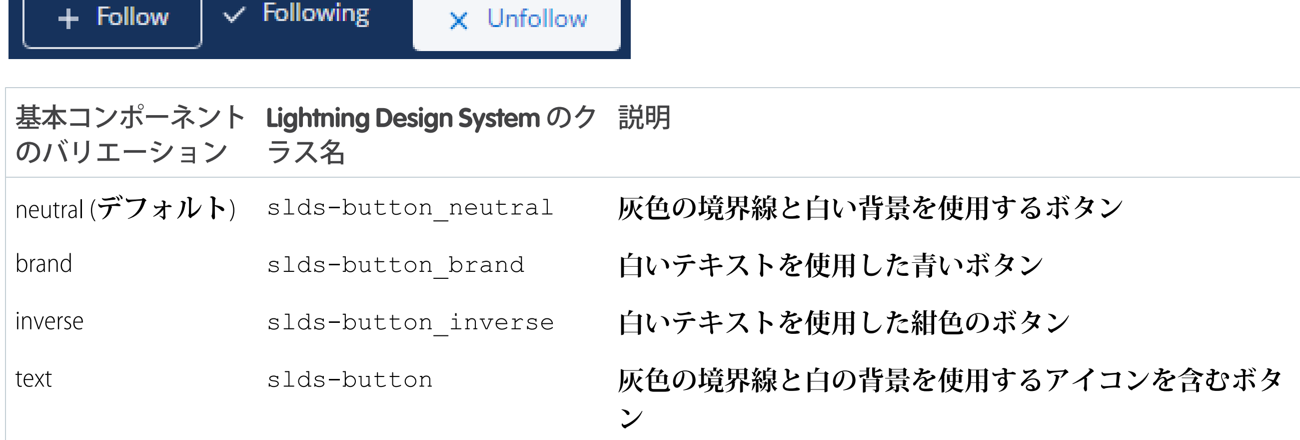
lightning:buttonMenu コンポーネントは、lightning:menuItem コンポーネントで表される、メニュー項目リストがあるドロップダウンメニューです。メニュー項目は、チェックマークを入れるまたは外すことが可能で、クライアント側コントローラでアクションを実行できます。さまざまなサイズのアイコンを使用してボタンメニューを作成したり、ボタンを基準にしたさまざまな位置にドロップダウンメニューを配置したりできます。バリエーションによってボタンの外観が変更されます。これは、ボタンアイコンのバリエーションと類似しています。lightning:buttonMenu は、Lightning Design System のメニューのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
border (デフォルト)	slds-button_icon-border	灰色の境界線を使用するアイコンを含むボタン
bare	slds-button_icon-bare	境界線のないアイコンを含むボタン
container	slds-button_icon-container	境界線のないアイコンを含む 32 x 32 ピクセルのボタン
border-filled	slds-button_icon-border-filled	灰色の境界線と白の背景を使用するアイコンを含むボタン
bare-inverse	slds-button_icon-inverse	暗色の背景用の境界線のない白いアイコンを含むボタン
border-inverse	slds-button_icon-border-inverse	暗色の背景を使用する白いアイコンを含むボタン

ボタン (ステートフル)

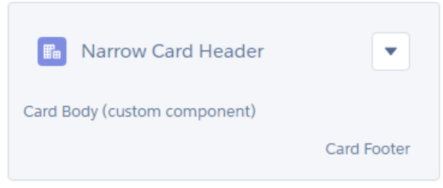
lightning:buttonStateful コンポーネントは、状態を切り替えるボタンです。ステートフルボタンには、状態に基づいて異なる表示ラベルやアイコンを表示できます。Lightning Design System のユーティリティアイコンのみがサポートされます。lightning:buttonStateful は、Lightning Design System のボタンのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
neutral (デフォルト)	slds-button_neutral	灰色の境界線と白い背景を使用するボタン
brand	slds-button_brand	白いテキストを使用した青いボタン
inverse	slds-button_inverse	白いテキストを使用した紺色のボタン
text	slds-button	灰色の境界線と白の背景を使用するアイコンを含むボタン

カード

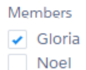
`lightning:card` コンポーネントは、HTML の `article` タグ内の関連情報のグループです。`lightning:card` は、Lightning Design System のカードのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
base (デフォルト)	slds-card	コンテナの幅を使用する関連情報のグループ。
narrow	slds-card_narrow	狭い幅を使用する関連情報のグループ。

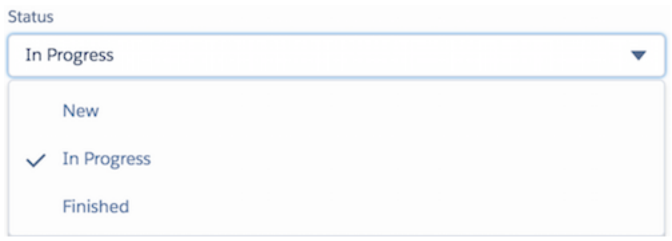
チェックボックスグループ

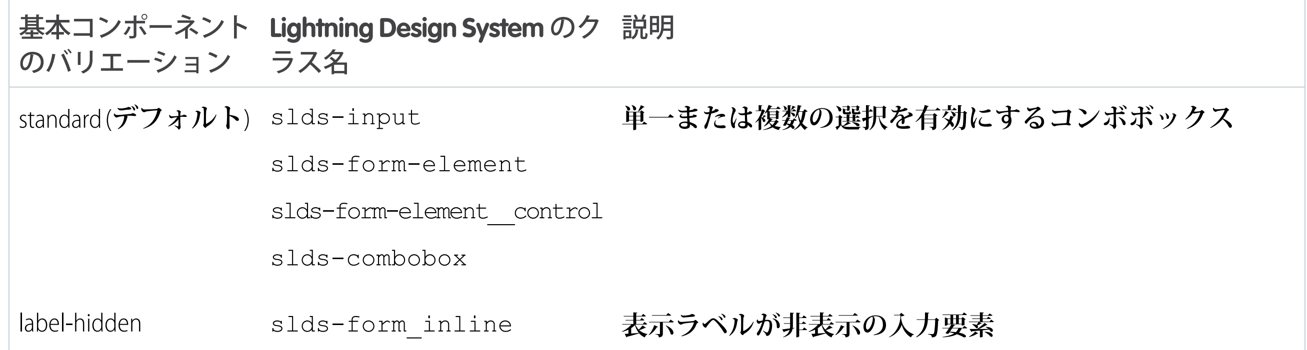
`lightning:checkboxGroup` コンポーネントは、単一または複数のオプションの選択を有効にするチェックボックスのグループです。このコンポーネントは、一連のチェックボックスの一斉グルーピングに適さない `type="checkbox"` の `lightning:input` とは異なります。チェックボックスグループでは `variant` 属性はサポートされませんが、`slds-form-element` クラスが、チェックボックスグループを囲む `fieldset` 要素に追加されます。`lightning:checkboxGroup` は、Lightning Design System のチェックボックスのスタイル設定を使用します。



コンボボックス

lightning:combobox コンポーネントは、オプションのリストから単一または複数の選択を有効にする入力要素です。選択結果は入力値として表示されます。複数選択コンボボックスでは、選択された各オプションが入力要素の下のピルに表示されます。lightning:combobox は、Lightning Design System のコンボボックスのスタイル設定を使用します。





基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	slds-input slds-form-element slds-form-element__control slds-combobox	単一または複数の選択を有効にするコンボボックス
label-hidden	slds-form_inline	表示ラベルが非表示の入力要素

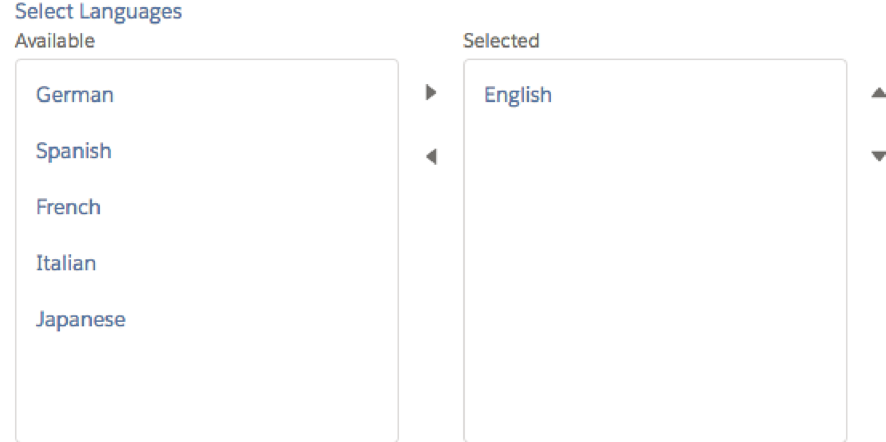
データテーブル

lightning:datatable コンポーネントは、型に応じて書式設定されたデータの列を表示するテーブルです。列のサイズ変更、行の選択、列の並び替えを有効にします。データテーブルでは variant 属性はサポートされませんが、slds-table と slds-table_bordered クラスが table 要素に追加されます。lightning:datatable は、Lightning Design System のデータテーブルのスタイル設定を使用します。

OPPORTUNITY ...	ACCOUNT NAME	CLOSE DATE	CONFIDENCE	AMOUNT
Schimmel, Schim...	Myra	7/23/2017	93%	€73,640.00
McKenzie, McKen...	Kaley	7/18/2017	29%	€29,027.00
Bartoletti-Bartoletti	Letitia	7/15/2017	80%	€13,000.00
Pagac-Pagac	Beryl	7/17/2017	37%	€70,094.00
Emard LLC	Myron	7/23/2017	35%	€90,457.00

デュアルリストボックス

lightning:dualListbox コンポーネントは、2 つのリストボックスを提供します。単一または複数のオプションを 2 番目のリストボックスに移動して、選択したオプションを並び替えることができます。lightning:dualListbox は、Lightning Design System のデュエル選択リストのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	slds-dueling-list	表示ラベルを表示するデュアルリストボックス
label-hidden	slds-form_inline	表示ラベルを非表示にするデュアルリストボックス

動的アイコン

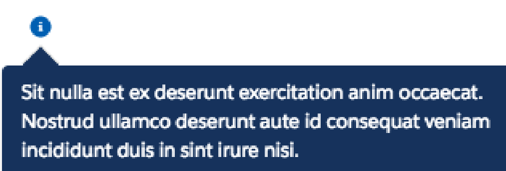
lightning:dynamicIcon コンポーネントは、さまざまなアニメーションアイコンを表します。type 属性は表示するアニメーションアイコンを指定します。また、Lightning Design System の動的アイコンに対応するものです。lightning:dynamicIcon は、Lightning Design System の動的アイコンのスタイル設定を使用します。

ファイルアップローダ

lightning:fileUpload コンポーネントは、レコードへのファイルのアップロードを有効にします。ファイルアップローダには、ドラッグアンドドロップ機能とファイル種別による絞り込みが含まれています。variant 属性はサポートされませんが、slds-file-selector クラスがコンポーネントに追加されます。lightning:fileUpload は、Lightning Design System のファイルセレクタのスタイル設定を使用します。

ヘルプテキスト (ツールチップ)

lightning:helptext コンポーネントは、画面の要素を説明する少量のテキストを含むポップオーバーがあるアイコンを表示します。ヘルプテキストでは variant 属性はサポートされませんが、slds-popover と slds-popover_tooltip クラスがツールチップに追加されます。lightning:helptext は、Lightning Design System のツールチップのスタイル設定を使用します。



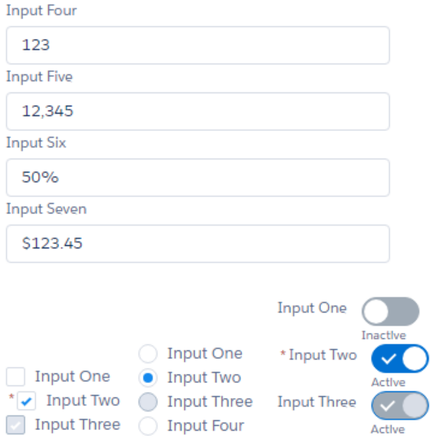
アイコン

lightning:icon コンポーネントは、コンテキストを示し、使いやすさを向上させるビジュアル要素です。Lightning Design System のすべてのアイコンがサポートされますが、バリエーションがサポートされるのはユーティリティアイコンのみです。アイコンはさまざまなサイズで作成できます。lightning:icon は、Lightning Design System のアイコンのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
inverse (デフォルト)	slds-icon	以前は「bare」と呼ばれていました。32 x 32 ピクセルのアイコン。
error	slds-icon-text-error	赤で塗りつぶされたアイコン
warning	slds-icon-text-warning	黄色で塗りつぶされたアイコン

入力

lightning:input コンポーネントは、入力項目やチェックボックスなど、ユーザ入力を受け入れる対話型コントロールです。lightning:input は、Lightning Design System の入力のスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	slds-input slds-form-element slds-form-element__control	入力要素。入力項目、チェックボックス、切り替え、ラジオボタンなどを使用できます。要素に追加されるクラスは、入力種別によって異なります。
label-hidden	slds-form_inline	表示ラベルが非表示の入力要素

レイアウト

`lightning:layout` コンポーネントは、ページ内または別のコンテナ内にコンテナを配置するための柔軟なグリッドシステムです。`variant` 属性を使用する代わりに、レイアウトのカスタマイズは `horizontalAlign`、`verticalAlign`、`pullToBoundary` で制御されます。`lightning:layout` は、Lightning Design System のグリッドのスタイル設定を使用します。詳細は、次のリソースを参照してください。

- lightning:layout
- Lightning Design System の考慮事項

ピル

`lightning:pill` コンポーネントは、角丸の境界線で囲まれたテキスト表示ラベルで、削除ボタンと一緒に表示されます。ピルには、テキスト表示ラベルの横にアイコンまたはアバターを含めることができます。このコンポーネントでは `variant` 属性はサポートされませんが、コンテンツおよびその他の属性で、適用されるスタイル設定が判断されます。たとえば、`hasError="true"` を使用するピルは、赤い境界線とエラーアイコンを使用するピルとして表示されます。`lightning:pill` は、Lightning Design System のピルのスタイル設定を使用します。

進行状況バー

`lightning:progressBar` コンポーネントは、左から右の水平方向の進行状況バーを表示して、操作の進行状況を示します。Lightning Design System の進行状況バーのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
base (デフォルト)	slds-progress-bar	基本の進行状況バー
circular	slds-progress-bar_circular	両端が丸まっている進行状況バー

進行状況インジケータとパス

lightning:progressIndicator コンポーネントは、処理のステップを表示し、完了している内容を示します。type 属性は、進行状況インジケータまたはパスを表示するかどうかを指定します。type="base" を使用する場合、variant 属性を使用できます。lightning:progressIndicator は、Lightning Design System の進行状況バーとパスのスタイル設定を使用します。

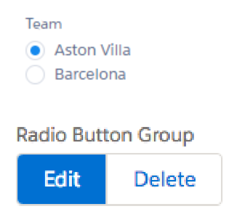
type="path" を使用する場合、variant 属性は使用できません。

また、lightning:path と lightning:picklistPath は、指定された選択リスト項目に基づいたレコードの処理の進行状況を表示できるようにします。lightning:path は [設定] の [パス設定] に基づいてパスを表示し、lightning:picklistPath は picklistFieldApiName 属性から派生したパスを表示します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
base (デフォルト)	slds-progress	type="base" の場合のみ。処理のステップを示します。
shaded	slds-progress_shade	type="base" の場合のみ。現在のステップに網掛け背景を追加します。

ラジオグループ

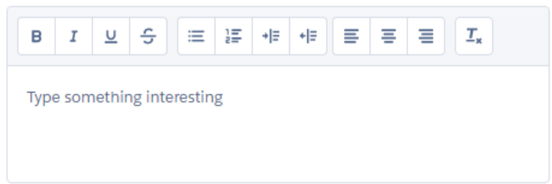
lightning:radioGroup コンポーネントは、単一のオプションの選択を有効にするラジオオプションおよびボタンのグループです。type 属性は、ラジオオプションまたはボタンのグループを表示するかどうかを指定します。lightning:radioGroup は、Lightning Design System のラジオグループのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	slds-radio slds-radio_button-group	ラジオオプションまたはラジオボタンのグループ
label-hidden	slds-form_inline	非表示の表示ラベル付きラジオオプションのグループ

リッチテキストエディタ

`lightning:inputRichText` コンポーネントは、カスタマイズ可能なツールバーがあるリッチテキストエディタです。ツールバーはエディタの上部に表示されますが、`bottom-toolbar` バリエーションを使用してエディタの下に位置を変更できます。`lightning:inputRichText` は、Lightning Design System のリッチテキストエディタのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
bottom-toolbar	slds-rich-text-editor__toolbar-bottom	エディタの下にツールバーが配置されるリッチテキストエディタ

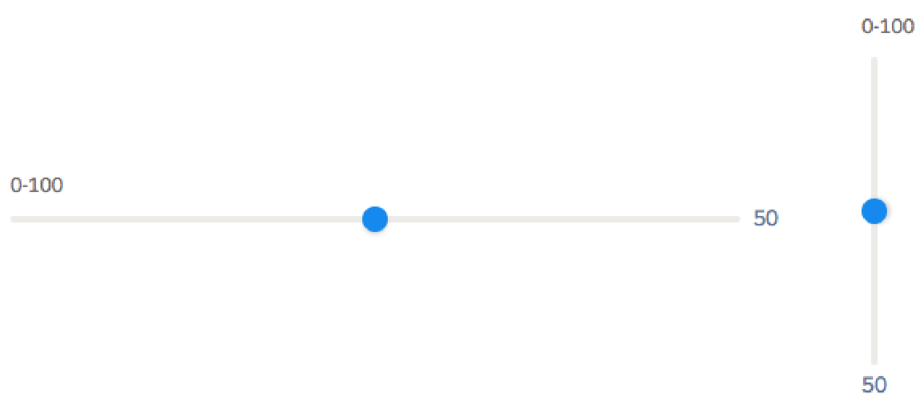
選択

`lightning:select` コンポーネントは、オプションを 1 つ選択できるドロップダウンリストです。`lightning:select` は、Lightning Design System の選択のスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	`slds-select`	値の単一選択をサポートする選択入力要素
label-hidden	`slds-form_inline`	表示ラベルが非表示の選択入力要素

スライダ

`lightning:slider` コンポーネントは、2 つの指定された数値の間の値を指定するスライダです。このコンポーネントでは `variant` 属性はサポートされません。`type` 属性は、横方向 (デフォルト) または縦方向のスライダを表示するかどうかを指定します。`lightning:slider` は、Lightning Design System のスライダのスタイル設定を使用します。



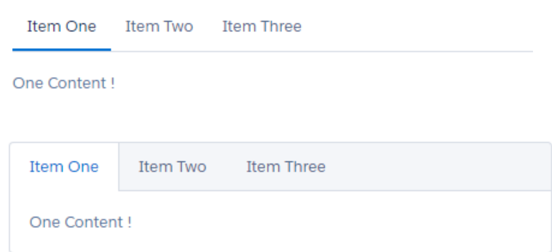
スピナー

lightning:spinner コンポーネントは、データを読み込み中であることを示すスピナーです。スピナーはさまざまなサイズで作成できます。lightning:spinner は、Lightning Design System のスピナーのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
base (デフォルト)	slds-spinner	灰色のスピナー
brand	slds-spinner_brand	青いスピナー
inverse	slds-spinner_inverse	暗色の背景用の白いスピナー

タブ

lightning:tabset コンポーネントは、lightning:tab コンポーネントで表される、対応するコンテンツ領域があるタブのリストです。lightning:tabset は、Lightning Design System のタブのスタイル設定を使用します。



基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
default	slds-tabs_default	境界線なしのタブとコンテンツ領域のリスト

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
scoped	`slds-tabs_scoped`	境界線ありのタブとコンテンツ領域のリスト
vertical	`slds-vertical-tabs`	コンテンツ領域の左側に縦方向に表示されるタブのリスト

テキストエリア

`lightning:textarea` コンポーネントは、複数行のテキストを入力するための入力項目です。Lightning Design System のテキストエリアのスタイル設定を使用します。

基本コンポーネントのバリエーション	Lightning Design System のクラス名	説明
standard (デフォルト)	`slds-form-element`	テキスト表示ラベルがある textarea 要素
label-hidden	`slds-form_inline`	表示ラベルが非表示の textarea 要素

タイル

`lightning:tile` コンポーネントは、関連情報のグループです。このコンポーネントではバリエーションはサポートされませんが、`slds-tile_board` クラスを渡してボードを作成できます。同様に、タイルボディで定義リストを使用してアイコンを表示するタイルを作成したり、順序なしリストを使用してアバターを表示するタイルのリストを作成したりできます。`lightning:tile` は、Lightning Design System のタイルのスタイル設定を使用します。

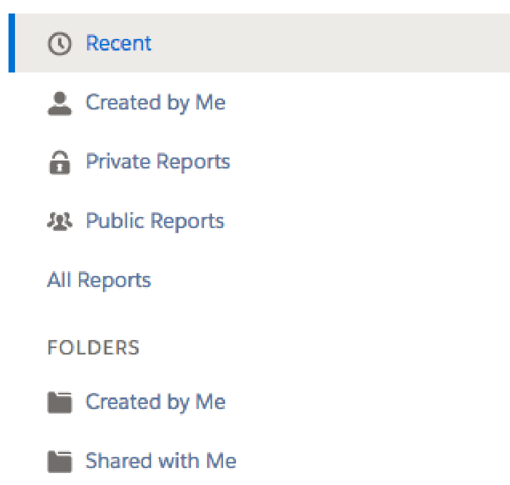
ツリー

`lightning:tree` コンポーネントは、階層構造を視覚化するものです。ツリー項目 (ブランチ) は、展開したり折りたたんだりできます。このコンポーネントでは `variant` 属性はサポートされませんが、Lightning Design System のツリーのスタイル設定を使用します。

ROLES
- Western Sales Director
  - Western Sales Manager
    - CA Sales Rep
    - OR Sales Rep
- Eastern Sales Director

垂直ナビゲーション

`lightning:verticalNavigation` コンポーネントは、単一のレベルのリンクのリストであり、アイコンと、折りたたんだり展開したりするオーバーフローセクションがサポートされます。このコンポーネントでは `variant` 属性はサポートされませんが、`compact` と `shaded` 属性を使用して、コンパクトスペース設定および網掛け背景のスタイル設定に対応できます。`lightning:verticalNavigation` は、Lightning Design System の垂直ナビゲーションのスタイル設定を使用します。



UI コンポーネントの操作

フレームワークには、共通のユーザーインターフェースコンポーネントが `ui` 名前空間に備えられています。これらのすべてのコンポーネントは、`aura:component` または `aura:component` の子コンポーネントのいずれかを拡張します。`aura:component` は、デフォルトの表示を行う抽象コンポーネントです。`ui:input` や `ui:output` などのユーザーインターフェースコンポーネントは、キーボード操作やマウス操作などの共通のユーザーインターフェースイベントを処理しやすくします。各コンポーネントは適宜スタイルを設定したり拡張したりできます。

メモ: Lightning Design System スタイル設定を適用するコンポーネントを探す場合、代わりに Lightning 基本コンポーネントを使用することを検討してください。

すべての使用できるコンポーネントについては、
`https://<myDomain>.lightning.force.com/auradocs/reference.app` (`<myDomain>` は、Salesforce カスタムドメインの名前) のコンポーネント参照を参照してください。

複雑なインタラクティブコンポーネント

次のコンポーネントは、1 つ以上のサブコンポーネントがあり、インタラクティブです。

型	主要コンポーネント	説明
メッセージ	ui:message	さまざまな重要度のメッセージ通知
メニュー	ui:menu	表示を制御するトリガを含むドロップダウンリスト
	ui:menuList	メニュー項目のリスト
	ui:actionMenuItem	アクションをトリガするメニュー項目
	ui:checkboxMenuItem	複数選択をサポートしてアクションをトリガできるメニュー項目
	ui:radioMenuItem	単一選択をサポートしてアクションをトリガできるメニュー項目
	ui:menuItemSeparator	メニュー項目の視覚的な分離線
	ui:menuItem	ui:menuList コンポーネントに含まれるメニュー項目の抽象かつ拡張可能なコンポーネント
	ui:menuTrigger	メニューを展開したり折りたたんだりするトリガ
	ui:menuTriggerLink	ドロップダウンメニューをトリガするリンク。このコンポーネントは ui:menuTrigger を拡張します

入力コントロールコンポーネント

次のコンポーネントは、たとえばボタンやチェックボックスなどがあり、インタラクティブです。

型	主要コンポーネント	説明
ボタン	ui:button	押したりクリックしたりできるアクションの実行が可能なボタン
チェックボックス	ui:inputCheckbox	複数選択をサポートする選択可能なオプション
	ui:outputCheckbox	参照のみのチェックボックスの値を表示します
ラジオボタン	ui:inputRadio	単一選択のみをサポートする選択可能なオプション
ドロップダウンリスト	ui:inputSelect	オプションを含むドロップダウンリスト
	ui:inputSelectOption	ui:inputSelect コンポーネントのオプション

ビジュアルコンポーネント

次のコンポーネントは、たとえばエラーメッセージや読み込みスピナーなどの情報キューを提供します。

型	主要コンポーネント	説明
項目レベルのエラー	ui:inputDefaultError	エラーが発生したときに表示されるエラーメッセージ
スピナー	ui:spinner	読み込みスピナー

項目コンポーネント

次のコンポーネントでは、値を入力または表示できます。

型	主要コンポーネント	説明
通貨	ui:inputCurrency	通貨を入力するための入力項目
	ui:outputCurrency	デフォルトまたは指定された形式で、通貨を表示します
メール	ui:inputEmail	メールアドレスを入力するための入力項目
	ui:outputEmail	クリック可能なメールアドレスを表示します
日時	ui:inputDate	日付を入力するための入力項目
	ui:inputDateTime	日時を入力するための入力項目
	ui:outputDate	デフォルトまたは指定された形式で、日付を表示します
	ui:outputDateTime	デフォルトまたは指定された形式で、日時を表示します
パスワード	ui:inputSecret	秘密のテキストを入力するための入力項目
電話番号	ui:inputPhone	電話番号を入力するための入力項目
	ui:outputPhone	電話番号を表示します
数値	ui:inputNumber	数値を入力するための入力項目
	ui:outputNumber	数値を表示します
範囲	ui:inputRange	範囲内の値を入力するための入力項目
リッチテキスト	ui:inputRichText	リッチテキストを入力するための入力項目
	ui:outputRichText	リッチテキストを表示します
テキスト	ui:inputText	1 行のテキストを入力するための入力項目
	ui:outputText	テキストを表示します
テキストエリア	ui:inputTextArea	複数行のテキストを入力するための入力項目

型	主要コンポーネント	説明
	ui:outputTextArea	参照のみのテキストエリアを表示します
URL	ui:inputURL	URL を入力するための入力項目
	ui:outputURL	クリック可能な URL を表示します

関連トピック:

UI コンポーネントの使用

コンポーネントの作成

コンポーネントのバンドル

UI コンポーネントのイベント処理

UIコンポーネントは、キーボード操作やマウス操作などのユーザインターフェースイベントを処理しやすくします。これらのイベントをリスンすると、updateon 属性を使用して UI 入力コンポーネントの値をバインドし、これらのイベントの起動時に値を更新することもできます。

コンポーネントにハンドラを定義して、UI イベントを取得します。たとえば、ui:inputTextArea コンポーネントで onblur という HTML DOM イベントをリスンすることができます。

```
<ui:inputTextArea aura:id="textarea" value="My text area" label="Type something"
      blur="{!c.handleBlur}" />
```

blur="{!c.handleBlur}" は、onblur イベントをリスンして、クライアント側コントローラに結び付けます。このイベントをトリガすると、次のクライアント側コントローラがイベントを処理します。

```
handleBlur : function(cmp, event, helper){
    var elem = cmp.find("textarea").getElement();
    //do something else
}
```

すべてのコンポーネントで使用可能な全イベントについては、「コンポーネントの参照」(ページ 437)を参照してください。

ブラウザイベントの値のバインド

UI の変更はコンポーネント属性に反映され、その属性の変更は UI に伝搬されます。コンポーネントを読み込むと、入力要素の値がコンポーネント属性の値に初期化されます。ユーザ入力が変更されると、コンポーネント変数の値が更新されます。たとえば、コンポーネント属性にバインドされている値を ui:inputText コンポーネントに含めて、同じコンポーネント属性に ui:outputText コンポーネントをバインドできます。ui:inputText コンポーネントは、onkeyup ブラウザイベントをリスンし、対応するコンポーネント属性値を更新します。

```
<aura:attribute name="first" type="String" default="John"/>
<aura:attribute name="last" type="String" default="Doe"/>

<ui:inputText label="First Name" value="{!v.first}" updateOn="keyup"/>
```

```
<ui:inputText label="Last Name" value="{!v.last}" updateOn="keyup"/>

<!-- Returns "John Doe" -->
<ui:outputText value="{!v.first +' '+ v.last}"/>
```

次の例では、数値の入力が取り込まれ、それらの数値の合計が返されます。ui:inputNumber コンポーネントは、onkeyup ブラウザイベントをリスンします。keyup イベントでこのコンポーネントの値が変更されると、ui:outputNumber コンポーネントの値も更新され、2 つの値の合計が返されます。

```
<aura:attribute name="number1" type="integer" default="1"/>
<aura:attribute name="number2" type="integer" default="2"/>

<ui:inputNumber label="Number 1" value="{!v.number1}" updateOn="keyup" />
<ui:inputNumber label="Number 2" value="{!v.number2}"  updateOn="keyup" />

<!-- Adds the numbers and returns the sum -->
<ui:outputNumber  value="{!(v.number1 * 1) + (v.number2 * 1)}"/>
```

メモ: 入力項目では文字列値が返されるため、適切に処理して数値に対応する必要があります。この例では、両方の値を 1 で乗算して、対応する数値を取得します。

UI コンポーネントの使用

ユーザは、値を選択または入力するために入力要素を使用してアプリケーションとやりとりします。ui:inputText や ui:inputCheckbox などのコンポーネントは、共通の入力要素に対応します。これらのコンポーネントは、ユーザインターフェースイベントのイベント処理を簡略化します。

メモ: すべての使用できるコンポーネントの属性とイベントについては、https://<myDomain>.lightning.force.com/auradocs/reference.app (<myDomain> は、Salesforce カスタムドメインの名前) のコンポーネント参照を参照してください。

独自のカスタムコンポーネントで入力コンポーネントを使用するには、.cmp または .app リソースに入力コンポーネントを追加します。次の例は、テキスト項目およびボタンの基本設定です。aura:id 属性は、cmp.find("myID"); を使用して JavaScript コードからコンポーネントを参照できるようにする一意の ID を定義します。

```
<ui:inputText label="Name" aura:id="name" placeholder="First, Last"/>
<ui:outputText aura:id="nameOutput" value=""/>
<ui:button aura:id="outputButton" label="Submit" press="{!c.getInput}"/>
```

メモ: すべてのテキスト項目に、項目のテキスト表示ラベルを表す label 属性が指定されている必要があります。表示ラベルをビューで非表示にする必要がある場合は、labelClass="assistiveText" を設定して表示ラベルを支援技術で使用できるようにします。

ui:outputText コンポーネントは、対応する ui:inputText コンポーネントの出力値のプレースホルダとして機能します。ui:outputText コンポーネントの値は、クライアント側の次のコントローラアクションを使用して設定できます。

```
getInput : function(cmp, event) {
      var fullName = cmp.find("name").get("v.value");
      var outName = cmp.find("nameOutput");
```

```
        outName.set("v.value", fullName);
    }
```

次の例は前の例と似ていますが、クライアント側コントローラなしで値のバインドを使用します。`ui:outputText` コンポーネントには、`onkeyup` ブラウザイベントが起動された時の `ui:inputText` コンポーネントの最新の値が反映されます。

```
<aura:attribute name="first" type="String" default="John"/>
<aura:attribute name="last" type="String" default="Doe"/>

<ui:inputText label="First Name" value="{!v.first}" updateOn="keyup"/>
<ui:inputText label="Last Name" value="{!v.last}" updateOn="keyup"/>

<!-- Returns "John Doe" -->
<ui:outputText value="{!v.first +' '+ v.last}"/>
```

ヒント: Salesforce1 でレコードを作成および編集するには、`force:createRecord` および `force:recordEdit` イベントを使用して、組み込みのレコードの作成ページおよびレコードの編集ページを使用します。

フロー Lightning コンポーネントの操作

Lightning コンポーネントにフローを埋め込んだら、JavaScript と Apex コードを使用して、実行時のフローを設定します。たとえば、値をフローに渡したり、フローの完了時の動作を制御したりします。`lightning:flow` では、自動起動フロー種別のフローのみがサポートされています。

フローは、Visual Workflow で構築された、Salesforce 情報を収集、更新、編集、作成するアプリケーションです。

フローを Lightning コンポーネントに埋め込むには、<lightning:flow> コンポーネントを追加します。

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" />
</aura:component>
```

```
({
    init : function (component) {
        // Find the component whose aura:id is "flowData"
        var flow = component.find("flowData");
        // In that component, start your flow. Reference the flow's Unique Name.
        flow.startFlow("myFlow");
    },
})
```

メモ: フローコンポーネントが含まれるページ (Lightning アプリケーションビルダーや有効な Lightning ページなど) をユーザが開くと、ページが読み込まれたときにフローが実行されます。フローの最初の画面が表示されるより前に、フローがレコードの作成や削除などを行わないことを確認してください。

このセクションの内容:

Lightning コンポーネントからのフロー変数値の設定

Lightning コンポーネントにフローを埋め込んだら、その変数、sObject 変数、コレクション変数、sObject コレクション変数を初期化して、フローにより多くのコンテキストを提供します。コンポーネントのコントローラで、対応付けのリストを作成し、そのリストを startFlow メソッドに渡します。

Lightning コンポーネントへのフロー変数値の取得

フロー変数値は、Lightning コンポーネントで表示または参照できます。フローをカスタム Lightning コンポーネントに埋め込んだら、`onstatuschange` アクションを使用して、フローの出力変数から値を取得します。出力変数は配列として返されます。

Lightning コンポーネントでのフローの完了動作の制御

デフォルトでは、フローユーザが [完了] をクリックすると、コンポーネントで新しいインタビューが開始されて、ユーザにフローの最初の画面が再度表示されます。ただし、`onstatuschange` アクションを使用して、フローの完了時の動作を指定できます。別のページにリダイレクトするには、いずれかの `force:navigateTo*` イベント (`force:navigateToObjectHome` や `force:navigateToUrl` など) を使用します。

Lightning コンポーネントからのフローインタビューの再開

デフォルトでは、ユーザは Salesforce Classic のホームページの [一時停止中のインタビュー] コンポーネントから一時停止中のインタビューを再開できます。ユーザがインタビューを再開できる方法と場所をカスタマイズするには、JavaScript コントローラでインタビュー ID を `resumeFlow` メソッドに渡します。

Lightning コンポーネントからのフロー変数値の設定

Lightning コンポーネントにフローを埋め込んだら、その変数、sObject 変数、コレクション変数、sObject コレクション変数を初期化して、フローにより多くのコンテキストを提供します。コンポーネントのコントローラで、対応付けのリストを作成し、そのリストを startFlow メソッドに渡します。

メモ: 変数を設定できるのはインタビューの開始時のみで、設定した変数で入力アクセスが許可されている必要があります。それぞれのフロー変数では、入力アクセスは次の項目によって制御されます。

- Cloud Flow Designer の [入力/出力種別] 変数項目
- Metadata API の `FlowVariable` の `isInput` 項目

入力アクセスを許可しない変数を参照している場合、その変数を設定しようとしても無視されます。

設定した各変数の `name`、`type`、`value` を指定します。type には、フローデータ型を使用します。

フロー変数の型	型	有効な値
テキスト	String	文字列値または同等の式
数値	Number	数値または同等の式
通貨	Currency	数値または同等の式
ブール	Boolean	• True 値: *true*、*1*、または同等の式 • False 値: *false*、*0*、または同等の式

フロー変数の型	型	有効な値
選択リスト	Picklist	文字列値または同等の式
複数選択リスト	Multipicklist	文字列値または同等の式
日付	Date	*"YYYY-MM-DD"* または同等の式
日付/時間	DateTime	*"YYYY-MM-DDThh:mm:ssZ"* または同等の式
sObject	sObject	キー - 値ペアの対応付けまたは同等の式

```
{
     name : "varName",
     type : "flowDataType",
     value : valueToSet
},
{
     name : "varName",
     type : "flowDataType",
     value : [ value1, value2]
}, ...
```

例: この JavaScript コントローラでは、数値変数、日付コレクション変数、および sObject 変数のペアの値を設定します。

```
({
   init : function (component) {
      // Find the component whose aura:id is "flowData"
      var flow = component.find("flowData");
      var inputVariables = [
         { name : "numVar", type : "Number", value: 30 },
         { name : "dateColl", type : "String", value: [ "2016-10-27", "2017-08-01" ]
},
         // Sets values for fields in the account sObject variable. Id uses the
         // value of the component's accountId attribute. Rating uses a string.
         { name : "account", type : "SObject", value: {
             "Id" : component.get("v.accountId"),
             "Rating" : "Warm"
             }
          },
          // Set the contact sObject variable to the value of the component's contact

          // attribute. We're assuming the attribute contains the entire sObject for
          // a contact record.
          { name : "contact", type : "SObject", value: component.get("v.contact") }
          },
       ];
       flow.startFlow("myFlow", inputVariables);
   }
})
```

例: 次に、Apex コントローラを介して取引先を取得するコンポーネントの例を示します。Apex コントローラは、JavaScript コントローラを介してデータをフローの sObject 変数に渡します。

```
<aura:component controller="AccountController" >
   <aura:attribute name="account" type="Account" />
   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <lightning:flow aura:id="flowData"/>
</aura:component>
```

```
public with sharing class AccountController {
   @AuraEnabled
   public static Account getAccount() {
      return [SELECT Id, Name, LastModifiedDate FROM Account LIMIT 1];
   }
}
```

```
({
   init : function (component) {
      // Create action to find an account
      var action = component.get("c.getAccount");

      // Add callback behavior for when response is received
      action.setCallback(this, function(response) {
         if (state === "SUCCESS") {
            // Pass the account data into the component's account attribute
            component.set("v.account", response.getReturnValue());
            // Find the component whose aura:id is "flowData"
            var flow = component.find("flowData");
            // Set the account sObject variable to the value of the component's
            // account attribute.
            var inputVariables = [
               {
                  name : "account",
                  type : "SObject",
                  value: component.get("v.account")
               }
            ];

            // In the component whose aura:id is "flowData, start your flow
            // and initialize the account sObject variable. Reference the flow's
            // Unique Name.
            flow.startFlow("myFlow", inputVariables);
         }
         else {
            console.log("Failed to get account date.");
         }
      });

      // Send action to be executed
      $A.enqueueAction(action);
   }
})
```

Lightning コンポーネントへのフロー変数値の取得

フロー変数値は、Lightning コンポーネントで表示または参照できます。フローをカスタム Lightning コンポーネントに埋め込んだら、onstatuschange アクションを使用して、フローの出力変数から値を取得します。出力変数は配列として返されます。

メモ: 変数で出力アクセスが許可されている必要があります。それぞれのフロー変数では、出力アクセスは次の項目によって制御されます。

- Cloud Flow Designer の [入力/出力種別] 変数項目
- Metadata API の FlowVariable の isInput 項目

出力アクセスを許可しない変数を参照している場合、その変数を取得しようとしても無視されます。

例: 次の例では、JavaScript コントローラを使用して、フローの accountName および numberOfEmployees 変数をコンポーネントの属性に渡します。次に、コンポーネントにより、これらの値が出力コンポーネントに表示されます。

```
<aura:component>
   <aura:attribute name="accountName" type="String" />
   <aura:attribute name="numberOfEmployees" type="Decimal" />

   <p><lightning:formattedText value="{!v.accountName}" /></p>
   <p><lightning:formattedNumber style="decimal" value="{!v.numberOfEmployees}" /></p>


   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <lightning:flow aura:id="flowData" onstatuschange="{!c.handleStatusChange}" />
</aura:component>
```

```
({
   init : function (component) {
      // Find the component whose aura:id is "flowData"
      var flow = component.find("flowData");
      // In that component, start your flow. Reference the flow's Unique Name.
      flow.startFlow("myFlow");
   },

   handleStatusChange : function (component, event) {
      if(event.getParam("status") === "FINISHED") {
         // Get the output variables and iterate over them
         var outputVariables = event.getParam("outputVariables");
         var outputVar;
         for(var i = 0; i < outputVariables.length; i++) {
            outputVar = outputVariables[i];
            // Pass the values to the component's attributes
            if(outputVar.name === "accountName") {
               component.set("v.accountName", outputVar.value);
            } else {
               component.set("v.numberOfEmployees", outputVar.value);
            }
         }
      }
```

```
    },
})
```

Lightning コンポーネントでのフローの完了動作の制御

デフォルトでは、フローユーザが[完了]をクリックすると、コンポーネントで新しいインタビューが開始されて、ユーザにフローの最初の画面が再度表示されます。ただし、onstatuschange アクションを使用して、フローの完了時の動作を指定できます。別のページにリダイレクトするには、いずれかの force:navigateTo* イベント (force:navigateToObjectHome や force:navigateToUrl など) を使用します。

メモ: 自動起動フローの終了時の動作を制御するには、FINISHED_SCREEN の状況をチェックします。

```
<aura:component access="global">
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" onstatuschange="{!c.handleStatusChange}" />
</aura:component>
```

```
// init function here
handleStatusChange : function (component, event) {
   if(event.getParam("status") === "FINISHED") {
        // Redirect to another page in Salesforce, or
        // Redirect to a page outside of Salesforce, or
        // Show a toast, or...
    }
}
```

例: 次の関数では、force:navigateToSObject イベントを使用して、フローで作成されたケースにユーザをリダイレクトします。

```
handleStatusChange : function (component, event) {
   if(event.getParam("status") === "FINISHED") {
      var outputVariables = event.getParam("outputVariables");
      var outputVar;
      for(var i = 0; i < outputVariables.length; i++) {
         outputVar = outputVariables[i];
         if(outputVar.name === "redirect") {
            var urlEvent = $A.get("e.force:navigateToSObject");
            urlEvent.setParams({
               "recordId": outputVar.value,
               "isredirect": "true"
            });
            urlEvent.fire();
         }
      }
   }
}
```

Lightning コンポーネントからのフローインタビューの再開

デフォルトでは、ユーザは Salesforce Classic のホームページの [一時停止中のインタビュー] コンポーネントから一時停止中のインタビューを再開できます。ユーザがインタビューを再開できる方法と場所をカスタマイズするには、JavaScript コントローラでインタビュー ID を `resumeFlow` メソッドに渡します。

```
({
    init : function (component) {
        // Find the component whose aura:id is "flowData"
        var flow = component.find("flowData");

        // In that component, resume a paused interview. Provide the method with
        // the ID of the interview that you want to resume.
        flow.resumeFlow("pausedInterviewId");
    },
})
```

例: 次の例では、インタビューを再開したり新しいインタビューを開始したりする方法を示します。ユーザが取引先責任者レコードから **[Survey Customer (顧客調査)]** をクリックすると、Lightning コンポーネントは次の 2 つのいずれかを実行します。

- [Survey Customers (顧客調査)] フローのインタビューが一時停止されている場合、最初のインタビューを再開する。
- [Survey Customers (顧客調査)] フローのインタビューが一時停止されていない場合、新しいインタビューを開始する。

```
<aura:component controller="InterviewsController">
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" />
</aura:component>
```

この Apex コントローラは、SOQL クエリを実行して一時停止中のインタビューのリストを取得します。クエリから何も返されない場合、`getPausedId()` は null 値を返し、コンポーネント新しいインタビューが開始されます。クエリから 1 つ以上のインタビューが返されると、コンポーネントでそのリスト内の最初のインタビューが再開されます。

```
public class InterviewsController {
   @AuraEnabled
   public static String getPausedId() {
      // Get the ID of the running user
      String currentUser = UserInfo.getUserId();
      // Find all of that user's paused interviews for the Survey customers flow
      List<FlowInterview> interviews =
         [ SELECT Id FROM FlowInterview
           WHERE CreatedById = :currentUser AND InterviewLabel LIKE '%Survey
customers%'];

      if (interviews == null || interviews.isEmpty()) {
         return null; // early out
      }
      // Return the ID for the first interview in the list
      return interviews.get(0).Id;
```

```
    }
}
```

JavaScript コントローラが Apex コントローラからインタビュー ID を取得すると、コンポーネントでそのインタビューが再開されます。Apex コントローラから null のインタビュー ID が返されると、コンポーネントで新しいインタビューが開始されます。

```
({
   init : function (component) {
       //Create request for interview ID
       var action = component.get("c.getPausedId");
       action.setCallback(this, function(response) {
          var interviewId = response.getReturnValue();
          // Find the component whose aura:id is "flowData"
          var flow = component.find("flowData");
          // If an interview ID was returned, resume it in the component
          // whose aura:id is "flowData".
          if ( interviewId !== null ) {
             flow.resumeFlow(interviewID);
          }
          // Otherwise, start a new interview in that component. Reference
          // the flow's Unique Name.
          else {
             flow.startFlow("Survey_customers");
          }
       });
       //Send request to be enqueued
       $A.enqueueAction(action);
   },
})
```

アクセシビリティのサポート

コンポーネントをカスタマイズする場合、アクセシビリティ (`aria` 属性など) を確保するコードが保持されるように注意してください。

アクセシビリティに対応したソフトウェアと支援技術によって、開発したアプリケーションを障害のあるユーザが使用および操作できます。Auraコンポーネントは W3C仕様に従って作成されるため、共通の支援技術で動作します。Lightningコンポーネントフレームワークを使用して開発する場合、アクセシビリティについては、WCAG ガイドラインに常に従うことをお勧めしますが、このガイドでは `ui` 名前空間でコンポーネントを使用する場合に活用できるアクセシビリティ機能について説明しています。

このセクションの内容:

ボタンの表示ラベル

音声メッセージ

フォーム、項目、および表示ラベル

イベント

メニュー

ボタンの表示ラベル

ボタンには、テキストのみ、アイコンとテキスト、またはテキストなしのアイコンを表示することができます。アクセシビリティに対応したボタンを作成するには、`lightning:button` を使用し、`label` 属性を使用してテキスト表示ラベルを設定します。詳細は、「lightning:button」を参照してください。

メモ: アクセシビリティに対応したボタンを作成するには `ui:button` を使用することもできますが、これは Lightning Design System スタイル設定に含まれていません。代わりに `lightning:button` を使用することをお勧めします。

テキストのみのボタン:

```
<lightning:button label="Search" onclick="{!c.doSomething}"/>
```

アイコンとテキストのボタン:

```
<lightning:button label="Download" iconName="utility:download" onclick="{!c.doSomething}"/>
```

アイコンのみのボタン:

```
<lightning:buttonIcon iconName="utility:settings" alternativeText="Settings"
onclick="{!c.doSomething}"/>
```

`alternativeText` 属性は、ビューには表示されず、支援技術に使用される表示ラベルを設定します。

`lightning:button` によって生成される HTML の例を次に示します。

```
<!-- Good: using span/assistiveText to hide the label visually, but show it to screen
readers -->
<button>
    ::before
    <span class="slds-assistive-text">Settings</span>
</button>
```

音声メッセージ

音声通知を送信するには、デフォルトで `role="alert"` が設定されている `ui:message` コンポーネントを使用します。`"alert"` `aria` ロールでは、div 内のテキストが取得され、ユーザが他の操作を行わなくてもテキストが読み上げられます。

```
<ui:message title="Error" severity="error" closable="true">
      This is an error message.
</ui:message>
```

フォーム、項目、および表示ラベル

入力コンポーネントは、フォーム項目に表示ラベルを割り当てやすいように設計されています。表示ラベルによって、フォーム項目とそのテキスト表示ラベルをプログラムで関連付けることができます。入力コンポーネントでプレースホルダを使用する場合は、アクセシビリティを考慮して `label` 属性を設定します。

アクセシビリティに対応した入力項目とフォームを作成するには、`lightning:input` を使用します。複数行のテキスト入力の場合は `<textarea>` タグではなく `lightning:textarea` を使用し、`<select>` タグではなく `lightning:select` を使用します。

```
<lightning:input name="myInput" label="Search" />
```

コードの実行に失敗した場合は、コンポーネントの表示中に表示ラベル要素を確認してください。表示ラベル要素に `for` 属性が存在し入力コントロール ID 属性の値と一致するか、入力が表示ラベルで囲まれている必要があります。入力コントロールには、`<input>`、`<textarea>`、および `<select>` があります。

`lightning:input` によって生成される HTML の例を次に示します。

```
<!-- Good: using label/for= -->
<label for="fullname">Enter your full name:</label>
<input type="text" id="fullname" />

<!-- Good: --using implicit label>
<label>Enter your full name:
    <input type="text" id="fullname"/>
</label>
```

関連トピック:

表示ラベルの使用

イベント

`onclick` イベントはどの種類の要素にも添付できますが、アクセシビリティに対応するには、デフォルトで HTML でのアクションの実行が可能な要素 (コンポーネントマークアップの `<a>`、`<button>`、`<input>` タグなど) にのみこのイベントを適用するように考慮してください。`<div>` タグで `onclick` イベントを使用して、クリックのイベントバブルを回避できます。

メニュー

メニューは、表示を制御するトリガを含むドロップダウンリストです。表示ラベルを表示するトリガとメニュー項目のリストを指定する必要があります。ドロップダウンメニューとそのメニュー項目は、デフォルトでは非表示になっています。この設定を変更するには、`ui:menuList` コンポーネントの `visible` 属性を `true` に設定します。メニュー項目は、`ui:menuTriggerLink` コンポーネントをクリックしたときにのみ表示されます。

次のコード例では、複数の項目が含まれるメニューを作成します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
              <ui:actionMenuItem aura:id="item2" label="Open"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item3" label="Closed"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item4" label="Closed Won"
click="{!c.updateTriggerLabel}"/>
```

```
        </ui:menuList>
</ui:menu>
```

メニューごとにその目的は異なります。目的の動作に対して正しいメニューを使用してください。メニューには次の 3 種類があります。

アクション
: 印刷、新規作成、保存などのアクションを作成する項目の `ui:actionMenuItem` を使用します。

ラジオボタン
: 複数の項目のリストから 1 つのみを選択する場合は、`ui:radioMenuItem` を使用します。

チェックボックススタイル
: 複数の項目のリストから複数の項目を選択できる場合は、`ui:checkboxMenuItem` を使用します。チェックボックスは、1 つの項目のオン/オフを切り替える場合にも使用できます。

メモ: ボタンであるトリガを使用したドロップダウンメニューを作成するには、`lightning:buttonMenu` を使用します。

第 4 章　コンポーネントの使用

トピック:

- Lightning Experience および Salesforce1 での Lightning コンポーネントの使用
- Lightning コンポーネントを Lightning ページで使用できるようにするための準備
- コミュニティビルダーでの Lightning コンポーネントの使用
- アプリケーションへのコンポーネントの追加
- Chatter パブリッシャーへのカスタムアプリケーションの統合
- Visualforce ページでの Lightning コンポーネントの使用
- Lightning Out を使用した任意のアプリケーションへの Lightning コンポーネントの追加 (ベータ)

コンポーネントは、多数の異なるコンテキストで使用できます。このセクションでは、その方法について説明します。

Lightning Experience および Salesforce1 での Lightning コンポーネントの使用

Lightning コンポーネントを使用して Lightning Experience や Salesforce1 をカスタマイズおよび拡張します。タブ、アプリケーション、およびアクションからコンポーネントを起動します。

このセクションの内容:

カスタムタブのコンポーネントの設定

`force:appHostable` インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でカスタムタブとして使用できるようにします。

Lightning Experience のカスタムタブとしての Lightning コンポーネントの追加

Lightning コンポーネントをカスタムタブに表示して Lightning Experience ユーザが使用できるようにします。

Salesforce1 のカスタムタブとしての Lightning コンポーネントの追加

Lightning コンポーネントをカスタムタブに表示して Salesforce1 ユーザが使用できるようにします。

Lightning コンポーネントアクション

Lightning コンポーネントアクションは、Lightning コンポーネントを呼び出すカスタムアクションです。Apex と JavaScript がサポートされるため、セキュアな方法でクライアント側のカスタム機能を開発できます。Lightning コンポーネントアクションは Salesforce1 および Lightning Experience でのみサポートされます。

Lightning コンポーネントでの標準アクションの上書き

`lightning:actionOverride` インターフェースを Lightning コンポーネントに追加して、コンポーネントを使用してオブジェクトに対する標準アクションを上書きできるようにします。ほとんどの標準コンポーネントおよびすべてのカスタムコンポーネントで、表示、新規、編集、およびタブ標準アクションを上書きできます。標準アクションを上書きすると、Lightning コンポーネントを使用して組織をカスタマイズできます。たとえば、レコードの参照、作成、編集方法を完全にカスタマイズできます。

カスタムタブのコンポーネントの設定

`force:appHostable` インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でカスタムタブとして使用できるようにします。

このインターフェースを実装するコンポーネントは、Lightning Experience と Salesforce1 の両方でタブを作成するために使用できます。

例: コンポーネントの例

```
<!--simpleTab.cmp-->
<aura:component implements="force:appHostable">

    <!-- Simple tab content -->

    <h1>Lightning Component Tab</h1>

</aura:component>
```

appHostable インターフェースにより、コンポーネントがカスタムタブとして使用可能になります。コンポーネントに他の要素を追加する必要はありません。

Lightning Experience のカスタムタブとしての Lightning コンポーネントの追加

Lightning コンポーネントをカスタムタブに表示して Lightning Experience ユーザが使用できるようにします。

Lightning Experience に含めるコンポーネントで、aura:component タグに implements="force:appHostable" を追加して変更を保存します。

エディション

使用可能なエディション: Salesforce Classic および Lightning Experience

使用可能なエディション: **Contact Manager** Edition、**Group** Edition、**Professional** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、および **Developer** Edition

UI を使用して Lightning コンポーネントを作成するエディション: **Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、**Developer** Edition、または Sandbox。

ユーザ権限

Lightning コンポーネントタブを作成する
- 「アプリケーションのカスタマイズ」

```
<aura:component implements="force:appHostable">
```

Lightning コンポーネントを作成するには、開発者コンソールを使用します。

次の手順に従って、コンポーネントを Lightning Experience に含め、組織でユーザが使用できるようにします。

1. このコンポーネントのカスタムタブを作成します。
   a. [設定] から、[クイック検索] ボックスに「タブ」と入力し、[タブ] を選択します。
   b. [Lightning コンポーネントタブ] 関連リストで [新規] をクリックします。
   c. ユーザが使用できるようにする Lightning コンポーネントを選択します。

d. タブに表示する表示ラベルを入力します。
e. タブのスタイルを選択し、[次へ] をクリックします。
f. プロファイルへのタブの追加を指示するメッセージが表示されたら、デフォルトを受け入れて[保存]をクリックします。

2. Lightning コンポーネントをアプリケーションランチャーに追加します。
   a. [設定]から、[クイック検索] ボックスに「アプリケーション」と入力し、[アプリケーション]を選択します。
   b. [新規] をクリックします。カスタムアプリケーション を選択し、[次へ] をクリックします。
   c. [アプリケーションの表示ラベル] に「`Lightning`」と入力し、[次へ] をクリックします。
   d. [選択可能なタブ] ドロップダウンメニューで、作成したLightningコンポーネントタブを選択し、右矢印ボタンをクリックしてカスタムアプリケーションに追加します。
   e. [次へ]をクリックします。[参照可能] チェックボックスをオンにしてアプリケーションをプロファイルに割り当て、[保存] をクリックします。

3. Lightning Experience でアプリケーションランチャーに移動して、出力を確認します。アプリケーションランチャーにカスタムアプリケーションが表示されます。カスタムアプリケーションをクリックすると、追加したコンポーネントが表示されます。

Salesforce1 のカスタムタブとしての Lightning コンポーネントの追加

Lightning コンポーネントをカスタムタブに表示して Salesforce1 ユーザが使用できるようにします。

追加するコンポーネントで、`aura:component` タグに `implements="force:appHostable"` を追加して変更を保存します。

エディション

使用可能なエディション: Salesforce Classic および Lightning Experience

使用可能なエディション: **Contact Manager** Edition、**Group** Edition、**Professional** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、および **Developer** Edition

UI を使用して Lightning コンポーネントを作成するエディション: **Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、**Developer** Edition、または Sandbox。

ユーザ権限

Lightning コンポーネントタブを作成する

- 「アプリケーションのカスタマイズ」

```
<aura:component implements="force:appHostable">
```

`appHostable` インターフェースにより、コンポーネントがカスタムタブとして使用可能になります。

Lightning コンポーネントを作成するには、開発者コンソールを使用します。

Salesforce1 ナビゲーションメニューにコンポーネントを追加するには、次の手順に従います。

1. コンポーネント用のカスタム Lightning コンポーネントタブを作成します。[設定] から、[クイック検索] ボックスに「タブ」と入力し、[タブ] を選択します。

   メモ: コンポーネントを Salesforce1 ナビゲーションメニューに追加するには、事前にカスタム Lightning コンポーネントタブを作成する必要があります。Salesforce フルサイトから Lightning コンポーネントへのアクセスはサポートされていません。

2. Salesforce1 ナビゲーションメニューに Lightning コンポーネントを追加します。
   a. [設定] から、[クイック検索] ボックスに「ナビゲーション」と入力し、**[Salesforce1 ナビゲーション]** を選択します。

b. 作成したカスタムタブを選択し、[追加] をクリックします。

c. 項目を選択し、[上へ] または [下へ] をクリックして並び替えます。

ナビゲーションメニューに、指定した順序で項目が表示されます。[選択済み] リストの最初の項目が、ユーザの Salesforce1 のランディングページに表示されます。

3. Salesforce1モバイルブラウザアプリケーションを起動して出力を確認します。ナビゲーションメニューに新しいメニュー項目が表示されます。

メモ: デフォルトで、組織のモバイルブラウザアプリケーションは有効になっています。Salesforce1 モバイルブラウザアプリケーションの使用についての詳細は、『*Salesforce1 アプリケーション開発者ガイド*』を参照してください。

Lightning コンポーネントアクション

Lightning コンポーネントアクションは、Lightning コンポーネントを呼び出すカスタムアクションです。Apex と JavaScript がサポートされるため、セキュアな方法でクライアント側のカスタム機能を開発できます。Lightning コンポーネントアクションは Salesforce1 および Lightning Experience でのみサポートされます。

エディション

使用可能なエディション: Salesforce1 と Lightning Experience の両方

使用可能なエディション: **Group** Edition、**Professional** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、**Contact Manager** Edition、および **Developer** Edition

メモ: Lightning コンポーネントアクションが正しく動作するには、組織に [私のドメイン] がリリースされている必要があります。

ページレイアウトエディタを使用して Lightning コンポーネントアクションをオブジェクトのページレイアウトに追加できます。組織に Lightning コンポーネントアクションがある場合は、ページレイアウトエディタのパレットの [Salesforce1 および Lightning アクション] カテゴリで見つけることができます。

Lightning コンポーネントアクションは、単に組織の Lightning コンポーネントをコールすることはできません。コンポーネントが Lightning コンポーネントアクションとして機能するには、その目的に特化して設定し、`force:LightningQuickAction` または `force:LightningQuickActionWithoutHeader` インターフェースを実装する必要があります。

Lightning コンポーネントアクションをパッケージ化する場合は、アクションが呼び出すコンポーネントは `access=global` とマークされている必要があります。

このセクションの内容:

カスタムアクション用のコンポーネントの設定

`force:lightningQuickAction` または `force:lightningQuickActionWithoutHeader` インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でカスタムアクションとして使用できるようにします。これらのインターフェースのいずれかを実装するコンポーネントを Lightning Experience と Salesforce1 の両方でオブジェクト固有のアクションまたはグローバルアクションとして使用できます。

レコード固有のアクションのコンポーネントの設定

force:hasRecordId インターフェースを Lightning コンポーネントに追加すると、現在のレコードの ID をコンポーネントに割り当てることができます。現在のレコード ID は、Lightning Experience または Salesforce1 のオブジェクト固有のカスタムアクションやアクション上書きなどとして、コンポーネントをLightning レコードページで使用する場合に便利です。

カスタムアクション用のコンポーネントの設定

force:lightningQuickAction または force:lightningQuickActionWithoutHeader インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でカスタムアクションとして使用できるようにします。これらのインターフェースのいずれかを実装するコンポーネントを Lightning Experience と Salesforce1 の両方でオブジェクト固有のアクションまたはグローバルアクションとして使用できます。

アクションとして使用する場合、force:lightningQuickAction インターフェースを実装するコンポーネントは [キャンセル] ボタンなどの標準アクションコントロールがあるパネルに表示されます。これらのコンポーネントにも専用のコントロールを表示でき、実装できますが、標準コントロールからのイベント用に設定されている必要があります。

force:lightningQuickActionWithoutHeader インターフェースを実装するコンポーネントは、追加のコントロールなしでパネルに表示され、アクションの完全なユーザインターフェースを提供します。

これらのインターフェースは相互に排他的です。つまり、コンポーネントに force:lightningQuickAction インターフェースまたは force:lightningQuickActionWithoutHeader インターフェースのどちらかを実装できますが、この両方を実装することはできません。コンポーネントは両方の標準ユーザインターフェース要素を表示できないため、両方の標準ユーザインターフェースを表示しません。

例: コンポーネントの例

カスタムアクションで使用できるコンポーネントの例を次に示します。カスタムアクションの名前は、「Quick Add」など自由に設定することができます(コンポーネントの名前と、そのコンポーネントを使用するアクションの名前が一致する必要はありません)。このコンポーネントでは、2つの数をすばやく合計することができます。

```
<!--quickAdd.cmp-->
<aura:component implements="force:lightningQuickAction">

    <!-- Very simple addition -->

    <lightning:input type="number" name="myNumber" aura:id="num1" label="Number 1"/>
+
    <lightning:input type="number" name="myNumber" aura:id="num2" label="Number 2"/>

    <br/>
    <lightning:button label="Add" onclick="{!c.clickAdd}"/>

</aura:component>
```

コンポーネントマークアップは 2 つの入力項目と [追加] ボタンを表示するだけです。

コンポーネントのコントローラが実際のすべての作業を実行します。

```
/*quickAddController.js*/
({
    clickAdd: function(component, event, helper) {

        // Get the values from the form
        var n1 = component.find("num1").get("v.value");
        var n2 = component.find("num2").get("v.value");

        // Display the total in a "toast" status message
        var resultsToast = $A.get("e.force:showToast");
        resultsToast.setParams({
            "title": "Quick Add: " + n1 + " + " + n2,
            "message": "The total is: " + (n1 + n2) + "."
        });
        resultsToast.fire();

        // Close the action panel
        var dismissActionPanel = $A.get("e.force:closeQuickAction");
        dismissActionPanel.fire();
    }

})
```

ユーザが入力した2つの数の取得は簡単ですが、より強固なコンポーネントでは、入力が有効かどうかの確認などを行います。この例の重要な部分は、数がどのように処理されるかと、カスタムアクションがどのように解決するかです。

加算の結果は「toast」に表示されます。これは、ページの上部に表示される状況メッセージです。このトーストは、`force:showToast` イベントを起動することで作成されます。トーストは結果を表示できる唯一の方法ではありません。また、アクションはトースト専用でもありません。これは、単に Lightning Experience または Salesforce1 の画面の上部にメッセージを表示するための便利な方法です。

ここでトーストを使用していることに関して重要なことは、その後どのように処理されるかということです。`clickAdd` コントローラアクションは `force:closeQuickAction` イベントを起動します。これにより、アクションパネルが閉じられます。ただし、アクションパネルが閉じられてもトーストは依然として表示されます。`force:closeQuickAction` イベントはアクションパネルによって処理されますが、アクションパネルは閉じられています。`force:showToast` イベントは `one.app` コンテナによって処理されるため、パネルが動作している必要はありません。

関連トピック:

レコード固有のアクションのコンポーネントの設定

レコード固有のアクションのコンポーネントの設定

`force:hasRecordId` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの ID をコンポーネントに割り当てることができます。現在のレコード ID は、Lightning Experience または Salesforce1 のオブジェクト固有のカスタムアクションやアクション上書きなどとして、コンポーネントを Lightning レコードページで使用する場合に便利です。

force:hasRecordId インターフェースは、このインターフェースを実装するコンポーネントに対して 2 つのことを行います。

- recordId という名前の属性をコンポーネントに追加します。この属性は文字列型であり、その値は 18 文字の Salesforce レコード ID (001xx000003DGSWAA4 など) です。これを自分で追加した場合、属性の定義は次のようなマークアップになります。

```
<aura:attribute name="recordId" type="String" />
```

  メモ: コンポーネントで force:hasRecordId を実装する場合、recordId 属性をコンポーネントに自分で追加する必要はありません。追加する場合は、属性のアクセスレベルまたは型を変更しないでください。変更すると、コンポーネントでランタイムエラーが発生します。

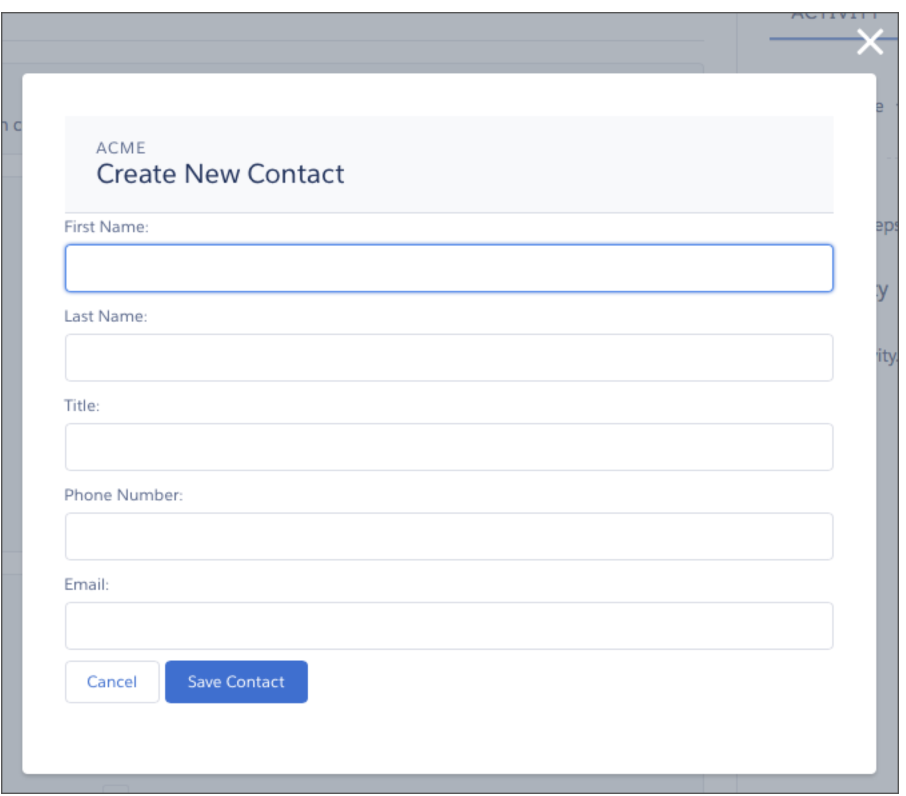
- Lightning Experience または Salesforce1 のレコードコンテキストでコンポーネントを呼び出す場合、recordId を、表示するレコードの ID に設定します。

この動作は、プログラミング言語のインターフェースで予想される動作とは異なります。これは、force:hasRecordId がマーカーインターフェースであるためです。マーカーインターフェースは、インターフェースの動作をコンポーネントに追加するよう伝える、コンポーネントのコンテナへの信号です。

recordId 属性は、レコードのコンテキストでコンポーネントを配置または呼び出す場合にのみ設定されます。たとえば、レコードページにコンポーネントを配置する場合、またはレコードページやオブジェクトホームからコンポーネントをアクションとして呼び出す場合などがこれに該当します。その他の場合(このコンポーネントをプログラムで別のコンポーネント内に作成する場合など)、recordId は設定されないため、コンポーネントでこれを使用しないでください。

例: レコード固有のアクションのコンポーネントの例

この拡張の例では、取引先レコードの詳細ページからカスタムアクションとして呼び出されるように設計されたコンポーネントを示します。コンポーネントを作成したら、取引先オブジェクトのカスタムアクションを作成し、取引先ページレイアウトに追加する必要があります。アクションを使用した開かれたコンポーネントは、次のようなアクションパネルに表示されます。



コンポーネント定義は、まず force:lightningQuickActionWithoutHeader と force:hasRecordId の両方のインターフェースで実装されます。最初のインターフェースは、コンポーネントをアクションとして使用できるようにし、標準コントロールが表示されないようにします。2番目のインターフェースは、レコードのコンテキストでコンポーネントが呼び出されたときにインターフェースのレコード ID 属性と値の自動割り当て動作を追加します。

quickContact.cmp

```
<aura:component controller="QuickContactController"
    implements="force:lightningQuickActionWithoutHeader,force:hasRecordId">

    <aura:attribute name="account" type="Account" />
    <aura:attribute name="newContact" type="Contact"
        default="{ 'sobjectType': 'Contact' }" /> <!-- default to empty record -->

    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />

    <!-- Display a header with details about the account -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading--label">{!v.account.Name}</p>
        <h1 class="slds-page-header__title slds-m-right--small
            slds-truncate slds-align-left">Create New Contact</h1>
    </div>

    <!-- Display the new contact form -->
     <lightning:input aura:id="contactField" name="firstName" label="First Name"
                      value="{!v.newContact.FirstName}" required="true"/>
```

```
    <lightning:input aura:id="contactField" name="lastname" label="Last Name"
                     value="{!v.newContact.LastName}" required="true"/>

    <lightning:input aura:id="contactField" name="title" label="Title"
                     value="{!v.newContact.Title}" />

    <lightning:input aura:id="contactField" type="phone" name="phone" label="Phone
Number"
                     pattern="^(1?(-?\d{3})-?)?(\d{3})(-?\d{4})$"
                     messageWhenPatternMismatch="The phone number must contain 7, 10,
 or 11 digits. Hyphens are optional."
                     value="{!v.newContact.Phone}" required="true"/>

    <lightning:input aura:id="contactField" type="email" name="email" label="Email"
                     value="{!v.newContact.Email}" />

    <lightning:button label="Cancel" onclick="{!c.handleCancel}"
class="slds-m-top--medium" />
    <lightning:button label="Save Contact" onclick="{!c.handleSaveContact}"
               variant="brand" class="slds-m-top--medium"/>

</aura:component>
```

コンポーネントでは、メンバー変数として使用される 3 つの属性が定義されます。

- *account* — init ハンドラで読み込まれた完全な取引先レコードを保持
- *newContact* — フォーム項目の値を取得するために使用される空の取引先責任者

残りのコンポーネント定義は、必須項目が空である場合や、電話項目が指定のパターンに一致していない場合に項目に関するエラーを表示する標準形式です。

コンポーネントのコントローラには、3 つのアクションハンドラの目的のすべてのコードがあります。

quickContactController.js

```
({
    doInit : function(component, event, helper) {

        // Prepare the action to load account record
        var action = component.get("c.getAccount");
        action.setParams({"accountId": component.get("v.recordId")});

        // Configure response handler
        action.setCallback(this, function(response) {
            var state = response.getState();
            if(state === "SUCCESS") {
                component.set("v.account", response.getReturnValue());
            } else {
                console.log('Problem getting account, response state: ' + state);
            }
        });
        $A.enqueueAction(action);
    },
```

```
    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {

            // Prepare the action to create the new contact
            var saveContactAction = component.get("c.saveContactWithAccount");
            saveContactAction.setParams({
                "contact": component.get("v.newContact"),
                "accountId": component.get("v.recordId")
            });

            // Configure the response handler for the action
            saveContactAction.setCallback(this, function(response) {
                var state = response.getState();
                if(state === "SUCCESS") {

                    // Prepare a toast UI message
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Contact Saved",
                        "message": "The new contact was created."
                    });

                    // Update the UI: close panel, show toast, refresh account page
                    $A.get("e.force:closeQuickAction").fire();
                    resultsToast.fire();
                    $A.get("e.force:refreshView").fire();
                }
                else if (state === "ERROR") {
                    console.log('Problem saving contact, response state: ' + state);
                }
                else {
                    console.log('Unknown problem, response state: ' + state);
                }
            });

            // Send the request to create the new contact
            $A.enqueueAction(saveContactAction);
        }

    },

 handleCancel: function(component, event, helper) {
     $A.get("e.force:closeQuickAction").fire();
    }
})
```

最初のアクションハンドラ doInit は init ハンドラです。このハンドラのジョブは、force:hasRecordId インターフェースで提供されるレコード ID を使用して、完全な取引先レコードを読み込むことです。このコンポーネントが別のオブジェクト (リード、商談、またはカスタムオブジェクト) のアクションで使用されないようにする手段は講じられていません。この場合、doInit ではレコードの読み込みに失敗しますが、フォームは表示されます。

handleSaveContact アクションハンドラは、ヘルパー関数をコールしてフォームを検証します。フォームが有効でない場合、項目レベルのエラーが表示されます。フォームが有効であった場合、アクションハンドラは次の作業を行います。

- 新しい取引先責任者を保存するサーバアクションを準備をする。
- サーバがアクションを完了したときのために*レスポンスハンドラ*と呼ばれるコールバック関数を定義する。レスポンスハンドラについては、後ほど説明します。
- サーバアクションをキューに追加する。

サーバアクションのレスポンスハンドラ自体はほとんど作業をしません。サーバアクションが成功すると、レスポンスハンドラは次の作業を行います。

- force:closeQuickAction イベントを起動してアクションパネルを閉じる。
- force:showToast イベントを起動して、取引先責任者が作成されたことを示す「トースト」メッセージを表示する。
- レコードページに自身を更新するように通知する force:refreshView イベントを起動して、レコードページを更新する。

更新イベントに対応して取引先責任者のリストが更新されると、リストの最後の項目に新しいレコードが表示されます。

handleCancel アクションハンドラは、force:closeQuickAction イベントを起動してアクションパネルを閉じます。

ここに記載されているコンポーネントヘルパーは、その用途を十分に説明できるだけの最小限のヘルパーです。本番品質のフォーム検証コードでは、さらに多くの作業が必要になります。

quickContactHelper.js

```
({
    validateContactForm: function(component) {
        var validContact = true;


        // Show error messages if required fields are blank
        var allValid = component.find('contactField').reduce(function (validFields,
inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validFields && inputCmp.get('v.validity').valid;
        }, true);

        if (allValid) {
        // Verify we have an account to attach it to
        var account = component.get("v.account");
        if($A.util.isEmpty(account)) {
            validContact = false;
            console.log("Quick action context doesn't have a valid account.");
        }

        return(validContact);
 }
    }
})
```

最後に、このコンポーネントのサーバ側コントローラとして使用される Apex クラスは、わかりやすいように意図的に簡略化されています。

QuickContactController.apxc

```
public with sharing class QuickContactController {

    @AuraEnabled
    public static Account getAccount(Id accountId) {
        // Perform isAccessible() checks here
        return [SELECT Name, BillingCity, BillingState FROM Account WHERE Id =
:accountId];
    }

    @AuraEnabled
    public static Contact saveContactWithAccount(Contact contact, Id accountId) {
        // Perform isAccessible() and isUpdateable() checks here
        contact.AccountId = accountId;
        upsert contact;
        return contact;
    }

}
```

一方のメソッドは、レコード ID に基づいて取引先を取得します。他方のメソッドは、新規取引先責任者レコードを取引先に割り当て、データベースに保存します。

関連トピック:

カスタムアクション用のコンポーネントの設定

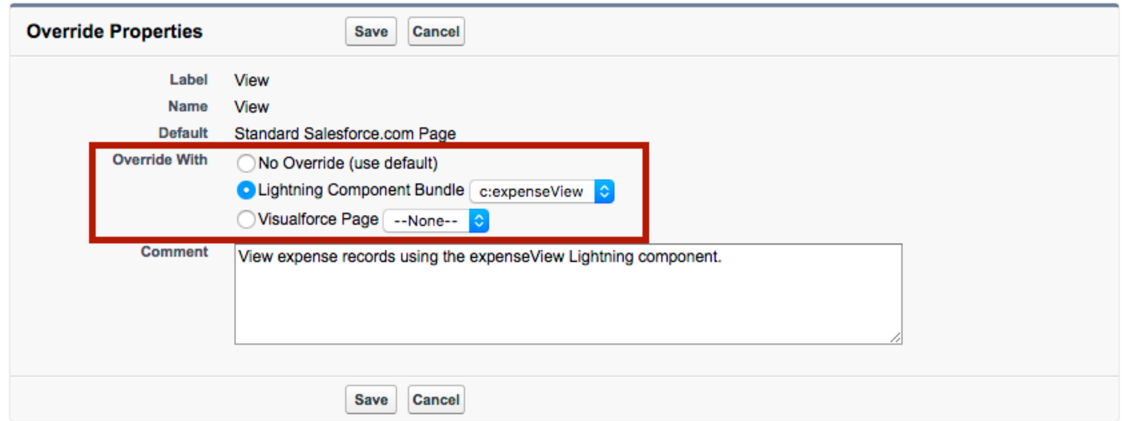
force:hasRecordId

force:hasSObjectName

Lightning コンポーネントでの標準アクションの上書き

lightning:actionOverride インターフェースを Lightning コンポーネントに追加して、コンポーネントを使用してオブジェクトに対する標準アクションを上書きできるようにします。ほとんどの標準コンポーネントおよびすべてのカスタムコンポーネントで、表示、新規、編集、およびタブ標準アクションを上書きできます。標準アクションを上書きすると、Lightning コンポーネントを使用して組織をカスタマイズできます。たとえば、レコードの参照、作成、編集方法を完全にカスタマイズできます。

Lightning コンポーネントでのアクションの上書きは、Visualforce ページでのアクションの上書きとよく似ています。アクションの [プロパティの上書き] で Visualforce ページの代わりに Lightning コンポーネントを選択します。



ただし、アクション上書きとして使用できる Lightning コンポーネントの作成方法や、Salesforce でのアクション上書きの使用方法に関して Visualforce との重要な違いがあります。始める前に詳細を徹底的に確認し、本番環境にリリースする前に Sandbox または Developer Edition 組織で慎重にテストすることをお勧めします。

このセクションの内容:

標準アクションと上書きの基本

ほとんどの標準オブジェクトとすべてのカスタムオブジェクトで 6 つの標準アクション (タブ、リスト、表示、編集、新規、削除) を使用できます。Salesforce Classic では、これらはすべて異なるアクションです。

Lightning コンポーネントでの標準アクションの上書き

Salesforce Classic と Lightning Experience の両方で標準アクションを上書きできます。目的は同じですが、操作経路は異なります。

アクション上書きとして使用する Lightning コンポーネントの作成

`lightning:actionOverride` インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でアクション上書きとして使用できるようにします。このインターフェースを実装するコンポーネントのみがオブジェクトアクションの [プロパティの上書き] パネルの **[Lightning コンポーネントバンドル]** メニューに表示されます。

アクション上書きのパッケージ化

カスタムオブジェクトのアクション上書きは、自動的にカスタムオブジェクトとパッケージ化されます。標準オブジェクトのアクション上書きはパッケージ化できません。

関連トピック:

lightning:actionOverride

標準アクションと上書きの基本

ほとんどの標準オブジェクトとすべてのカスタムオブジェクトで6つの標準アクション(タブ、リスト、表示、編集、新規、削除) を使用できます。Salesforce Classic では、これらはすべて異なるアクションです。

Lightning Experience と Salesforce1 では、タブアクションとリストアクションが 1 つのアクションに統合され、オブジェクトホームになりました。ただし、オブジェクトホームには Lightning Experience の [タブ] アクションと Salesforce1 の [リスト] アクション経由でアクセスします。このリリースでは、Lightning コンポーネントを使用してタブアクションのみを上書きできます。したがって、コンポーネントを使用して Salesforce1 のリストアクションを上書きすることはできません。最後に、Salesforce1 には固有の検索アクション (タブ経由でアクセス) があります(これは少し厄介であり複雑なためです)。

次の表に、[設定] でアクションを指定するときに上書きできる、オブジェクトの標準アクションと、3 種類のユーザエクスペリエンスで上書きされた結果のアクションを示します。

[設定] で上書き	Salesforce Classic	Lightning Experience	Saleforce1
タブ	オブジェクトタブ	オブジェクトホーム	検索
リスト	オブジェクトリスト	なし	オブジェクトホーム
ビュー	レコードビュー	レコードホーム	レコードホーム
編集	レコード編集	レコード編集	レコード編集
新規	レコードの作成	レコードの作成	レコードの作成
削除	レコード削除	レコード削除	レコード削除

メモ:

- 「N/A」は、標準の動作にアクセスできないわけでも、標準の動作を*上書き*できないわけでもありません。*上書きにアクセス*できないということです。使用できないのは上書きの機能です。
- 他にも [承認] と [コピー] の 2 つの標準アクションがあります。これらのアクションはより複雑であるため、これらの上書きは高度なプロジェクトになります。Lightning コンポーネントでこれらのアクションを上書きすることはできません。

Lightning コンポーネントのアクション上書きを使用する方法と場所

Lightning Experience と Salesforce1 で Lightning コンポーネントを使用して、表示、新規、編集、およびタブ標準アクションを上書きできます。Visualforce とは異なり、Lightning コンポーネントを使用する上書きは Salesforce Classic に影響しません。つまり、次のようになります。

- Visualforce ページで標準アクションを上書きした場合、Salesforce Classic、Lightning Experience、および Salesforce1 のアクションが上書きされます。
- Lightning コンポーネントで標準アクションを上書きした場合、Lightning Experience と Salesforce1 のアクションが上書きされますが、Salesforce Classic では標準の Salesforce ページが使用されます。

オブジェクトの Lightning レコードページがオブジェクトの表示アクションの上書きよりも優先されます。つまり、オブジェクトの表示アクションを上書きするときにオブジェクトの Lightning レコードページの作成と割り

当ても行う場合は、Lightning レコードページが使用されます。上書きは無効になります。上書きで Lightning コンポーネントまたは Visualforce ページのどちらを使用するかに関係なくこれは適用されます。

アクションの上書きはLightningコンソールアプリケーションでサポートされておらず、実行しても無視され何の通知もありません。Lightningコンソールアプリケーションユーザが、上書きされたアクションをトリガすると、代わりに標準アクションが表示されます。Lightningコンソールアプリケーション外で同じアクションをトリガすると、上書きされたアクションが表示されます。この動作によってユーザエクスペリエンスの一貫性が損なわれる可能性があるため、その点をユーザに警告する必要があります。また、アクション上書きでのみ実行されるコードではなく、トリガと入力規則を使用してデータ検証要件を満たしていることを確認します。この方法により、標準アクションとアクション上書きのどちらを使用してデータを変更しても、データの有効性が確保されます。

Lightning コンポーネントでの標準アクションの上書き

Salesforce Classic と Lightning Experience の両方で標準アクションを上書きできます。目的は同じですが、操作経路は異なります。

組織には `lightning:actionOverride` インターフェースを実装する 1 つ以上の Lightning コンポーネントが必要です。独自のカスタムコンポーネントや管理パッケージのコンポーネントを使用できます。

上書きするアクションがあるオブジェクトのオブジェクト管理設定に移動します。

1. [ボタン、リンク、およびアクション] を選択します。
2. 上書きするアクションの [編集] を選択します。
3. [上書き手段] で **[Lightning コンポーネントバンドル]** を選択します。
4. ドロップダウンメニューから、アクション上書きとして使用する Lightning コンポーネントの名前を選択します。
5. [保存] を選択します。

メモ: ユーザが Lightning Experience または Salesforce1 を再読み込みするまでアクション上書きへの変更はユーザに表示されません。

関連トピック:

Salesforce ヘルプ: オブジェクト管理設定の検索

Salesforce ヘルプ: 標準ボタンとタブのホームページの上書き

アクション上書きとして使用する Lightning コンポーネントの作成

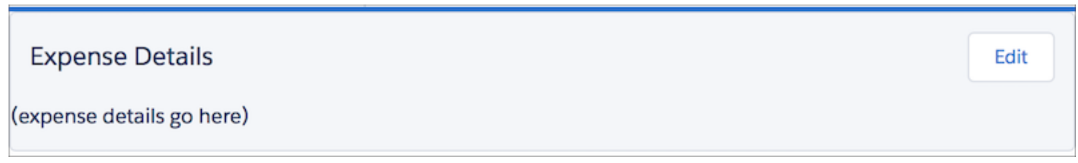
`lightning:actionOverride` インターフェースを Lightning コンポーネントに追加し、Lightning Experience または Salesforce1 でアクション上書きとして使用できるようにします。このインターフェースを実装するコンポーネントのみがオブジェクトアクションの [プロパティの上書き] パネルの **[Lightning コンポーネントバンドル]** メニューに表示されます。

```
<aura:component
    implements="lightning:actionOverride,force:hasRecordId,force:hasSObjectName">

    <article class="slds-card">
```

```
      <div class="slds-card__header slds-grid">
        <header class="slds-media slds-media--center slds-has-flexi-truncate">
          <div class="slds-media__body">
            <h2><span class="slds-text-heading--small">Expense Details</span></h2>
          </div>
        </header>
        <div class="slds-no-flex">
            <lightning:button label="Edit" onclick="{!c.handleEdit}"/>
        </div>
      </div>
      <div class="slds-card__body">(expense details go here)</div>
    </article>
</aura:component>
```

Lightning Experience では、標準のタブアクションと表示アクションはページとして表示され、標準の新規アクションと編集アクションは、フロート表示されたパネルに表示されます。アクション上書きとして使用する場合、`lightning:actionOverride` インターフェースを実装する Lightning コンポーネントは標準の動作を完全に置き換えます。ただし、上書きされたアクションはパネルとしてではなく常にページとして表示されます。コンポーネントに、Lightning Experience のメインナビゲーションバー以外のコントロールは表示されません。コンポーネントでは、ナビゲーションバーのほかに、ナビゲーションやアクションを含めアクションの完全なユーザインターフェースが提供されることが予想されます。



注目すべき Visualforce との重要な違いの 1 つは、[Lightning コンポーネントバンドル] メニューへのコンポーネントの追加方法です。Visualforce ページのメニューには、特定のオブジェクトの標準コントローラを使用するページ、または標準コントローラを一切使用しないページのリストが表示されます。この絞り込みにより、オブジェクトごとにメニューオプションが変化し、オブジェクトに固有のページまたは完全に汎用的なページのみが提供されます。

[Lightning コンポーネントバンドル] メニューには、`lightning:actionOverride` インターフェースを実装するすべてのコンポーネントが含まれます。`lightning:actionOverride` を実装するコンポーネントでは、システム管理者が特定のアクションまたは特定のオブジェクトのアクションのみを上書きできるように制限できません。目的のオブジェクトの目的のアクションを上書きするためだけにコンポーネントが使用されるように、組織でプロセスおよびコンポーネントの命名規則を採用することをお勧めします。このようにしても、コンポーネント開発者には、任意のオブジェクトの任意のアクションで使用できるように、`lightning:actionOverride` インターフェースを実装するコンポーネントで適切に対応する責任があります。

現在のレコードの詳細へのアクセス

通常、アクション上書きとして使用するコンポーネントには、使用するオブジェクト種別に関する詳細が必要です。多くの場合、それは現在のレコードの ID です。コンポーネントは、次のインターフェースを実装して次のオブジェクトおよびレコードの詳細にアクセスできます。

force:hasRecordId

`force:hasRecordId` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの ID をコンポーネントに割り当てることができます。現在のレコード ID は、Lightning Experience または Salesforce1 のオブジェクト固有のカスタムアクションやアクション上書きなどとして、コンポーネントを Lightning レコードページで使用する場合に便利です。

force:hasSObjectName

`force:hasSObjectName` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの sObject 型の API 参照名をコンポーネントに割り当てることができます。さまざまな sObject 型のレコードでコンポーネントを使用でき、現在のレコードの特定のタイプに適応する必要がある場合は、sObject 名が便利です。

関連トピック:

force:hasRecordId

force:hasSObjectName

アクション上書きのパッケージ化

カスタムオブジェクトのアクション上書きは、自動的にカスタムオブジェクトとパッケージ化されます。標準オブジェクトのアクション上書きはパッケージ化できません。

カスタムオブジェクトをパッケージ化すると、そのオブジェクトの標準アクションの上書きも一緒にパッケージ化されます。これには、上書きで使用される Lightning コンポーネントが含まれます。これは問題なく機能します。

ただし、標準オブジェクトはパッケージ化できません。結果として、標準オブジェクトの標準アクションの上書きをパッケージ化する方法はありません。

パッケージの標準オブジェクトの標準アクションを上書きするには、次の操作を実行します。

- 上書きで使用される Lightning コンポーネントを手動でパッケージ化する。
- 影響を受ける標準オブジェクトの関連する標準アクションを手動で上書きする手順を登録組織に提供する。

関連トピック:

Lightning コンポーネントでの標準アクションの上書き

メタデータ API 開発者ガイド: ActionOverride

Lightning コンポーネントを Lightning ページで使用できるようにするための準備

カスタム Lightning コンポーネントは、標準では Lightning ページまたは Lightning アプリケーションビルダーで動作しません。このどちらかの場所でカスタムコンポーネントを使用するには、互換性が確保されるようコンポーネントとそのコンポーネントバンドルを設定します。

このセクションの内容:

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定

Lightningページまたは Lightning アプリケーションビルダーでカスタム Lightning コンポーネントを使用する前に、いくつかのステップを実行します。

Lightning コンポーネントバンドルのデザインリソース

デザインリソースを使用して、Lightning アプリケーションビルダーなどのビルダーツールに公開する属性を制御します。デザインリソースは.cmp リソースと同じフォルダに存在します。このリソースには、Lightning コンポーネントの設計時の動作(ページまたはアプリケーションでコンポーネントを表示するためにビジュアルツールが必要とする情報) が記述されます。

Lightning Experience のレコードホームページのコンポーネントの設定

Lightningページおよび Lightning アプリケーションビルダーで動作するようにコンポーネントを設定したら、次のガイドラインを使用して、Lightning Experience のレコードページで動作するようにコンポーネントを設定します。

Lightning for Outlook および Lightning for Gmail のコンポーネントの作成

Lightning for Outlook および Lightning for Gmail のメールアプリケーションペインにドラッグアンドドロップできるカスタムの Lightning コンポーネントを作成します。

カスタムコンポーネントの動的選択リストの作成

Lightning アプリケーションビルダーでコンポーネントが設定されるときに、コンポーネントのプロパティを選択リストとして公開できます。選択リストの値は、作成する Apex クラスによって提供されます。

カスタム Lightning ページテンプレートコンポーネントの作成

すべての標準 Lightning ページは、ページの領域とページに含まれるコンポーネントを定義するデフォルトテンプレートコンポーネントに関連付けられています。カスタム Lightning ページテンプレートコンポーネントでは、定義した構造とコンポーネントを使用して、ビジネスニーズに合ったページテンプレートを作成できます。カスタムテンプレートを実装すると、ページ作成者は Lightning アプリケーションビルダーの新規ページウィザードでそのカスタムテンプレートを使用できます。

Lightning ページテンプレートコンポーネントのベストプラクティス

Lightning ページテンプレートコンポーネントを作成する場合、次のベストプラクティスと制限に留意してください。

lightning:flexipageRegionInfo による Lightning ページコンポーネントでの幅の認識

Lightning アプリケーションビルダーでページの範囲にコンポーネントを追加するときに、`lightning:flexipageRegionInfo` サブコンポーネントが親コンポーネントにその範囲の幅を渡します。`lightning:flexipageRegionInfo` といくつかの戦略的CSS を使用すれば、実行時、範囲ごとに異なる表示で親コンポーネントに表示できます。

Lightningページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項

Lightningページおよび Lightning アプリケーションビルダーのコンポーネントおよびコンポーネントバンドルを作成する場合、次のガイドラインを参考にしてください。

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定

Lightning ページまたは Lightning アプリケーションビルダーでカスタム Lightning コンポーネントを使用する前に、いくつかのステップを実行します。

1.組織で [私のドメイン] をリリースする

Lightning コンポーネントを Lightning タブや Lightning ページで使用する場合や、カスタム Lightning ページテンプレートまたはスタンドアロンアプリケーションとして使用する場合は、組織で[私のドメイン]をリリースする必要があります。

[私のドメイン] についての詳細は、Salesforce ヘルプを参照してください。

2.新規インターフェースをコンポーネントに追加する

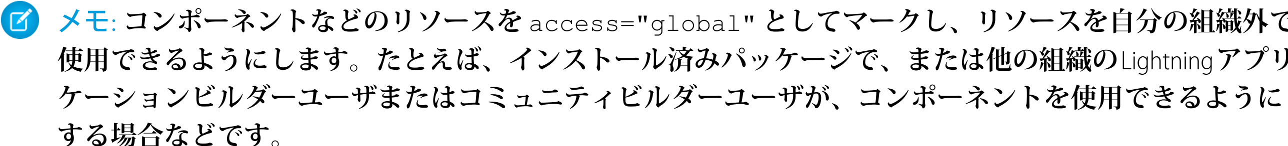
コンポーネントを Lightning アプリケーションビルダーまたは Lightning ページに表示するには、コンポーネントに次のいずれかのインターフェースを実装する必要があります。

インターフェース	説明
`flexipage:availableForAllPageTypes`	レコードページと他の種別のページ (Lightning アプリケーションのユーティリティバーを含む) でコンポーネントを使用できます。
`flexipage:availableForRecordHome`	コンポーネントがレコードページ専用に設計されている場合は、このインターフェースを `flexipage:availableForAllPageTypes` の代わりに実装します。 詳細は、「Lightning Experience のレコードホームページのコンポーネントの設定」(ページ 143)を参照してください。
`clients:availableForMailAppAppPage`	Lightning アプリケーションビルダーのメールアプリケーション Lightning ページ、Lightning for Outlook、または Lightning for Gmail にコンポーネントを表示できます。

シンプルな「Hello World」コンポーネントのサンプルコードを次に示します。

```
<aura:component implements="flexipage:availableForAllPageTypes" access="global">
    <aura:attribute name="greeting" type="String" default="Hello" access="global" />
    <aura:attribute name="subject" type="String" default="World" access="global" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```



メモ: コンポーネントなどのリソースを access="global" としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織の Lightning アプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

3.デザインリソースをコンポーネントバンドルに追加する

デザインリソースを使用して、Lightning アプリケーションビルダーなどのビルダーツールに公開する属性を制御します。デザインリソースは .cmp リソースと同じフォルダに存在します。このリソースには、Lightning コンポーネントの設計時の動作 (ページまたはアプリケーションでコンポーネントを表示するためにビジュアルツールが必要とする情報) が記述されます。

たとえば、コンポーネントを 1 つ以上のオブジェクトに制限したり、属性のデフォルト値を設定したり、Lightning コンポーネントの属性をシステム管理者が Lightning アプリケーションビルダーで編集できるようにしたりするには、コンポーネントバンドルのデザインリソースが必要です。

「Hello World」コンポーネントと一緒にバンドルするデザインリソースを次に示します。

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you want to greet" />
    <design:attribute name="greeting" label="Greeting" />
</design:component>
```

デザインリソースの名前は、*componentName*.design にする必要があります。

省略可能: SVG リソースをコンポーネントバンドルに追加する

SVG リソースを使用して、コンポーネントが Lightning アプリケーションビルダーのコンポーネントペインに表示されるときのカスタムアイコンを定義できます。これをコンポーネントバンドルに追加します。

「Hello World」コンポーネントと一緒に表示するシンプルな赤い円の SVG リソースの例を次に示します。

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     width="400" height="400">
  <circle cx="100" cy="100" r="50" stroke="black"
    stroke-width="5" fill="red" />
</svg>
```

SVG リソースの名前は *componentName*.svg にする必要があります。

関連トピック:

Lightning コンポーネントバンドルのデザインリソース

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項

コンポーネントのバンドル

インターフェースの参照

Lightning コンポーネントバンドルのデザインリソース

デザインリソースを使用して、Lightning アプリケーションビルダーなどのビルダーツールに公開する属性を制御します。デザインリソースは .cmp リソースと同じフォルダに存在します。このリソースには、Lightning コンポーネントの設計時の動作 (ページまたはアプリケーションでコンポーネントを表示するためにビジュアルツールが必要とする情報) が記述されます。

たとえば、「Hello World」コンポーネントと一緒にバンドルする簡単なデザインリソースを示します。

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
</design:component>
```

design:component

これは、デザインリソースのルート要素です。これには、使用するツール (アプリケーションビルダーなど) に関する、コンポーネントの設計時の設定が含まれます。

属性	説明
label	Lightning アプリケーションビルダーに表示するときのコンポーネントの表示ラベルを設定します。カスタム Lightning ページテンプレートコンポーネントを作成するときに、このテキストが Lightning アプリケーションビルダーの新規ページウィザードにテンプレートの名前として表示されます。

design:attribute

Lightning コンポーネントの属性をシステム管理者が Lightning アプリケーションビルダーで編集できるようにするには、属性の `design:attribute` ノードをデザインリソースに追加します。コンポーネント定義で必須とマークされた属性は、デフォルト値が割り当てられている場合を除き、Lightning アプリケーションビルダーで自動的にユーザに表示されます。

デザインリソースでは、`Integer`、`String`、または `Boolean` 型の属性のみがサポートされます。

属性	説明
datasource	静的値を含む選択リストとして項目を表示します。文字列属性でのみサポートされます。 `<design:attribute name="Name" datasource="value1,value2,value3" />` Apex クラスを使用して選択リスト値を動的に設定することもできます。詳細は、「カスタムコンポーネントの動的選択リストの作成」(ページ 150)を参照してください。 デザインリソースに datasource が設定された文字列属性はすべて選択リストとして処理されます。
default	デザインリソース内の属性のデフォルト値を設定します。 `<design:attribute name="Name" datasource="value1,value2,value3" default="value1" />`
description	プロパティペインに属性の i バブルとして表示されます。
label	プロパティペインに表示する属性の表示ラベル。
max	属性が Integer の場合、属性の最大許容値が設定されます。属性が String の場合、これは属性の最大許容長です。
min	属性が Integer の場合、属性の最小許容値が設定されます。属性が String の場合、これは属性の最小許容長です。
name	必須属性。この値は .cmp リソースの aura:attribute 名の値と一致する必要があります。
placeholder	プロパティペインに表示するときの属性の入力プレースホルダテキスト。
required	属性が必須かどうかを示します。省略した場合、デフォルトの false になります。

<sfdc:object> および <sfdc:objects>

これらのタグセットを使用して、コンポーネントを 1 つ以上のオブジェクトに制限します。

メモ: <sfdc:object> および <sfdc:objects> はコミュニティビルダーではサポートされません。コンポーネントをオブジェクト固有のアクションとして使用するように設定する場合、または標準アクションを上書きするように設定する場合も、これらは無視されます。

次に、2 つのオブジェクトに制限された同じ「Hello World」コンポーネントのデザインリソースを示します。

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
    <sfdc:objects>
```

```
        <sfdc:object>Custom__c</sfdc:object>
        <sfdc:object>Opportunity</sfdc:object>
    </sfdc:objects>
</design:component>
```

オブジェクトをパッケージからインストールする場合、`<sfdc:object>` タグセットにオブジェクトを含めるときにオブジェクト名の先頭に ***namespace***__ の文字列を追加します。たとえば、objectNamespace__ObjectApiName__c です。

関連トピック:

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項

Lightning Experience のレコードホームページのコンポーネントの設定

Lightning ページおよび Lightning アプリケーションビルダーで動作するようにコンポーネントを設定したら、次のガイドラインを使用して、Lightning Experience のレコードページで動作するようにコンポーネントを設定します。

レコードページがアプリケーションページと明確に異なる点は、レコードのコンテキストがあるということです。現在のレコードに基づいてコンポーネントにコンテンツを表示するには、インターフェースと属性の組み合わせを使用します。

- コンポーネントをレコードページと他の種別のページの両方で使用できる場合、flexipage:availableForAllPageTypes を実装します。
- コンポーネントがレコードページ専用に設計されている場合は、flexipage:availableForAllPageTypes の代わりに flexipage:availableForRecordHome インターフェースを実装します。
- コンポーネントにレコード ID が必要な場合は、force:hasRecordId インターフェースも実装します。
- コンポーネントにオブジェクトの API 名が必要な場合は、force:hasSObjectName インターフェースも実装します。

メモ: 管理コンポーネントが flexipage または forceCommunity インターフェースを実装している場合、コンポーネントとその属性が access="global" に設定されていないとアップロードがブロックされます。アクセスチェックについての詳細は、「アクセスの制御」を参照してください。

force:hasRecordId

レコードページのコンポーネント、カスタムオブジェクトのアクションなど、特定のレコードに関連付けられたコンテキストで呼び出されるコンポーネントで役立ちます。現在表示されているレコードの ID をコンポーネントで取得する場合は、このインターフェースを追加します。

force:hasRecordId インターフェースは、このインターフェースを実装するコンポーネントに対して 2 つのことを行います。

- `recordId` という名前の属性をコンポーネントに追加します。この属性は文字列型であり、その値は 18 文字の Salesforce レコード ID (001xx000003DGSWAA4 など) です。これを自分で追加した場合、属性の定義は次のようなマークアップになります。

  ```
  <aura:attribute name="recordId" type="String" />
  ```

  メモ: コンポーネントで `force:hasRecordId` を実装する場合、`recordId` 属性をコンポーネントに自分で追加する必要はありません。追加する場合は、属性のアクセスレベルまたは型を変更しないでください。変更すると、コンポーネントでランタイムエラーが発生します。

- Lightning Experience または Salesforce1 のレコードコンテキストでコンポーネントを呼び出す場合、`recordId` を、表示するレコードの ID に設定します。

`recordId` 属性を Lightning アプリケーションビルダーに公開しないでください。つまり、この属性をコンポーネントのデザインリソースに配置しないでください。システム管理者がレコード ID を提供するのは望ましくありません。

`recordId` 属性は、レコードのコンテキストでコンポーネントを配置または呼び出す場合にのみ設定されます。たとえば、レコードページにコンポーネントを配置する場合、またはレコードページやオブジェクトホームからコンポーネントをアクションとして呼び出す場合などがこれに該当します。その他の場合(このコンポーネントをプログラムで別のコンポーネント内に作成する場合など)、`recordId` は設定されないため、コンポーネントでこれを使用しないでください。

`force:hasSObjectName`

レコードページコンポーネントに役立ちます。現在表示されているレコードのオブジェクトのAPI名をコンポーネントで認識する必要がある場合は、このインターフェースを実装します。

このインターフェースは、`sObjectName` という名前の属性をコンポーネントに追加します。この属性は文字列型であり、その値は `Account` や `myNamespace__myObject__c` のようなオブジェクトの API 名です。次に例を示します。

```
<aura:attribute name="sObjectName" type="String" />
```

メモ: コンポーネントで `force:hasSObjectName` を実装する場合、`sObjectName` 属性をコンポーネントに自分で追加する必要はありません。追加する場合は、属性のアクセスレベルまたは型を変更しないでください。変更すると、コンポーネントでランタイムエラーが発生します。

`sObjectName` 属性は、レコードのコンテキストでコンポーネントを配置または呼び出す場合にのみ設定されます。たとえば、レコードページにコンポーネントを配置する場合、またはレコードページやオブジェクトホームからコンポーネントをアクションとして呼び出す場合などがこれに該当します。その他の場合 (このコ

ンポーネントをプログラムで別のコンポーネント内に作成する場合など)、sObjectName は設定されないため、コンポーネントでこれを使用しないでください。

関連トピック:

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項

Salesforce レコードの操作

force:hasRecordId

force:hasSObjectName

Lightning for Outlook および Lightning for Gmail のコンポーネントの作成

Lightning for Outlook および Lightning for Gmail のメールアプリケーションペインにドラッグアンドドロップできるカスタムの Lightning コンポーネントを作成します。

Lightning for Outlook または Lightning for Gmail のメールアプリケーションペインにコンポーネントを追加するには、clients:availableForMailAppAppPage インターフェースを実装します。

メールまたはカレンダーの行動にコンポーネントがアクセスできるようにするには、clients:hasItemContext インターフェースを実装します。

clients:hasItemContext インターフェースは、レコードまたはコンテンツ固有のロジックを実装するために使用できるコンポーネントに属性を追加します。次の属性が含まれます。

- source 属性。メールまたは予約のソースを示します。使用できる値は、email と event です。

```
<aura:attribute name="source" type="String" />
```

- people 属性。現在のメールまたは予約の受信者のメールアドレスを示します。

```
<aura:attribute name="people" type="Object" />
```

  people 属性の形状は、source 属性の値によって変化します。

  ソース属性がメールに設定されている場合は、人オブジェクトに次の要素が含まれます。

```
{
    to: [ { name: nameString, email: emailString }, ... ],
    cc: [ ... ],
    from: [ { name: senderName, email: senderEmail } ],
}
```

  ソース属性が行動に設定されている場合は、人オブジェクトに次の要素が含まれます。

```
{
    requiredAttendees: [ { name: attendeenameString, email: emailString }, ... ],
    optionalAttendees: [ { name: optattendeenameString, email: emailString }, ... ],
    organizer: [ { name: organizerName, email: senderEmail } ],
}
```

- subject は、現在のメールの件名を示します。

```
<aura:attribute name="subject" type="String" />
```

- messageBody は、現在のメールのメッセージを示します。

```
<aura:attribute name="messageBody" type="String" />
```

コンポーネントに行動の日付または場所を指定するには、clients:hasEventContext インターフェースを実装します。

```
    dates: {
            "start": value (String),
            "end": value (String),
    }
```

Lightning for Outlook および Lightning for Gmail は、次の行動をサポートしません。

- force:navigateToList
- force:navigateToRelatedList
- force:navigateToObjectHome
- force:refreshView

メモ: カスタムコンポーネントを Lightning for Outlook または Lightning for Gmail に正しく表示するには、可変幅に合わせてカスタムコンポーネントを調整できるようにします。

このセクションの内容:

Lightning for Outlook および Lightning for Gmail のカスタムコンポーネントのサンプル
Lightning for Outlook および Lightning for Gmail のメールアプリケーションペインに実装できるカスタムの Lightning コンポーネントのサンプルを見ていきます。

Lightning for Outlook および Lightning for Gmail のカスタムコンポーネントのサンプル

Lightning for Outlook および Lightning for Gmail のメールアプリケーションペインに実装できるカスタムの Lightning コンポーネントのサンプルを見ていきます。

以下に、Lightning for Outlook および Lightning for Gmail のメールアプリケーションペインに搭載できるカスタム Lightning コンポーネントの例を示します。このコンポーネントは、選択したメールまたは予約のコンテキストを利用します。

```
<aura:component implements="clients:availableForMailAppAppPage,clients:hasItemContext">

<!--
    Add these handlers to customize what happens when the attributes change
    <aura:handler name="change" value="{!v.subject}" action="{!c.handleSubjectChange}" />

    <aura:handler name="change" value="{!v.people}" action="{!c.handlePeopleChange}" />
-->
```

```
    <div id="content">
        <h1><b>Email subject</b></h1>
        <span id="subject">{!v.subject}</span>

        <h1>To:</h1>
        <aura:iteration items="{!v.people.to}" var="to">
            {!to.name} - {!to.email} <br/>
        </aura:iteration>

        <h1>From:</h1>
        {!v.people.from.name} - {!v.people.from.email}

        <h1>CC:</h1>
        <aura:iteration items="{!v.people.cc}" var="cc">
            {!cc.name} - {!cc.email} <br/>
        </aura:iteration>

        <span class="greeting">New Email Arrived</span>, {!v.subject}!
    </div>
</aura:component>
```

この例では、カスタムコンポーネントに、メール受信者のメールアドレスに基づく取引先および商談情報が表示されます。このコンポーネントは、初期化時に JavaScript コントローラ関数の `handlePeopleChange()` をコールします。JavaScript コントローラは、Apex サーバ側コントローラのメソッドをコールして、情報をクエリし、取引先の対応期間および商談が完了するまでの日数を計算します。以下に、Apex コントローラ、JavaScript コントローラ、ヘルパーを示します。

```
<!--
This component handles the email context on initialization.
It retrieves accounts and opportunities based on the email addresses included
in the email recipients list.
It then calculates the account and opportunity ages based on when the accounts
were created and when the opportunities will close.
-->

<aura:component
    implements="clients:availableForMailAppAppPage,clients:hasItemContext"
    controller="ComponentController">

    <aura:handler name="init" value="{!this}" action="{!c.handlePeopleChange}" />
    <aura:attribute name="accounts" type="List" />
    <aura:attribute name="opportunities" type="List" />
    <aura:iteration items="{!v.accounts}" var="acc">
            {!acc.name} => {!acc.age}
    </aura:iteration>
    <aura:iteration items="{!v.opportunities}" var="opp">
            {!opp.name} => {!opp.closesIn} Days till closing
    </aura:iteration>
```

```
</aura:component>
```

```
/*
On the server side, the Apex controller includes
Aura-enabled methods that accept a list of emails as parameters.
*/

public class ComponentController {
    /*
    This method searches for Contacts with matching emails in the email list,
    and includes Account information in the fields. Then, it filters the
    information to return a list of objects to use on the client side.
    */
    @AuraEnabled
    public static List<Map<String, Object>> findAccountAges(List<String> emails) {
    List<Map<String, Object>> ret = new List<Map<String, Object>>();
    List<Contact> contacts = [SELECT Name, Account.Name, Account.CreatedDate
                              FROM Contact
                              WHERE Contact.Email IN :emails];
    for (Contact c: contacts) {
            Map<String, Object> item = new Map<String, Object>();
            item.put('name', c.Account.Name);
            item.put('age',
                     Date.valueOf(c.Account.CreatedDate).daysBetween(
                         System.Date.today()));
            ret.add(item);
    }
     return ret;
}

    /*
    This method searches for OpportunityContactRoles with matching emails
    in the email list.
    Then, it calculates the number of days until closing to return a list
    of objects to use on the client side.
    */
    @AuraEnabled
    public static List<Map<String, Object>> findOpportunityCloseDateTime(List<String>
emails) {
    List<Map<String, Object>> ret = new List<Map<String, Object>>();
    List<OpportunityContactRole> contacts =
           [SELECT Opportunity.Name, Opportunity.CloseDate
            FROM OpportunityContactRole
            WHERE isPrimary=true AND Contact.Email IN :emails];
    for (OpportunityContactRole c: contacts) {
            Map<String, Object> item = new Map<String, Object>();
            item.put('name', c.Opportunity.Name);
            item.put('closesIn',
                     System.Date.today().daysBetween(
                         Date.valueOf(c.Opportunity.CloseDate)));
            ret.add(item);
    }
```

```
    return ret;
  }
}
```

```
({
/*
This JavaScript controller is called on component initialization and relies
on the helper functionality to build a list of email addresses from the
available people. It then makes a caller to the server to run the actions to
display information.
Once the server returns the values, it sets the appropriate values to display
on the client side.
*/
    handlePeopleChange: function(component, event, helper){
            var people = component.get("v.people");
            var peopleEmails = helper.filterEmails(people);
            var action = component.get("c.findOpportunityCloseDateTime");
            action.setParam("emails", peopleEmails);

            action.setCallback(this, function(response){
            var state = response.getState();
            if(state === "SUCCESS"){
               component.set("v.opportunities", response.getReturnValue());
            } else{
               component.set("v.opportunities",[]);
            }
});
            $A.enqueueAction(action);
            var action = component.get("c.findAccountAges");
            action.setParam("emails", peopleEmails);

            action.setCallback(this, function(response){
            var state = response.getState();
            if(state === "SUCCESS"){
               component.set("v.accounts", response.getReturnValue());
            } else{
               component.set("v.accounts",[]);
            }
});
$A.enqueueAction(action);
}
})
```

```
({
    /*
    This helper function filters emails from objects.
    */
    filterEmails : function(people){
            return this.getEmailsFromList(people.to).concat(
                this.getEmailsFromList(people.cc));
    },
```

```
    getEmailsFromList : function(list){
            var ret = [];
            for (var i in list) {
            ret.push(list[i].email);
    }
     return ret;
  }
})
```

カスタムコンポーネントの動的選択リストの作成

Lightningアプリケーションビルダーでコンポーネントが設定されるときに、コンポーネントのプロパティを選択リストとして公開できます。選択リストの値は、作成する Apex クラスによって提供されます。

たとえば、カスタムお知らせレコードを表示するホームページのコンポーネントを作成するとします。Apexクラスを使用して、Lightningアプリケーションビルダーのコンポーネントプロパティの選択リストにすべてのお知らせレコードのタイトルを表示できます。これで、システム管理者がホームページにコンポーネントを追加するときに、ページに配置する適切なお知らせを簡単に選択できます。

1. 選択リストのデータソースとして使用するカスタム Apex クラスを作成します。Apex クラスは、`VisualEditor.DynamicPickList` 抽象クラスを拡張する必要があります。
2. カスタム Apex クラスをデータソースとして指定するデザインファイルに属性を追加します。

次に、簡単な例を示します。

Apex クラスの作成

```
global class MyCustomPickList extends VisualEditor.DynamicPickList{

    global override VisualEditor.DataRow getDefaultValue(){
        VisualEditor.DataRow defaultValue = new VisualEditor.DataRow('red', 'RED');
        return defaultValue;
    }
    global override VisualEditor.DynamicPickListRows getValues() {
        VisualEditor.DataRow value1 = new VisualEditor.DataRow('red', 'RED');
        VisualEditor.DataRow value2 = new VisualEditor.DataRow('yellow', 'YELLOW');
       VisualEditor.DynamicPickListRows  myValues = new VisualEditor.DynamicPickListRows();

        myValues.addRow(value1);
        myValues.addRow(value2);
        return myValues;
    }
}
```

メモ: `VisualEditor.DataRow` では、任意のオブジェクトをその値として指定できますが、文字列属性にはデータソースのみを指定できます。`isValid()` と `getLabel()` のデフォルトの実装では、パラメータで渡されるオブジェクトが比較のための文字列であることを前提としています。

`VisualEditor.DynamicPickList` 抽象クラスについての詳細は、『Apex 開発者ガイド』を参照してください。

デザインファイルへの Apex クラスの追加

既存のコンポーネントのデータソースとして Apex クラスを指定するには、Apex 名前空間と Apex クラス名で構成される値がある属性にデータソースプロパティを追加します。

```
<design:component>
        <design:attribute name="property1" datasource="apex://MyCustomPickList"/>
</design:component>
```

動的選択リストの考慮事項

- Apex データソースを public として指定しても、管理パッケージでは考慮されません。Apex クラスが public で、管理パッケージに含まれている場合、登録者組織のカスタムコンポーネントのデータソースとして使用できます。
- Apex クラスがデータソースとして使用されている場合、Apex クラスのプロファイルアクセスは考慮されません。システム管理者のプロファイルに Apex クラスへのアクセス権はないが、カスタムコンポーネントへのアクセス権がある場合、システム管理者の Lightning アプリケーションビルダーには、コンポーネントの Apex クラスによって提供される値が表示されます。

カスタム Lightning ページテンプレートコンポーネントの作成

すべての標準Lightningページは、ページの領域とページに含まれるコンポーネントを定義するデフォルトテンプレートコンポーネントに関連付けられています。カスタム Lightning ページテンプレートコンポーネントでは、定義した構造とコンポーネントを使用して、ビジネスニーズに合ったページテンプレートを作成できます。カスタムテンプレートを実装すると、ページ作成者は Lightning アプリケーションビルダーの新規ページウィザードでそのカスタムテンプレートを使用できます。

メモ: Lightning アプリケーションビルダーでカスタムテンプレートコンポーネントを使用するには、組織で [私のドメイン] が有効になっている必要があります。

カスタムLightningページテンプレートコンポーネントは、レコードページ、アプリケーションページ、ホームページでサポートされています。テンプレートコンポーネントが実装する必要のあるインターフェースは、ページ種別によって異なります。

- `lightning:appHomeTemplate`
- `lightning:homeTemplate`
- `lightning:recordHomeTemplate`

重要: 各テンプレートコンポーネントでは、1 つのテンプレートインターフェースのみを実装する必要があります。テンプレートコンポーネントで `flexipage:availableForAllPageTypes` や `force:hasRecordId` などの他のインターフェース型を実装しないでください。テンプレートコンポーネントでは、通常のLightningコンポーネントのようにマルチタスクを実行できません。これはテンプレートまたはそれ以外になります。

1.テンプレートコンポーネント構造の構築

カスタムテンプレートは、1 つの .cmp リソースとデザインリソースが含まれている必要のある Lightning コンポーネントバンドルです。.cmp リソースでは、テンプレートインターフェースを実装し、テンプレート領域ご

とに Aura.Component[] 型の属性を宣言する必要があります。Aura.Component[] 型では、コンポーネントのコレクションとして属性を定義します。

メモ: Aura.Component[] 属性は、デザインリソースの領域としても指定されている場合にのみ領域として解釈されます。

次に、lightning:layout コンポーネントと Salesforce Lightning Design System (SLDS) を使用してスタイルを設定する 2 列のアプリケーションページテンプレートの .cmp リソースの例を示します。

テンプレートがデスクトップで表示される場合、右列が 30% (4 つの SLDS 列)、左列がページ幅の残りの 70% を占めます。デスクトップ以外のフォーム要素以外の場合、列は 50/50 で表示されます。

```
<aura:component implements="lightning:appHomeTemplate" description="Main column
 and right sidebar. On a phone, the regions are of equal width">
    <aura:attribute name="left" type="Aura.Component[]" />
    <aura:attribute name="right" type="Aura.Component[]" />

    <div>
        <lightning:layout horizontalAlign="spread">
            <lightning:layoutItem flexibility="grow"
                                  class="slds-m-right_small">
                {!v.left}
            </lightning:layoutItem>
            <lightning:layoutItem size="{! $Browser.isDesktop ? '4' : '6' }"
                                  class="slds-m-left_small">
                {!v.right}
            </lightning:layoutItem>
        </lightning:layout>
    </div>

</aura:component>
```

aura:component タグの description 属性は、省略可能ですが推奨属性です。description を定義すると、Lightning アプリケーションビルダーの新規ページウィザードのテンプレート画像の下にテンプレートの説明として表示されます。

2. デザインリソースでのテンプレート領域とコンポーネントの設定

デザインリソースでは、テンプレートを使用するページに含める必要がある領域やそれらの領域に配置できるコンポーネントの種類を指定して、テンプレートで構築できるページの種類を制御します。

領域は、.cmp リソースの設定に従って、ページ全体に設定されたインターフェースの割り当てを継承します。

次のタグを使用して、領域とコンポーネントを指定します。

flexipage:template

このタグには属性はなく、flexipage:region タグのラッパーとして動作します。テキストリテラルは使用できません。

flexipage:region

このタグには次の属性があり、テンプレートの領域を定義します。テキストリテラルは使用できません。

属性	説明
name	Aura.Component[] 型としてマークされる .cmp リソースの属性の名前。領域として属性にフラグを設定します。
defaultWidth	領域のデフォルトの幅を指定します。この属性はすべての領域で必須です。有効な値は、Small、Medium、Large、Xlarge です。

flexipage:formFactor

このタグを使用して、表示されるデバイス種別に基づいてページ上のコンポーネントの占有スペースを指定します。このタグは、flexipage:region タグの子として指定する必要があります。flexipage:region ごとに複数の flexipage:formFactor タグを使用して、フォーム要素の柔軟な動作を定義します。

属性	説明
type	テンプレートが表示されるフォーム要素またはデバイス (デスクトップやタブレットなど) の種別。有効な値は、Medium (タブレット) と Large (デスクトップ) です。Small フォーム要素 (電話) の適切な幅の値は Small のみなので、Small 種別を指定する必要はありません。その関連付けは、Salesforce によって自動的に行われます。
width	この領域のコンポーネントが表示される領域の使用可能なサイズ。有効な値は、Small、Medium、Large、Xlarge です。

たとえば、このコードスニペットでは、テンプレートがLargeのフォーム要素に表示されるときに領域の表示幅は Large になり、テンプレートが Medium のフォーム要素に表示されるときに領域の表示幅が Small になります。

```
        <flexipage:region name="right" defaultWidth="Large">
            <flexipage:formFactor type="Large" width="Large" />
            <flexipage:formFactor type="Medium" width="Small" />
        </flexipage:region>
```

ヒント: lightning:flexipageRegionInfo サブコンポーネントを使用して、領域の幅情報をコンポーネントに渡すことができます。これにより、表示される領域サイズに基づいて、ページコンポーネントの異なる表示を設定できます。

次に、サンプル .cmp リソースで使用するデザインファイルを示します。デザインファイルの表示ラベルテキストは、新規ページウィザードのテンプレートの名前として表示されます。

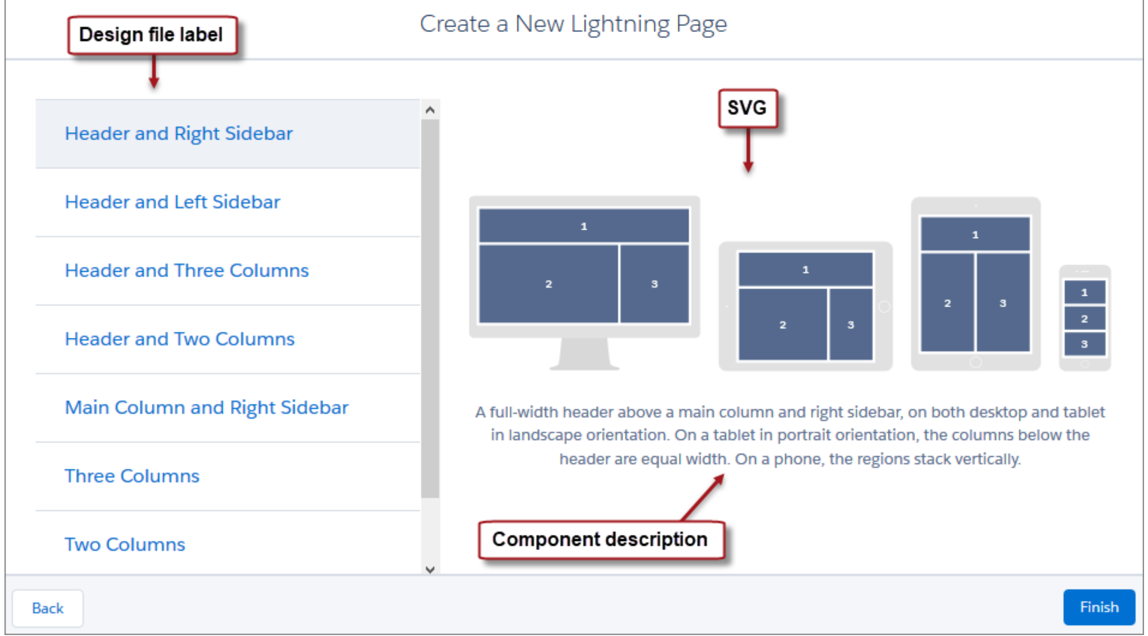
```
<design:component label="Two Column Custom App Page Template">
    <flexipage:template >
        <!-- The default width for the "left" region is "MEDIUM". In tablets,
        the width is "SMALL" -->
            <flexipage:region name="left" defaultWidth="MEDIUM">
                <flexipage:formfactor type="MEDIUM" width="SMALL" />
            </flexipage:region>
            <flexipage:region name="right" defaultWidth="SMALL" />
```

```
        </flexipage:template>
</design:component>
```

3.(省略可能) テンプレート画像の追加

.cmp リソースに description を追加した場合、ユーザが Lightning アプリケーションビルダーの新規ページウィザードでテンプレートを選択したときに説明とテンプレート画像の両方が表示されます。

SVG リソースを使用して、カスタムテンプレート画像を定義できます。



SVG リソースのサイズは 150KB 以下、高さは 160px 以下、幅は 560px 以下にすることをお勧めします。

関連トピック:

Lightning ページテンプレートコンポーネントのベストプラクティス

lightning:flexipageRegionInfo による Lightning ページコンポーネントでの幅の認識

Lightning コンポーネント開発者ガイド: `lightning:layout`

Lightning ページテンプレートコンポーネントのベストプラクティス

Lightning ページテンプレートコンポーネントを作成する場合、次のベストプラクティスと制限に留意してください。

- カスタム背景スタイル設定をテンプレートコンポーネントに追加しないでください。Salesforce の Lightning Experience ページのテーマに干渉します。
- スクロール領域をテンプレートコンポーネントに含めると、Lightning アプリケーションビルダーで表示しようとするときに問題が発生する可能性があります。

- カスタムテンプレートは拡張したり、拡張可能にしたりできません。また、他の場所からテンプレートを拡張することも、テンプレートから他の要素を拡張することもできません。
- 実行時ではなく設計時に getter を使用して領域を変数として取得できます。次に、その例を示します。

```
<aura:component implements="lightning:appHomeTemplate">
    <aura:attribute name="region" type="Aura.Component[]" />
    <aura:handler name="init" value="{!this}" action="{!c.init}" />

    <div>
        {!region}
    </div>

</aura:component>
```

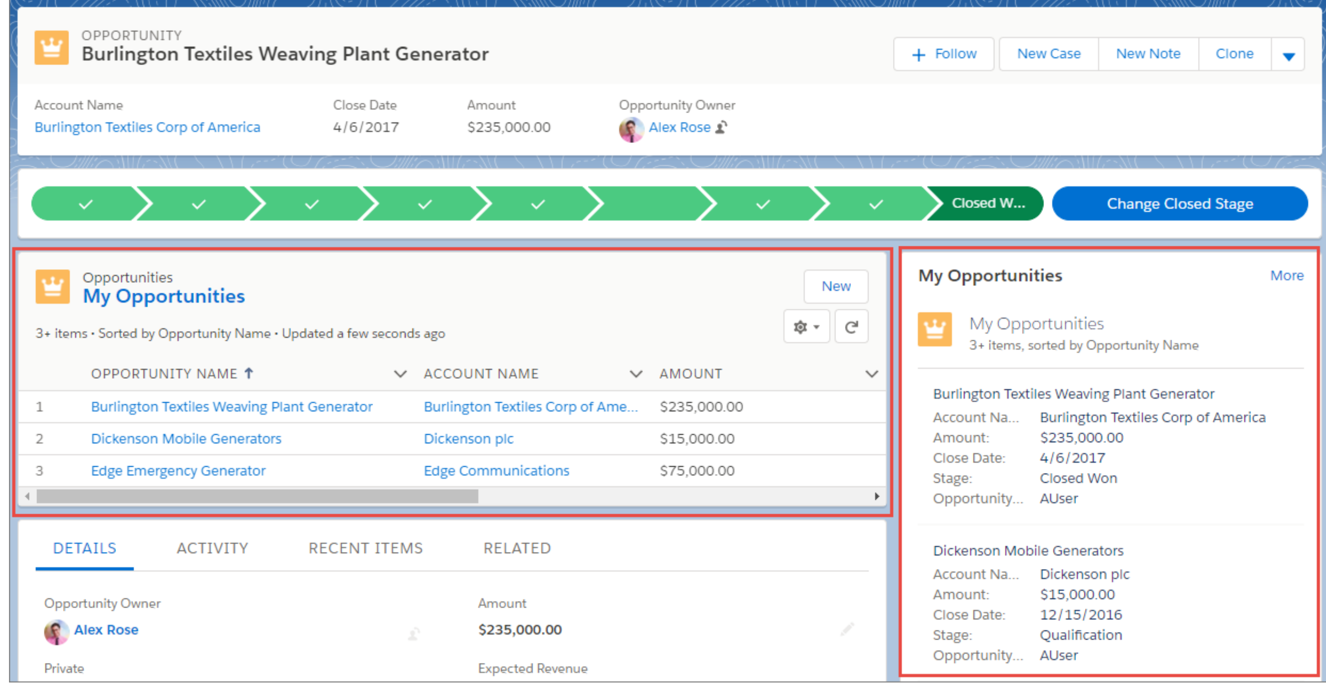
```
{
    init : function(component, event, helper) {
        var region = cmp.get('v.region'); // This will fail at run time.
        ...
    }
}
```

- Lightning ページで使用されておらず、access=global に設定されていなければ、テンプレートから領域を削除できます。領域はいつでも追加できます。
- 領域はコードで複数回使用できますが、実行時に表示される領域のインスタンスは 1 つのみになるようにする必要があります。
- テンプレートコンポーネントには最大 25 個の領域を含めることができます。

`lightning:flexipageRegionInfo` による Lightning ページコンポーネントでの幅の認識

Lightning アプリケーションビルダーでページの範囲にコンポーネントを追加するときに、`lightning:flexipageRegionInfo` サブコンポーネントが親コンポーネントにその範囲の幅を渡します。`lightning:flexipageRegionInfo` といくつかの戦略的 CSS を使用すれば、実行時、範囲ごとに異なる表示で親コンポーネントに表示できます。

たとえば、リストビューコンポーネントは幅認識コンポーネントであるため、小さい範囲と大きい範囲で異なって表示されます。



有効な範囲幅の値は、Small、Medium、Large、Xlarge です。

CSS を使用してコンポーネントをスタイル設定し、コンポーネントが表示される方法を判断できます。次に例を示します。

この単純なコンポーネントには、field1 と field2 の 2 つの項目があります。このコンポーネントが Small の範囲内にない場合、その範囲の幅の 50% を使用して項目が横並びで表示されます。このコンポーネントが Small の範囲内にある場合、その範囲の幅の 100% を使用して項目がリストとして表示されます。

```
<aura:component implements="flexipage:availableForAllPageTypes">
    <aura:attribute name="width" type="String"/>
    <lightning:flexipageRegionInfo width="{!v.width}"/>
    <div class="{! 'container' + (v.width=='SMALL'?' narrowRegion':'')}">
        <div class="{! 'eachField f1' + (v.width=='SMALL'?' narrowRegion':'')}">
            <lightning:input name="field1" label="First Name"/>
        </div>
        <div class="{! 'eachField f2' + (v.width=='SMALL'?' narrowRegion':'')}">
            <lightning:input name="field2" label="Last Name"/>
        </div>
    </div>
</aura:component>
```

このコンポーネントで使用する CSS ファイルを次に示します。

```
.THIS .eachField.narrowRegion{
    width:100%;
}
.THIS .eachField{
    width:50%;
    display:inline-block;
}
```

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定に関するヒントおよび考慮事項

Lightning ページおよび Lightning アプリケーションビルダーのコンポーネントおよびコンポーネントバンドルを作成する場合、次のガイドラインを参考にしてください。

メモ: コンポーネントなどのリソースを `access="global"` としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織の Lightning アプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

コンポーネント

- デザインファイルの要素で `label` 属性を使用して、`<design:component label="foo">` などのわかりやすい名前をコンポーネントに設定します。
- 表示領域の 100% の幅 (余白を含む) を占めるようにコンポーネントを設計します。
- ユーザ操作を必要とする場合、宣言型ツールでコンポーネントの適切なプレースホルダ動作を指定する必要があります。
- コンポーネントに空白のボックスが表示されないようにする必要があります。他のサイトがどのように動作するかを考えます。たとえば、Facebook では、実際のフィード項目がサーバから返されるまでフィードの概要が表示されます。これにより、UI 応答のユーザの認識が向上します。
- 起動されたイベントにコンポーネントが連動する場合は、イベントが起動される前に表示するデフォルトの状態を指定します。
- Lightning Experience のスタイル設定および Salesforce Design System と一貫性のある方法で、コンポーネントのスタイルを設定します。
- 組織で [私のドメイン] が有効になっていない場合、カスタムコンポーネントが含まれる Lightning ページを有効化すると、コンポーネントは実行時にページからドロップされます。

属性

- デザインファイルを使用して、Lightning アプリケーションビルダーに公開する属性を制御します。
- システム管理者にとって使いやすくわかりやすい属性にします。SOQL クエリ、JSON オブジェクト、Apex クラス名は公開しません。
- 必須属性にはデフォルト値を指定します。デフォルト値のない必須属性を持つコンポーネントをアプリケーションビルダーに追加すると、無効と表示され、ユーザの操作性が低下します。
- 公開される属性には、サポートされる基本のデータ型 (string、integer、boolean) を使用します。
- `<design:attribute>` 要素の整数属性に最小値と最大値を指定して、有効な値範囲を制御します。
- 文字列属性では、事前定義された一連の値を持つデータソースを指定して、属性の設定を選択リストとして公開できます。
- すべての属性に、わかりやすい表示名を使用した表示ラベルを指定します。
- 説明を提供して、データ形式や予期される値範囲など、予期されるデータおよびガイドラインを説明します。説明テキストは、プロパティエディタにツールチップとして表示されます。

- flexipage:availableForAllPageTypes または forceCommunity:availableForAllPageTypes インターフェースを実装するコンポーネントの設計属性を削除するには、まずコンポーネントからインターフェースを削除した後、設計属性を削除します。その後でインターフェースを再実装します。コンポーネントが Lightning ページで参照されている場合は、変更する前にページからコンポーネントを削除する必要があります。

制限事項

Lightning アプリケーションビルダー は、Map 型、Object 型、または java:// 複合型をサポートしません。

関連トピック:

Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定
Lightning Experience のレコードホームページのコンポーネントの設定

コミュニティビルダーでの Lightning コンポーネントの使用

コミュニティビルダーでカスタムLightningコンポーネントを使用するには、コンポーネントとそのコンポーネントバンドルの互換性が確保されるように設定する必要があります。

このセクションの内容:

コミュニティのコンポーネントの設定
カスタム Lightning コンポーネントを、コミュニティビルダー の [Lightning コンポーネント] ペインでドラッグアンドドロップできるようにします。

コミュニティ用のカスタムテーマレイアウトコンポーネントの作成
カスタムテーマレイアウトを作成して、カスタマーサービス (Napili) テンプレートのページの外観および全体的な構造を変換します。

コミュニティ用のカスタムの検索およびプロファイルメニューコンポーネントの作成
コミュニティビルダーで、カスタマーサービス (Napili) テンプレートの標準 [プロファイルヘッダー] コンポーネントや [検索パブリッシャーと投稿パブリッシャー] コンポーネントと交換するカスタムコンポーネントを作成します。

コミュニティのカスタムコンテンツレイアウトコンポーネントの作成
コミュニティビルダーには、比率 2:1 の 2 列レイアウトなど、ページのコンテンツ領域を定義するレイアウトがいくつか用意され、すぐに使用できます。ただし、コミュニティ用にカスタマイズされたレイアウトが必要な場合は、カスタムコンテンツレイアウトコンポーネントを作成し、それをコミュニティビルダーで新規ページを作成するときに使用します。コミュニティテンプレートに付属するデフォルトページのコンテンツレイアウトを更新することもできます。

コミュニティのコンポーネントの設定

カスタム Lightning コンポーネントを、コミュニティビルダー の [Lightning コンポーネント] ペインでドラッグアンドドロップできるようにします。

新規インターフェースをコンポーネントに追加する

コミュニティビルダーに表示するには、コンポーネントに forceCommunity:availableForAllPageTypes インターフェースを実装する必要があります。

シンプルな「Hello World」コンポーネントのサンプルコードを次に示します。

```
<aura:component implements="forceCommunity:availableForAllPageTypes" access="global">
    <aura:attribute name="greeting" type="String" default="Hello" access="global" />
    <aura:attribute name="subject" type="String" default="World" access="global" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```

メモ: コンポーネントなどのリソースを access="global" としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織の Lightning アプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

次に、デザインリソースをコンポーネントバンドルに追加します。デザインリソースには、Lightning コンポーネントの設計時の動作 (ページまたはアプリケーションへのコンポーネントの追加を可能にするためにビジュアルツールが必要とする情報) が記述されます。システム管理者がコミュニティビルダーで編集できる属性が含まれています。

このリソースの追加は、Lightning アプリケーションビルダーでのリソースの追加と似ています。詳細は、「Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定」を参照してください。

重要: コミュニティにカスタムコンポーネントを追加すると、ゲストユーザプロファイルに設定したオブジェクトレベルセキュリティと項目レベルセキュリティ (FLS) をスキップできます。Lightning コンポーネントでは、オブジェクトを参照したり、Apex コントローラからオブジェクトを取得したりするときに、CRUD および FLS が自動的に適用されることはありません。つまり、このフレームワークでは、ユーザに CRUD 権限および FLS 表示権限がないレコードと項目は引き続き表示されます。CRUD と FLS は、Apex コントローラで手動によって適用する必要があります。

関連トピック:

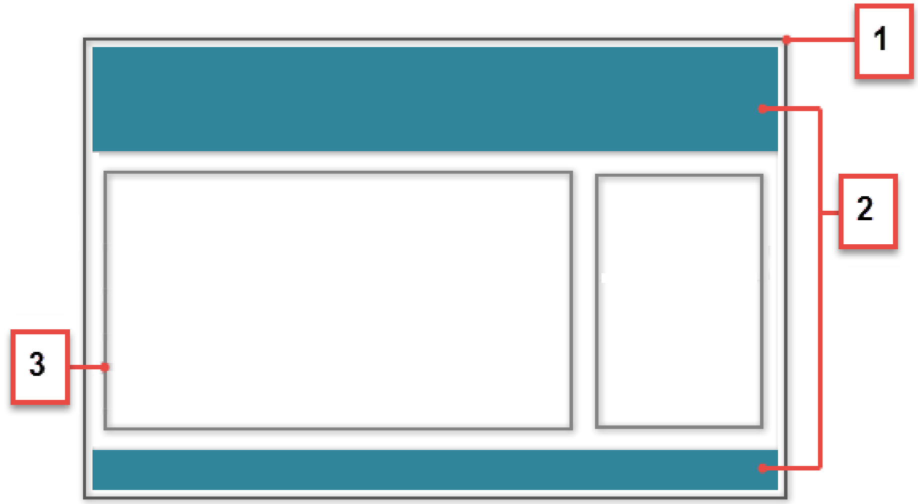
コンポーネントのバンドル

コミュニティの標準設計トークン

コミュニティ用のカスタムテーマレイアウトコンポーネントの作成

カスタムテーマレイアウトを作成して、カスタマーサービス (Napili) テンプレートのページの外観および全体的な構造を変換します。

テーマレイアウトは、コミュニティのテンプレートページの最上位のテンプレート (1) です。共通のヘッダーとフッター (2) が含まれ、多くの場合、ナビゲーション、検索、およびユーザプロファイルメニューが含まれます。一方、コンテンツレイアウト (3) では、2 列レイアウトなど、ページのコンテンツ領域を定義します。



テーマレイアウト種別は、コミュニティ内で同じテーマレイアウトを共有するページを分類します。

開発者コンソールでカスタムテーマレイアウトコンポーネントを作成すると、そのコンポーネントはコミュニティビルダーの [設定] > [テーマ] 領域に表示されます。ここで、コンポーネントに新規または既存のテーマレイアウト種別を割り当てることができます。その後、ページのプロパティでテーマレイアウト種別(つまりテーマレイアウト) を適用します。

1.インターフェースのテーマレイアウトコンポーネントへの追加

テーマレイアウトコンポーネントは、`forceCommunity:themeLayout` インターフェースを実装して、コミュニティビルダーの [設定] > [テーマ] 領域に表示する必要があります。

コードで `{!v.body}` を明示的に宣言して、テーマレイアウトにコンテンツレイアウトが含まれていることを確認します。ページのコンテンツをテーマレイアウト内に表示したい場合は必ず、`{!v.body}` を追加します。

コンポーネントをマークアップ内の領域に追加することも、ユーザがコンポーネントをドラッグアンドドロップできるように領域をオープンにしておくこともできます。`Aura.Component[]` として宣言された属性や、マークアップに含まれる属性はすべて、ユーザがコンポーネントを追加できるテーマレイアウトのオープン領域として表示されます。

カスタマーサービス (Napili) では、テンプレートヘッダーが次のロック済みの領域で構成されます。

- `search` ([検索パブリッシャー] コンポーネントを含む)
- `profileMenu` ([ユーザプロファイルメニュー] コンポーネントを含む)
- `navBar` ([ナビゲーションメニュー] コンポーネントを含む)

[テンプレートのヘッダー] 領域で既存のコンポーネントを再利用するカスタムテーマレイアウトを作成するには、必要に応じて、`search`、`profileMenu`、`navBar` などを属性名の値として宣言します。次に例を示します。

```
<aura:attribute name="navBar" type="Aura.Component[]" required="false" />
```

ヒント: カスタムプロファイルメニューまたは検索コンポーネントを作成する場合、属性名の値を宣言すると、テーマレイアウトを使用するときにユーザもカスタムコンポーネントを選択できます。

以下は、シンプルなテーマレイアウトのサンプルコードです。

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample
 Custom Theme Layout">
    <aura:attribute name="search" type="Aura.Component[]" required="false"/>
    <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
    <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
    <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
    <div>
        <div class="searchRegion">
            {!v.search}
        </div>
        <div class="profileMenuRegion">
            {!v.profileMenu}
        </div>
        <div class="navigation">
            {!v.navBar}
        </div>
        <div class="newHeader">
            {!v.newHeader}
        </div>
        <div class="mainContentArea">
            {!v.body}
        </div>
    </div>
</aura:component>
```

メモ: コンポーネントなどのリソースを `access="global"` としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織のLightningアプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

2.テーマプロパティを含めるためのデザインリソースの追加

コミュニティビルダーでテーマレイアウトプロパティを公開するには、デザインリソースをバンドルに追加します。

次の例では、Small Header というテーマレイアウトに 2 つチェックボックスを追加します。

```
<design:component label="Small Header">
    <design:attribute name="blueBackground" label="Blue Background"/>
    <design:attribute name="smallLogo" label="Small Logo"/>
</design:component>
```

デザインリソースでは、プロパティのみを公開します。コンポーネントでプロパティを実装する必要があります。

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Small
 Header">
    <aura:attribute name="blueBackground" type="Boolean" default="false"/>
    <aura:attribute name="smallLogo" type="Boolean" default="false" />
    ...
```

デザインリソースの名前は、*componentName*`.design` にする必要があります。

3。CSS リソースの追加による重複の問題の回避

必要に応じて、CSS リソースをバンドルに追加して、テーマレイアウトのスタイルを設定します。

ダイアログボックスやフロート表示など、位置付けられた要素の重複の問題を回避する手順は、次のとおりです。

- CSS スタイルを適用します。

```
.THIS {
    position: relative;
    z-index: 1;
}
```

- 要素をカスタムテーマレイアウトの div タグでラップします。

```
<div class="mainContentArea">
    {!v.body}
</div>
```

メモ: カスタムテーマレイアウトの場合、デフォルトで SLDS が読み込まれます。

CSS リソースの名前は *componentName*.css にする必要があります。

関連トピック:

コミュニティ用のカスタムの検索およびプロファイルメニューコンポーネントの作成

forceCommunity:navigationMenuBase

Salesforce ヘルプ: カスタムテーマレイアウトとテーマレイアウト種別

コミュニティ用のカスタムの検索およびプロファイルメニューコンポーネントの作成

コミュニティビルダーで、カスタマーサービス (Napili) テンプレートの標準 [プロファイルヘッダー] コンポーネントや [検索パブリッシャーと投稿パブリッシャー] コンポーネントと交換するカスタムコンポーネントを作成します。

forceCommunity:profileMenuInterface

forceCommunity:profileMenuInterface インターフェースを Lightning コンポーネントに追加して、カスタマーサービス (Napili) コミュニティテンプレートのカスタムプロファイルメニューコンポーネントとして使用できるようにします。作成したカスタムプロファイルメニューコンポーネントは、システム管理者がコミュニティビルダーの [設定] > [テーマ] で選択して、テンプレートの標準 [プロファイルヘッダー] コンポーネントと交換できます。

以下は、シンプルなプロファイルメニューコンポーネントのサンプルコードです。

```
<aura:component implements="forceCommunity:profileMenuInterface" access="global">
    <aura:attribute name="options" type="String[]" default="Option 1, Option 2"/>
    <ui:menu >
```

```
        <ui:menuTriggerLink aura:id="trigger" label="Profile Menu"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
            <aura:iteration items="{!v.options}" var="itemLabel">
                <ui:actionMenuItem label="{!itemLabel}" click="{!c.handleClick}"/>
            </aura:iteration>
        </ui:menuList>
    </ui:menu>
</aura:component>
```

forceCommunity:searchInterface

forceCommunity:searchInterface インターフェースを Lightning コンポーネントに追加して、カスタマーサービス (Napili) コミュニティテンプレートのカスタム検索コンポーネントとして使用できるようにします。作成したカスタム検索コンポーネントは、システム管理者がコミュニティビルダーの [設定] > [テーマ] で選択して、テンプレートの標準 [検索パブリッシャーと投稿パブリッシャー] コンポーネントと交換できます。

以下は、シンプルな検索コンポーネントのサンプルコードです。

```
<aura:component implements="forceCommunity:searchInterface" access="global">
    <div class="search">
        <div class="search-wrapper">
            <form class="search-form">
                <div class="search-input-wrapper">
                    <input class="search-input" type="text" placeholder="My Search"/>
                </div>
                <input type="hidden" name="language" value="en" />
            </form>
        </div>
    </div>
</aura:component>
```

関連トピック:

コミュニティ用のカスタムテーマレイアウトコンポーネントの作成

forceCommunity:navigationMenuBase

Salesforce ヘルプ: カスタムテーマレイアウトとテーマレイアウト種別

コミュニティのカスタムコンテンツレイアウトコンポーネントの作成

コミュニティビルダーには、比率 2:1 の 2 列レイアウトなど、ページのコンテンツ領域を定義するレイアウトがいくつか用意され、すぐに使用できます。ただし、コミュニティ用にカスタマイズされたレイアウトが必要な場合は、カスタムコンテンツレイアウトコンポーネントを作成し、それをコミュニティビルダーで新規ページを作成するときに使用します。コミュニティテンプレートに付属するデフォルトページのコンテンツレイアウトを更新することもできます。

開発者コンソールでカスタムコンテンツレイアウトコンポーネントを作成すると、そのコンポーネントはコミュニティビルダーの [新規ページ] および [レイアウトを変更] ダイアログボックスに表示されます。

1.新規インターフェースをコンテンツレイアウトコンポーネントに追加する

コミュニティビルダーの [新規ページ] および [レイアウトを変更] ダイアログボックスに表示するには、コンテンツレイアウトコンポーネントに forceCommunity:layout インターフェースを実装する必要があります。

シンプルな 2 列のコンテンツレイアウトのサンプルコードを次に示します。

```
<aura:component implements="forceCommunity:layout" description="Custom Content Layout"
access="global">
    <aura:attribute name="column1" type="Aura.Component[]" required="false"></aura:attribute>

    <aura:attribute name="column2" type="Aura.Component[]" required="false"></aura:attribute>


    <div class="container">
        <div class="contentPanel">
            <div class="left">
                {!v.column1}
            </div>
            <div class="right">
                {!v.column2}
            </div>
        </div>
    </div>
</aura:component>
```

メモ: コンポーネントなどのリソースを access="global" としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織の Lightning アプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

2.CSS リソースをコンポーネントバンドルに追加する

次に、必要に応じて CSS リソースを追加してコンテンツレイアウトのスタイルを設定します。

シンプルな 2 列のコンテンツレイアウトのサンプル CSS を次に示します。

```
.THIS .contentPanel:before,
.THIS .contentPanel:after {
    content: " ";
    display: table;
}
.THIS .contentPanel:after {
    clear: both;
}
.THIS .left {
    float: left;
    width: 50%;
}
.THIS .right {
    float: right;
    width: 50%;
}
```

CSS リソースの名前は *componentName*.css にする必要があります。

3.省略可能: SVG リソースをコンポーネントバンドルに追加する

SVG リソースをコンポーネントバンドルに追加して、コミュニティビルダーに表示されるときのコンテンツレイアウトコンポーネントのカスタムアイコンを定義できます。

コミュニティビルダーのコンテンツレイアウトコンポーネントで推奨される画像サイズは、170 x 170 ピクセルです。ただし、画像のサイズが異なる場合、コミュニティビルダーが適合するように画像を拡大縮小します。

SVG リソースの名前は *componentName*.svg にする必要があります。

関連トピック:

コンポーネントのバンドル

コミュニティの標準設計トークン

アプリケーションへのコンポーネントの追加

アプリケーションにコンポーネントを追加する準備ができたら、最初にフレームワークに標準で付属しているコンポーネントを検討します。これらのコンポーネントは、拡張したり、作成するカスタムコンポーネントにコンポジションを使用して追加したりして利用することもできます。

メモ: すべての標準コンポーネントについては、https://<myDomain>.lightning.force.com/auradocs/reference.app (<myDomain> は、Salesforce カスタムドメインの名前) の Components フォルダを参照してください。ui 名前空間には、Web ページでよく使用される多くのコンポーネントが含まれています。

コンポーネントはカプセル化され、内部は非公開に保たれますが、公開形状はコンポーネントのコンシューマから参照できます。この強固な分離により、コンポーネント作成者は自由に内部実装の詳細を変更することができ、コンポーネントのコンシューマはこうした変更から隔離されます。

コンポーネントの公開形状は、設定可能な属性とコンポーネントとやりとりするイベントによって定義されます。公開形状は、基本的には開発者がコンポーネントとやりとりするための API です。新しいコンポーネントを設計するには、公開する属性と、コンポーネントが開始または応答するイベントについて検討します。

新しいコンポーネントの形状を定義したら、複数の開発者が並行してそのコンポーネントを開発できます。これは、チームでアプリケーションを開発する場合に便利なアプローチです。

アプリケーションに新しいカスタムコンポーネントを追加する場合は、「開発者コンソールの使用」 (ページ 7)を参照してください。

関連トピック:

コンポーネントのコンポジション

オブジェクト指向開発の使用

コンポーネントの属性

イベントとの通信

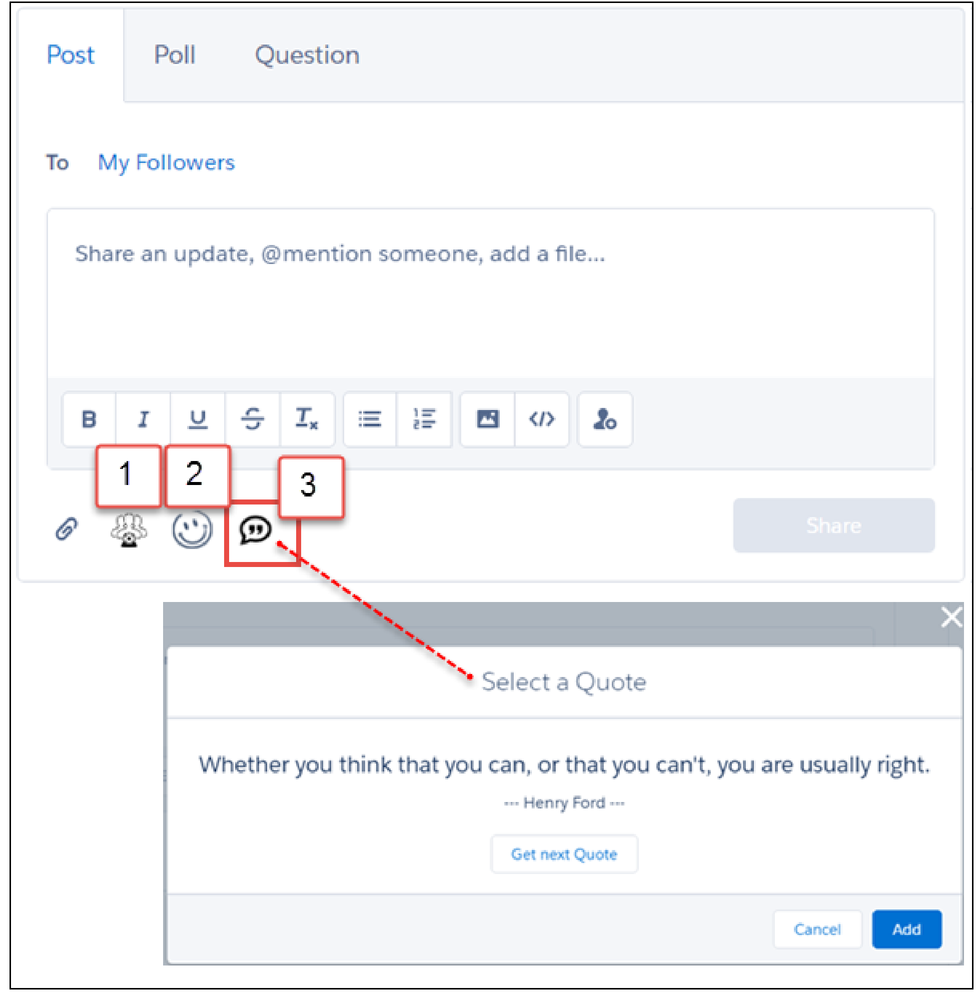
Chatter パブリッシャーへのカスタムアプリケーションの統合

Chatter のリッチパブリッシャーアプリケーション API を使用して、最大 5 個のカスタムアプリケーションを Chatter パブリッシャーに統合します。リッチパブリッシャーアプリケーション API を使用すると、開発者はカスタムペイロードをフィード項目に添付できます。リッチパブリッシャーアプリケーションは、Lightning コンポーネントを使用して構成や表示を行います。インテグレーションに役立つよう、2 つの Lightning インターフェースと 1 つの Lightning イベントが用意されています。アプリケーションをパッケージ化して、AppExchange にアップロードします。コミュニティ管理者ページには、Chatter パブリッシャーに追加する 5 個のアプリケーションを選択するためのセレクタがあります。

`lightning:sendChatterExtensionPayload` イベントと共に、`lightning:availableForChatterExtensionComposer` および `lightning:availableForChatterExtensionRenderer` インターフェースを使用して、カスタムアプリケーションを Chatter パブリッシャーに統合し、アプリケーションのペイロードを Chatter フィードに表示します。

例: **Chatter パブリッシャーと統合されたカスタムアプリケーションの例**

次の例では、3 つのカスタムアプリケーションインテグレーションがある Chatter パブリッシャーを示します。ビデオミーティングアプリケーション (1)、絵文字アプリケーション (2)、毎日の引用句を選択するアプリケーション (3) のアイコンがあります。



Chatter フィード投稿のカスタムアプリケーションペイロードの例

次の例では、Chatter フィードに含まれるカスタムアプリケーションのペイロードを示します。

次のセクションでは、カスタム引用句アプリケーションが Chatter パブリッシャーにどのように統合されたのかを示します。

1.コンポーザコンポーネントの設定

コンポーザコンポーネントでは、コンポーネントファイル、コントローラファイル、ヘルパーファイル、スタイルファイルが作成されています。

次に、quotesCompose.cmp のコンポーネントマークアップを示します。このファイルでは、lightning:availableForChatterExtensionComposer インターフェースが実装されます。

```
<aura:component implements="lightning:availableForChatterExtensionComposer">
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <div class="container">
     <span class="quote" aura:id="quote"></span>
        <span class="author" aura:id="author"></span>
        <ui:button label="Get next Quote" press="{!c.getQuote}"/>
    </div>

</aura:component>
```

コントローラとヘルパーを使用して、コンポーザコンポーネントを初期化し、ソースから引用句を取得します。引用句を取得したら、プラットフォームでアプリケーションのペイロードを、パブリッシャーで作成されるフィード項目に関連付けることができるようにイベント sendChatterExtensionPayload を起動します。

```
getQuote: function(cmp, event, helper) {
    // get quote from the source
    var compEvent = cmp.getEvent("sendChatterExtensionPayload");
    compEvent.setParams({
        "payload" : "<payload object>",
        "extensionTitle" : "<title to use when extension is rendered>",
        "extensionDescription" : "<description to use when extension is rendered>"
    });
    compEvent.fire();
}
```

CSS リソースをコンポーネントバンドルに追加して、コンポジションコンポーネントのスタイルを設定します。

2.レンダラコンポーネントの設定

レンダラコンポーネントでは、コンポーネントファイル、コントローラファイル、スタイルファイルが作成されています。

次に、quotesRender.cmp のコンポーネントマークアップを示します。このファイルでは、lightning:availableForChatterExtensionRenderer インターフェースが実装されます。

```
<aura:component implements="lightning:availableForChatterExtensionRenderer">
    <aura:attribute name="_quote" type="String"/>
    <aura:attribute name="_author" type="String"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>
```

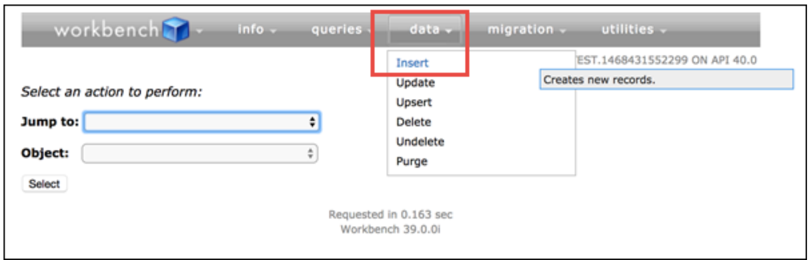
```
    <div class="container">
     <span class="quote" aura:id="quote">{!v._quote}</span>
        <span class="author" aura:id="author">--- {!v._author} ---</span>
    </div>
</aura:component>
```

コントローラを使用して、`lightning:availableForChatterExtensionRenderer` インターフェースによって提供されるペイロードを解析し、属性を設定します。CSS リソースをレンダラバンドルに追加して、レンダラコンポーネントのスタイルを設定します。

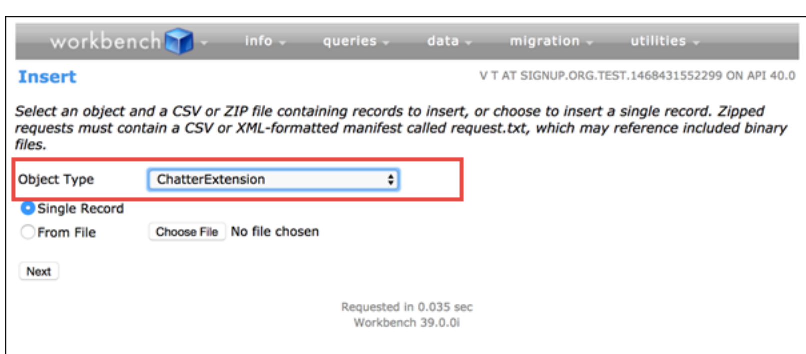
3.新しい ChatterExtension エンティティの設定

これらのコンポーネントの作成後、ワークベンチに移動して組織にログインします。ChatterExtension エンティティを作成します。

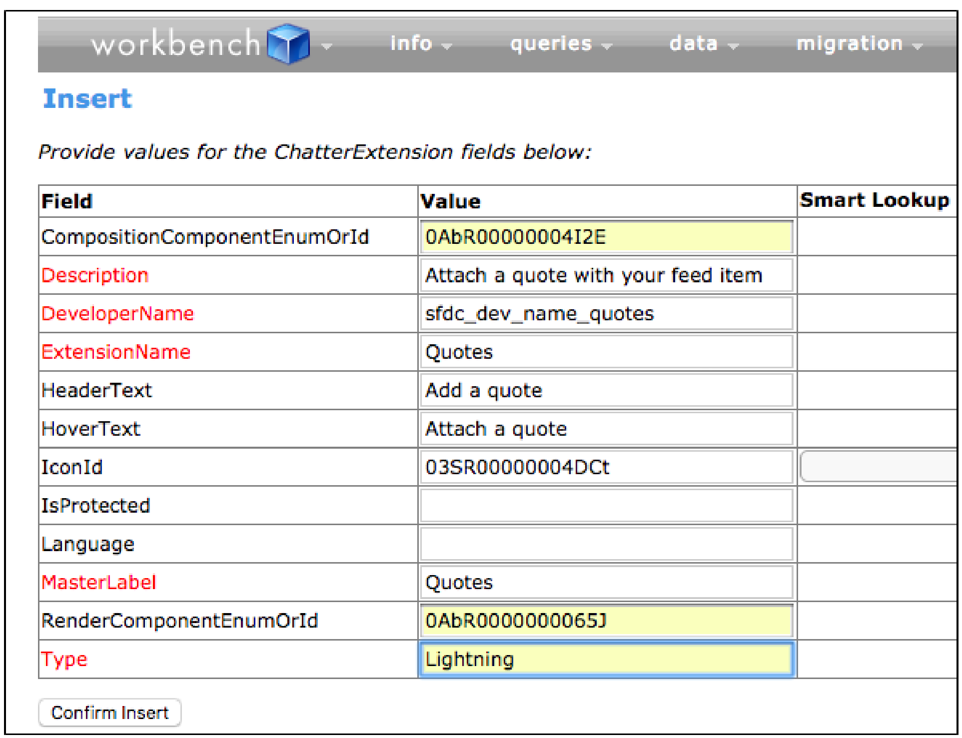
[Data (データ)] メニューから **[Insert (挿入)]** を選択します。



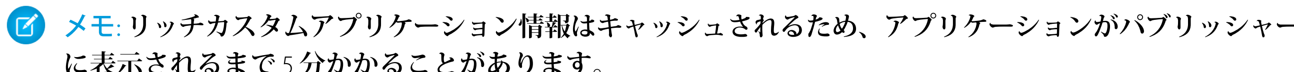
[Object Type (オブジェクト種別)] リストから *`ChatterExtension`* を選択します。



[Value (値)] 列で、`ChatterExtension` 項目の値を指定します (値と説明については「ChatterExtension」を参照)。



作業内容を保存して、Chatterパブリッシャーの投稿ビューおよび質問ビューにカスタムアプリケーションを表示します。



メモ: リッチカスタムアプリケーション情報はキャッシュされるため、アプリケーションがパブリッシャーに表示されるまで 5 分かかることがあります。

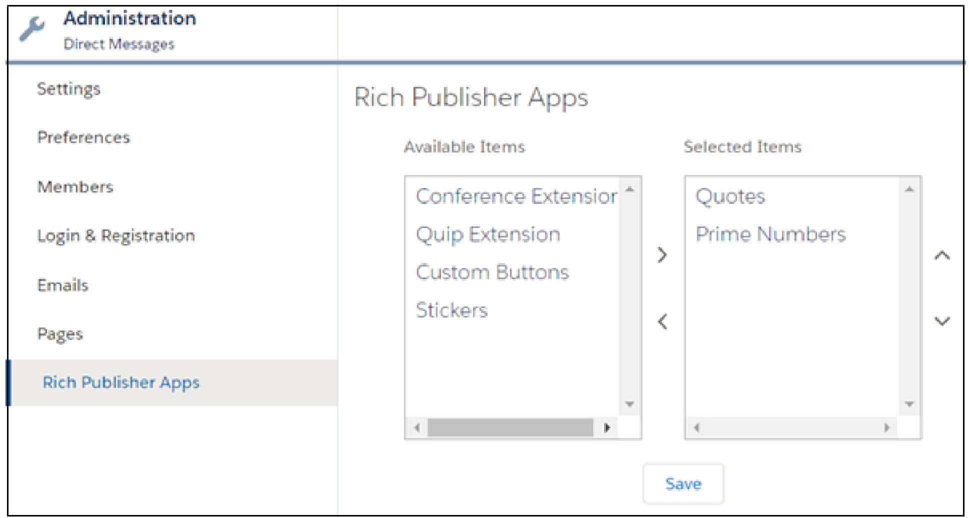
4.アプリケーションのパッケージ化と App Exchange へのアップロード

『ISVforce ガイド』には、アプリケーションのパッケージ化と AppExchange へのアップロードに関する有益な情報が記載されています。

5.Chatter パブリッシャーに埋め込むアプリケーションの選択

堅牢なアプリケーションのカタログがある場合、管理者ページを使用して表示するアプリケーションを選択および配置できます。最大5個のアプリケーションを選択し、任意の順序で配置します。ここで設定する順序によって、パブリッシャーに表示されるアイコンの順序が決まります。

コミュニティで、コミュニティワークスペースに移動して、[管理] ワークスペースをクリックします。[リッチパブリッシャーアプリケーション] をクリックして、ページを開きます。



関連トピック:

lightning:availableForChatterExtensionComposer

lightning:availableForChatterExtensionRenderer

lightning:sendChatterExtensionPayload

Visualforce ページでの Lightning コンポーネントの使用

Lightning コンポーネントを Visualforce ページに追加して、両方のソリューションを使用して作成した機能を組み合わせることができます。Lightning コンポーネントを使用して新機能を実装し、それを既存の Visualforce ページで使用できます。

重要: Visualforce 用 Lightning コンポーネントは、Lightning Out (Lightning コンポーネントをほぼすべての Web ページに埋め込むことができる強力かつ柔軟な機能) に基づいています。Visualforce と共に使用すると、いくつのかの細かな要素が簡略化されます。たとえば、認証を処理したり、接続アプリケーションを設定したりする必要がなくなります。

その他の点では、Visualforce 用の Lightning コンポーネントは Lightning Out と同じように使用できます。詳細は、このガイドの「Lightning Out」セクションを参照してください。

次の 3 ステップで Lightning コンポーネントを Visualforce ページに追加します。

1. `<apex:includeLightning/>` コンポーネントを使用して、Visualforce 用 Lightning コンポーネントの JavaScript ライブラリを Visualforce ページに追加します。
2. コンポーネントの連動関係を宣言する Lightning アプリケーションを作成して参照します。
3. `$Lightning.createComponent()` を使用して、ページにコンポーネントを作成する JavaScript 関数を記述します。

Visualforce 用 Lightning コンポーネントの JavaScript ライブラリの追加

ページの先頭に `<apex:includeLightning/>` を追加します。このコンポーネントは Visualforce 用の Lightning コンポーネントで使用される JavaScript ファイルを読み込みます。

Lightning 連動関係アプリケーションの作成および参照

Visualforce 用の Lightning コンポーネントを使用するには、Lightning 連動関係アプリケーションを参照してコンポーネントの連動関係を定義します。このアプリケーションはグローバルにアクセス可能で、`ltng:outApp` を拡張します。アプリケーションによって、使用するすべての Lightning 定義 (コンポーネントなど) の連動関係が宣言されます。

以下に、`lcvfTest.app` という簡単なアプリケーションの例を示します。このアプリケーションは `<aura:dependency>` タグを使用して、標準 Lightning コンポーネントの `ui:button` を使用することを示します。

```
<aura:application access="GLOBAL" extends="ltng:outApp">
    <aura:dependency resource="ui:button"/>
</aura:application>
```

メモ: `ltng:outApp` から拡張すると、SLDS リソースがページに追加されて、Salesforce Lightning Design System (SLDS) で Lightning コンポーネントのスタイルを設定できるようになります。SLDS リソースをページに追加しない場合、代わりに `ltng:outAppUnstyled` から拡張します。

このアプリケーションをページで参照するには、次の JavaScript コードを使用します。*theNamespace* はアプリケーションの名前空間プレフィックスです。つまり、組織の名前空間か、アプリケーションを提供する管理パッケージの名前空間のいずれかになります。

```
$Lightning.use("theNamespace:lcvfTest", function() {});
```

アプリケーションが組織で定義されている場合 (管理パッケージに含まれていない場合)、次の例に示すように、代わりにデフォルトの「c」名前空間を使用できます。組織で名前空間が定義されていない場合は、デフォルトの名前空間を*使用する必要*があります。

Lightning 連動関係アプリケーションの作成についての詳細は、「Lightning Out の連動関係」を参照してください。

ページでのコンポーネントの作成

最後に、`$Lightning.createComponent(String type, Object attributes, String locator, function callback)` を使用して、最上位コンポーネントをページに追加します。この関数は `$A.createComponent()` に似ていますが、コンポーネントの挿入先となる DOM 要素を指定する追加の `domLocator` パラメータが含まれています。

前の例の `lcvfTest.app` を使用して `ui:button` を作成するサンプル Visualforce ページを見てみましょう。

```
<apex:page>
    <apex:includeLightning />

    <div id="lightning" />

```

```
    <script>
        $Lightning.use("c:lcvfTest", function() {
          $Lightning.createComponent("ui:button",
          { label : "Press Me!" },
          "lightning",
          function(cmp) {
            // do some stuff
          });
        });
    </script>
</apex:page>
```

このコードでは、「lightning」という ID を持つ DOM 要素が作成され、`$Lightning.createComponent()` メソッドで参照されます。このメソッドでは、「Press Me!」と表示される `ui:button` が作成され、コールバック関数が実行されます。

重要: ページ上で `$Lightning.use()` を複数回コールできますが、すべてのコールで同じ Lightning 連動関係アプリケーションを参照する必要があります。

`$Lightning.use()` および `$Lightning.createComponent()` の使用についての詳細は、「Lightning Out のマークアップ」を参照してください。

関連トピック:

Lightning Out の連動関係

Lightning Out を使用した任意のアプリケーションへの Lightning コンポーネントの追加 (ベータ)

Lightning Out のマークアップ

未認証ユーザとの Lightning Out アプリケーションの共有

Lightning Out の考慮事項と制限

Lightning Out を使用した任意のアプリケーションへの Lightning コンポーネントの追加 (ベータ)

Lightning Out を使用して、Salesforce サーバの外部で Lightning コンポーネントアプリケーションを実行します。Heroku、ファイアウォール内の部門サーバ、または SharePoint のいずれで実行される Node.js アプリケーションであっても、Force.com を使用してカスタムアプリケーションを作成してあらゆる場所でアプリケーションを実行できます。

メモ: このリリースには、Lightning Out のベータバージョンが含まれています。ベータバージョンは機能の品質は高いですが、既知の制限があります。Lightning Out についてのフィードバックと提案は、IdeaExchange からお寄せください。

任意の場所にリリースできる Lightning コンポーネントの開発は、Salesforce 内で実行するものとほぼ同じです。Lightning コンポーネントの開発についてすでに知っていることは、そのまま適用できます。唯一の実質的な違いは、Lightning コンポーネントアプリケーションをリモート Web コンテナ (*発信元サーバ*) に埋め込む方法です。

Lightning Out は、発信元サーバのページに含める JavaScript ライブラリとして、そして Lightning コンポーネントアプリケーションを設定し有効化するために追加するマークアップとして、外部アプリケーションに追加されます。初期化が完了すると、Lightning Out は Lightning コンポーネントアプリケーションをセキュアな接続上で取り込み、設定し、実行するページの DOM に挿入します。それが完了すると、「通常」の Lightning コンポーネントコードが後を引き継ぎ、実行します。

メモ: この方法は、iframe を使用したアプリケーションの埋め込みとは大きく異なります。Lightning Out によって実行される Lightning コンポーネントはページ上で完全に機能します。必要に応じて、Lightning コンポーネントアプリケーションと、埋め込み先のページまたはアプリケーションとのインタラクションも有効にできます。このインタラクションは、Lightning イベントを使用して処理されます。

Salesforce と発信元サーバの間のセキュアな接続を有効にするには、簡単なマークアップの他に、Salesforce 内での若干の設定と準備が必要です。さらに、発信元サーバはアプリケーションをホストしているため、認証を独自のコードで管理する必要があります。

この設定プロセスは、Force.com REST API を使用して Salesforce に接続するアプリケーションに対するプロセスと似ており、同程度の作業が必要です。

このセクションの内容:

Lightning Out の要件

Lightning Out を使用して Lightning コンポーネントをリリースするには、接続とセキュリティを確保するための若干の要件があります。

Lightning Out の連動関係

特別な Lightning 連動関係アプリケーションを作成して、Lightning Out またはVisualforce の Lightning コンポーネントを使用してリリースされる Lightning コンポーネントアプリケーションのコンポーネント連動関係を説明します。

Lightning Out のマークアップ

Lightning Out では、ページ上にシンプルなマークアップがいくつか必要です。また、有効化には簡単な JavaScript 関数を 2 つ使用します。

Lightning Out からの認証

Lightning Out では認証は処理されません。代わりに、Lightning Out アプリケーションを初期化するときに、手動で Salesforce セッション ID または認証トークンを入力します。

未認証ユーザとの Lightning Out アプリケーションの共有

Lightning Out 連動関係アプリケーションに `ltng:allowGuestAccess` インターフェースを追加すると、ユーザは Salesforce の認証を行わずにそのアプリケーションにアクセスできます。このインターフェースでは、Lightning コンポーネントを含むアプリケーションを作成し、あらゆる場所のすべてのユーザにリリースできます。

Lightning Out の考慮事項と制限

Lightning Outを使用したアプリケーションの作成の大部分は、Lightningコンポーネントを使用したアプリケーションの作成とよく似ています。ただし、コンポーネントは Salesforce の「外部」で実行されるため、いくつかの問題を認識しておく必要があります。そして、コンポーネントやアプリケーションに変更を加える必要があることもあります。

関連トピック:

Idea Exchange: Lightning Components Anywhere / Everywhere (どこでも使える Lightning コンポーネント)

Lightning Out の要件

Lightning Out を使用して Lightning コンポーネントをリリースするには、接続とセキュリティを確保するための若干の要件があります。

リモート Web コンテナ (*発信元サーバ*) は、次のものをサポートしている必要があります。

- クライアントブラウザに提供されるマークアップ (HTML と JavaScript の両方) を変更する機能。Lightning Out マークアップを追加できる必要があります。
- 有効な Salesforce セッション ID を取得する機能。このためには、多くの場合、発信元サーバ用の接続アプリケーションを設定する必要があります。
- Salesforce インスタンスにアクセスする機能。たとえば、発信元サーバがファイアウォールの内側にある場合、インターネットにアクセスして少なくとも Salesforce に到達する権限が必要です。

Salesforce 組織は、次のものを許可するように設定されている必要があります。

- 発信元サーバが認証と接続を行う機能。このためには、多くの場合、発信元サーバ用の接続アプリケーションを設定する必要があります。
- 発信元サーバがクロスオリジンリソース共有 (CORS) ホワイトリストに追加されている必要があります。

最後に、発信元サーバ上でホストされるLightning コンポーネントの連動関係情報が含まれる特別な Lightning コンポーネントアプリケーションを作成します。このアプリケーションはLightning OutまたはVisualforceのLightning コンポーネントによってのみ使用されます。

Lightning Out の連動関係

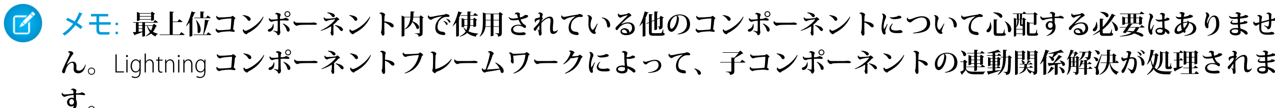
特別な Lightning 連動関係アプリケーションを作成して、Lightning Out またはVisualforce の Lightning コンポーネントを使用してリリースされるLightningコンポーネントアプリケーションのコンポーネント連動関係を説明します。

Lightning Out を使用して Lightning コンポーネントアプリケーションが初期化されるときに、Lightning Out はアプリケーション内のコンポーネントの連動関係を読み込みます。これを効率的に行うには、事前にコンポーネントの連動関係を指定して、Lightning Out の起動時に定義を一度に読み込めるようにする必要があります。

連動関係を指定するメカニズムが、*Lightning 連動関係アプリケーション*です。連動関係アプリケーションは、単なる `<aura:application>` で、いくつかの属性と `<aura:dependency>` タグによる連動関係コンポーネントの説明が含まれています。Lightning連動関係アプリケーションは、ユーザが直接使用するために実際にリリースされるアプリケーションではありません。**Lightning**連動関係アプリケーションは、**Lightning Out**の連動

関係を指定するためだけに使用します(または、Lightning Out を内部で使用するVisualforce の Lightning コンポーネントのために使用)。

基本的な Lightning 連動関係アプリケーションは、次のようになります。



```
<aura:application access="GLOBAL" extends="ltng:outApp">
    <aura:dependency resource="c:myAppComponent"/>
</aura:application>
```

Lightning 連動関係アプリケーションは、次の操作を実行する必要があります。

- アクセス制御を `GLOBAL` に設定する。
- `ltng:outApp` または `ltng:outAppUnstyled` から拡張する。
- `$Lightning.createComponent()` へのコールで参照されているすべてのコンポーネントを連動関係としてリストする。

この例では、`<c:myAppComponent>` は、`$Lightning.createComponent()` を使用して 発信元サーバ 上に作成しようとしている Lightning コンポーネントアプリケーションの最上位コンポーネントです。`$Lightning.createComponent()` を使用してページに追加する異なるコンポーネントについてそれぞれ連動関係を作成します。

メモ: 最上位コンポーネント内で使用されている他のコンポーネントについて心配する必要はありません。Lightning コンポーネントフレームワークによって、子コンポーネントの連動関係解決が処理されます。

スタイル設定の連動関係の定義

Lightning Out アプリケーションをスタイル設定するために、Salesforce Lightning Design System とスタイル設定なしという 2つのオプションがあります。Lightning Design System スタイル設定はデフォルトであり、Lightning Out は、現在のバージョンの Lightning Design System を、Lightning Out を使用しているページに自動的に含めます。Lightning Design System リソースを除外し、スタイルを柔軟に制御し、可能であれば発信元サーバのスタイル設定と一致させるには、`ltng:outApp` ではなく `ltng:outAppUnstyled` から拡張するように連動関係アプリケーションを設定します。

使用上の注意

Lightning 連動関係アプリケーションは、通常の Lightning アプリケーションではないため、そのようには扱わないでください。Lightning 連動関係アプリケーションは、Lightning Out アプリケーションの連動関係を指定するためだけに使用します。

特に、次の点に注意してください。

- Lightning 連動関係アプリケーションにはテンプレートを追加できません。
- Lightning 連動関係アプリケーションのボディに追加するコンテンツは表示されません。

関連トピック:

aura:dependency

アプリケーションでの Salesforce Lightning Design System の使用

Lightning Out のマークアップ

Lightning Out では、ページ上にシンプルなマークアップがいくつか必要です。また、有効化には簡単な JavaScript 関数を 2 つ使用します。

Lightning Out に固有のものは、マークアップと Lightning Out ライブラリの JavaScript 関数のみです。その他のすべては、すでにご存じで愛用されている Lightning コンポーネントコードです。

ページへの Lightning Out の追加

発信元サーバで Lightning Out の使用を有効にするには、Lightning コンポーネントアプリケーションをホストしているアプリケーションまたはページに Lightning Out JavaScript ライブラリを含めます。ライブラリを含めるには、1 行のマークアップが必要です。

```
<script src="https://myDomain.my.salesforce.com/lightning/lightning.out.js"></script>
```

重要: ホストには自分のカスタムドメインを使用します。サンプルソースコードから他者のインスタンスをコピーして貼り付けることはやめてください。これを行うと、Salesforce インスタンスと Lightning Out ライブラリを読み込む元のインスタンスの間でバージョンの不一致があるたびにアプリケーションが破損します。これは、Salesforce の定期アップグレード中に、年間 3 回以上起こっています。そうならないようにしてください。

Lightning コンポーネントアプリケーションの読み込みと初期化

`$Lightning.use()` 関数を使用して、Lightning コンポーネントフレームワークと Lightning コンポーネントアプリケーションの読み込みと初期化を行います。

`$Lightning.use()` 関数は 4 つの引数を取ります。

名前	型	説明
`appName`	string	必須。Lightning 連動関係アプリケーションの名前 (名前空間を含む)。たとえば、「`"c:expenseAppDependencies"`」などです。
`callback`	function	Lightning コンポーネントフレームワークとアプリケーションが完全に読み込まれた後にコールされる関数。コールバックは引数を受信しません。 通常、このコールバックで `$Lightning.createComponent()` をコールしてアプリケーションをページに追加します (次のセクションを参照)。また、別の方法で表示を更新するか、他の方法で Lightning コンポーネントの準備が整ったことに応答する場合もあります。
`lightningEndPointURI`	string	Salesforce インスタンス上の Lightning ドメインの URL。たとえば、「https://*myDomain*.lightning.force.com」などです。

名前	型	説明
`authToken`	string	有効な Salesforce セッションのセッション ID または OAuth アクセストークン。 メモ: 独自のコード内でこのトークンを取得する必要があります。Lightning Out では認証は処理されません。「Lightning Out からの認証」を参照してください。

`appName` は必須項目です。他の 3 つのパラメータは省略可能です。通常の使用では、4 つのパラメータすべてを指定します。

メモ: ページ上で、複数の Lightning 連動関係アプリケーションを使用することはできません。`$Lightning.use()` を複数回コールすることはできますが、どのコールでも同じ連動関係アプリケーションを参照する必要があります。

ページへの Lightning コンポーネントの追加

`$Lightning.createComponent()` 関数を使用して、ページに Lightning コンポーネントを追加し、有効にします。

`$Lightning.createComponent()` 関数は 4 つの引数を取ります。

名前	型	説明
`componentName`	string	必須。ページに追加する Lightning コンポーネントの名前 (名前空間を含む)。たとえば、「`"c:newExpenseForm"`」などです。
`attributes`	Object	必須。作成時にコンポーネントに設定する属性。たとえば、「`{ name: theName, amount: theAmount }`」などです。コンポーネントに属性が必要ない場合は、空のオブジェクト `{ }` を渡します。
`domLocator`	Element または string	必須。作成したコンポーネントを挿入するページ上の位置を示す DOM 要素または要素 ID。
`callback`	function	コンポーネントが追加されてページ上で有効になった後にコールされる関数。コールバックは、作成されたコンポーネントを唯一の引数として受信します。

メモ: ページに複数の Lightning コンポーネントを追加できます。つまり、複数の DOM ロケータを使用して `$Lightning.createComponent()` を複数回コールすることによって、ページの異なる部分にコンポーネントを追加できます。この方法で作成された各コンポーネントは、ページの Lightning 連動関係アプリケーション内で指定する必要があります。

内部的には、`$Lightning.createComponent()` は標準の `$A.createComponent()` 関数をコールします。DOM ロケータ以外の引数は同じです。さらに、コールを Lightning Out セマンティックでラップしている以外は動作も同じです。

関連トピック:

コンポーネントの動的な作成

Lightning Out からの認証

Lightning Out では認証は処理されません。代わりに、Lightning Out アプリケーションを初期化するときに、手動で Salesforce セッション ID または認証トークンを入力します。

Lightning Out で使用する認証トークンを取得する方法は 2 つあります。

- Visualforce ページで、Lightning Components for Visualforce を使用して、式 `{! $Api.Session_ID }` で、現在の Visualforce セッション ID を取得します。このセッションは、Visualforce ページのみで使用するためのものです。
- その他のページでは、認証済みセッションの取得には OAuth を使用し、Force.com REST API で使用する認証済みセッションの取得と同じプロセスに従います。この場合、OAuth トークンを取得して、どのページでも使用できます。

理解しておくべき重要な点は、Lightning Out は認証のためのものではないということです。`$Lightning.use()` 関数は、提供された認証トークンが何であれ、それをセキュリティサブシステムに渡すだけです。ほとんどの組織からの認証トークンはセッション ID または OAuth トークンです。

未認証ユーザとの Lightning Out アプリケーションの共有

Lightning Out 連動関係アプリケーションに `ltng:allowGuestAccess` インターフェースを追加すると、ユーザは Salesforce の認証を行わずにそのアプリケーションにアクセスできます。このインターフェースでは、Lightning コンポーネントを含むアプリケーションを作成し、あらゆる場所のすべてのユーザにリリースできます。

`ltng:allowGuestAccess` インターフェースのある Lightning Out 連動関係アプリケーションは、Lightning Components for Visualforce および Lightning Out で使用できます。

- Lightning Components for Visualforce を使用すると、Lightning アプリケーションを Visualforce ページに追加してから、そのページを Salesforce タブ + Visualforce コミュニティで使用できます。その後、そのページへの公開アクセスを許可できます。
- Lightning Out を使用すると、Lightning Out がサポートされているすべての場所、つまりほぼすべての場所に Lightning アプリケーションをリリースできます。

`ltng:allowGuestAccess` インターフェースは、コミュニティが有効な組織でのみ使用可能で、Lightning Out アプリケーションは組織で定義したすべてのコミュニティエンドポイントに関連付けられています。

重要: `ltng:allowGuestAccess` インターフェースを追加して Lightning アプリケーションをゲストユーザがアクセスできるようにすると、コミュニティで公開アクセスが有効になっているかどうかに関わらず、組織内のすべてのコミュニティからアクセスできます。コミュニティ URL を介したアクセスを防ぐことはできず、一部のコミュニティのみに対してアクセス可能にすることはできません。

警告: アプリケーションのゲストアクセスへの開放は慎重に行ってください。ゲストアクセスが有効なアプリケーションは、コミュニティのゲストユーザプロファイルに設定したオブジェクトレベルセキュリティと項目レベルセキュリティ (FLS) を無視します。Lightning コンポーネントでは、オブジェクトを参照したり、Apex コントローラからオブジェクトを取得したりするときに、CRUD および FLS が自動的に適用されることはありません。つまり、このフレームワークでは、ユーザに CRUD アクセス権および FLS 表示権限がないレコードと項目は引き続き表示されます。CRUD と FLS は、Apex コントローラで手動によって適用する必要があります。ゲストアクセスが有効なアプリケーションで使用されるコード内のエラーによって、組織のデータが世界中に開放されてしまう可能性があります。

Lightning Out Lightning Components for Visualforce

使用方法

まず、`ltng:allowGuestAccess` インターフェースを Lightning Out 連動関係アプリケーションに追加します。この例を次に示します。

```
<aura:application access="GLOBAL" extends="ltng:outApp"
    implements="ltng:allowGuestAccess">

    <aura:dependency resource="c:storeLocatorMain"/>

</aura:application>
```

メモ: 個別のコンポーネントではなく、Lightning アプリケーションのみに `ltng:allowGuestAccess` インターフェースを追加できます。

次に、Lightning Out JavaScript ライブラリをページに追加します。

- Lightning Components for Visualforce では、`<apex:includeLightning />` タグをページの任意の場所に追加するだけです。
- Lightning Out では、コミュニティのエンドポイント URL を使用して、ライブラリを直接参照する `<script>` タグを追加します。この例を次に示します。

```
<script
src="https://yourCommunityDomain/communityURL/lightning/lightning.out.js"></script>
```

例: `https://universalcontainers.force.com/ourstores/lightning/lightning.out.js`

最後に、Lightning アプリケーションを読み込んで有効化する JavaScript コードを追加します。このコードは、エンドポイントに組織のいずれかのコミュニティ URL を使用する必要がある重要な追加が含まれる、標準の Lightning Out です。エンドポイント URL は、`https://yourCommunityDomain/communityURL/` の形式を取ります。次のサンプルでは、関連する行が強調表示されています。

```
<script>
    $Lightning.use("c:locatorApp",     // name of the Lightning app
        function() {                   // Callback once framework and app loaded
            $Lightning.createComponent(
                "c:storeLocatorMain", // top-level component of your app
                { },                  // attributes to set on the component when created
                "lightningLocator",   // the DOM location to insert the component
                function(cmp) {
                    // callback when component is created and active on the page
```

```
                }
            );
        },
        'https://universalcontainers.force.com/ourstores/'  // Community endpoint
    );
</script>
```

関連トピック:

Salesforce ヘルプ*:* コミュニティの作成

Visualforce ページでの Lightning コンポーネントの使用

Lightning Out の考慮事項と制限

Lightning Out を使用したアプリケーションの作成の大部分は、Lightning コンポーネントを使用したアプリケーションの作成とよく似ています。ただし、コンポーネントはSalesforceの「外部」で実行されるため、いくつかの問題を認識しておく必要があります。そして、コンポーネントやアプリケーションに変更を加える必要があることもあります。

認識しておくべき問題は、2 つのカテゴリに分類できます。

Lightning Out の使用に関する考慮事項

Lightning Out アプリケーションは Salesforce コンテナの外部で実行されるため、いくつかのことに留意し、場合によっては対処する必要があります。

1 つ目に、Lightning コンポーネントは、ユーザのブラウザの Cookie 設定に連動します。Lightning Out は Salesforce *の外部*で Lightning コンポーネントを実行するため、これらの Cookie は「サードパーティ」Cookie となります。ユーザは、ブラウザ設定でサードパーティの Cookie を許可する必要があります。

最も影響が大きくて明白な問題は、認証です。認証を処理するSalesforceコンテナがないため、自分で処理する必要があります。この重要なトピックは、「Lightning Out からの認証」で詳しく説明しています。

もう1つの重要な考慮事項は、それほど明白ではありません。アプリケーションでサポートされる多くの重要なアクションは、さまざまなLightningイベントを起動することによって実現されます。ただし、イベントは森の中で倒れる木のようなものです。誰も耳を澄ましていなければ、効果はあるでしょうか?多くの主要なLightningイベントの場合、「リスナー」はLightning ExperienceまたはSalesforce1コンテナである `one.app` です。そして、イベントを処理する `one.app` がなければ、実際にイベントは無効です。それらのイベントを起動しても、エラーを表示せずに失敗します。

標準イベントは、「イベントの参照」に記載されています。Lightning Out での使用がサポートされていないイベントには、次の注意事項があります。

メモ: このイベントは、`one.app` コンテナによって処理されます。Lightning Experience および Salesforce1 でのみサポートされています。

標準コンポーネントの制限事項

Lightning Out の主要機能は安定していて完成していますが、その他の Salesforce 機能とのインタラクションについては、まだ作業中のものがあります。

そのうちで主要なものは、Lightning コンポーネントフレームワークに組み込まれた標準コンポーネントです。Lightning Out、および Lightning Out に基づく Visualforce の Lightning コンポーネントなどの多くの標準コンポーネントは、スタンドアロンコンテキストで使用すると正しく動作しません。これは、コンポーネントが、`one.app` コンテナで使用できるリソースに暗黙的に連動するためです。

所有するコンポーネントのすべての連動関係を明示的にすることで、この問題を避けることができます。`ltng:require` を使用して、コンポーネント自体に埋め込まれていない必要な JavaScript リソースおよび CSS リソースをすべて参照します。

アプリケーションで標準コンポーネントを使用している場合、Lightning Out または Visualforce の Lightning コンポーネント で使用されているときに、スタイルや動作が記載されているとおりでないことがあります。

関連トピック:

Lightning コンポーネントのブラウザサポートの考慮事項
Lightning Out からの認証
システムイベントの参照
Visualforce ページでの Lightning コンポーネントの使用

第 5 章　イベントとの通信

トピック:

- アクションとイベント
- クライアント側コントローラを使用したイベントの処理
- コンポーネントイベント
- アプリケーションイベント
- イベント処理のライフサイクル
- 高度なイベントの例
- 非 Lightning コードからの Lightning イベントの起動
- イベントのベストプラクティス
- 表示ライフサイクル中に起動されたイベント
- Salesforce1 と Lightning Experience で処理されるイベント
- システムイベント

フレームワークでは、イベント駆動型プログラミングが使用されます。ここでは、インターフェースイベントが発生したときに応答するハンドラを記述します。イベントは、ユーザ操作によってトリガされている場合もあれば、それ以外の場合もあります。

Lightning コンポーネントフレームワークでは、JavaScript コントローラのアクションからイベントが起動されます。イベントには、そのイベントの起動前に設定可能で、処理時に読み取り可能な属性を含められます。

イベントは、`.evt` リソースの `aura:event` タグによって宣言され、コンポーネントとアプリケーションのいずれかのタイプを設定できます。

コンポーネントイベント
: コンポーネントイベントは、コンポーネントのインスタンスから起動されます。コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

アプリケーションイベント
: アプリケーションイベントは、従来の公開/登録モデルに従います。アプリケーションイベントは、コンポーネントのインスタンスから起動されます。イベントのハンドラを提供するすべてのコンポーネントに通知されます。

> メモ: 可能な場合は常に、アプリケーションイベントではなくコンポーネントイベントを使用します。コンポーネントイベントを処理できるのは、コンテインメント階層で上位にあるコンポーネントのみであるため、それらのイベントを把握する必要があるコンポーネントにのみ使用が限定されます。アプリケーションイベントは、特定のレコードへの移動など、アプリケーションレベルで処理する必要があるものに適しています。アプリケーションイベントにより、アプリケーションの別々の部分にあって直接的なコンテインメント関係がないコンポーネント間で通信が可能になります。

アクションとイベント

フレームワークでは、イベントを使用してコンポーネント間でデータを通信します。通常、イベントはユーザアクションでトリガされます。

アクション
: コンポーネントまたはアプリケーションの要素に対するユーザ操作。ユーザアクションでイベントがトリガされますが、イベントは常にユーザアクションで明示的にトリガされるわけではありません。この種別のアクションは、クライアント側のJavaScriptコントローラのアクション(コントローラアクションとも呼ばれる) と同じではありません。次のボタンは、ボタンのクリックに応じてブラウザの `onclick` イベントに結び付けられます。

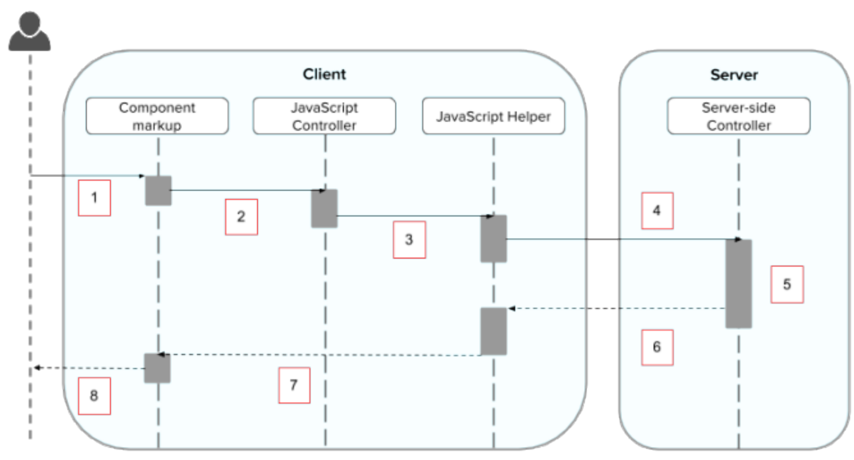
```
<lightning:button label = "Click Me" onclick = "{!c.handleClick}" />
```

ボタンをクリックすると、コンポーネントのクライアント側コントローラで `handleClick` メソッドが呼び出されます。

イベント
: アクションに関するブラウザからの通知。ブラウザイベントは、前の例のようにクライアント側のJavaScriptコントローラで処理されます。ブラウザイベントは、コンポーネント間でデータを通信するためにJavaScriptコントローラで作成して起動できるフレームワークのコンポーネントイベントやアプリケーションイベントと同じではありません。たとえば、チェックボックスのクリックイベントをクライアント側コントローラに結び付け、そのコントローラからコンポーネントイベントを起動して、関連データを親コンポーネントと通信できます。

システムイベントと呼ばれる別のタイプのイベントは、ライフサイクル(コンポーネントの初期化、属性値の変更、表示など)の間にフレームワークによって自動的に起動されます。コンポーネントは、コンポーネントのマークアップでイベントを登録してシステムイベントを処理できます。

次の図は、サーバからのデータの取得をコンポーネントに要求するボタンをユーザがクリックしたときの様子を示しています。



1. ユーザがボタンのクリックまたはコンポーネントに対する操作を行うと、ブラウザイベントがトリガされます。たとえば、ボタンのクリック時にサーバのデータを保存できます。
2. ボタンをクリックすると、ヘルパー関数を呼び出す前にカスタムロジックを提供するクライアント側の JavaScript コントローラが呼び出されます。
3. JavaScript コントローラにより、ヘルパー関数が呼び出されます。ヘルパー関数を使用すると、コードの再利用が促進されますが、この例では省略可能です。
4. ヘルパー関数により、Apex コントローラメソッドがコールされ、アクションがキューに入ります。
5. Apex メソッドが呼び出され、データが返されます。
6. Apex メソッドが完了すると、JavaScript コールバック関数が呼び出されます。
7. JavaScript コールバック関数により、ロジックが評価されて、コンポーネントの UI が更新されます。
8. ユーザに更新されたコンポーネントが表示されます。

関連トピック:

クライアント側コントローラを使用したイベントの処理
変更ハンドラを使用したデータ変更の検出
サーバ側のアクションのコール
表示ライフサイクル中に起動されたイベント

クライアント側コントローラを使用したイベントの処理

クライアント側のコントローラは、コンポーネント内のイベントを処理します。これは、すべてのコンポーネントのアクションの関数を定義する JavaScript リソースです。

クライアント側コントローラは、名前-値のペアの対応付けを含む、オブジェクトリテラル表記の JavaScript オブジェクトです。この名前はそれぞれクライアント側のアクションに対応します。この値は、アクションに関連付けられた関数コードです。クライアント側コントローラは括弧および中括弧で囲まれます。アクションハンドラはカンマで区切ります (JavaScript マップと同様に)。

```
({
    myAction : function(cmp, event, helper) {
        // add code for the action
    },

    anotherAction : function(cmp, event, helper) {
        // add code for the action
    }
})
```

各アクション関数は、次の 3 つのパラメータを取得します。

1. `cmp` — コントローラが属するコンポーネント。
2. `event` — アクションが処理しているイベント。
3. `helper` — コンポーネントのヘルパーであり、省略可能です。ヘルパーには、コンポーネントのバンドルの JavaScript コードで再利用できる関数があります。

クライアント側のコントローラの作成

クライアント側のコントローラは、コンポーネントのバンドルの一部です。これは、*componentName*Controller.js という命名規則で自動的に結び付けられます。

開発者コンソールを使用してクライアント側のコントローラを作成するには、コンポーネントのサイドバーで **[CONTROLLER (コントローラ)]** をクリックします。

クライアント側のコントローラアクションのコール

標準の Lightning コンポーネントである、<lightning:button> を含む HTML ボタンと対照的に、次のコンポーネントの例では 2 つのボタンが作成されます。これらのボタンをクリックすると、text コンポーネントの属性が指定された値で更新されます。target.get("v.label") は、ボタンの label 属性の値を参照します。

コンポーネントのソース

```
<aura:component>
    <aura:attribute name="text" type="String" default="Just a string. Waiting for change."/>

    <input type="button" value="Flawed HTML Button"
        onclick="alert('this will not work')"/>
    <br/>
    <lightning:button label="Framework Button" onclick="{!c.handleClick}"/>
    <br/>
    {!v.text}
</aura:component>
```

JavaScript についての知識があるユーザなら、HTML タグがフレームワークの第一級オブジェクトであることを知っているため、1 番目の「Flawed」ボタンのようなものを自分で記述して試してみることをお勧めします。ただし、コンポーネント内の任意の JavaScript (alert() コールなど) は無視されるため、この「Flawed」ボタンは機能しません。

フレームワークには独自のイベントシステムがあります。HTML タグは Lightning コンポーネントに対応付けられるため、DOM イベントは Lightning イベントに対応付けられます。

on で始まるブラウザの DOM 要素イベント (onclick や onkeypress など) はすべて、コントローラアクションに結び付けることができます。コントローラアクションに結び付けることができるのはブラウザイベントのみです。

「Framework」ボタンは、<lightning:button> コンポーネントの onclick 属性をコントローラの handleClick アクションに結び付けます。

クライアント側コントローラのソース

```
({
    handleClick : function(cmp, event) {
        var attributeValue = cmp.get("v.text");
        console.log("current text: " + attributeValue);

        var target = event.getSource();
        cmp.set("v.text", target.get("v.label"));
    }
})
```

`handleClick` アクションは `event.getSource()` を使用して、このコンポーネントイベントを起動するソースコンポーネントを取得します。この場合、ソースコンポーネントはマークアップの `<lightning:button>` です。

次に、コードは `text` コンポーネント属性の値をボタンの `label` 属性の値に設定します。`text` コンポーネント属性は、マークアップの `<aura:attribute>` タグに定義されています。

ヒント: コンポーネント内のクライアント側アクションとサーバ側アクションには一意の名前を使用します。JavaScript 関数 (クライアント側アクション) と Apex メソッド (サーバ側アクション) が同じ名前だと、問題が発生したときにデバッグしにくくなるおそれがあります。デバッグモードでは、フレームワークによって、クライアント側アクション名とサーバ側アクション名の競合に関するブラウザコンソールの警告が記録されます。

フレームワークイベントの処理

クライアント側のコンポーネントコントローラのアクションを使用して、フレームワークイベントを処理します。マウスおよびキーボードの一般的な操作に対応するフレームワークイベントは、標準コンポーネントで使用できます。

コンポーネントの属性へのアクセス

`handleClick` 関数の各アクションの最初の引数は、コントローラが属するコンポーネントです。このコンポーネントに対して最もよく行われる操作の 1 つは、その属性値の参照と変更です。

`cmp.get("v.attributeName")` では、***attributeName*** 属性の値が返されます。

`cmp.set("v.attributeName", "attribute value")` では、***attributeName*** 属性の値が設定されます。

コントローラでの別のアクションの呼び出し

アクションメソッドを別のメソッドからコールするには、共通のコードをヘルパー関数に配置し、`helper.someFunction(cmp)` で呼び出します。

関連トピック:

コンポーネントのバンドル内の JavaScript コードの共有
イベント処理のライフサイクル
コントローラのサーバ側ロジックの作成

コンポーネントイベント

コンポーネントイベントは、コンポーネントのインスタンスから起動されます。コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

このセクションの内容:

コンポーネントイベント伝達
このフレームワークでは、キャプチャフェーズとバブルフェーズがコンポーネントイベントの伝達でサポートされます。これらのフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。

カスタムコンポーネントイベントの作成
カスタムコンポーネントイベントは、`.evt` リソースの `<aura:event>` タグを使用して作成します。イベントには、そのイベントの起動前に設定可能で、処理時に読み取り可能な属性を含められます。

コンポーネントイベントの起動
別のコンポーネントにデータを通信するには、コンポーネントイベントを起動します。コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

コンポーネントイベントの処理
コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

関連トピック:

aura:method

アプリケーションイベント

クライアント側コントローラを使用したイベントの処理

高度なイベントの例

継承とは?

コンポーネントイベント伝達

このフレームワークでは、キャプチャフェーズとバブルフェーズがコンポーネントイベントの伝達でサポートされます。これらのフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。

イベントを起動するコンポーネントは、ソースコンポーネントと呼ばれます。フレームワークでは、異なるフェーズでイベントを処理できます。これらのフェーズにより、アプリケーションのイベントの最適な処理を柔軟に行うことができます。

次のフェーズがあります。

キャプチャ
イベントがキャプチャされ、アプリケーションルートからソースコンポーネントに伝達していきます。イベントはコンテインメント階層内のキャプチャイベントを受信するコンポーネントによって処理されます。

アプリケーションルートからイベントを起動したソースコンポーネントへと順番にイベントハンドラが呼び出されていきます。

このフェーズの任意の登録ハンドラでイベントの伝達を停止できます。停止した時点でこのフェーズまたはバブルフェーズのハンドラはそれ以上コールされなくなります。

バブル

イベントを起動したコンポーネントはそのイベントを処理できます。その後イベントはソースコンポーネントからアプリケーションルートにバブルアップしていきます。イベントはコンテインメント階層内のバブルイベントを受信するコンポーネントによって処理されます。

イベントを起動したソースコンポーネントからアプリケーションルートへと順番にイベントハンドラが呼び出されていきます。

このフェーズの任意の登録ハンドラでイベントの伝達を停止できます。停止した時点でこのフェーズのハンドラはそれ以上コールされなくなります。

次に、コンポーネントイベントの伝達シーケンスを示します。

1. イベントの起動 — コンポーネントイベントが起動します。
2. キャプチャフェーズ — フレームワークはアプリケーションルートからソースコンポーネントへのキャプチャフェーズをすべてのコンポーネントがトラバースされるまで実行します。任意のイベント処理で、イベントに対して `stopPropagation()` をコールして伝達を停止できます。
3. バブルフェーズ—フレームワークはソースコンポーネントからアプリケーションルートへのバブルフェーズをすべてのコンポーネントがトラバースされるまで、または `stopPropagation()` がコールされるまで実行します。

メモ: アプリケーションイベントには別個のデフォルトフェーズがあります。コンポーネントイベントには別個のデフォルトフェーズがありません。デフォルトフェーズはバブルフェーズです。

カスタムコンポーネントイベントの作成

カスタムコンポーネントイベントは、`.evt` リソースの `<aura:event>` タグを使用して作成します。イベントには、そのイベントの起動前に設定可能で、処理時に読み取り可能な属性を含められます。

コンポーネントイベントの場合は、`<aura:event>` タグに `type="COMPONENT"` を使用します。たとえば、次の `c:compEvent` コンポーネントイベントには属性が 1 つあり、その名前は `message` です。

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

イベントを起動するコンポーネントは、イベントのデータを設定できます。属性値を設定するには、`event.setParam()` または `event.setParams()` をコールします。イベントに設定されるパラメータ名は、イベントの `<aura:attribute>` の `name` 属性と一致している必要があります。たとえば、`c:compEvent` を起動する場合、次のコードを使用することが考えられます。

```
event.setParam("message", "event message here");
```

イベントを処理するコンポーネントは、イベントデータを取得できます。このイベントの属性値を取得するには、ハンドラのクライアント側コントローラで `event.getParam("message")` をコールします。

コンポーネントイベントの起動

別のコンポーネントにデータを通信するには、コンポーネントイベントを起動します。コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

イベントの登録

コンポーネントは、マークアップに `<aura:registerEvent>` を使用して、イベントを起動できるように登録します。次に例を示します。

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
```

ここでは、イベントを起動して処理する場合に `name` 属性の値がどのように使用されるかを確認します。

イベントの起動

JavaScript でコンポーネントイベントへの参照を取得するには、`cmp.getEvent("evtName")` を使用します。この `evtName` は、`<aura:registerEvent>` の `name` 属性と一致します。

`fire()` を使用して、コンポーネントのインスタンスからイベントを起動します。たとえば、クライアント側コントローラの次のアクション関数でイベントを起動します。

```
var compEvent = cmp.getEvent("sampleComponentEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
// compEvent.setParams({"myParam" : myValue });
compEvent.fire();
```

関連トピック:

アプリケーションイベントの起動

コンポーネントイベントの処理

コンポーネントイベントは、イベントを起動したコンポーネント、またはコンテインメント階層内のイベントを受信するコンポーネントによって処理されます。

ハンドラコンポーネントのマークアップで `<aura:handler>` を使用します。次に例を示します。

```
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleComponentEvent}"/>
```

`<aura:handler>` の `name` 属性は、イベントを起動するコンポーネントの `<aura:registerEvent>` タグの `name` 属性に一致する必要があります。

`<aura:handler>` の `action` 属性は、イベントを処理するクライアント側コントローラのアクションを設定します。

`event` 属性では、処理するイベントを指定します。形式は ***namespace:eventName*** です。

この例では、イベントが起動されると、クライアント側コントローラの handleComponentEvent アクションがコールされます。

イベント処理のフェーズ

コンポーネントイベントハンドラは、デフォルトでバブルフェーズに関連付けられます。その代わりにキャプチャフェーズのハンドラを追加するには、phase 属性を使用します。

```
<aura:handler name="sampleComponentEvent" event="ns:eventName"
    action="{!c.handleComponentEvent}" phase="capture" />
```

イベントのソースの取得

<aura:handler> タグのクライアント側コントローラアクションで、evt.getSource() を使用して、どのコンポーネントがイベントを起動したかを確認します。evt はイベントへの参照です。ソース要素を取得するには、evt.getSource().getElement() を使用します。

このセクションの内容:

それ自体のイベントを処理するコンポーネント
コンポーネントは、マークアップの <aura:handler> タグを使用して、それ自体のイベントを処理できます。

バブルまたはキャプチャのコンポーネントイベントの処理
イベント伝達ルールにより、コンテインメント階層のどのコンポーネントがデフォルトでバブルフェーズまたはキャプチャフェーズのイベントを処理できるのが決まります。ルールや、バブルフェーズまたはキャプチャフェーズのイベントの処理方法について説明します。

コンポーネントイベントの動的な処理
コンポーネントには、JavaScript を使用してハンドラを動的にバインドできます。この方法は、コンポーネントがクライアント側で JavaScript を使用して作成されている場合に役立ちます。

関連トピック:

コンポーネントイベント伝達
アプリケーションイベントの処理

それ自体のイベントを処理するコンポーネント

コンポーネントは、マークアップの <aura:handler> タグを使用して、それ自体のイベントを処理できます。

<aura:handler> の action 属性は、イベントを処理するクライアント側コントローラのアクションを設定します。次に例を示します。

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleSampleEvent}"/>
```

メモ: イベントはそれぞれその名前で定義されるため、`<aura:registerEvent>` と `<aura:handler>` の `name` 属性は一致している必要があります。

バブルまたはキャプチャのコンポーネントイベントの処理

イベント伝達ルールにより、コンテインメント階層のどのコンポーネントがデフォルトでバブルフェーズまたはキャプチャフェーズのイベントを処理できるのかが決まります。ルールや、バブルフェーズまたはキャプチャフェーズのイベントの処理方法について説明します。

このフレームワークでは、キャプチャフェーズとバブルフェーズがコンポーネントイベントの伝達でサポートされます。これらのフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。キャプチャフェーズはバブルフェーズの前に実行されます。

デフォルトのイベント伝達ルール

デフォルトでは、コンテインメント階層のすべての親がキャプチャフェーズおよびバブルフェーズでイベントを処理できるわけではありません。代わりに、イベントはコンテインメント階層のすべての所有者に伝達されます。

コンポーネントの所有者は、その作成を行うコンポーネントです。宣言的に作成されたコンポーネントの場合、所有者はイベントの起動コンポーネントを参照するマークアップが含まれる、最も外側のコンポーネントになります。プログラムで作成されたコンポーネントの場合、所有者コンポーネントはそのコンポーネントを作成するために `$A.createComponent` を呼び出したコンポーネントになります。

イベント伝達の方向 (下) はバブルフェーズ (上) と反対になりますが、キャプチャフェーズにも同じルールが適用されます。

わかりやすく、バブルフェーズの例を使用して説明します。

`c:owner` には `c:container` が含まれ、さらにそこには `c:eventSource` が含まれます。

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

`c:eventSource` がイベントを起動すると、このコンポーネント自体がイベントを処理します。次に、イベントはコンテインメント階層をバブルアップします。

`c:container` には `c:eventSource` が含まれますが、マークアップの最も外側のコンポーネントではないことが原因で所有者にはならないため、バブルイベントを処理できません。

`c:owner` は、`c:container` がそのマークアップ内にあるため、所有者です。`c:owner` はイベントを処理できます。

すべてのコンテナコンポーネントへの伝達

デフォルトの動作では、コンテインメント階層内のすべての親がイベントを処理できるわけではありません。他のコンポーネントが含まれているが、それらのコンポーネントの所有者ではないコンポーネントがありま

す。これらのコンポーネントは、コンテナコンポーネントと呼ばれます。この例の c:container は、c:eventSource の所有者でないため、コンテナコンポーネントになります。デフォルトでは、c:container は c:eventSource が起動したイベントを処理できません。

コンテナコンポーネントには、Aura.Component[] 型のファセット属性 (デフォルトの body 属性など) があります。コンテナコンポーネントには、定義で {!v.body} などの式が使用されているコンポーネントが含まれます。コンテナコンポーネントは、その式で表されるコンポーネントの所有者ではありません。

コンテナコンポーネントでイベントを処理できるようにするには、コンテナコンポーネントの <aura:handler> タグに includeFacets="true" を追加します。たとえば、includeFacets="true" をコンテナコンポーネント c:container のハンドラに追加すると、c:eventSource からバブルされたコンポーネントイベントを処理できるようになります。

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

バブルイベントを処理する

コンポーネントイベントを起動したコンポーネントは、<aura:registerEvent> タグを使用してイベントを起動したことを登録します。

```
<aura:component>
    <aura:registerEvent name="compEvent" type="c:compEvent" />
</aura:component>
```

バブルフェーズのイベントを処理するコンポーネントは、<aura:handler> タグを使用してそのクライアント側コントローラの処理アクションを割り当てます。

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleBubbling}"/>
</aura:component>
```

メモ: <aura:handler> の name 属性は、イベントを起動するコンポーネントの <aura:registerEvent> タグの name 属性に一致する必要があります。

キャプチャイベントを処理する

キャプチャフェーズのイベントを処理するコンポーネントは、<aura:handler> タグを使用してそのクライアント側コントローラの処理アクションを割り当てます。

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleCapture}"
        phase="capture" />
</aura:component>
```

phase 属性を設定しない場合、コンポーネントイベントのデフォルト処理フェーズはバブルです。

イベント伝達を停止する

他のコンポーネントへのイベント伝達を停止するには、Event オブジェクトの stopPropagation() メソッドを使用します。

非同期コード実行のイベント伝達を一時停止する

`event.pause()` を使用して、`event.resume()` がコールされるまでイベントの処理と伝達を一時停止します。このフロー制御メカニズムは、非同期コードの実行からの応答に基づいて決定を行う場合に便利です。たとえば、ネイティブモバイルコードに対する非同期コールからの応答に基づいてイベント伝達に関する決定を行うことができます。

`pause()` や `resume()` は、キャプチャフェーズまたはバブルフェーズでコールできます。

イベントバブルの例

自分でいろいろと試せるように 1 つの例を見てみましょう。

```
<!--c:eventBubblingParent-->
<aura:component>
    <c:eventBubblingChild>
        <c:eventBubblingGrandchild />
    </c:eventBubblingChild>
</aura:component>
```

メモ: このサンプルコードでは、デフォルトの c 名前空間を使用しています。自分の組織に名前空間がある場合は、その名前空間を使用してください。

最初に、単純なコンポーネントイベントを定義します。

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!--simple event with no attributes-->
</aura:event>
```

`c:eventBubblingEmitter` は、`c:compEvent` を起動するコンポーネントです。

```
<!--c:eventBubblingEmitter-->
<aura:component>
    <aura:registerEvent name="bubblingEvent" type="c:compEvent" />
    <lightning:button onclick="{!c.fireEvent}" label="Start Bubbling"/>
</aura:component>
```

`c:eventBubblingEmitter` のコントローラは次のようになります。ボタンをクリックすると、マークアップに登録された `bubblingEvent` イベントが起動します。

```
/*eventBubblingEmitterController.js*/
{
    fireEvent : function(cmp) {
        var cmpEvent = cmp.getEvent("bubblingEvent");
        cmpEvent.fire();
    }
}
```

`c:eventBubblingGrandchild` には `c:eventBubblingEmitter` が含まれ、`<aura:handler>` を使用してイベントのハンドラを割り当てます。

```
<!--c:eventBubblingGrandchild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>
```

```


    <div class="grandchild">
        <c:eventBubblingEmitter />
    </div>
</aura:component>
```

c:eventBubblingGrandchild のコントローラは次のようになります。

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
    }
}
```

ハンドラがコールされると、コントローラはイベント名をログに記録します。

c:eventBubblingChild のマークアップは次のようになります。c:eventBubblingGrandchild は、この例で後から c:eventBubblingParent を作成するときに c:eventBubblingChild のボディとして渡します。

```
<!--c:eventBubblingChild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="child">
        {!v.body}
    </div>
</aura:component>
```

c:eventBubblingChild のコントローラは次のようになります。

```
/*eventBubblingChildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Child handler for " + event.getName());
    }
}
```

c:eventBubblingParent には c:eventBubblingChild が含まれ、さらにそこには c:eventBubblingGrandchild が含まれます。

```
<!--c:eventBubblingParent-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="parent">
        <c:eventBubblingChild>
            <c:eventBubblingGrandchild />
        </c:eventBubblingChild>
    </div>
</aura:component>
```

c:eventBubblingParent のコントローラは次のようになります。

```
/*eventBubblingParentController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Parent handler for " + event.getName());
    }
}
```

次に、コードを実行したらどうなるか確認しましょう。

1. ブラウザで、c:eventBubblingParent に移動します。<c:eventBubblingParent /> が含まれる .app リソースを作成します。
2. c:eventBubblingEmitter のマークアップに含まれる [バブルを開始] ボタンをクリックします。
3. ブラウザのコンソールで出力を確認します。

```
Grandchild handler for bubblingEvent
Parent handler for bubblingEvent
```

c:compEvent イベントは、c:eventBubblingGrandchild と c:eventBubblingParent (コンテインメント階層で所有者) に対してバブルになります。イベントが c:eventBubblingChild によって処理されることはありません。c:eventBubblingChild は c:eventBubblingParent のマークアップに含まれますが、そのマークアップの最も外側にあるコンポーネントではないために所有者にならないからです。

では、イベント伝達を停止する方法について説明します。伝達を停止するには、c:eventBubblingGrandchild のコントローラを編集します。

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
        event.stopPropagation();
    }
}
```

次に、c:eventBubblingParent に移動して、[バブルを開始] ボタンをクリックします。

ブラウザのコンソールで出力を確認します。

```
Grandchild handler for bubblingEvent
```

イベントは c:eventBubblingParent コンポーネントにバブルアップされなくなりました。

関連トピック:

コンポーネントイベント伝達

コンポーネントイベントの動的な処理

コンポーネントには、JavaScriptを使用してハンドラを動的にバインドできます。この方法は、コンポーネントがクライアント側で JavaScript を使用して作成されている場合に役立ちます。

詳細は、「コンポーネントへのイベントハンドラの動的な追加」(ページ 311)を参照してください。

コンポーネントイベントの例

以下に、コンポーネントイベントを使用して、別のコンポーネントの属性を更新する簡単な使用事例を示します。

1. ユーザがノーティファイアコンポーネント `ceNotifier.cmp` のボタンをクリックします。
2. `ceNotifier.cmp` のクライアント側コントローラが、コンポーネントイベントにメッセージを設定し、イベントを起動します。
3. ハンドラコンポーネント `ceHandler.cmp` にはノーティファイアコンポーネントが含まれ、起動されたイベントを処理します。
4. `ceHandler.cmp` のクライアント側コントローラが、イベントで送信されたデータに基づいて `ceHandler.cmp` の属性を設定します。

メモ: この例のイベントおよびコンポーネントは、デフォルトの c 名前空間を使用します。自分の組織に名前空間がある場合は、その名前空間を使用してください。

コンポーネントイベント

`ceEvent.evt` コンポーネントイベントには属性が 1 つ設定されています。この場合は、起動時にこの属性を使用してイベントに一定のデータを渡します。

```
<!--c:ceEvent-->
<aura:event type="COMPONENT">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

ノーティファイアコンポーネント

`c:ceNotifier` コンポーネントは `aura:registerEvent` を使用して、コンポーネントイベントを起動する可能性があることを宣言します。

コンポーネントのボタンには、`onclick` ブラウザイベントがあり、クライアント側コントローラの `fireComponentEvent` アクションに結び付けられています。ボタンをクリックすると、アクションが呼び出されます。

```
<!--c:ceNotifier-->
<aura:component>
    <aura:registerEvent name="cmpEvent" type="c:ceEvent"/>

    <h1>Simple Component Event Sample</h1>
    <p><lightning:button
        label="Click here to fire a component event"
        onclick="{!c.fireComponentEvent}" />
    </p>
</aura:component>
```

クライアント側コントローラが、cmp.getEvent("cmpEvent") をコールして、イベントのインスタンスを取得します。この cmpEvent は、コンポーネントのマークアップにある <aura:registerEvent> タグの名前属性の値と一致します。このコントローラがイベントの message 属性を設定して、イベントを起動します。

```
/* ceNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        // Get the component event by using the
        // name value from aura:registerEvent
        var cmpEvent = cmp.getEvent("cmpEvent");
        cmpEvent.setParams({
            "message" : "A component event fired me. " +
            "It all happened so fast. Now, I'm here!" });
        cmpEvent.fire();
    }
}
```

ハンドラコンポーネント

c:ceHandler ハンドラコンポーネントには、c:ceNotifier コンポーネントが含まれます。<aura:handler> タグでは、c:ceNotifier にある <aura:registerEvent> タグの name 属性である cmpEvent と同じ値が使用されます。これは、c:ceNotifier からバブルアップされたイベントを処理するための c:ceHandler を結び付けます。

イベントが起動されると、ハンドラコンポーネントのクライアント側コントローラで handleComponentEvent アクションが呼び出されます。

```
<!--c:ceHandler-->
<aura:component>
    <aura:attribute name="messageFromEvent" type="String"/>
    <aura:attribute name="numEvents" type="Integer" default="0"/>

    <!-- Note that name="cmpEvent" in aura:registerEvent
     in ceNotifier.cmp -->
    <aura:handler name="cmpEvent" event="c:ceEvent" action="{!c.handleComponentEvent}"/>

    <!-- handler contains the notifier component -->
    <c:ceNotifier />

    <p>{!v.messageFromEvent}</p>
    <p>Number of events: {!v.numEvents}</p>

</aura:component>
```

コントローラがイベントで送信されたデータを取得し、そのデータを使用してハンドラコンポーネントの messageFromEvent 属性を更新します。

```
/* ceHandlerController.js */
{
    handleComponentEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
```

```
            cmp.set("v.messageFromEvent", message);
            var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
            cmp.set("v.numEvents", numEventsHandled);
        }
}
```

すべてをまとめる

c:ceHandler コンポーネントを c:ceHandlerApp アプリケーションに追加します。アプリケーションに移動し、ボタンをクリックしてコンポーネントイベントを起動します。

https://<myDomain>.lightning.force.com/c/ceHandlerApp.app (<myDomain> はカスタム Salesforce ドメインの名前)

サーバ上のデータにアクセスする場合は、この例を拡張して、ハンドラのクライアント側コントローラからサーバ側コントローラをコールします。

関連トピック:

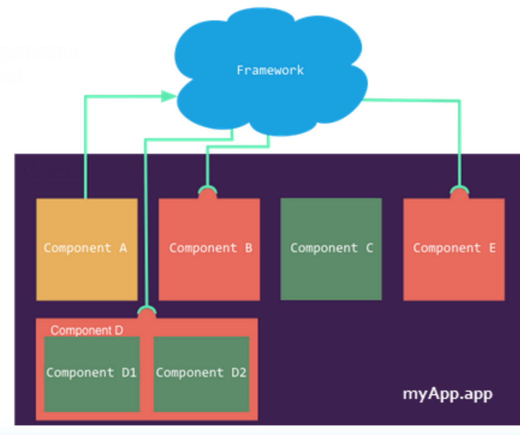
コンポーネントイベント

コントローラのサーバ側ロジックの作成

アプリケーションイベントの例

アプリケーションイベント

アプリケーションイベントは、従来の公開/登録モデルに従います。アプリケーションイベントは、コンポーネントのインスタンスから起動されます。イベントのハンドラを提供するすべてのコンポーネントに通知されます。



このセクションの内容:

アプリケーションイベントの伝達

フレームワークでは、アプリケーションイベントの伝達のキャプチャフェーズ、バブルフェーズ、デフォルトフェーズがサポートされます。キャプチャフェーズとバブルフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。デフォルトフェーズでは、フレームワークの元の処理動作が保持されます。

カスタムアプリケーションイベントの作成

カスタムアプリケーションイベントは、.evt リソースの <aura:event> タグを使用して作成します。イベントには、そのイベントの起動前に設定可能で、処理時に読み取り可能な属性を含められます。

アプリケーションイベントの起動

アプリケーションイベントは、従来の公開/登録モデルに従います。アプリケーションイベントは、コンポーネントのインスタンスから起動されます。イベントのハンドラを提供するすべてのコンポーネントに通知されます。

アプリケーションイベントの処理

ハンドラコンポーネントのマークアップで <aura:handler> を使用します。

関連トピック:

コンポーネントイベント

クライアント側コントローラを使用したイベントの処理

アプリケーションイベントの伝達

高度なイベントの例

アプリケーションイベントの伝達

フレームワークでは、アプリケーションイベントの伝達のキャプチャフェーズ、バブルフェーズ、デフォルトフェーズがサポートされます。キャプチャフェーズとバブルフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。デフォルトフェーズでは、フレームワークの元の処理動作が保持されます。

イベントを起動するコンポーネントは、ソースコンポーネントと呼ばれます。フレームワークでは、異なるフェーズでイベントを処理できます。これらのフェーズにより、アプリケーションのイベントの最適な処理を柔軟に行うことができます。

次のフェーズがあります。

キャプチャ

イベントがキャプチャされ、アプリケーションルートからソースコンポーネントに伝達していきます。イベントはコンテインメント階層内のキャプチャイベントを受信するコンポーネントによって処理されます。

アプリケーションルートからイベントを起動したソースコンポーネントへと順番にイベントハンドラが呼び出されていきます。

このフェーズの任意の登録ハンドラでイベントの伝達を停止できます。停止した時点でこのフェーズまたはバブルフェーズのハンドラはそれ以上コールされなくなります。コンポーネントで

`event.stopPropagation()` を使用してイベント伝達を停止すると、そのコンポーネントはデフォルトフェーズで使用されるルートノードになります。

このフェーズの登録ハンドラは、`event.preventDefault()` をコールしてイベントのデフォルトの動作をキャンセルできます。このコールにより、デフォルトフェーズのハンドラが実行されなくなります。

バブル

イベントを起動したコンポーネントはそのイベントを処理できます。その後イベントはソースコンポーネントからアプリケーションルートにバブルアップしていきます。イベントはコンテインメント階層内のバブルイベントを受信するコンポーネントによって処理されます。

イベントを起動したソースコンポーネントからアプリケーションルートへと順番にイベントハンドラが呼び出されていきます。

このフェーズの任意の登録ハンドラでイベントの伝達を停止できます。停止した時点でこのフェーズのハンドラはそれ以上コールされなくなります。コンポーネントで `event.stopPropagation()` を使用してイベント伝達を停止すると、そのコンポーネントはデフォルトフェーズで使用されるルートノードになります。

このフェーズの登録ハンドラは、`event.preventDefault()` をコールしてイベントのデフォルトの動作をキャンセルできます。このコールにより、デフォルトフェーズのハンドラが実行されなくなります。

デフォルト

イベントハンドラは、ルートノードからそのサブツリーを経由して非決定的な順序で呼び出されます。デフォルトフェーズのコンポーネント階層に関する伝達ルールは、キャプチャフェーズやバブルフェーズとは異なります。デフォルトフェーズは、アプリケーションの別のサブツリーにあるコンポーネントに影響するアプリケーションイベントを処理する場合に便利です。

前のフェーズでイベントの伝達が停止されていない場合、ルートノードはデフォルトのアプリケーションルートになります。前のフェーズでイベントの伝達が停止されている場合、ルートノードはハンドラで `event.stopPropagation()` を呼び出したコンポーネントになります。

次に、アプリケーションイベントの伝達シーケンスを示します。

1. イベントの起動—アプリケーションイベントが起動します。イベントを起動するコンポーネントは、ソースコンポーネントと呼ばれます。
2. キャプチャフェーズ — フレームワークはアプリケーションルートからソースコンポーネントへのキャプチャフェーズをすべてのコンポーネントがトラバースされるまで実行します。任意のイベント処理で、イベントに対して `stopPropagation()` をコールして伝達を停止できます。
3. バブルフェーズ—フレームワークはソースコンポーネントからアプリケーションルートへのバブルフェーズをすべてのコンポーネントがトラバースされるまで、または `stopPropagation()` がコールされるまで実行します。
4. デフォルトフェーズ — フレームワークはルートノードからキャプチャフェーズまたはバブルフェーズで `preventDefault()` がコールされるまでデフォルトフェーズを実行します。前のフェーズでイベントの伝達が停止されていない場合、ルートノードはデフォルトのアプリケーションルートになります。前のフェーズでイベントの伝達が停止されている場合、ルートノードはハンドラで `event.stopPropagation()` を呼び出したコンポーネントになります。

カスタムアプリケーションイベントの作成

カスタムアプリケーションイベントは、.evt リソースの <aura:event> タグを使用して作成します。イベントには、そのイベントの起動前に設定可能で、処理時に読み取り可能な属性を含められます。

アプリケーションイベントの場合は、<aura:event> タグに type="APPLICATION" を使用します。たとえば、次の c:appEvent アプリケーションイベントには、message という名前の属性が1つ設定されています。

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

イベントを起動するコンポーネントは、イベントのデータを設定できます。属性値を設定するには、event.setParam() または event.setParams() をコールします。イベントに設定されるパラメータ名は、イベントの <aura:attribute> の name 属性と一致している必要があります。たとえば、c:appEvent を起動する場合、次のコードを使用することが考えられます。

```
event.setParam("message", "event message here");
```

イベントを処理するコンポーネントは、イベントデータを取得できます。このイベントの属性を取得するには、ハンドラのクライアント側コントローラで event.getParam("message") をコールします。

アプリケーションイベントの起動

アプリケーションイベントは、従来の公開/登録モデルに従います。アプリケーションイベントは、コンポーネントのインスタンスから起動されます。イベントのハンドラを提供するすべてのコンポーネントに通知されます。

イベントの登録

コンポーネントは、マークアップに <aura:registerEvent> を使用して、アプリケーションイベントを起動できるように登録します。name 属性は必須ですが、アプリケーションイベントでは使用されません。name 属性が関係するのは、コンポーネントイベントのみです。次の例では、name="appEvent" を使用していますが、この値はどこにも使用されていません。

```
<aura:registerEvent name="appEvent" type="c:appEvent"/>
```

イベントの起動

JavaScript で $A.get("e.myNamespace:myAppEvent") を使用して、myNamespace 名前空間の myAppEvent イベントのインスタンスを取得します。

メモ: アプリケーションイベントのインスタンスを取得する構文は、コンポーネントイベントを取得する cmp.getEvent("***evtName***") 構文とは異なります。

fire() を使用して、イベントを起動します。

```
var appEvent = $A.get("e.c:appEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
//appEvent.setParams({ "myParam" : myValue });
appEvent.fire();
```

アプリケーションの表示中に起動されるイベント

いくつかのイベントは、アプリケーションを表示中に起動されます。すべての init イベントは、コンポーネントまたはアプリケーションが初期化されたことを示すために起動されます。コンポーネントが別のコンポーネントまたはアプリケーションに含まれる場合は、まず内部のコンポーネントから初期化されます。

表示中にサーバコールが実行された場合は、aura:waiting が起動されます。フレームワークでサーバ応答を受信すると、aura:doneWaiting が起動されます。

最後に、すべての表示が完了すると、aura:doneRendering が起動されます。

メモ: 従来の aura:waiting、aura:doneWaiting、および aura:doneRendering アプリケーションイベントは、最後の手段としてのみ使用することをお勧めします。aura:waiting および aura:doneWaiting アプリケーションイベントは、一括処理されるサーバ要求 (アプリケーションの他のコンポーネントからの要求も含む) ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、これらのアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。

詳細は、「表示ライフサイクル中に起動されたイベント」 (ページ 217) を参照してください。

関連トピック:

コンポーネントイベントの起動

アプリケーションイベントの処理

ハンドラコンポーネントのマークアップで <aura:handler> を使用します。

次に例を示します。

```
<aura:handler event="c:appEvent" action="{!c.handleApplicationEvent}"/>
```

event 属性では、処理するイベントを指定します。形式は ***namespace:eventName*** です。

<aura:handler> の action 属性は、イベントを処理するクライアント側コントローラのアクションを設定します。

メモ: <aura:handler> で name 属性を設定すると、アプリケーションイベントのハンドラは機能しません。name 属性は、コンポーネントイベントを処理する場合にのみ使用します。

この例では、イベントが起動されると、クライアント側コントローラの handleApplicationEvent アクションがコールされます。

イベント処理のフェーズ

フレームワークでは、異なるフェーズでイベントを処理できます。これらのフェーズにより、アプリケーションのイベントの最適な処理を柔軟に行うことができます。

アプリケーションイベントハンドラは、デフォルトフェーズに関連付けられます。その代わりにキャプチャフェーズまたはバブルフェーズのハンドラを追加するには、`phase` 属性を使用します。

イベントのソースの取得

`<aura:handler>` タグのクライアント側コントローラアクションで、`evt.getSource()` を使用して、どのコンポーネントがイベントを起動したかを確認します。`evt` はイベントへの参照です。ソース要素を取得するには、`evt.getSource().getElement()` を使用します。

このセクションの内容:

アプリケーションのバブルイベントとキャプチャイベントの処理

イベント伝達ルールにより、コンテインメント階層のどのコンポーネントがデフォルトでバブルフェーズまたはキャプチャフェーズのイベントを処理できるのが決まります。ルールや、バブルフェーズまたはキャプチャフェーズのイベントの処理方法について説明します。

関連トピック:

コンポーネントイベントの処理

アプリケーションのバブルイベントとキャプチャイベントの処理

イベント伝達ルールにより、コンテインメント階層のどのコンポーネントがデフォルトでバブルフェーズまたはキャプチャフェーズのイベントを処理できるのが決まります。ルールや、バブルフェーズまたはキャプチャフェーズのイベントの処理方法について説明します。

フレームワークでは、アプリケーションイベントの伝達のキャプチャフェーズ、バブルフェーズ、デフォルトフェーズがサポートされます。キャプチャフェーズとバブルフェーズは DOM の処理パターンと似ており、対象のコンポーネントがイベントに対応したり、場合によっては後続のハンドラの動作を制御したりできます。デフォルトフェーズでは、フレームワークの元の処理動作が保持されます。

デフォルトのイベント伝達ルール

デフォルトでは、コンテインメント階層のすべての親がキャプチャフェーズおよびバブルフェーズでイベントを処理できるわけではありません。代わりに、イベントはコンテインメント階層のすべての所有者に伝達されます。

コンポーネントの所有者は、その作成を行うコンポーネントです。宣言的に作成されたコンポーネントの場合、所有者はイベントの起動コンポーネントを参照するマークアップが含まれる、最も外側のコンポーネントになります。プログラムで作成されたコンポーネントの場合、所有者コンポーネントはそのコンポーネントを作成するために `$A.createComponent` を呼び出したコンポーネントになります。

イベント伝達の方向 (下) はバブルフェーズ (上) と反対になりますが、キャプチャフェーズにも同じルールが適用されます。

わかりやすく、バブルフェーズの例を使用して説明します。

c:owner には c:container が含まれ、さらにそこには c:eventSource が含まれます。

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

c:eventSource がイベントを起動すると、このコンポーネント自体がイベントを処理します。次に、イベントはコンテインメント階層をバブルアップします。

c:container には c:eventSource が含まれますが、マークアップの最も外側のコンポーネントではないことが原因で所有者にはならないため、バブルイベントを処理できません。

c:owner は、c:container がそのマークアップ内にあるため、所有者です。c:owner はイベントを処理できます。

すべてのコンテナコンポーネントへの伝達

デフォルトの動作では、コンテインメント階層内のすべての親がイベントを処理できるわけではありません。他のコンポーネントが含まれているが、それらのコンポーネントの所有者ではないコンポーネントがあります。これらのコンポーネントは、コンテナコンポーネントと呼ばれます。この例の c:container は、c:eventSource の所有者でないため、コンテナコンポーネントになります。デフォルトでは、c:container は c:eventSource が起動したイベントを処理できません。

コンテナコンポーネントには、Aura.Component[] 型のファセット属性 (デフォルトの body 属性など) があります。コンテナコンポーネントには、定義で {!v.body} などの式が使用されているコンポーネントが含まれます。コンテナコンポーネントは、その式で表されるコンポーネントの所有者ではありません。

コンテナコンポーネントでイベントを処理できるようにするには、コンテナコンポーネントの <aura:handler> タグに includeFacets="true" を追加します。たとえば、includeFacets="true" をコンテナコンポーネント c:container のハンドラに追加すると、c:eventSource からバブルされたコンポーネントイベントを処理できるようになります。

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

バブルイベントを処理する

バブルフェーズのハンドラを追加するには、phase="bubble" を設定します。

```
<aura:handler event="c:appEvent" action="{!c.handleBubbledEvent}"
    phase="bubble" />
```

event 属性では、処理するイベントを指定します。形式は ***namespace*:*eventName*** です。

<aura:handler> の action 属性は、イベントを処理するクライアント側コントローラのアクションを設定します。

キャプチャイベントを処理する

キャプチャフェーズのハンドラを追加するには、phase="capture" をせていします。

```
<aura:handler event="c:appEvent" action="{!c.handleCapturedEvent}"
    phase="capture" />
```

イベント伝達を停止する

他のコンポーネントへのイベント伝達を停止するには、Event オブジェクトの stopPropagation() メソッドを使用します。

非同期コード実行のイベント伝達を一時停止する

event.pause() を使用して、event.resume() がコールされるまでイベントの処理と伝達を一時停止します。このフロー制御メカニズムは、非同期コードの実行からの応答に基づいて決定を行う場合に便利です。たとえば、ネイティブモバイルコードに対する非同期コールからの応答に基づいてイベント伝達に関する決定を行うことができます。

pause() や resume() は、キャプチャフェーズまたはバブルフェーズでコールできます。

アプリケーションイベントの例

以下に、アプリケーションイベントを使用して、別のコンポーネントの属性を更新する簡単な使用事例を示します。

1. ユーザがノーティファイアコンポーネント aeNotifier.cmp のボタンをクリックします。
2. aeNotifier.cmp のクライアント側コントローラが、コンポーネントイベントにメッセージを設定し、イベントを起動します。
3. ハンドラコンポーネント aeHandler.cmp が、起動されたイベントを処理します。
4. aeHandler.cmp のクライアント側コントローラが、イベントで送信されたデータに基づいて aeHandler.cmp の属性を設定します。

メモ: この例のイベントおよびコンポーネントは、デフォルトの c 名前空間を使用します。自分の組織に名前空間がある場合は、その名前空間を使用してください。

アプリケーションイベント

aeEvent.evt アプリケーションイベントには属性が 1 つ設定されています。この場合は、起動時にこの属性を使用してイベントに一定のデータを渡します。

```
<!--c:aeEvent-->
<aura:event type="APPLICATION">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

ノーティファイアコンポーネント

aeNotifier.cmp ノーティファイアコンポーネントは aura:registerEvent を使用して、アプリケーションイベントを起動する可能性があることを宣言します。name 属性は必須ですが、アプリケーションイベントでは使用されません。name 属性が関係するのは、コンポーネントイベントのみです。

コンポーネントのボタンには、onclick ブラウザイベントがあり、クライアント側コントローラの fireApplicationEvent アクションに結び付けられています。このボタンをクリックすると、アクションが呼び出されます。

```
<!--c:aeNotifier-->
<aura:component>
    <aura:registerEvent name="appEvent" type="c:aeEvent"/>

    <h1>Simple Application Event Sample</h1>
    <p><lightning:button
        label="Click here to fire an application event"
        onclick="{!c.fireApplicationEvent}" />
    </p>
</aura:component>
```

クライアント側コントローラが、$A.get("e.c:aeEvent") をコールして、イベントのインスタンスを取得します。このコントローラがイベントの message 属性を設定して、イベントを起動します。

```
/* aeNotifierController.js */
{
    fireApplicationEvent : function(cmp, event) {
        // Get the application event by using the
        // e.<namespace>.<event> syntax
        var appEvent = $A.get("e.c:aeEvent");
        appEvent.setParams({
            "message" : "An application event fired me. " +
            "It all happened so fast. Now, I'm everywhere!" });
        appEvent.fire();
    }
}
```

ハンドラコンポーネント

aeHandler.cmp ハンドラコンポーネントは、<aura:handler> タグを使用して、アプリケーションイベントを処理することを登録します。

メモ: <aura:handler> で name 属性を設定すると、アプリケーションイベントのハンドラは機能しません。name 属性は、コンポーネントイベントを処理する場合にのみ使用します。

イベントが起動されると、ハンドラコンポーネントのクライアント側コントローラで handleApplicationEvent アクションが呼び出されます。

```
<!--c:aeHandler-->
<aura:component>
    <aura:attribute name="messageFromEvent" type="String"/>
    <aura:attribute name="numEvents" type="Integer" default="0"/>
```

```
    <aura:handler event="c:aeEvent" action="{!c.handleApplicationEvent}"/>

    <p>{!v.messageFromEvent}</p>
    <p>Number of events: {!v.numEvents}</p>
</aura:component>
```

コントローラがイベントで送信されたデータを取得し、そのデータを使用してハンドラコンポーネントの messageFromEvent 属性を更新します。

```
/* aeHandlerController.js */
{
    handleApplicationEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
        cmp.set("v.messageFromEvent", message);
        var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
        cmp.set("v.numEvents", numEventsHandled);
    }
}
```

コンテナコンポーネント

aeContainer.cmp コンテナコンポーネントには、ノーティファイアコンポーネントとハンドラコンポーネントが含まれます。この点は、ハンドラにノーティファイアコンポーネントが含まれるコンポーネントイベントの例とは異なります。

```
<!--c:aeContainer-->
<aura:component>
    <c:aeNotifier/>
    <c:aeHandler/>
</aura:component>
```

すべてをまとめる

このコードをテストする場合は、<c:aeContainer> をサンプル aeWrapper.app アプリケーションに追加して、アプリケーションに移動します。

https://<myDomain>.lightning.force.com/c/aeWrapper.app (<myDomain> はカスタム Salesforce ドメインの名前)

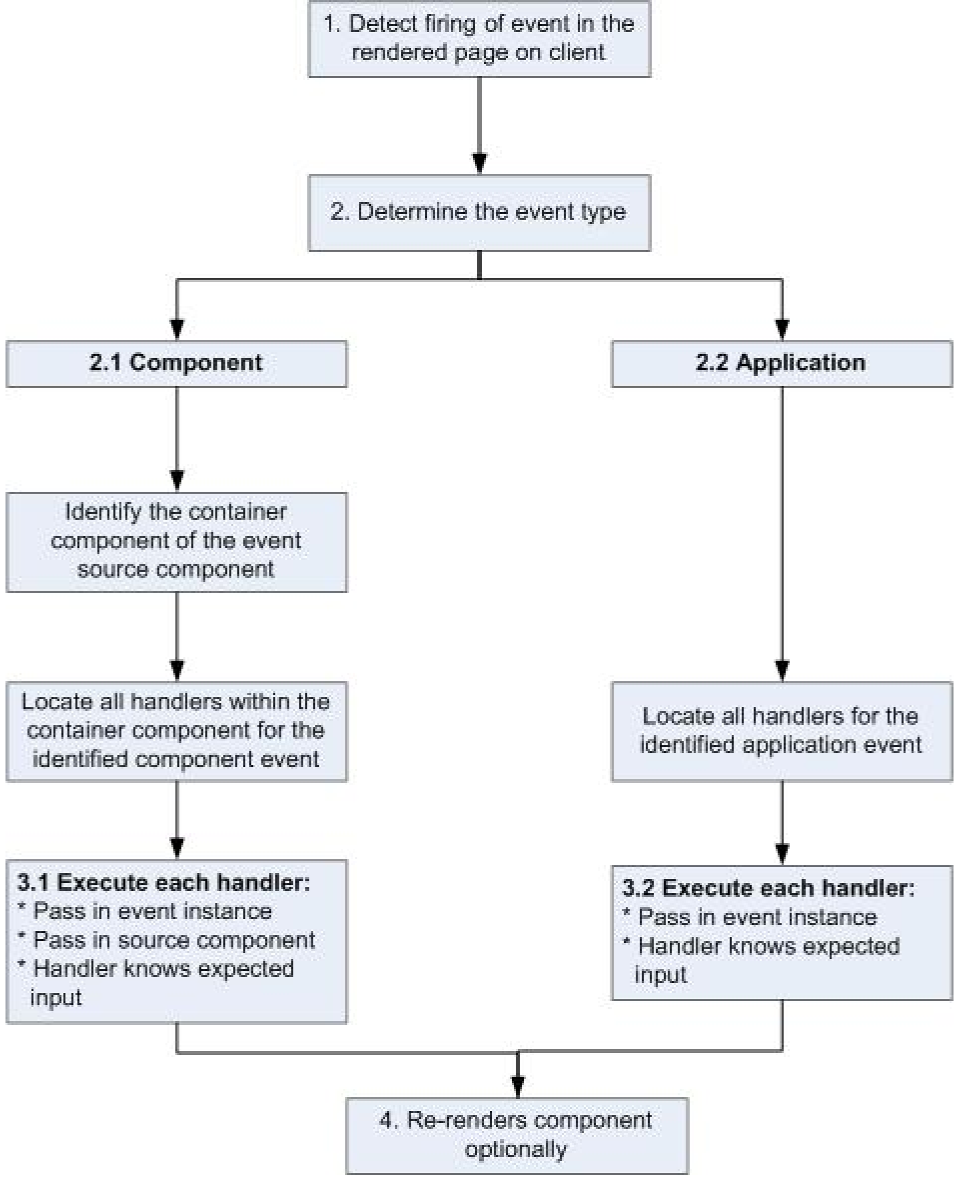
サーバ上のデータにアクセスする場合は、この例を拡張して、ハンドラのクライアント側コントローラからサーバ側コントローラをコールします。

関連トピック:

アプリケーションイベント
コントローラのサーバ側ロジックの作成
コンポーネントイベントの例

イベント処理のライフサイクル

次のチャートは、フレームワークによるイベントの処理の概要を示しています。



1 イベントの起動を検出する

フレームワークがイベントの起動を検出します。たとえば、ノーティファイアコンポーネントのボタンクリックでイベントがトリガされていることがあります。

2 イベントタイプを判断する

2.1 コンポーネントイベント

イベントを起動した親コンポーネントまたはコンテナコンポーネントのインスタンスが特定されます。このコンテナコンポーネントが関連するすべてのイベントハンドラの場所を確認し、さらなる処理が行えるようにします。

2.2 アプリケーションイベント

どのコンポーネントにもこのイベントのイベントハンドラを指定できます。関連するすべてのイベントハンドラの場所が確認されます。

3 各ハンドラを実行する

3.1 コンポーネントイベントハンドラの実行

イベントのコンテナコンポーネントで定義された各イベントハンドラが、ハンドラコントローラによって実行されます。このときに次の操作も実行できます。

- 属性を設定する、またはコンポーネント上のデータを変更する (コンポーネントが再表示されます)。
- 別のイベントを起動する、またはクライアント側あるいはサーバ側のアクションを呼び出す。

3.2 アプリケーションイベントハンドラの実行

すべてのイベントハンドラが実行されます。イベントハンドラが実行されると、イベントインスタンスがイベントハンドラに渡されます。

4 コンポーネントを再表示する (省略可能)

イベント処理中にコンポーネントが変更された場合、イベントハンドラおよびコールバックアクションの実行後にコンポーネントを自動的に再表示することができます。

関連トピック:

カスタムレンダラの作成

高度なイベントの例

次の例は、比較的簡単なコンポーネントイベントおよびアプリケーションイベントの例に基づいています。コンポーネントイベントとアプリケーションイベントの両方で機能する、1 つのノーティファイアコンポーネントと 1 つのハンドラコンポーネントを使用します。イベントに結び付けられたコンポーネントについて説明する前に、関与する個々のリソースを見ていきます。

次の表は、この例で使用する各種リソースの役割をまとめたものです。これらのリソースのソースコードは、表の後に記載されています。

リソース	リソース名	使用方法
イベントファイル	コンポーネントイベント (`compEvent.evt`)およびアプリケーションイベント (`appEvent.evt`)	コンポーネントイベントとアプリケーションイベントを別々のリソースに定義します。`eventsContainer.cmp` に、コンポーネントイベントとアプリケーションイベントの両方の使用方法が示されます。
ノーティファイア	コンポーネント (`eventsNotifier.cmp`) およびそのコントローラ (`eventsNotifierController.js`)	ノーティファイアには、イベントを開始する `onclick` ブラウザイベントが含まれます。このコントローラはイベントを起動します。

リソース	リソース名	使用方法
ハンドラ	コンポーネント (eventsHandler.cmp) およびそのコントローラ (eventsHandlerController.js)	ハンドラコンポーネントには、ノーティファイアコンポーネント (またはアプリケーションイベントの <aura:handler> タグ) が含まれ、イベントの起動後に実行されるコントローラアクションをコールします。
コンテナコンポーネント	eventsContainer.cmp	完全デモの UI にイベントハンドラを表示します。

コンポーネントイベントおよびアプリケーションイベントの定義は別々の .evt リソースに保存されますが、ノーティファイアコンポーネントとハンドラコンポーネントの個別のバンドルに、どちらのイベントでも機能するコードを含めることができます。

コンポーネントとアプリケーションのどちらのイベントにも、イベントの形状を定義する context 属性が含まれます。このデータがイベントのハンドラに渡されます。

コンポーネントイベント

compEvent.evt のマークアップは次のようになります。

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

アプリケーションイベント

appEvent.evt のマークアップは次のようになります。

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

ノーティファイアコンポーネント

eventsNotifier.cmp ノーティファイアコンポーネントには、コンポーネントイベントまたはアプリケーションイベントを開始するボタンが含まれます。

ノーティファイアコンポーネントは aura:registerEvent タグを使用して、コンポーネントイベントおよびアプリケーションイベントを起動する可能性があることを宣言します。name 属性は必要ですが、値が関係するのはコンポーネントイベントのみであり、アプリケーションイベントの他の場所で値が使用されることはありません。

parentName 属性はまだ設定されていません。以下に、この属性がどのように設定され、eventsContainer.cmp に表示されるのかを示します。

```
<!--c:eventsNotifier-->
<aura:component>
  <aura:attribute name="parentName" type="String"/>
  <aura:registerEvent name="componentEventFired" type="c:compEvent"/>
  <aura:registerEvent name="appEvent" type="c:appEvent"/>

  <div>
    <h3>This is {!v.parentName}'s eventsNotifier.cmp instance</h3>
    <p><ui:button
        label="Click here to fire a component event"
        press="{!c.fireComponentEvent}" />
    </p>
    <p><ui:button
        label="Click here to fire an application event"
        press="{!c.fireApplicationEvent}" />
    </p>
  </div>
</aura:component>
```

CSS ソース

CSS は eventsNotifier.css にあります。

```
/* eventsNotifier.css */
.cEventsNotifier {
    display: block;
    margin: 10px;
    padding: 10px;
    border: 1px solid black;
}
```

クライアント側コントローラのソース

eventsNotifierController.js コントローラはイベントを起動します。

```
/* eventsNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // Look up event by name, not by type
        var compEvents = cmp.getEvent("componentEventFired");

        compEvents.setParams({ "context" : parentName });
        compEvents.fire();
    },

    fireApplicationEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // note different syntax for getting application event
        var appEvent = $A.get("e.c:appEvent");
```

```
        appEvent.setParams({ "context" : parentName });
        appEvent.fire();
    }
}
```

ボタンをクリックしてコンポーネントイベントやアプリケーションイベントを起動することはできますが、まだハンドラコンポーネントをイベントに結び付けて応答するようにしていないため、出力に変化はありません。

コントローラがイベントを起動する前に、コンポーネントイベントまたはアプリケーションイベントの context 属性をノーティファイアコンポーネントの parentName に設定します。ハンドラコンポーネントを確認しながら、この設定が出力にどのように影響するかについて説明します。

ハンドラコンポーネント

eventsHandler.cmp ハンドラコンポーネントには、アプリケーションイベントおよびコンポーネントイベントの c:eventsNotifier ノーティファイアコンポーネントと <aura:handler> タグが含まれます。

```
<!--c:eventsHandler-->
<aura:component>
  <aura:attribute name="name" type="String"/>
  <aura:attribute name="mostRecentEvent" type="String" default="Most recent event handled:"/>

  <aura:attribute name="numComponentEventsHandled" type="Integer" default="0"/>
  <aura:attribute name="numApplicationEventsHandled" type="Integer" default="0"/>

  <aura:handler event="c:appEvent" action="{!c.handleApplicationEventFired}"/>
  <aura:handler name="componentEventFired" event="c:compEvent"
action="{!c.handleComponentEventFired}"/>

  <div>
    <h3>This is {!v.name}</h3>
    <p>{!v.mostRecentEvent}</p>
    <p># component events handled: {!v.numComponentEventsHandled}</p>
    <p># application events handled: {!v.numApplicationEventsHandled}</p>
    <c:eventsNotifier parentName="{#v.name}" />
  </div>
</aura:component>
```

メモ: {#v.name} は非バインド式です。つまり、c:eventsNotifier の parentName 属性値を変更しても、逆伝播して c:eventsHandler の name 属性値に影響が及ぶことはありません。詳細は、「コンポーネント間のデータバインド」 (ページ 48)を参照してください。

CSS ソース

CSS は eventsHandler.css にあります。

```
/* eventsHandler.css */
.cEventsHandler {
  display: block;
  margin: 10px;
  padding: 10px;
```

```
  border: 1px solid black;
}
```

クライアント側コントローラのソース

クライアント側コントローラは eventsHandlerController.js にあります。

```
/* eventsHandlerController.js */
{
    handleComponentEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: COMPONENT event, from " + context);

        var numComponentEventsHandled =
            parseInt(cmp.get("v.numComponentEventsHandled")) + 1;
        cmp.set("v.numComponentEventsHandled", numComponentEventsHandled);
    },

    handleApplicationEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: APPLICATION event, from " + context);

        var numApplicationEventsHandled =
            parseInt(cmp.get("v.numApplicationEventsHandled")) + 1;
        cmp.set("v.numApplicationEventsHandled", numApplicationEventsHandled);
    }
}
```

name 属性はまだ設定されていません。以下に、この属性がどのように設定され、eventsContainer.cmp に表示されるのかを示します。

ボタンをクリックでき、UI がイベントタイプを示すものに変更されます。クリック数が 1 つ増え、コンポーネントイベントかアプリケーションイベントかを示します。これで終了ではありません。イベントの context 属性が設定されていないため、イベントのソースが未定義です。

コンテナコンポーネント

eventsContainer.cmp のマークアップは次のようになります。

```
<!--c:eventsContainer-->
<aura:component>
    <c:eventsHandler name="eventsHandler1"/>
    <c:eventsHandler name="eventsHandler2"/>
</aura:component>
```

コンテナコンポーネントには、2 つのハンドラコンポーネントが含まれます。このコンテナコンポーネントは、両方のハンドラコンポーネントの name 属性を設定します。この属性がパススルーされ、ノーティファイアコンポーネントの parentName 属性が設定されます。この操作によって、ノーティファイアコンポーネントまたはハンドラコンポーネント自体の説明で確認した UI テキストのギャップが埋められます。

c:eventsContainer コンポーネントを c:eventsContainerApp アプリケーションに追加します。アプリケーションに移動します。

https://<myDomain>.lightning.force.com/c/eventsContainerApp.app (<myDomain> はカスタム Salesforce ドメインの名前)

いずれかのイベントハンドラの [Click here to fire a component event (コンポーネントイベントを起動する場合はここをクリック)] をクリックします。処理されたコンポーネントイベント数のカウンタには、起動元のコンポーネントのハンドラのみが通知されるため、このコンポーネントのみのイベント数が増加します。

いずれかのイベントハンドラの [Click here to fire an application event (アプリケーションイベントを起動する場合はここをクリック)]をクリックします。処理されたアプリケーションイベント数のカウンタには、処理しているすべてのコンポーネントが通知されるため、両方のコンポーネントのイベント数が増加します。

関連トピック:

コンポーネントイベントの例

アプリケーションイベントの例

イベント処理のライフサイクル

非 Lightning コードからの Lightning イベントの起動

Lightning イベントは、Lightning アプリケーション外の JavaScript から起動できます。たとえば、Lightning アプリケーションで一定の非 Lightning コードをコールし、終了後にそのコードが Lightning アプリケーションと通信するようにする必要のある場合があります。

たとえば、別のシステムにログインする必要のある外部コードをコールして、一部のデータを Lightning アプリケーションに返すことができます。このイベント mynamespace:externalEvent をコールしてみましょう。この JavaScript を非 Lightning コードに含めて、非 Lightning コードの終了時にこのイベントを起動します。

```
var myExternalEvent;
    if(window.opener.$A &&
      (myExternalEvent = window.opener.$A.get("e.mynamespace:externalEvent"))) {
         myExternalEvent.setParams({isOauthed:true});
         myExternalEvent.fire();
       }
```

window.opener.$A.get() は、Lightning アプリケーションが読み込まれているマスタウィンドウを参照します。

関連トピック:

アプリケーションイベント

フレームワークのライフサイクル外のコンポーネントの変更

イベントのベストプラクティス

以下にイベントを使用する場合のベストプラクティスをいくつか示します。

できる限りコンポーネントイベントを使用する

可能な場合は常に、アプリケーションイベントではなくコンポーネントイベントを使用します。コンポーネントイベントを処理できるのは、コンテインメント階層で上位にあるコンポーネントのみであるため、それらのイベントを把握する必要があるコンポーネントにのみ使用が限定されます。アプリケーションイベントは、特定のレコードへの移動など、アプリケーションレベルで処理する必要があるものに適しています。アプリケーションイベントにより、アプリケーションの別々の部分にあって直接的なコンテインメント関係がないコンポーネント間で通信が可能になります。

低レベルのイベントをビジネスロジックイベントと区別する

クリックなどの低レベルのイベントをイベントハンドラで処理し、`approvalChange` イベントやビジネスロジックイベントに相当するものなどは、高レベルのイベントとして再起動することをお勧めします。

コンポーネントの状態に基づく動的アクション

コンポーネントの状態に応じてクリックイベント時に異なるアクションを呼び出す必要がある場合は、次のアプローチを試します。

1. コンポーネントの状態を、New (新規) や Pending (待機中) などの非連続値としてコンポーネントの属性に保存します。
2. クライアント側コントローラに、次に実行するアクションを判断するロジックを配置します。
3. コンポーネントのバンドルでロジックを再利用する必要がある場合は、ロジックをヘルパーに配置します。

次に例を示します。

1. コンポーネントのマークアップに `<ui:button label="do something" press="{!c.click}" />` が含まれる。
2. コントローラで、`click` 関数を定義している。この関数は適切なヘルパー関数に代行させますが、正しいイベントを起動する可能性もあります。

ディスパッチャコンポーネントを使用したイベントのリスンおよびリレー

イベントをリスンしているハンドラコンポーネントのインスタンスが多数あるときは、イベントをリスンするディスパッチャコンポーネントを指定したほうがよい場合があります。ディスパッチャコンポーネントは、コンポーネントのどのインスタンスで詳細情報を受け取るかを判断する一定のロジックを実行し、別のコンポーネントイベントまたはアプリケーションイベントをこれらのコンポーネントのインスタンスで起動することができます。

関連トピック:

クライアント側コントローラを使用したイベントの処理
イベントのアンチパターン

イベントのアンチパターン

イベントを使用する場合に回避すべきいくつかのアンチパターンが存在します。

レンダラでイベントを起動しない

レンダラでイベントを起動すると、無限の表示ループが生じることがあります。

次のようなコードは記述しないでください。

```
afterRender: function(cmp, helper) {
    this.superAfterRender();
    $A.get("e.myns:mycmp").fire();
}
```

代わりに、init フックを使用して、コンポーネントを構築してから表示するまでの間にコントローラのアクションを実行します。コンポーネントに次のコードを追加します。

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

詳細は、「コンポーネントの初期化時のアクションの呼び出し」 (ページ 271)を参照してください。

onclick および ontouchend イベントを使用しない

コンポーネントの onclick イベントと ontouchend イベントに異なるアクションを使用することはできません。フレームワークは、タッチ/タップイベントをクリックに変換し、存在する onclick ハンドラを有効にします。

関連トピック:

カスタムレンダラの作成

イベントのベストプラクティス

表示ライフサイクル中に起動されたイベント

コンポーネントはそのライフサイクルの間にインスタンス化され、表示され、さらに再表示されます。コンポーネントが再表示されるのは、プログラムまたは値が変更されて再表示が必要になった場合のみです。たとえば、ブラウザイベントがアクションをトリガしてデータが更新された場合などです。

コンポーネントの作成

コンポーネントのライフサイクルは、クライアントが HTTP 要求をサーバに送信し、コンポーネント設定データがクライアントに返されると開始します。以前の要求によってコンポーネント定義がすでにクライアント側にあり、コンポーネントにサーバとの連動関係がない場合は、このサーバとの往復のやりとりは行われません。

ネストされたいくつかのコンポーネントを含むアプリケーションを見てみましょう。フレームワークは、アプリケーションをインスタンス化し、v.body facet の子を通って、各コンポーネントを作成します。まず、コン

ポーネント定義とその親階層全体を作成してから、コンポーネント内で facet を作成します。また、属性、インターフェース、コントローラ、アクションの定義を含め、すべてのコンポーネントの連動関係もサーバに作成します。

コンポーネントインスタンスが作成されると、逐次化されたコンポーネント定義とインスタンスがクライアントに送信されます。定義はキャッシュされますが、インスタンスデータはキャッシュされません。クライアントは、応答の逐次化を解除して JavaScript オブジェクトまたは対応付けを作成します。その結果、コンポーネントインスタンスの表示に使用するインスタンスツリーが作成されます。コンポーネントツリーの準備が整うと、すべてのコンポーネントに対して `init` イベントが起動されます。起動は、子コンポーネントから開始し、親コンポーネントで終了します。

コンポーネントの表示

表示ライフサイクルは、コンポーネントが明示的に非表示にされない限りコンポーネントの有効期間内に 1 回発生します。コンポーネントを作成すると、次の処理が行われます。

1. `init` イベントは、コンポーネントを構成するコンポーネントサービスによって起動され、初期化が完了したこと通知します。

   ```
   <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
   ```

   コンポーネントの表示が開始される前に、`init` ハンドラをカスタマイズして、独自のコントローラロジックを追加できます。詳細は、「コンポーネントの初期化時のアクションの呼び出し」(ページ 271) を参照してください。

2. ツリーのコンポーネントごとに、`render()` の基本実装またはカスタムレンダラがコールされ、コンポーネントの表示が開始されます。詳細は、「カスタムレンダラの作成」(ページ 284) を参照してください。コンポーネントの作成プロセスと同様に、表示はルートコンポーネントで開始され、子コンポーネントとスーパーコンポーネントの順に処理され (存在する場合)、子サブコンポーネントで終了します。

3. コンポーネントが DOM に表示されると、`afterRender()` がコールされ、これらの各コンポーネント定義について表示が完了したことが通知されます。これにより、フレームワークの表示サービスで DOM 要素が作成されたら、DOM ツリーを操作できます。

4. クライアントがサーバ要求 XHR への応答の待機を終了したことを示すために、`aura:doneWaiting` イベントが起動されます。このイベントは、クライアント側コントローラアクションに結び付けられたハンドラを追加することで処理できます。

   メモ: `aura:doneWaiting` イベントは、最後の手段としてのみ使用することをお勧めします。`aura:doneWaiting` アプリケーションイベントは、サーバ応答 (アプリケーションの他のコンポーネントからの応答も含む) ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。

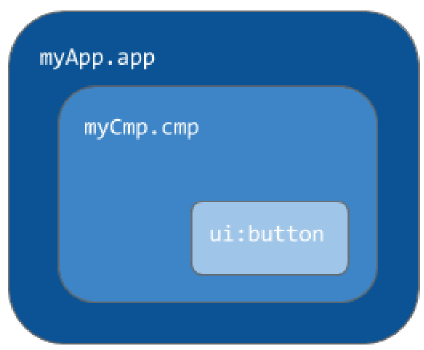
5. フレームワークの表示サービスによって DOM 要素が挿入されたら、フレームワークによって `render` イベントが起動され、DOM ツリーを操作できるようになります。`render` イベントの処理は、カスタムレンダラの作成や `afterRender()` の上書きよりも優先されます。詳細は、「render イベントの処理」を参照してください。

6. 最後に、`aura:doneRendering` イベントが表示ライフサイクルの終了時に起動されます。

メモ: aura:doneRendering イベントは、最後の手段としてのみ使用することをお勧めします。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience や Salesforce1 などの複雑なアプリケーションに含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、イベントハンドラを複数回トリガすることがあります。

ネストされたコンポーネントの表示

myApp.app というアプリケーションに myCmp.cmp コンポーネントが含まれ、そのコンポーネントに ui:button コンポーネントが含まれるとします。



初期化中、init() イベントは、ui:button、ui:myCmp、myApp.app の順序で起動されます。

関連トピック:

[カスタムレンダラの作成](#)
[システムイベントの参照](#)

Salesforce1 と Lightning Experience で処理されるイベント

Salesforce1 と Lightning Experience では、Lightning コンポーネントで起動できるいくつかのイベントが処理されます。

これらの force または lightning イベントのいずれかを Salesforce1 または Lightning Experience 外の Lightning アプリケーション/コンポーネントで起動する場合、次のようになります。

- 処理コンポーネントの <aura:handler> タグを使用して、イベントを処理する必要があります。
- 必要に応じて、イベントがクライアントに送信されるように <aura:registerEvent> または <aura:dependency> タグを使用します。

イベント名	説明
force:closeQuickAction	クイックアクションパネルを閉じます。アプリケーションで一度に開くことができるクイックアクションパネルは 1 つのみです。

イベント名	説明
`force:createRecord`	指定した `entityApiName` (「Account」や「myNamespace__MyObject__c」など) のレコードを作成するページを開きます。
`force:editRecord`	`recordId` で指定したレコードを編集するページを開きます。
`force:navigateToComponent` (ベータ)	ある Lightning コンポーネントから別のコンポーネントに移動します。
`force:navigateToList`	`listViewId` で指定したリストビューに移動します。
`force:navigateToObjectHome`	`scope` 属性で指定したオブジェクトホームに移動します。
`force:navigateToRelatedList`	`parentRecordId` で指定した関連リストに移動します。
`force:navigateToSObject`	`recordId` で指定した sObject レコードに移動します。
`force:navigateToURL`	指定した URL に移動します。
`force:recordSave`	レコードを保存します。
`force:recordSaveSuccess`	レコードが正常に保存されたことを示します。
`force:refreshView`	ビューを再読み込みします。
`force:showToast`	トースト通知とメッセージを表示します。
`lightning:openFiles`	ContentDocument および ContentHubItem オブジェクトの 1 つ以上のファイルレコードを開きます。

Salesforce1、Lightning Experience、およびスタンドアロンアプリケーションのクライアント側ロジックのカスタマイズ

Salesforce1 および Lightning Experience では多くのイベントが自動的に処理されますが、コンポーネントがスタンドアロンアプリケーションで実行される場合には追加作業が必要です。`$A.get()` を使用してイベントをインスタンス化すると、コンポーネントが実行されている場所 (Salesforce1 および Lightning Experience またはスタンドアロンアプリケーション) を判断するのに役立ちます。たとえば、コンポーネントを Salesforce1 および Lightning Experience で読み込む場合にトーストを表示するとします。その場合、`force:showToast` イベントを起動し、Salesforce1 および Lightning Experience 用にパラメータを設定できますが、スタンドアロンアプリケーション用に独自の実装を作成する必要があります。

```
displayToast : function (component, event, helper) {
    var toast = $A.get("e.force:showToast");
    if (toast){
        //fire the toast event in Salesforce1 and Lightning Experience
        toast.setParams({
            "title": "Success!",
            "message": "The component loaded successfully."
        });
```

```
        toast.fire();
    } else {
        //your toast implementation for a standalone app here
    }
}
```

関連トピック:

イベントの参照

aura:dependency

コンポーネントイベントの起動

アプリケーションイベントの起動

システムイベント

ライフサイクルの間にフレームワークによっていくつかのシステムイベントが起動されます。

これらのイベントは、Lightning アプリケーション/コンポーネント、および Salesforce1 内で処理できます。

イベント名	説明
`aura:doneRendering`	ルートアプリケーションの初期表示が完了したことを示します。`aura:doneRendering` イベントは、最後の手段としてのみ使用することをお勧めします。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience や Salesforce1 などの複雑なアプリケーションに含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、イベントハンドラを複数回トリガすることがあります。
`aura:doneWaiting`	アプリケーションでサーバ要求への応答の待機が終了したことを示します。このイベントの前には `aura:waiting` イベントがあります。`aura:doneWaiting` イベントは、最後の手段としてのみ使用することをお勧めします。`aura:doneWaiting` アプリケーションイベントは、サーバ応答(アプリケーションの他のコンポーネントからの応答も含む) ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。
`aura:locationChange`	URL のハッシュ部分が変更されたことを示します。
`aura:noAccess`	要求したリソースのセキュリティ上の制約により、そのリソースにアクセスできないことを示します。

イベント名	説明
aura:systemError	エラーが発生したことを示します。
aura:valueChange	属性値が変更されたことを示します。
aura:valueDestroy	コンポーネントが破棄されたことを示します。
aura:valueInit	アプリケーションまたはコンポーネントが初期化されたことを示します。
aura:valueRender	アプリケーションまたはコンポーネントが表示または再表示されたことを示します。
aura:waiting	アプリケーションでサーバ要求への応答を待機していることを示します。最後の手段として使用する場合を除き、従来の aura:waiting イベントの使用はおすすめしません。アプリケーションの他のコンポーネントからの要求であっても、aura:waiting アプリケーションイベントはサーバー要求ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。

関連トピック:

システムイベントの参照

第 6 章　アプリケーションの作成

トピック:

- アプリケーションの概要
- アプリケーションの UI の設計
- アプリケーションテンプレートの作成
- セキュアなコードの開発
- アプリケーションのスタイル設定
- JavaScript の使用
- JavaScript Cookbook
- Apex の使用
- Lightning データサービス
- Lightning コンテナ
- アクセスの制御
- オブジェクト指向開発の使用
- AppCache の使用
- アプリケーションとコンポーネントの配布

コンポーネントは、アプリケーションのビルディングブロックです。このセクションでは、さまざまなビルディングブロックをまとめて新しいアプリケーションを作成するための典型的なワークフローを説明します。

まず、スタンドアロンアプリケーションまたは Salesforce アプリケーション (Lightning Experience や Salesforce1 など) のどちらのコンポーネントを作成するのかを決定する必要があります。どちらのコンポーネントも Salesforce データにアクセスできますが、特筆すべき点として、Lightning Experience または Salesforce1 用に作成されたコンポーネントのみが、レコード作成および編集ページを利用する Salesforce イベントを自動的に処理できます。

「クイックスタート」 (ページ 8)では、スタンドアロンアプリケーションのコンポーネントおよび Salesforce1 のコンポーネントの作成について説明しており、どちらのコンポーネントが必要かを判断するのに役立ちます。

アプリケーションの概要

アプリケーションは、.app リソース内にマークアップが含まれている特殊な最上位コンポーネントです。

本番サーバでは、.app リソースは、ブラウザ URL 内で唯一アドレス指定が可能な単位です。アプリケーションには、次のような URL を使用してアクセスします。

https://<myDomain>.lightning.force.com/<namespace>/<appName>.app (<myDomain> はカスタム Salesforce ドメインの名前)

関連トピック:

aura:application

サポートされる HTML タグ

アプリケーションの UI の設計

アプリケーションの UI を設計するには、.app リソースにマークアップを挿入します。UI の各部分がコンポーネントに対応し、コンポーネントにはネストされたコンポーネントを含めることができます。高度なアプリケーションを作成するには、複数のコンポーネントを構成します。

アプリケーションのマークアップは <aura:application> タグで開始します。

メモ: スタンドアロンアプリケーションを作成すると、Lightning Out や Visualforce ページの Lightning コンポーネントなどを使用して、Salesforce1 や Lightning Experience 外でコンポーネントをホストできます。<aura:application> タグについての詳細は、「aura:application」を参照してください。

sample.app ファイルを見てみましょう。<aura:application> タグで開始しています。

```
<aura:application extends="force:slds">
    <lightning:layout>
        <lightning:layoutItem padding="around-large">
            <h1 class="slds-text-heading--large">Sample App</h1>
        </lightning:layoutItem>
    </lightning:layout>
    <lightning:layout>
        <lightning:layoutItem padding="around-small">
            Sidebar
            <!-- Other component markup here -->
        </lightning:layoutItem>
        <lightning:layoutItem padding="around-small">
            Content
            <!-- Other component markup here -->
        </lightning:layoutItem>
    </lightning:layout>

</aura:application>
```

sample.app ファイルには HTML タグ (<h1> など) とコンポーネント (<lightning:layout> など) が含まれています。ここでは各コンポーネントの詳細な説明は省きますが、マークアップがいかに簡単かを確認してくだ

さい。<lightning:layoutItem> コンポーネントには、他のコンポーネントまたは HTML マークアップを含めることができます。

関連トピック:

aura:application

アプリケーションテンプレートの作成

アプリケーションテンプレートは、フレームワークとアプリケーションの読み込みのブートストラップを行います。デフォルトの aura:template テンプレートを拡張するコンポーネントを作成して、アプリケーションのテンプレートをカスタマイズします。

テンプレートでは、<aura:component> タグの isTemplate システム属性を true に設定する必要があります。これにより、通常のコンポーネントでは許可されない <script> タグなどの制限項目を許可するようフレームワークに指示されます。

たとえば、サンプルアプリケーションには、aura:template を拡張する np:template テンプレートがあります。np:template は次のようになります。

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="title" value="My App"/>
    ...
</aura:component>
```

コンポーネントで aura:template の拡張と、aura:set を使用した title 属性の設定がどのように行われるかを注目してください。

<aura:application> で template システム属性を設定することで、アプリケーションはカスタムテンプレートを示します。

```
<aura:application template="np:template">
    ...
</aura:application>
```

テンプレートで拡張できるのは、コンポーネントまたは別のテンプレートのみです。コンポーネントまたはアプリケーションでテンプレートを拡張することはできません。

セキュアなコードの開発

LockerService アーキテクチャレイヤでは、個々の Lightning コンポーネントが各自のコンテナで分離され、コーディングのベストプラクティスが適用されるためセキュリティが向上します。

このフレームワークでは、コンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むコンテンツのソースを制御します。

このセクションの内容:

LockerService とは?

LockerService は、Lightning コンポーネントのための強力なセキュリティアーキテクチャです。LockerService では、Lightning コンポーネントが各自の名前空間で分離されるためセキュリティが向上します。また、LockerService では、コードのサポート機能が向上するベストプラクティスが促進されます。これは、サポートされる API へのアクセスのみを許可し、公開されていないフレームワーク内部へのアクセスを排除することで実現します。

コンテンツセキュリティポリシーの概要

Lightning コンポーネントフレームワークでは、W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御します。

Salesforce Lightning CLI

Lightning CLI は Heroku Toolbelt プラグインで、コードをスキャンして一般的な JavaScript コーディングの問題や Lightning 固有の問題がないか調べることができます。このツールは、LockerService を有効にするための Lightning コンポーネントコードを準備する上で有用です。

LockerService とは?

LockerService は、Lightning コンポーネントのための強力なセキュリティアーキテクチャです。LockerService では、Lightning コンポーネントが各自の名前空間で分離されるためセキュリティが向上します。また、LockerService では、コードのサポート機能が向上するベストプラクティスが促進されます。これは、サポートされる API へのアクセスのみを許可し、公開されていないフレームワーク内部へのアクセスを排除することで実現します。

このセクションの内容:

JavaScript ES5 の厳格モードの適用

LockerService では JavaScript ES5 の厳格モードが暗黙的に有効になります。コードで `"use strict"` を指定する必要がありません。JavaScript の厳格モードにより、コードの堅牢性とサポート範囲が拡大されます。たとえば、このモードでない場合は抑制される一部のエラーを発生させることができます。

DOM のアクセスコンテインメント

コンポーネントは、DOM をトラバースして、同じ名前空間内のコンポーネントで作成された要素にアクセスすることしかできません。この動作により、他の名前空間内のコンポーネントで所有される DOM 要素にアクセスするアンチパターンを回避できます。

グローバル参照のセキュアなラッパー

LockerService はグローバル参照に制限を適用します。`window` など、組み込み以外のオブジェクトは LockerService がセキュアなバージョンを用意します。たとえば、`window` のセキュアなバージョンは `SecureWindow` です。組み込み以外のオブジェクトと同じ方法でセキュアなラッパーとやりとりできますが、セキュアなラッパーではオブジェクトとそのプロパティへのアクセスが制限されます。セキュアなラッパーでは、基盤となるオブジェクトの API のサブセットが公開されます。

サポートされている JavaScript API フレームワークのメソッドのみへのアクセス

サポートされている公開済みの JavaScript API フレームワークのメソッドにのみアクセスできます。これらのメソッドは、`https://<myDomain>.lightning.force.com/auradocs/reference.app` (<myDomain> はカスタム Salesforce ドメインの名前) にあるリファレンスドキュメントアプリケーションで公開されています。以前はサポートされていないメソッドにアクセスできたため、サポートされていないメソッドが変更または削除されたときにコードが破損するおそれがありました。

LockerService による影響

LockService の影響を受けるものと、受けないものについて見てみましょう。

コンポーネントの LockerService の無効化

コンポーネントの API バージョンを 39.0 以下に設定すると、コンポーネントの LockerService を無効にできます。コンポーネントが API バージョン 40.0 以上に設定されている場合は、LockerService が有効になります。API バージョン 40.0 は、LockService がすべての組織で有効化された Summer '17 に相当します。

コンポーネントの API バージョンを混在させない

一貫性を保ちデバッグを容易にするため、アプリケーション、コンテインメント階層(コンポーネント内のコンポーネント)、または拡張階層 (コンポーネントを拡張するコンポーネント) のすべてのコンポーネントに同じ API バージョンを設定することをお勧めします。

サポートされていないブラウザで無効化された LockService

LockerService は、ブラウザの JavaScript 機能である厳格モード、`Map` オブジェクト、`Proxy` オブジェクトのサポートに依存します。ブラウザが要件を満たしていない場合、LockerService はそのすべてのセキュリティ機能を適用できず、無効になります。

関連トピック:

コンテンツセキュリティポリシーの概要

DOM の変更

リファレンスドキュメントアプリケーション

Salesforce Lightning CLI

Salesforce ヘルプ: Lightning Experience でサポートされるブラウザ

JavaScript ES5 の厳格モードの適用

LockerService では JavaScript ES5 の厳格モードが暗黙的に有効になります。コードで `"use strict"` を指定する必要がありません。JavaScript の厳格モードにより、コードの堅牢性とサポート範囲が拡大されます。たとえば、このモードでない場合は抑制される一部のエラーを発生させることができます。

厳格モードを使用する場合のいくつかの考慮事項を次に示します。

- `var` キーワードを使用して変数を宣言する必要があります。
- ライブラリ外で使用可能な変数を作成するには、`window` オブジェクトにその変数を明示的に関連付ける必要があります。詳細は、「コンポーネント間の JavaScript コードの共有」を参照してください。
- コンポーネントが使用するライブラリも厳格モードで動作する必要があります。

JavaScript の厳格モードについての詳細は、Mozilla Developer Networkの記事を参照してください。

DOM のアクセスコンテインメント

コンポーネントは、DOMをトラバースして、同じ名前空間内のコンポーネントで作成された要素にアクセスすることしかできません。この動作により、他の名前空間内のコンポーネントで所有される DOM 要素にアクセスするアンチパターンを回避できます。

メモ: これは、名前空間に関係なく別のコンポーネントにアクセスするコンポーネントを対象としたアンチパターンです。LockerService では、クロス名前空間アクセスのみが防止されます。独自の名前空間内のクロスコンポーネントアクセスを防ぐことで、コンポーネントが緊密に連動し、問題が発生する可能性が低くなります。

DOM コンテインメントを表すサンプルコンポーネントを見てみましょう。

```
<!--c:domLocker-->
<aura:component>
    <div id="myDiv" aura:id="div1">
        <p>See how LockerService restricts DOM access</p>
    </div>
    <lightning:button name="myButton" label="Peek in DOM"
                aura:id="button1" onclick="{!c.peekInDom}"/>
</aura:component>
```

c:domLocker コンポーネントは、<div> 要素と <lightning:button> コンポーネントを作成します。

次に、DOM 内でピークするクライアント側コントローラを示します。

```
({ /* domLockerController.js */
    peekInDom : function(cmp, event, helper) {
        console.log("cmp.getElements(): ", cmp.getElements());
        // access the DOM in c:domLocker
        console.log("div1: ", cmp.find("div1").getElement());
        console.log("button1: ", cmp.find("button1"));
        console.log("button name: ", event.getSource().get("v.name"));

        // returns an error
        //console.log("button1 element: ", cmp.find("button1").getElement());
    }
})
```

有効な DOM アクセス

要素が c:domLocker によって作成されるため、次のメソッドは有効な DOM アクセスです。

cmp.getElements()
コンポーネントによって表示される DOM 内の要素を返します。

cmp.find()
aura:id 属性によって識別される、div およびボタンコンポーネントを返します。

cmp.find("div1").getElement()
c:domLocker が div を作成したときに、div の DOM 要素を返します。

event.getSource().get("v.name")
イベントをディスパッチしたボタンの名前を返します。この場合は、myButton です。

無効な DOM アクセス

`<lightning:button>` で作成された DOM 要素にアクセスするために `cmp.find("button1").getElement()` を使用することはできません。ボタンが `lightning` 名前空間にあり、`c:domLocker` が `c` 名前空間にあるため、LockerService は `c:domLocker` が `<lightning:button>` の DOM に アクセスすることを許可しません。

`cmp.find("button1").getElement()` のコードのコメントを解除すると、エラーが表示されます。

```
c:domLocker$controller$peekInDom [cmp.find(...).getElement is not a function]
```

このセクションの内容:

LockerService で Proxy オブジェクトを使用する方法

LockerService は、標準 JavaScript `Proxy` オブジェクトを使用して、基盤となる JavaScript オブジェクトへのコンポーネントのアクセスを絞り込みます。`Proxy` オブジェクトは、同じ名前空間のコンポーネントによって作成された DOM 要素のみがコンポーネントに表示されるようにします。

関連トピック:

LockerService とは?

JavaScript の使用

LockerService で **`Proxy`** オブジェクトを使用する方法

LockerService は、標準 JavaScript `Proxy` オブジェクトを使用して、基盤となる JavaScript オブジェクトへのコンポーネントのアクセスを絞り込みます。`Proxy` オブジェクトは、同じ名前空間のコンポーネントによって作成された DOM 要素のみがコンポーネントに表示されるようにします。

`Proxy` オブジェクトは、未加工の JavaScript オブジェクトと同じ方法で操作できますが、ブラウザのコンソールに `Proxy` として表示されます。これは、ブラウザのデバッガを使用して調査を開始する場合に、LockerService による `Proxy` の使用状況を理解するのに役立ちます。

コンポーネントで組み込み JavaScript オブジェクトを作成すると、LockerService から未加工の JavaScript オブジェクトが返されます。LockerService でオブジェクトを絞り込むと、`Proxy` オブジェクトが返されます。次に、LockerService でオブジェクトを絞り込み、`Proxy` オブジェクトを返すシナリオを示します。

- オブジェクトを異なる名前空間のコンポーネントに渡す。
- `cmp.get()` をコールして、ネイティブ JavaScript オブジェクトまたは配列の値で設定した属性値を取得する。オブジェクトまたは配列が初めて作成されたときは絞り込まれません。

次のオブジェクトにアクセスすると、LockerService から `Proxy` オブジェクトが返されます。

- HTMLCollection インターフェースを実装するオブジェクト
- HTML 要素を表す `SecureElement` オブジェクト

標準 JavaScript `Proxy` オブジェクトについての詳細は、「Mozilla Developer Network (Mozilla 開発者ネットワーク)」を参照してください。

関連トピック:

DOM のアクセスコンテインメント

グローバル参照のセキュアなラッパー

グローバル参照のセキュアなラッパー

LockerService はグローバル参照に制限を適用します。`window` など、組み込み以外のオブジェクトは LockerService がセキュアなバージョンを用意します。たとえば、`window` のセキュアなバージョンは `SecureWindow` です。組み込み以外のオブジェクトと同じ方法でセキュアなラッパーとやりとりできますが、セキュアなラッパーではオブジェクトとそのプロパティへのアクセスが制限されます。セキュアなラッパーでは、基盤となるオブジェクトの API のサブセットが公開されます。

最も一般的に使用されるセキュアなオブジェクトのリストを次に示します。

`SecureAura`

`$A` のセキュアなラッパー。JavaScript コードのフレームワークを使用するためのエントリポイントです。

`SecureComponent`

`Component` オブジェクトのセキュアなラッパー。

`SecureComponentRef`

`SecureComponentRef` は、異なる名前空間のコンポーネントの外部 API を提供する `SecureComponent` のサブセットです。

コントローラまたはヘルパーの使用中は、`SecureComponent` (基本的に `this` オブジェクト) にアクセスできます。コンポーネントを操作する他のコンテキストでは、異なる名前空間のコンポーネントを参照すると、`SecureComponentRef` が返されます。たとえば、マークアップに `lightning:button` が含まれ、`cmp.find("buttonAuraId")` をコールすると、`lightning:button` はボタンマークアップを含むコンポーネントとは異なる名前空間にあるため、`SecureComponentRef` が返されます。

`SecureDocument`

`Document` オブジェクトのセキュアなラッパー。HTML ドキュメントまたはページのルートノードを表します。`Document` オブジェクトは、ページのコンテンツ (DOM ツリー) へのエントリポイントです。

`SecureElement`

`Element` オブジェクトのセキュアなラッパー。HTML 要素を表します。`SecureElement` は `Proxy` オブジェクトでラップされているため、そのデータにアクセスされたときに遅延絞り込みを実行できるようにパフォーマンスが最適化されます。ブラウザコンソールでデバッグしている場合、HTML 要素は Proxy オブジェクトによって表されます。

`SecureObject`

LockerService によってラップされるオブジェクトのセキュアなラッパー。`SecureObject` がある場合、通常はそのオブジェクトへのアクセス権がないため、一部のプロパティは使用できません。

`SecureWindow`

`Window` オブジェクトのセキュアなラッパー。DOM ドキュメントがあるウィンドウを表します。

例

いくつかのセキュアなラッパーを表すサンプルコンポーネントを見てみましょう。

```
<!--c:secureWrappers-->
<aura:component >
    <div id="myDiv" aura:id="div1">
        <p>See how LockerService uses secure wrappers</p>
    </div>
    <lightning:button name="myButton" label="Peek in DOM"
        aura:id="button1" onclick="{!c.peekInDom}"/>
</aura:component>
```

c:secureWrappers コンポーネントは、<div> HTML 要素と <lightning:button> コンポーネントを作成します。

次に、DOM 内でピークするクライアント側コントローラを示します。

```
({ /* secureWrappersController.js */
    peekInDom : function(cmp, event, helper) {
        console.log("div1: ", cmp.find("div1").getElement());

        console.log("button1: ", cmp.find("button1"));
        console.log("button name: ", event.getSource().get("v.name"));
        // add debugger statement for inspection
        // always remove this from production code
        debugger;
    }
})
```

console.log() を使用して、<div> 要素とボタンを調べます。<div> SecureElement は Proxy オブジェクトでラップされているため、そのデータにアクセスされたときに遅延絞り込みを実行できるようにパフォーマンスが最適化されます。

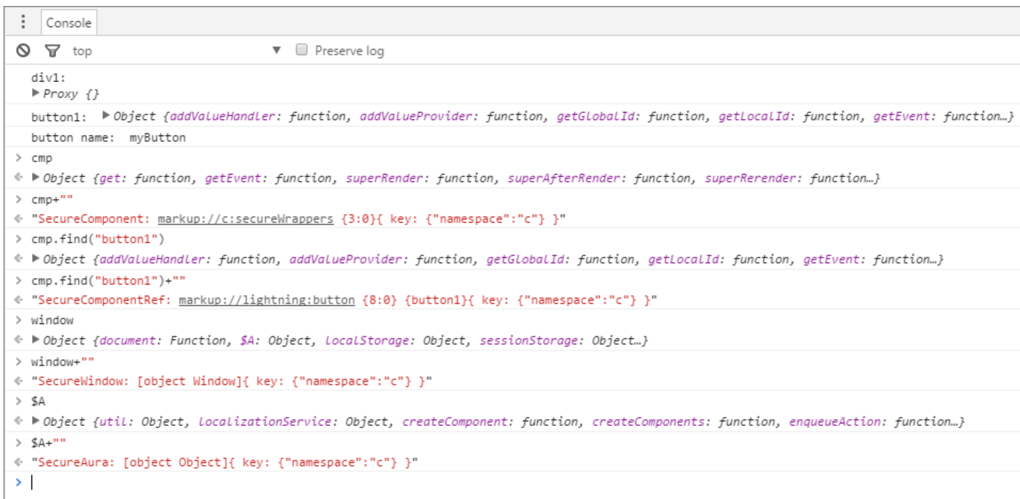
ブラウザコンソールの要素を調べることができるように、コードに debugger ステートメントが追加されています。

ブラウザコンソールに次の式を入力し、結果を確認します。

```
cmp
cmp+""
cmp.find("button1")
cmp.find("button1")+""
window
window+""
$A
$A+""
```

オブジェクトが String に変換されるように、いくつかの式に空の文字列が追加されています。toString() メソッドを使用することもできます。

出力は次のとおりです。



出力の一部を調べてみましょう。

cmp+""

`c:secureWrappers` コンポーネントを表す `cmp` の `SecureComponent` オブジェクトを返します。

cmp.find("button1")+""

異なる名前空間のコンポーネントの外部 API を表す `SecureComponentRef` を返します。この例では、このコンポーネントは `lightning:button` です。

window+""

`SecureWindow` オブジェクトを返します。

$A+""

`SecureAura` オブジェクトを返します。

このセクションの内容:

セキュアなラッパーの JavaScript API

`SecureWindow` などのセキュアなラッパーは、ラップするオブジェクトの API のサブセットを公開します。セキュアなラッパーの API は、LockerService API ビューアアプリケーションまたはリファレンスドキュメントアプリケーションで文書化されています。

関連トピック:

LockerService で Proxy オブジェクトを使用する方法

セキュアなラッパーの JavaScript API

`SecureWindow` などのセキュアなラッパーは、ラップするオブジェクトの API のサブセットを公開します。セキュアなラッパーの API は、LockerService API ビューアアプリケーションまたはリファレンスドキュメントアプリケーションで文書化されています。

LockerService API ビューア

LockerService API Viewer には、LockerService によって公開される DOM API が表示されます。API Viewer アプリケーションには、SecureDocument、SecureElement、および SecureWindow の API が表示されます。

API Viewer の現在の UI はまだ完璧ではありません。改善されるまでしばらくお待ちください。多くの情報がありますが、重要なポイントは緑色の背景は DOM メソッドがサポートされていることを示すということです。

Salesforce Lightning CLI ツールを使用して、コードをスキャンし、Lightning 固有の問題がないか調べます。

リファレンスドキュメントアプリケーション

リファレンスドキュメントアプリケーションの **[JavaScript API]** > **[コンポーネント]** には、SecureComponent の API がリストされます。

SecureAura は $A のラッパーです。

リファレンスドキュメントアプリケーションには、次の場所からアクセスします。

https://<myDomain>.lightning.force.com/auradocs/reference.app。<myDomain> は、カスタム Salesforce ドメインの名前です。

関連トピック:

グローバル参照のセキュアなラッパー

サポートされている JavaScript API フレームワークのメソッドのみへのアクセス

サポートされている公開済みの JavaScript API フレームワークのメソッドにのみアクセスできます。これらのメソッドは、https://<myDomain>.lightning.force.com/auradocs/reference.app (<myDomain> はカスタム Salesforce ドメインの名前) にあるリファレンスドキュメントアプリケーションで公開されています。以前はサポートされていないメソッドにアクセスできたため、サポートされていないメソッドが変更または削除されたときにコードが破損するおそれがありました。

LockerService による影響

LockService の影響を受けるものと、受けないものについて見てみましょう。

LockerService は、次で使用するカスタム Lightning コンポーネントのセキュリティとベストプラクティスを適用します。

- Lightning Experience
- Salesforce1
- Lightning コミュニティ
- 独自に作成したスタンドアロンアプリケーション (myApp.app など)
- カスタム Lightning コンポーネントを追加できるその他のアプリケーション (Lightning Experience の Salesforce コンソールなど)
- Lightning Out

LockerService は次には影響しません (これらのコンテキストでの Visualforce の Lightning コンポーネントの使用は除く)。

- Salesforce Classic
- Visualforce ベースのコミュニティ
- Salesforce Classic のアプリケーション (Salesforce Classic の Salesforce コンソールなど)

コンポーネントの LockerService の無効化

コンポーネントの API バージョンを 39.0 以下に設定すると、コンポーネントの LockerService を無効にできます。コンポーネントが API バージョン 40.0 以上に設定されている場合は、LockerService が有効になります。API バージョン 40.0 は、LockService がすべての組織で有効化された Summer '17 に相当します。

Summer '17 より前に作成されたコンポーネントは API バージョンが 40.0 未満であるため、LockerService は無効です。

コンポーネントにバージョン設定すると、コンポーネントを API バージョンに関連付けることができます。コンポーネントを作成するときのデフォルトのバージョンは最新の API バージョンです。開発者コンソールで、右パネルの **[Bundle Version Settings (バージョン設定を対応付ける)]** をクリックしてコンポーネントのバージョンを設定します。

一貫性を保ちデバッグを容易にするため、可能な場合はアプリケーションのすべてのコンポーネントに同じ API バージョンを設定することをお勧めします。

関連トピック:

コンポーネントの API バージョンを混在させない
コンポーネントのバージョン設定
開発者コンソールで Lightning コンポーネントを作成する

コンポーネントの API バージョンを混在させない

一貫性を保ちデバッグを容易にするため、アプリケーション、コンテインメント階層 (コンポーネント内のコンポーネント)、または拡張階層 (コンポーネントを拡張するコンポーネント) のすべてのコンポーネントに同じ API バージョンを設定することをお勧めします。

コンテインメント階層または拡張階層に API バージョンが混在し、LockerService が有効なコンポーネントと無効なコンポーネントがある場合、アプリケーションのデバッグが難しくなります。

拡張階層

LockerService は、コンポーネントのバージョンのみに基づいてコンポーネントまたはアプリケーションで有効化されます。コンポーネントの拡張階層は、LockerService の適用には考慮されません。

`Car` コンポーネントが `Vehicle` コンポーネントを拡張する例を見てみましょう。`Car` の API バージョンは 39.0 であるため、LockerService は無効です。`Vehicle` の API バージョンは 40.0 であるため、LockerService は有効です。

ここで、`window._counter` に値を割り当てることで、`Vehicle` の expando プロパティ `_counter` を `window` オブジェクトに追加するとします。`Vehicle` では LockerService が有効なため、`_counter` プロパティはコンポー

ネントの名前空間での window のセキュアなラッパー、SecureWindow に追加されます。このプロパティはネイティブの window オブジェクトには追加されません。

Car では LockerService が無効なため、コンポーネントはネイティブの window オブジェクトにアクセスできます。_counter プロパティは SecureWindow でのみ使用できるため、Car はこのプロパティを参照できません。

この微妙な動作によって、コードが期待どおりに機能しない場合の対処に苦労する可能性があります。そのデバッグに費やした時間は戻ってきません。拡張階層のすべてのコンポーネントで同じ API バージョンを使用すれば、このような問題を回避できます。

コンテインメント階層

アプリケーションまたはコンポーネント内のコンテインメント階層は、LockerService の適用には考慮されません。

Bicycle コンポーネントに Wheel コンポーネントが含まれる例を見てみましょう。Bicycle の API バージョンが 40.0 の場合、LockerService は有効です。Wheel の API バージョンが 39.0 の場合、LockerService が有効な Bicycle コンポーネントに含まれているにも関わらず、Wheel では LockerService が無効になります。

コンポーネントの API バージョンの混在によって、拡張階層と同様の問題が発生する可能性が高くなります。可能な場合は、アプリケーションまたはコンポーネント階層のすべてのコンポーネントに同じ API バージョンを設定することをお勧めします。

関連トピック:

コンポーネントのバージョン設定
コンポーネントの LockerService の無効化
グローバル参照のセキュアなラッパー
コンポーネント間の JavaScript コードの共有

サポートされていないブラウザで無効化された LockService

LockerService は、ブラウザの JavaScript 機能である厳格モード、Map オブジェクト、Proxy オブジェクトのサポートに依存します。ブラウザが要件を満たしていない場合、LockerService はそのすべてのセキュリティ機能を適用できず、無効になります。

LockService はサポートされていないブラウザで無効になります。サポートされていないブラウザを使用する場合、修正できない問題が発生する可能性があります。サポートされているブラウザを使用すれば、問題を回避してブラウザをより安全に操作できます。

メモ: LockerService 要件は、Lightning Experience でサポートされているブラウザ (IE11 を除く) と連携しています。IE11 では LockerService は無効になります。セキュリティを強化するため、IE11 以外のサポートされているブラウザを使用することをお勧めします。

関連トピック:

Lightning コンポーネントのブラウザサポートの考慮事項
Salesforce ヘルプ: Lightning Experience でサポートされるブラウザ

コンテンツセキュリティポリシーの概要

Lightning コンポーネントフレームワークでは、W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御します。

CSP は、Web アプリケーションセキュリティに関する W3C ワーキンググループの勧告候補です。このフレームワークでは、W3C が推奨する `Content-Security-Policy` HTTP ヘッダーを使用しています。

フレームワークの CSP は、次のリソースに対応しています。

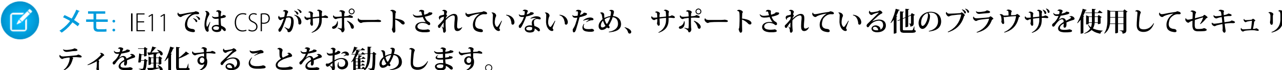
JavaScript ライブラリ
すべての JavaScript ライブラリは、Salesforce 静的リソースにアップロードする必要があります。詳細は、「外部 JavaScript ライブラリの使用」(ページ 276) を参照してください。

リソースの HTTPS 接続
すべての外部フォント、画像、フレーム、および CSS は、HTTPS URL を使用する必要があります。

CSP 信頼済みサイトを追加すれば、CSP ポリシーを変更して、サードパーティリソースへのアクセスを拡張できます。

ブラウザサポート

CSP が適用されないブラウザもあります。CSP が適用されるブラウザのリストについては、`caniuse.com` を参照してください。



メモ: IE11 では CSP がサポートされていないため、サポートされている他のブラウザを使用してセキュリティを強化することをお勧めします。

CSP 違反の検出

ポリシー違反は、ブラウザの開発者コンソールのログに記録されます。違反は次のようなメッセージになります。

```
Refused to load the script 'https://externaljs.docsample.com/externalLib.js'
because it violates the following Content Security Policy directive: ...
```

アプリケーションの機能に影響がない場合は、CSP 違反を無視できます。

このセクションの内容:

厳格な CSP 制限の重要な更新

Lightning コンポーネントフレームワークでは、すでに W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御しています。重要な更新「Lightning コンポーネントの厳格なコンテンツセキュリティポリシーの有効化」は、CSP を強化して、クロスサイトスクリプティング攻撃のリスクを軽減します。厳格な CSP は、Sandbox 組織および Developer Edition 組織でのみ適用されます。

関連トピック:

Lightning コンポーネントのブラウザサポートの考慮事項

コンポーネントからの API コールの実行

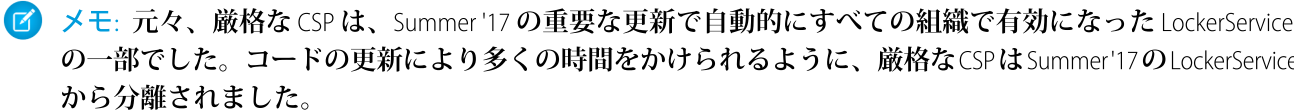
サードパーティ API にアクセスするための CSP 信頼済みサイトの作成

Salesforce ヘルプ: Lightning Experience でサポートされるブラウザ

厳格な CSP 制限の重要な更新

Lightning コンポーネントフレームワークでは、すでに W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御しています。重要な更新「Lightning コンポーネントの厳格なコンテンツセキュリティポリシーの有効化」は、CSP を強化して、クロスサイトスクリプティング攻撃のリスクを軽減します。厳格な CSP は、Sandbox 組織および Developer Edition 組織でのみ適用されます。

厳格な CSP によってインラインスクリプト (`script-src`) の `unsafe-inline` および `unsafe-eval` キーワードが禁じられます。`eval()` またはインライン JavaScript コード実行を使用してすべてのコールを削除し、使用するコードとサードパーティライブラリがこれらのルールに従っていることを確認してください。サードパーティライブラリを `unsafe-inline` や `unsafe-eval` に依存しない最近のバージョンに更新する必要がある可能性があります。



メモ: 元々、厳格な CSP は、Summer '17 の重要な更新で自動的にすべての組織で有効になった LockerService の一部でした。コードの更新により多くの時間をかけられるように、厳格な CSP は Summer '17 の LockerService から分離されました。

重要な更新のスケジュール

厳格な CSP は、他の組織でも段階的に使用可能になります。予定されているスケジュールを次に示しますが、このスケジュールは今後のリリースで変更される可能性があります。

Summer '17

重要な更新は、Sandbox 組織と Developer Edition 組織でのみ利用可能です。

Winter '18 (将来の計画)

重要な更新は、本番組織を含むすべての組織に適用されます。

Winter '19 (将来の計画)

重要な更新が期限切れになったときに、すべての組織で自動的に有効化されます。

重要な更新の有効化

厳格な CSP は、重要な更新「Lightning LockerService セキュリティの有効化」が以前に有効化されている Sandbox 組織と Developer Edition 組織でデフォルトで有効になります。その他すべての Sandbox 組織と Developer Edition 組織では、厳格な CSP はデフォルトで無効になります。

厳格な CSP を有効にする手順は、次のとおりです。

1. [設定] から、[クイック検索] ボックスに「*重要な更新*」と入力して、[重要な更新] を選択します。
2. 「Lightning コンポーネントの厳格なコンテンツセキュリティポリシーの有効化」で、[有効化]をクリックします。
3. ブラウザページを更新して、厳格な CSP の有効化に進みます。

この重要な更新による影響

重要な更新「Lightning コンポーネントの厳格なコンテンツセキュリティポリシーの有効化」により、Sandbox 組織と Developer Edition 組織で CSP を厳格化します。

- Lightning Experience
- Salesforce1
- 独自に作成したスタンドアロンアプリケーション (`myApp.app` など)

メモ: コミュニティの CSP を厳格化するものとして、別個の重要な更新「コミュニティでの Lightning コンポーネントの厳格なコンテンツセキュリティポリシーの有効化」があります。

この重要な更新は次には影響しません。

- Salesforce Classic
- Salesforce Classic のアプリケーション (Salesforce Classic の Salesforce コンソールなど)
- Lightning アプリケーション外のコンテナにある Lightning コンポーネント (Visualforce や Visualforce ベースのコミュニティの Lightning コンポーネントなど) を実行できる Lightning Out。そのコンテナで CSP ルールが定義されます。

Salesforce Lightning CLI

Lightning CLI は Heroku Toolbelt プラグインで、コードをスキャンして一般的な JavaScript コーディングの問題や Lightning 固有の問題がないか調べることができます。このツールは、LockerService を有効にするための Lightning コンポーネントコードを準備する上で有用です。

Lightning CLI は、オープンソースの ESLint プロジェクトを基盤としたリンティングツールです。ESLint と同様に、CLI ツールはコードに一般的な JavaScript の問題が見つかるとフラグを設定します。

Lightning CLI は、LockerService に関連する特定の問題に対してアラームを表示します。フラグが設定される問題として、Lightning コンポーネントのコードの誤りや、サポート対象外または非公開の JavaScript API メソッドの使用などが挙げられます。Lightning CLI は Heroku Toolbelt 内にインストールされ、コマンドラインで使用します。

このセクションの内容:

Salesforce Lightning CLI のインストール

Lightning CLI を Heroku Toolbelt プラグインとしてインストールします。次に、Heroku Toolbelt を更新して最新の Lightning CLI ルールを取得します。

Salesforce Lightning CLI の使用

Lightning CLI は、他の lint コマンドラインツールと同じように実行できます。ただし、heroku コマンドを使用して呼び出す必要があります。シェルウィンドウに結果が表示されます。

問題の確認と解決

Lightning コンポーネントコードに対して Lightning CLI を実行すると、スキャンされたファイルで見つかった問題ごとに結果が出力されます。結果を確認し、コードの問題を解決します。

Salesforce Lightning CLI ルール

Lightning CLI に組み込まれたルールは、LockerService での制限、Lightning API の適切な使用、および Lightning コンポーネントコードの記述に関する多くのベストプラクティスに対応しています。各ルールはコードでトリガされると、コード内の問題が疑われる領域を指摘します。

Salesforce Lightning CLI のオプション

Lightning CLI の動作を変更する複数のオプションがあります。

Salesforce Lightning CLI のインストール

Lightning CLI を Heroku Toolbelt プラグインとしてインストールします。次に、Heroku Toolbelt を更新して最新の Lightning CLI ルールを取得します。

Lightning CLI は Heroku Toolbelt に依存します。Lightning CLI の使用を試みる前に、`heroku` コマンドが正しくインストールされていることを確認します。Heroku Toolbelt についての詳細は、次のページを参照してください。

https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up

Heroku Toolbelt が稼働したら、次のコマンドを使用して Lightning CLI プラグインをインストールします。

```
heroku plugins:install salesforce-lightning-cli
```

インストールされたプラグインは、`heroku update` コマンドを使用して Heroku Toolbelt を更新するたびに更新されます。週 1 回程度、更新コマンドを実行して Lightning CLI ルールを最新の状態に維持してください。

Salesforce Lightning CLI の使用

Lightning CLI は、他の lint コマンドラインツールと同じように実行できます。ただし、`heroku` コマンドを使用して呼び出す必要があります。シェルウィンドウに結果が表示されます。

通常の使用方法

リンティングツールである Lightning CLI は、Lightning コンポーネントが含まれるフォルダに対して実行できます。

```
heroku lightning:lint ./path/to/lightning/components/
```

メモ: Lightning CLI は、ローカルファイルに対してのみ実行されます。メタデータ API またはツール (Force.com IDE、Force.com 移行ツール、さまざまなサードパーティオプションなど) を使用して、コンポーネントコードをマシンにダウンロードします。

デフォルトの出力にはエラーのみが表示されます。警告も表示するには、verbose モードオプションを使用します。

Lightning CLI 実行の出力の確認方法については、「問題の確認と解決」を参照してください。

一般的なオプション

ファイルの絞り込み

特定の種類のファイルのみをスキャンする場合もあります。`--files` 引数を使用すると、ファイルを照合するパターンを設定できます。

たとえば、次のコマンドではコントローラのみをスキャンできます。

```
heroku lightning:lint ./path/to/lightning/components/ --files **/*Controller.js
```

冗長モード

デフォルトの出力にはエラーのみが表示されるため、より大きな問題に集中できます。`--verbose` 引数を使用すると、リンティングプロセス中に警告メッセージとエラーを表示できます。

関連トピック:

Salesforce Lightning CLI のオプション

問題の確認と解決

Lightning コンポーネントコードに対して Lightning CLI を実行すると、スキャンされたファイルで見つかった問題ごとに結果が出力されます。結果を確認し、コードの問題を解決します。

出力例を次に示します。

```
error     secure-document    Invalid SecureDocument API
  Line:109:29
  scrapping = document.innerHTML;
  ^

  warning   no-plusplus   Unary operator '++' used
  Line:120:50
  for (var i = (index+1); i < sibs.length; i++) {
  ^

  error    secure-window  Invalid SecureWindow API
  Line:33:21
  var req = new XMLHttpRequest();
  ^

  error  default-case  Expected a default case
  Line:108:13
```

```
switch (e.keyCode) {
^
```

問題が警告またはエラーごとに1つずつ表示されます。各問題には、行番号、重要度、および問題の短い説明が含まれます。ルール名も含まれます。これを使用して、問題のより詳細な説明を検索できます。Lightning CLI で適用されるルールと、可能な解決策とオプションについては、「Salesforce Lightning CLI ルール」を参照してください。

各問題を確認し、当該のコードを調査し、実際に問題があった場合は修正してすべて排除する必要があります。

完璧な自動ツールはありませんが、コードに含まれる実際の問題の大半はLightning CLI で生成されるエラーと警告により指摘されるものと考えられます。LockerService を有効にしてコードを使用する前にこうした問題を修正することを計画する必要があります。

関連トピック:

Salesforce Lightning CLI ルール

Salesforce Lightning CLI ルール

Lightning CLI に組み込まれたルールは、LockerService での制限、Lightning API の適切な使用、および Lightning コンポーネントコードの記述に関する多くのベストプラクティスに対応しています。各ルールはコードでトリガされると、コード内の問題が疑われる領域を指摘します。

Lightning CLI では、Salesforce で作成された Lightning 固有のルールに加え、ESLint から追加された他のルールも有効化されています。こうしたルールに関するドキュメントは、ESLint プロジェクトサイトから入手できます。ここで説明されていないエラーまたは警告がルールから表示された場合は、ESLint ルールページで検索してください。

このセクションの内容:

JavaScript 組み込み API (ecma-intrinsics) の検証

このルールは、JavaScript の組み込み API (正式名称: ECMAScript) を処理します。

Aura API (aura-api) の検証

このルールは、フレームワーク APIの使用が公開されているドキュメントに従っているかどうかを検証します。ドキュメント化されていない機能や非公開機能の使用は許可されません。

Lightning コンポーネントの公開 API (secure-component) の検証

このルールは、公開されているサポート対象のフレームワーク API関数のみを検証し、プロパティが使用されます。

セキュアドキュメント公開 API (secure-document) の検証

このルールは、サポート対象の関数のみを検証し、`document` グローバルのプロパティが使用されます。

セキュアウィンドウ公開 API (secure-window) の検証

このルールは、サポート対象の関数のみを検証し、`window` グローバルのプロパティが使用されます。

独自のカスタムルール

Salesforce Lightning CLI がコードに適用する JavaScript スタイルルールをカスタマイズできます。

JavaScript 組み込み API (ecma-intrinsics) の検証

このルールは、JavaScript の組み込み API (正式名称: ECMAScript) を処理します。

LockerService が有効化されている場合、フレームワークによってサポート対象外の API オブジェクトまたはコールの使用が阻止されます。つまり、Lightning コードで使用が許可されるのは次の機能です。

- JavaScript に組み込まれた機能 (「組み込み」機能)
- 公開され、サポートされている、Lightning コンポーネントフレームワークに組み込まれた機能
- 公開され、サポートされている、LockerService *SecureObject* オブジェクトに組み込まれた機能

「組み込み API」とは何でしょうか?それは、ECMAScript 言語仕様で定義された API のことです。つまり、JavaScript に組み込まれている API です。これには仕様の付録 B が含まれます。この付録では、JavaScript の「コア」には含まれていないにも関わらず、Web ブラウザ内で実行される JavaScript ではまだサポートされている従来のブラウザ機能を扱っています。

組み込みと考えられている JavaScript の機能 (`window` および `document` グローバル変数など) には、より制約の多い API を提供する *SecureObject* オブジェクトに置き換えられたものもあります。

ルールの詳細

このルールは、組み込み JavaScript API の使用が公開されている仕様に従っているかどうかを検証します。非標準、非推奨、削除済みの言語機能の使用は許可されません。

関連資料

- ECMAScript 仕様
- Annex B: Additional ECMAScript Features for Web Browsers (付録 B: Web ブラウザ用のその他の ECMAScript 機能)
- Intrinsic Objects (JavaScript) (組み込みオブジェクト (JavaScript))

関連トピック:

Aura API (aura-api) の検証

Lightning コンポーネントの公開 API (secure-component) の検証

セキュアドキュメント公開 API (secure-document) の検証

セキュアウィンドウ公開 API (secure-window) の検証

Aura API (aura-api) の検証

このルールは、フレームワーク API の使用が公開されているドキュメントに従っているかどうかを検証します。ドキュメント化されていない機能や非公開機能の使用は許可されません。

LockerService が有効化されている場合、フレームワークによってサポート対象外の API オブジェクトまたはコールの使用が阻止されます。つまり、Lightning コードで使用が許可されるのは次の機能です。

- JavaScript に組み込まれた機能 (「組み込み」機能)
- 公開され、サポートされている、Lightning コンポーネントフレームワークに組み込まれた機能
- 公開され、サポートされている、LockerService *SecureObject* オブジェクトに組み込まれた機能

このルールは、サポートされている公開フレームワーク API を対象とします (フレームワークグローバル `$A` で使用可能な API など)。

このルールは「Aura API」と呼ばれています。それは、Lightning コンポーネントフレームワークの基礎となるのがオープンソースの Aura フレームワークであるためです。さらにこのルールは、Lightning コンポーネント固有の使用ではなく、Aura フレームワークで許可されている使用を検証します。

ルールの詳細

次のパターンは問題と見なされます。

```
Aura.something(); // Use $A instead
$A.util.fake(); // fake is not available in $A.util
```

関連資料

`$A` を含む、フレームワークで使用可能なすべてのメソッドについての詳細は、`https://myDomain.lightning.force.com/auradocs/reference.app` の JavaScript API を参照してください (*`myDomain`* はカスタム Salesforce ドメインの名前)。

関連トピック:

Lightning コンポーネントの公開 API (secure-component) の検証

セキュアドキュメント公開 API (secure-document) の検証

セキュアウィンドウ公開 API (secure-window) の検証

Lightning コンポーネントの公開 API (`secure-component`) の検証

このルールは、公開されているサポート対象のフレームワーク API 関数のみを検証し、プロパティが使用されます。

LockerService が有効化されている場合、フレームワークによってサポート対象外の API オブジェクトまたはコールの使用が阻止されます。つまり、Lightning コードで使用が許可されるのは次の機能です。

- JavaScript に組み込まれた機能 (「組み込み」機能)
- 公開され、サポートされている、Lightning コンポーネントフレームワークに組み込まれた機能
- 公開され、サポートされている、LockerService *SecureObject* オブジェクトに組み込まれた機能

LockerService より前にコンポーネントへの参照を作成または取得する場合、公開されていないものも含め、任意の関数をコールし、そのコンポーネントで使用可能な任意のプロパティにアクセスできます。LockerService が有効化されると、コンポーネントは新しい SecureComponent オブジェクトで「ラップ」されます。このオブジェクトがコンポーネントとその関数およびプロパティへのアクセスを制御します。SecureComponent によって、公開されているサポート対象のコンポーネント API のみを使用するように制限されます。

ルールの詳細

リファレンスドキュメントアプリケーションには、`SecureComponent` の API がリストされます。リファレンスドキュメントアプリケーションには、次の場所からアクセスします。

https://<myDomain>.lightning.force.com/auradocs/reference.app。<myDomain> は、カスタム Salesforce ドメインの名前です。

SecureComponent の API は、**[JavaScript API]** > [コンポーネント] にリストされます。

関連資料

- SecureComponent.js 実装

関連トピック:

Aura API (aura-api) の検証

セキュアドキュメント公開 API (secure-document) の検証

セキュアウィンドウ公開 API (secure-window) の検証

セキュアドキュメント公開 API (secure-document) の検証

このルールは、サポート対象の関数のみを検証し、document グローバルのプロパティが使用されます。

LockerService が有効化されている場合、フレームワークによってサポート対象外の API オブジェクトまたはコールの使用が阻止されます。つまり、Lightning コードで使用が許可されるのは次の機能です。

- JavaScript に組み込まれた機能 (「組み込み」機能)
- 公開され、サポートされている、Lightning コンポーネントフレームワークに組み込まれた機能
- 公開され、サポートされている、LockerService *SecureObject* オブジェクトに組み込まれた機能

LockerService より前に document グローバルにアクセスする場合、任意の関数をコールし、使用可能なプロパティにアクセスできます。LockerService が有効化されると、document グローバルは新しい SecureDocument オブジェクトで「ラップ」されます。このオブジェクトが document とその関数およびプロパティへのアクセスを制御します。SecureDocument によって、document グローバルの「安全な」機能のみを使用するように制限されます。

関連資料

- SecureDocument.js 実装

関連トピック:

Aura API (aura-api) の検証

Lightning コンポーネントの公開 API (secure-component) の検証

セキュアウィンドウ公開 API (secure-window) の検証

セキュアウィンドウ公開 API (secure-window) の検証

このルールは、サポート対象の関数のみを検証し、window グローバルのプロパティが使用されます。

LockerService が有効化されている場合、フレームワークによってサポート対象外の API オブジェクトまたはコールの使用が阻止されます。つまり、Lightning コードで使用が許可されるのは次の機能です。

- JavaScript に組み込まれた機能 (「組み込み」機能)

- 公開され、サポートされている、Lightning コンポーネントフレームワークに組み込まれた機能
- 公開され、サポートされている、LockerService *SecureObject* オブジェクトに組み込まれた機能

LockerService より前に `window` グローバルにアクセスする場合、任意の関数をコールし、使用可能なプロパティにアクセスできます。LockerService が有効化されると、`window` グローバルは新しい SecureWindow オブジェクトで「ラップ」されます。このオブジェクトが `window` とその関数およびプロパティへのアクセスを制御します。SecureWindow によって、`window` グローバルの「安全な」機能のみを使用するように制限されます。

関連資料

- SecureWindow.js 実装

関連トピック:

Aura API (aura-api) の検証
Lightning コンポーネントの公開 API (secure-component) の検証
セキュアドキュメント公開 API (secure-document) の検証

独自のカスタムルール

Salesforce Lightning CLI がコードに適用する JavaScript スタイルルールをカスタマイズできます。

組織やプロジェクトが異なれば、採用される JavaScript ルールも異なることはよくあります。Lightning CLI ツールを使用すると、Salesforce コーディング規則を強制せずに、LockerService に対応できるようになります。そのために、Lightning CLI ルールは、セキュリティルールとスタイルルールの 2 つのセットに分かれています。セキュリティルールは変更できませんが、スタイルルールの変更や追加はできます。

カスタムルール設定ファイルを指定するには、`--config` 引数を使用します。カスタムルール設定ファイルを使用すると、独自のコードスタイルルールを定義し、Lightning CLI ツールで使用されるスタイルルールに反映させることができます。

メモ: カスタムルールの失敗によって警告が生成されても、デフォルトの出力に警告は表示されません。警告を表示するには、`--verbose` フラグを使用します。

Lightning CLI のデフォルトのスタイルルールは下記を参照してください。このルールを新しいファイルにコピーし、目的のスタイルルールに合わせて変更します。または、既存の ESLint ルール設定ファイルを直接使用してください。以下に例を示します。

```
heroku lightning:lint ./path/to/lightning/components/ --config ~/.eslintrc
```

メモ: `--config` を使用した追加や変更ができない ESLint ルールもあります。Lightning プラットフォームのコンテキストで安全または中立と見なされたルールのみが、Lightning CLI で有効化されます。セキュリティルールは上書きできません。

デフォルトのスタイルルール

次に、Lightning CLI で使用されるデフォルトのスタイルルールを示します。

```
/*
 * Copyright (C) 2016 salesforce.com, inc.
```

```
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*         http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

module.exports = {
   rules: {
       // code style rules, these are the default value, but the user can
       // customize them via --config in the linter by providing custom values
       // for each of these rules.
       "no-trailing-spaces": 1,
       "no-spaced-func": 1,
       "no-mixed-spaces-and-tabs": 0,
       "no-multi-spaces": 0,
       "no-multiple-empty-lines": 0,
       "no-lone-blocks": 1,
       "no-lonely-if": 1,
       "no-inline-comments": 0,
       "no-extra-parens": 0,
       "no-extra-semi": 1,
      "no-warning-comments": [0, { "terms": ["todo", "fixme", "xxx"], "location": "start"
}],
       "block-scoped-var": 1,
       "brace-style": [1, "1tbs"],
       "camelcase": 1,
       "comma-dangle": [1, "never"],
       "comma-spacing": 1,
       "comma-style": 1,
       "complexity": [0, 11],
       "consistent-this": [0, "that"],
       "curly": [1, "all"],
       "eol-last": 0,
       "func-names": 0,
       "func-style": [0, "declaration"],
       "generator-star-spacing": 0,
       "indent": 0,
       "key-spacing": 0,
       "keyword-spacing": [0, "always"],
       "max-depth": [0, 4],
       "max-len": [0, 80, 4],
       "max-nested-callbacks": [0, 2],
       "max-params": [0, 3],
       "max-statements": [0, 10],
       "new-cap": 0,
       "newline-after-var": 0,
```

```
        "one-var": [0, "never"],
        "operator-assignment": [0, "always"],
        "padded-blocks": 0,
        "quote-props": 0,
        "quotes": 0,
        "semi": 1,
        "semi-spacing": [0, {"before": false, "after": true}],
        "sort-vars": 0,
        "space-after-function-name": [0, "never"],
        "space-before-blocks": [0, "always"],
        "space-before-function-paren": [0, "always"],
        "space-before-function-parentheses": [0, "always"],
        "space-in-brackets": [0, "never"],
        "space-in-parens": [0, "never"],
        "space-infix-ops": 0,
        "space-unary-ops": [1, { "words": true, "nonwords": false }],
        "spaced-comment": [0, "always"],
        "vars-on-top": 0,
        "valid-jsdoc": 0,
        "wrap-regex": 0,
        "yoda": [1, "never"]
    }
};
```

Salesforce Lightning CLI のオプション

Lightning CLI の動作を変更する複数のオプションがあります。

次のオプションを使用できます。

オプション	説明
`-i`、`--ignore` *IGNORE*	フォルダを無視するためのパターン。たとえば、`foo` という名前をフォルダを無視する場合、次のようになります。 `--ignore **/foo/**`
`--files` *FILES*	特定のファイルのみを含めるためのパターン。デフォルトはすべての `.js` ファイルです。たとえば、クライアント側コントローラのみを含める場合、次のようになります。 `--files **/*Controller.js`
`-j`、`--json`	JSON を出力して、他のツールとのインテグレーションを促進します。このオプションを指定しないと、デフォルトの標準テキスト出力形式になります。
`--config` *CONFIG*	カスタム ESLint 設定へのパス。コードスタイルルールのみが選択され、残りは無視されます。以下に例を示します。 `--config path/to/.eslintrc`

オプション	説明
--verbose	エラーと警告をレポートします。デフォルトでは、Lightning CLI はエラーのみをレポートします。

Lightning CLI では、いくつかの組み込みヘルプも提供します。このヘルプには次のコマンドでいつでもアクセスできます。

```
heroku lightning --help
heroku lightning:lint --help
```

関連トピック:

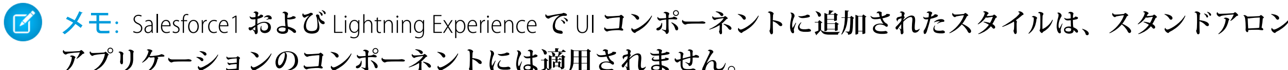
Salesforce Lightning CLI の使用

アプリケーションのスタイル設定

アプリケーションは、.app リソース内にマークアップが含まれている特殊な最上位コンポーネントです。他のコンポーネントと同様に、そのバンドルに CSS を <appName>.css というリソースで入れることができます。

たとえば、アプリケーションのマークアップが notes.app にある場合、そのアプリケーションの CSS は notes.css です。

Salesforce1 および Lightning Experience で表示される UI コンポーネントには、その表示テーマと一致するスタイル設定が含まれます。たとえば、ui:button には、ニュートラルスタイルを表示する button--neutral クラスが含まれます。ui:input を拡張する入力コンポーネントには、他のスタイル設定に加えてカスタムフォントを使用する入力項目を表示する uiInput--input クラスが含まれます。



メモ: Salesforce1 および Lightning Experience で UI コンポーネントに追加されたスタイルは、スタンドアロンアプリケーションのコンポーネントには適用されません。

このセクションの内容:

アプリケーションでの Salesforce Lightning Design System の使用

Salesforce Lightning Design System では、Lightning Experience と一貫性のあるデザインを作成できます。Lightning Design System スタイルを使用すれば、標準のスタイルをリバースエンジニアリングしなくても Salesforce と一貫性のある UI をカスタムアプリケーションに設定できます。

外部 CSS の使用

静的リソースとしてアップロードした外部 CSS リソースを参照するには、.cmp または .app マークアップで <ltng:require> タグを使用します。

join 式を使用したスタイル設定マークアップの可読性の向上

コンポーネントの属性値に基づいて適用するクラス名を指定するときに、マークアップがややこしくなることがあります。マークアップを読みやすくするために、join を使用することを検討します。

コンポーネントの CSS のヒント

Lightning ページ、Lightning アプリケーションビルダー、またはコミュニティビルダーで使用するコンポーネントの CSS を設定する場合のいくつかのヒントを次に示します。

設計トークンを使用したスタイル設定

ビジュアルデザインの重要な値を名前付きトークンに取得します。トークンの値を一度定義すると、Lightning コンポーネントの CSS リソース全体で再利用できます。トークンを使用することで、設計の一貫性を確保しやすくなり、設計の変化に伴った更新がさらに簡単になります。

関連トピック:

コンポーネント内の CSS

Salesforce1 のカスタムタブとしての Lightning コンポーネントの追加

アプリケーションでの Salesforce Lightning Design System の使用

Salesforce Lightning Design System では、Lightning Experience と一貫性のあるデザインを作成できます。Lightning Design System スタイルを使用すれば、標準のスタイルをリバースエンジニアリングしなくても Salesforce と一貫性のある UI をカスタムアプリケーションに設定できます。

独自アプリケーションで `force:slds` を拡張すると、自動的に Lightning Design System のスタイルとデザイントークンが使用されます。これが、Lightning Design System 機能強化が常に最新の状態になり、一貫性を維持する最も簡単な方法です。

`force:slds` を拡張する方法は、次のとおりです。

```
<aura:application extends="force:slds">
    <!-- customize your application here -->
</aura:application>
```

静的リソースの使用

`force:slds` を拡張すると、CSS が変更されるたびに Lightning Design System スタイルのバージョンが自動的に更新されます。特定の Lightning Design System バージョンを使用する場合は、そのバージョンをダウンロードし、組織に静的リソースとして追加します。

メモ: 最新の Lightning Design System スタイルが自動的に使用できるよう、`force:slds` を拡張することをお勧めします。特定の Lightning Design System バージョンを使用し続けると、アプリケーションのスタイルが徐々に Lightning Experience のその後のバージョンからずれてしまったり、重複した CSS をダウンロードしてしまったりします。

Lightning Design System の最新バージョンをダウンロードするには、Lightning Design System を生成してダウンロードします。

Lightning Design System アーカイブの静的リソースの名前は SLDS*###* という形式にすることをお勧めします。*###* は Lightning Design System のバージョン番号 (例: *SLDS203*) です。こうすることで、複数のバージョンの Lightning Design System をインストールでき、コンポーネントのバージョンの利用状況を管理できます。

コンポーネントで Lightning Design System の静的バージョンを使用するには、<ltng:require/> を使用してその静的バージョンを含めます。以下に例を示します。

```
<aura:component>
    <ltng:require
       styles="{!$Resource.SLDS203 + '/assets/styles/lightning-design-system-ltng.css'}"/>
</aura:component>
```

関連トピック:

設計トークンを使用したスタイル設定

外部 CSS の使用

静的リソースとしてアップロードした外部 CSS リソースを参照するには、.cmp または .app マークアップで <ltng:require> タグを使用します。

次に <ltng:require> の使用例を示します。

```
<ltng:require styles="{!$Resource.resourceName}" />
```

resourceName は、静的リソースの [名前] です。管理パッケージでは、リソース名にパッケージ名前空間プレフィックス ($Resource.yourNamespace__resourceName など) を含める必要があります。個々のグラフィックやスクリプトなど、スタンドアロンの静的リソースの場合、必要なのはそれだけです。アーカイブ静的リソース内の項目を参照するには、項目へのパスの残りを文字列の連結を使用して含めます。

次に、スタイルの読み込みに関する考慮事項を示します。

CSS のセットの読み込み

CSS のセットを読み込むには、styles 属性でリソースのカンマ区切りのリストを指定します。

メモ: 式で $Resource が解析される方法に予測できない動作があるため、複数の $Resource 参照を 1 つの属性に含めるには join 演算子を使用します。たとえば、コンポーネントに追加するスタイルシートが複数ある場合、styles 属性は次のようにする必要があります。

```
styles="{!join(',',
    $Resource.myStyles + '/stylesheetOne.css',
    $Resource.myStyles + '/moreStyles.css')}"
```

読み込み順序

スタイルはリストの順序で読み込まれます。

1 回のみの読み込み

同じコンポーネントまたは異なるコンポーネントの複数の <ltng:require> タグでスタイルが指定されていても、スタイルが読み込まれるのは 1 回のみです。

カプセル化

カプセル化と再利用性を確保するには、CSS リソースを使用するすべての .cmp または .app リソースに <ltng:require> タグを追加します。

<ltng:require> には、JavaScript ライブラリのリストを読み込む scripts 属性もあります。afterScriptsLoaded イベントを使用すると、scripts の読み込み後にコントローラアクションをコールで

きます。これは scripts を読み込むことによってのみトリガされ、styles の CSS が読み込まれたときにトリガされることはありません。

静的リソースについての詳細は、Salesforce オンラインヘルプの「静的リソース」を参照してください。

Lightning Experience または Salesforce1 のスタイル設定コンポーネント

Lightning Experience または Salesforce1 でのスタイル設定の競合を回避するには、外部 CSS に一意の名前空間プレフィックスを追加します。たとえば、外部CSS宣言に .myBootstrap をプレフィックスとして追加する場合、myBootstrap クラスを指定する <div> タグでコンポーネントマークアップを囲みます。

```
<ltng:require styles="{!$Resource.bootstrap}"/>
<div class="myBootstrap">
    <c:myComponent />
    <!-- Other component markup -->
</div>
```

メモ: CSS に一意の名前空間プレフィックスを追加する必要があるのは外部 CSS のみです。コンポーネントバンドル内の CSS を使用している場合、実行中は .THIS キーワードが .namespaceComponentName になります。

関連トピック:

外部 JavaScript ライブラリの使用

コンポーネント内の CSS

$Resource

join 式を使用したスタイル設定マークアップの可読性の向上

コンポーネントの属性値に基づいて適用するクラス名を指定するときに、マークアップがややこしくなることがあります。マークアップを読みやすくするために、join を使用することを検討します。

次の例では、コンポーネントの属性値に基づいて適用するクラス名を設定します。このコードは判読可能ですが、クラス名間のスペースを忘れがちです。

```
<li class="{! 'calendarEvent ' +
    v.zoomDirection + ' ' +
    (v.past ? 'pastEvent ' : '') +
    (v.zoomed ? 'zoom ' : '') +
    (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

マークアップが数行に分割されていないような場合には、わかりづらくイライラします。

```
<li class="{! 'calendarEvent ' + v.zoomDirection + ' ' + (v.past ? 'pastEvent ' : '') +
(v.zoomed ? 'zoom ' : '') + (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

マークアップを読みやすくするために、代わりに join を使用することを検討します。次の例の join 式は、' ' を最初の引数として設定するため、式の後続の引数ごとに指定し直す必要がありません。

```
<li
    class="{! join(' ',
        'calendarEvent',
        v.zoomDirection,
        v.past ? 'pastEvent' : '',
        v.zoomed ? 'zoom' : '',
        v.multiDayFragment ? 'multiDayFragment' : ''
    )}">
    <!-- content here -->
</li>
```

また、join 式は動的なスタイル設定にも使用できます。

```
<div style="{! join(';',
    'top:' + v.timeOffsetTop + '%',
    'left:' + v.timeOffsetLeft + '%',
    'width:' + v.timeOffsetWidth + '%'
)}">
    <!-- content here -->
</div>
```

関連トピック:

式の関数のリファレンス

コンポーネントの CSS のヒント

Lightning ページ、Lightning アプリケーションビルダー、またはコミュニティビルダーで使用するコンポーネントの CSS を設定する場合のいくつかのヒントを次に示します。

コンポーネントの幅は 100% に設定する必要がある

コンポーネントは Lightning ページの別の場所に移動されることがあるため、固有の幅や左右の余白は設定しないでください。コンポーネントが表示されるコンテナの幅の 100% に設定する必要があります。左右の余白を追加すると、コンポーネントの幅が変更されてページのレイアウトが崩れる可能性があります。

ドキュメントのフローから HTML 要素を削除しない

一部の CSS ルールは、ドキュメントのフローから HTML 要素を削除します。以下に例を示します。

```
float: left;
float: right;
position: absolute;
position: fixed;
```

コンポーネントはページの別の場所に移動されることがあるだけでなく、完全に異なるページで使用されることもあるため、通常のドキュメントフローに従う必要があります。フロートと絶対位置または固定位置を使用すると、コンポーネントが配置されたページのレイアウトが崩れます。*自分が表示しているページ*のレイアウトが崩れていないとしても、コンポーネントを配置できる*他の*ページのレイアウトは崩れる可能性があります。

子要素にはルート要素よりも大きいスタイルを設定しない

Lightning ページはコンポーネント間のスペースを一定に保ちますが、子要素がルート要素よりも大きい場合はスペースを一定にできません。

たとえば、次のパターンは使用しないでください。

```
<div style="height: 100px">
  <div style="height: 200px">
    <!--Other markup here-->
  </div>
</div>
```

```
<!--Margin increases the element's effective size-->
<div style="height: 100px">
  <div style="height: 100px margin: 10px">
    <!--Other markup here-->
  </div>
</div>
```

ベンダープレフィックス

`-moz-` や `-webkit-` および他のベンダープレフィックスは、Lightning で自動的に追加されます。

プレフィックスなしのバージョンを作成するだけで十分です。フレームワークにより、CSS 出力の生成時に必要なプレフィックスが自動的に追加されます。プレフィックスの追加を選択すると、そのままの状態で使用されます。これにより、特定のプレフィックスに対して代替値を指定できます。

例: 次の例は、`border-radius` のプレフィックスなしのバージョンです。

```
.class {
  border-radius: 2px;
}
```

前述の宣言の結果、次の宣言になります。

```
.class {
  -webkit-border-radius: 2px;
  -moz-border-radius: 2px;
  border-radius: 2px;
}
```

設計トークンを使用したスタイル設定

ビジュアルデザインの重要な値を名前付きトークンに取得します。トークンの値を一度定義すると、Lightning コンポーネントの CSS リソース全体で再利用できます。トークンを使用することで、設計の一貫性を確保しやすくなり、設計の変化に伴った更新がさらに簡単になります。

設計トークンはコンポーネントやアプリケーションの設計を作成するためのビジュアルデザイン「原子」です。具体的には、設計トークンは、余白やスペース設定のピクセル値、フォントサイズやフォントファミリ、色の 16 進数値などのビジュアルデザイン属性を保存する名前付きエンティティです。トークンは低レベルの

値を一元管理する優れた方法です。トークンを使用して、コンポーネントやアプリケーションの設計を構成するスタイルを作成できます。

このセクションの内容:

トークンバンドル
トークンは、コンポーネント、イベント、インターフェースと同様にバンドルの種別です。

トークンバンドルの作成
開発者コンソールを使用して組織のトークンバンドルを作成します。

トークンの定義と使用
トークンは、`<aura:token>` コンポーネントを使用して指定する名前-値のペアです。トークンをトークンバンドル内で定義してから、コンポーネントの CSS スタイルリソースでトークンを使用します。

トークンでの式の使用
トークンでは、限られた式のセットがサポートされています。式を使用すると、1 つのトークン値を別のトークンで再利用したり、トークンを組み合わせせることによってより複雑なスタイルのプロパティを形成したりできます。

トークンバンドルの拡張
`extends` 属性を使用して 1 つのトークンバンドルを別のトークンバンドルから拡張します。

標準設計トークンの使用
Salesforce では、コンポーネントスタイルリソースでアクセスできる一連の「基本」トークンが公開されます。これらの標準トークンを使用すると、独自のカスタムコンポーネントで Salesforce Lightning Design System (SLDS) の外観を模倣することができます。SLDS の変化に伴って、標準設計トークンを使用してスタイル設定されたコンポーネントも変化します。

トークンバンドル

トークンは、コンポーネント、イベント、インターフェースと同様にバンドルの種別です。

トークンバンドルに含まれるリソースは 1 つのみで、それはトークンコレクション定義です。

リソース	リソース名	使用方法
トークンコレクション	`defaultTokens.tokens`	トークンバンドル内の唯一の必須リソース。1 つ以上のトークンのマークアップが含まれます。各トークンバンドルに含まれるトークンリソースは 1 つのみです。

メモ: トークンバンドルの名前や説明を作成後に開発者コンソールで編集することはできません。バンドルの `AuraBundleDefinition` は、メタデータ API を使用して変更できます。

トークンコレクションは `<aura:tokens>` タグで始まり、トークンを定義する `<aura:token>` タグのみを含めることができます。

トークンコレクションは、式をサポートしていますが制限があります。「トークンでの式の使用」を参照してください。トークンコレクションでは、その他のマークアップ、レンダラ、コントローラ、およびその他のすべてのものを使用できません。

関連トピック:

トークンでの式の使用

トークンバンドルの作成

開発者コンソールを使用して組織のトークンバンドルを作成します。

トークンバンドルを作成する手順は、次のとおりです。

1. 開発者コンソールで、**[File (ファイル)]** > **[New (新規)]** > **[Lightning Tokens]** を選択します。
2. トークンバンドルの名前を入力します。

   最初のトークンバンドルには「*defaultTokens*」という名前を付ける必要があります。defaultTokens 内で定義されたトークンは Lightning コンポーネントで自動的にアクセス可能になります。その他のバンドルで定義されたトークンは、defaultTokens バンドルにインポートしないかぎり、コンポーネントでアクセス可能になりません。

空のトークンバンドルが作成されたので、編集準備が整いました。

```
<aura:tokens>

</aura:tokens>
```

メモ: トークンバンドルの名前や説明を作成後に開発者コンソールで編集することはできません。バンドルの AuraBundleDefinition は、メタデータ API を使用して変更できます。トークンバンドルにはバージョンを設定できますが、無効となります。

トークンの定義と使用

トークンは、<aura:token> コンポーネントを使用して指定する名前-値のペアです。トークンをトークンバンドル内で定義してから、コンポーネントの CSS スタイルリソースでトークンを使用します。

トークンの定義

新しいトークンをバンドルの <aura:tokens> コンポーネントの子コンポーネントとして追加します。たとえば、次のように使用します。

```
<aura:tokens>
    <aura:token name="myBodyTextFontFace"
                value="'Salesforce Sans', Helvetica, Arial, sans-serif"/>
    <aura:token name="myBodyTextFontWeight" value="normal"/>
    <aura:token name="myBackgroundColor" value="#f4f6f9"/>
    <aura:token name="myDefaultMargin" value="6px"/>
</aura:tokens>
```

<aura:token> タグで使用できる属性は name と value のみです。

トークンの使用

defaultTokens バンドルで作成されたトークンは、名前空間内のコンポーネントで自動的に使用可能になります。設計トークンを使用するには、token() 関数およびコンポーネントバンドルのCSS リソースのトークン名を使用して参照します。以下に例を示します。

```
.THIS p {
    font-family: token(myBodyTextFontFace);
    font-weight: token(myBodyTextFontWeight);
}
```

トークンを参照するために、より簡潔な関数名を使用するには、token() 関数の代わりに t() 関数を使用してください。この2つは同じものです。トークン名が命名規則に従っている場合や十分に内容が伝わるものであれば、簡潔な関数名を使用しても CSS スタイルの明確さが損なわれません。

トークンでの式の使用

トークンでは、限られた式のセットがサポートされています。式を使用すると、1つのトークン値を別のトークンで再利用したり、トークンを組み合わせることによってより複雑なスタイルのプロパティを形成したりできます。

相互参照トークン

1つのトークンの値を別のトークンの定義内で参照するには、トークンをラップして標準的な式の構文で参照できるようにします。

次の例では、Salesforceによって提供されたトークンをカスタムトークン内で参照しています。標準トークンは直接表示されていませんが、次のようなものであると想像します。

```
<!-- force:base tokens (SLDS standard tokens) -->
<aura:tokens>
  ...
  <aura:token name="colorBackground" value="rgb(244, 246, 249)" />
  <aura:token name="fontFamily" value="'Salesforce Sans', Arial, sans-serif" />
  ...
</aura:tokens>
```

上記のことを念頭に置いて、カスタムトークン内で次のように標準トークンを参照できます。

```
<!-- defaultTokens.tokens (your tokens) -->
<aura:tokens extends="force:base">
  <aura:token name="mainColor" value="{! colorBackground }" />
  <aura:token name="btnColor" value="{! mainColor }" />
  <aura:token name="myFont" value="{! fontFamily }" />
</aura:tokens>
```

相互参照できるのは、同じファイルまたは親で定義されているトークンのみです。

トークンリソースでの式の構文は、他のトークンへの参照のみに限定されています。

トークンの組み合わせ

個々のトークン値を組み合わせた、より複雑なCSSスタイルプロパティの形成をサポートするために、`token()` 関数は文字列の連結をサポートしています。たとえば、次のトークンが定義されているとします。

```
<!-- defaultTokens.tokens (your tokens) -->
<aura:tokens>
  <aura:token name="defaultHorizonalSpacing" value="12px" />
  <aura:token name="defaultVerticalSpacing" value="6px" />
</aura:tokens>
```

これらの 2 つのトークンを CSS スタイル定義内で組み合わせることができます。以下に例を示します。

```
/* myComponent.css */
.THIS div.notification {
  margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing);
  /* more styles here */
}
```

適切なスタイル定義を作成するために、何度でもトークンと文字列を組み合わせることができます。たとえば、`margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing + ' 3px');` を使用すれば、前の定義で下部の間隔をハードコードできます。

`token()` 関数内でサポートされている演算子は文字列連結のための「+」のみです。

関連トピック:

[トークンの定義と使用]

トークンバンドルの拡張

`extends` 属性を使用して 1 つのトークンバンドルを別のトークンバンドルから拡張します。

1つのバンドルから別のバンドルへトークンを追加するには、次のように、「親」トークンバンドルから「子」トークンバンドルに拡張します。

```
<aura:tokens extends="yourNamespace:parentTokens">
    <!-- additional tokens here -->
</aura:tokens>
```

トークン値を上書きすると大半は期待どおりに動作しますが、子トークンバンドル内のトークンでは親バンドルの同じ名前でトークンが上書きされます。標準トークンを使用している場合は例外です。Lightning Experience や Salesforce1 では標準トークンを上書きできません。

重要: 標準トークン値の上書きは定義されていない動作で、サポートされていません。標準トークンと同じ名前のトークンを作成すると、コンテキストによって標準トークンの値を上書きする場合と影響を及ぼさない場合があります。今後のリリースではこの動作が変更されるため、使用しないでください。

関連トピック:

[標準設計トークンの使用]

標準設計トークンの使用

Salesforceでは、コンポーネントスタイルリソースでアクセスできる一連の「基本」トークンが公開されます。これらの標準トークンを使用すると、独自のカスタムコンポーネントでSalesforce Lightning Design System (SLDS) の外観を模倣することができます。SLDSの変化に伴って、標準設計トークンを使用してスタイル設定されたコンポーネントも変化します。

組織に標準トークンを追加するには、次のようにトークンバンドルを基本トークンから拡張します。

```
<aura:tokens extends="force:base">
    <!-- your own tokens here -->
</aura:tokens>
```

トークンが defaultTokens (または defaultTokens が拡張する別のトークンバンドル) に追加されると、token() 関数およびトークン名を使用して、独自のトークンのように force:base からトークンを参照できます。以下に例を示します。

```
.THIS p {
    font-family: token(fontFamily);
    font-weight: token(fontWeightRegular);
}
```

自分のトークンを標準トークンと組み合わせることができます。独自のトークンを標準トークンと簡単に区別できるような命名方法を使用することをお勧めします。トークン名に「my」などの簡単に識別できるプレフィックスを使用することを検討してください。

このセクションの内容:

標準トークンの上書き (開発者プレビュー)

標準トークンは、カスタムコンポーネントでLightning Design Systemのデザインを指定します。標準トークンを上書きし、ブランド設定をカスタマイズして Lightning アプリケーションに適用できます。

標準設計トークン - force:base

使用可能な標準トークンは、Salesforce Lightning Design System (SLDS) で提供される設計トークンのサブセットです。次のトークンは、force:base から拡張した場合に使用できます。

コミュニティの標準設計トークン

標準設計トークンのサブセットを使用して、コミュニティビルダーで [ブランド] パネルと互換性のあるコンポーネントを作成します。[ブランド] パネルでは、システム管理者がブランドプロパティを使用してコミュニティ全体のスタイル設定をすばやく行うことができます。[ブランド] パネルの各プロパティは、1つ以上の標準設計トークンと対応付けられます。システム管理者が [ブランド] パネルのプロパティを更新すると、そのブランドプロパティに関連付けられたトークンを使用する Lightning コンポーネントが自動的に更新されます。

関連トピック:

トークンバンドルの拡張

標準トークンの上書き (開発者プレビュー)

標準トークンは、カスタムコンポーネントで Lightning Design System のデザインを指定します。標準トークンを上書きし、ブランド設定をカスタマイズして Lightning アプリケーションに適用できます。

メモ: 標準トークンの上書きは、開発者プレビューとして利用できます。この機能は、Salesforce がドキュメント、プレスリリース、または公式声明で正式リリースを発表しない限り、正式リリースされません。この機能についてのフィードバックと提案は、IdeaExchange からお寄せください。

Lightning アプリケーションの標準トークンを上書きするには、一意の名前 (`myOverrides` など) でトークンバンドルを作成します。トークンリソースで、標準トークンの値を再定義します。

```
<aura:tokens>
    <aura:token name="colorTextBrand" value="#8d7d74"/>
</aura:tokens>
```

Lightning アプリケーションで、`tokens` 属性のトークンバンドルを指定します。

```
<aura:application tokens="c:myOverrides">
    <!-- Your app markup here -->
</aura:application>
```

トークンの上書きは、Salesforceが提供するリソースとコンポーネント、およびトークンを使用する独自のコンポーネントを含む、アプリケーション全体に適用されます。

トークン属性を使用するアプリケーションのパッケージ化はサポートされていません。

重要: トークンバンドルの必須リソース、`defaultTokens.tokens` 内の標準トークン値の上書きはサポートされていません。標準トークンと同じ名前のトークンを作成すると、コンテキストによっては標準トークンの値を上書きする場合と影響を及ぼさない場合があります。上書きは、上記のように個別のリソースでのみ行ってください。

関連トピック:

標準設計トークン - force:base

標準設計トークン - `force:base`

使用可能な標準トークンは、Salesforce Lightning Design System (SLDS) で提供される設計トークンのサブセットです。次のトークンは、`force:base` から拡張した場合に使用できます。

使用可能なトークン

重要: 標準トークン値は SLDS に合わせて変化します。使用可能なトークンとその値は、通知なしで変更される可能性があります。ここに示したトークン値は単なる例です。

トークン名	値の例
`borderWidthThin`	1px
`borderWidthThick`	2px
`spacingXxxSmall`	0.125rem

トークン名	値の例
spacingXxSmall	0.25rem
spacingXSmall	0.5rem
spacingSmall	0.75rem
spacingMedium	1rem
spacingLarge	1.5rem
spacingXLarge	2rem
sizeXxSmall	6rem
sizeXSmall	12rem
sizeSmall	15rem
sizeMedium	20rem
sizeLarge	25rem
sizeXLarge	40rem
sizeXxLarge	60rem
squareIconUtilitySmall	1rem
squareIconUtilityMedium	1.25rem
squareIconUtilityLarge	1.5rem
squareIconLargeBoundary	3rem
squareIconLargeBoundaryAlt	5rem
squareIconLargeContent	2rem
squareIconMediumBoundary	2rem
squareIconMediumBoundaryAlt	2.25rem
squareIconMediumContent	1rem
squareIconSmallBoundary	1.5rem
squareIconSmallContent	.75rem
squareIconXSmallBoundary	1.25rem
squareIconXSmallContent	.5rem
fontWeightLight	300
fontWeightRegular	400
fontWeightBold	700
lineHeightHeading	1.25

トークン名	値の例
lineHeightText	1.375
lineHeightReset	1
lineHeightTab	2.5rem
fontFamily	'Salesforce Sans', Arial, sans-serif
borderRadiusSmall	.125rem
borderRadiusMedium	.25rem
borderRadiusLarge	.5rem
borderRadiusPill	15rem
borderRadiusCircle	50%
colorBorder	rgb(216, 221, 230)
colorBorderBrand	rgb(21, 137, 238)
colorBorderError	rgb(194, 57, 52)
colorBorderSuccess	rgb(75, 202, 129)
colorBorderWarning	rgb(255, 183, 93)
colorBorderTabSelected	rgb(0, 112, 210)
colorBorderSeparator	rgb(244, 246, 249)
colorBorderSeparatorAlt	rgb(216, 221, 230)
colorBorderSeparatorInverse	rgb(42, 66, 108)
colorBorderRowSelected	rgb(0, 112, 210)
colorBorderRowSelectedHover	rgb(21, 137, 238)
colorBorderButtonBrand	rgb(0, 112, 210)
colorBorderButtonBrandDisabled	rgba(0, 0, 0, 0)
colorBorderButtonDefault	rgb(216, 221, 230)
colorBorderButtonInverseDisabled	rgba(255, 255, 255, 0.15)
colorBorderInput	rgb(216, 221, 230)
colorBorderInputActive	rgb(21, 137, 238)
colorBorderInputDisabled	rgb(168, 183, 199)
colorBorderInputCheckboxSelectedCheckmark	rgb(255, 255, 255)
colorBackground	rgb(244, 246, 249)
colorBackgroundAlt	rgb(255, 255, 255)

トークン名	値の例
colorBackgroundAltInverse	rgb(22, 50, 92)
colorBackgroundRowHover	rgb(244, 246, 249)
colorBackgroundRowActive	rgb(238, 241, 246)
colorBackgroundRowSelected	rgb(240, 248, 252)
colorBackgroundRowNew	rgb(217, 255, 223)
colorBackgroundInverse	rgb(6, 28, 63)
colorBackgroundBrowser	rgb(84, 105, 141)
colorBackgroundChromeMobile	rgb(0, 112, 210)
colorBackgroundChromeDesktop	rgb(255, 255, 255)
colorBackgroundHighlight	rgb(250, 255, 189)
colorBackgroundModal	rgb(255, 255, 255)
colorBackgroundModalBrand	rgb(0, 112, 210)
colorBackgroundNotificationBadge	rgb(194, 57, 52)
colorBackgroundNotificationBadgeHover	rgb(0, 95, 178)
colorBackgroundNotificationBadgeFocus	rgb(0, 95, 178)
colorBackgroundNotificationBadgeActive	rgb(0, 57, 107)
colorBackgroundNotificationNew	rgb(240, 248, 252)
colorBackgroundPayload	rgb(244, 246, 249)
colorBackgroundShade	rgb(224, 229, 238)
colorBackgroundStencil	rgb(238, 241, 246)
colorBackgroundStencilAlt	rgb(224, 229, 238)
colorBackgroundScrollbar	rgb(224, 229, 238)
colorBackgroundScrollbarTrack	rgb(168, 183, 199)
colorBrand	rgb(21, 137, 238)
colorBrandDark	rgb(0, 112, 210)
colorBackgroundModalButton	rgba(0, 0, 0, 0.07)
colorBackgroundModalButtonActive	rgba(0, 0, 0, 0.16)
colorBackgroundInput	rgb(255, 255, 255)
colorBackgroundInputActive	rgb(255, 255, 255)
colorBackgroundInputCheckbox	rgb(255, 255, 255)

トークン名	値の例
colorBackgroundInputCheckboxDisabled	rgb(216, 221, 230)
colorBackgroundInputCheckboxSelected	rgb(21, 137, 238)
colorBackgroundInputDisabled	rgb(224, 229, 238)
colorBackgroundInputError	rgb(255, 221, 225)
colorBackgroundPill	rgb(255, 255, 255)
colorBackgroundToast	rgba(84, 105, 141, 0.95)
colorBackgroundToastSuccess	rgb(4, 132, 75)
colorBackgroundToastError	rgba(194, 57, 52, 0.95)
shadowDrag	0 2px 4px 0 rgba(0, 0, 0, 0.40)
shadowDropDown	0 2px 3px 0 rgba(0, 0, 0, 0.16)
shadowHeader	0 2px 4px rgba(0,0,0,0.07)
shadowButtonFocus	0 0 3px #0070D2
shadowButtonFocusInverse	0 0 3px #E0E5EE
colorTextActionLabel	rgb(84, 105, 141)
colorTextActionLabelActive	rgb(22, 50, 92)
colorTextBrand	rgb(21, 137, 238)
colorTextBrowser	rgb(255, 255, 255)
colorTextBrowserActive	rgba(0, 0, 0, 0.4)
colorTextDefault	rgb(22, 50, 92)
colorTextError	rgb(194, 57, 52)
colorTextInputDisabled	rgb(84, 105, 141)
colorTextInputFocusInverse	rgb(22, 50, 92)
colorTextInputIcon	rgb(159, 170, 181)
colorTextInverse	rgb(255, 255, 255)
colorTextInverseWeak	rgb(159, 170, 181)
colorTextInverseActive	rgb(94, 180, 255)
colorTextInverseHover	rgb(159, 170, 181)
colorTextLink	rgb(0, 112, 210)
colorTextLinkActive	rgb(0, 57, 107)
colorTextLinkDisabled	rgb(22, 50, 92)

トークン名	値の例
colorTextLinkFocus	rgb(0, 95, 178)
colorTextLinkHover	rgb(0, 95, 178)
colorTextLinkInverse	rgb(255, 255, 255)
colorTextLinkInverseHover	rgba(255, 255, 255, 0.75)
colorTextLinkInverseActive	rgba(255, 255, 255, 0.5)
colorTextLinkInverseDisabled	rgba(255, 255, 255, 0.15)
colorTextModal	rgb(255, 255, 255)
colorTextModalButton	rgb(84, 105, 141)
colorTextStageLeft	rgb(224, 229, 238)
colorTextTabLabel	rgb(22, 50, 92)
colorTextTabLabelSelected	rgb(0, 112, 210)
colorTextTabLabelHover	rgb(0, 95, 178)
colorTextTabLabelFocus	rgb(0, 95, 178)
colorTextTabLabelActive	rgb(0, 57, 107)
colorTextTabLabelDisabled	rgb(224, 229, 238)
colorTextToast	rgb(224, 229, 238)
colorTextWeak	rgb(84, 105, 141)
colorTextIconBrand	rgb(0, 112, 210)
colorTextButtonBrand	rgb(255, 255, 255)
colorTextButtonBrandHover	rgb(255, 255, 255)
colorTextButtonBrandActive	rgb(255, 255, 255)
colorTextButtonBrandDisabled	rgb(255, 255, 255)
colorTextButtonDefault	rgb(0, 112, 210)
colorTextButtonDefaultHover	rgb(0, 112, 210)
colorTextButtonDefaultActive	rgb(0, 112, 210)
colorTextButtonDefaultDisabled	rgb(216, 221, 230)
colorTextButtonDefaultHint	rgb(159, 170, 181)
colorTextButtonInverse	rgb(224, 229, 238)
colorTextButtonInverseDisabled	rgba(255, 255, 255, 0.15)
colorTextIconDefault	rgb(84, 105, 141)

トークン名	値の例
colorTextIconDefaultHint	rgb(159, 170, 181)
colorTextIconDefaultHover	rgb(0, 112, 210)
colorTextIconDefaultActive	rgb(0, 57, 107)
colorTextIconDefaultDisabled	rgb(216, 221, 230)
colorTextIconInverse	rgb(255, 255, 255)
colorTextIconInverseHover	rgb(255, 255, 255)
colorTextIconInverseActive	rgb(255, 255, 255)
colorTextIconInverseDisabled	rgba(255, 255, 255, 0.15)
colorTextLabel	rgb(84, 105, 141)
colorTextPlaceholder	rgb(84, 105, 141)
colorTextPlaceholderInverse	rgb(224, 229, 238)
colorTextRequired	rgb(194, 57, 52)
colorTextPill	rgb(0, 112, 210)
durationInstantly	0s
durationImmediately	0.05s
durationQuickly	0.1s
durationPromptly	0.2s
durationSlowly	0.4s
durationPaused	3.2s
colorBackgroundButtonBrand	rgb(0, 112, 210)
colorBackgroundButtonBrandActive	rgb(0, 57, 107)
colorBackgroundButtonBrandHover	rgb(0, 95, 178)
colorBackgroundButtonBrandDisabled	rgb(224, 229, 238)
colorBackgroundButtonDefault	rgb(255, 255, 255)
colorBackgroundButtonDefaultHover	rgb(244, 246, 249)
colorBackgroundButtonDefaultFocus	rgb(244, 246, 249)
colorBackgroundButtonDefaultActive	rgb(238, 241, 246)
colorBackgroundButtonDefaultDisabled	rgb(255, 255, 255)
colorBackgroundButtonIcon	rgba(0, 0, 0, 0)
colorBackgroundButtonIconHover	rgb(244, 246, 249)

トークン名	値の例
colorBackgroundButtonIconFocus	rgb(244, 246, 249)
colorBackgroundButtonIconActive	rgb(238, 241, 246)
colorBackgroundButtonIconDisabled	rgb(255, 255, 255)
colorBackgroundButtonInverse	rgba(0, 0, 0, 0)
colorBackgroundButtonInverseActive	rgba(0, 0, 0, 0.24)
colorBackgroundButtonInverseDisabled	rgba(0, 0, 0, 0)
lineHeightButton	1.875rem
lineHeightButtonSmall	1.75rem
colorBackgroundAnchor	rgb(244, 246, 249)

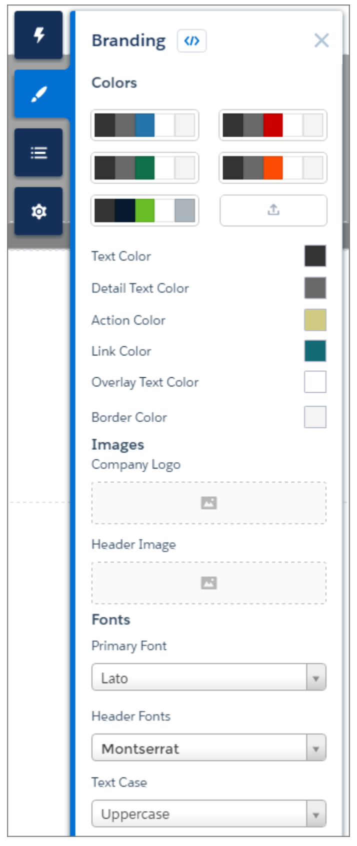
SLDS で使用可能な設計トークンの完全なリストについては、Lightning Design System サイトの「Design Tokens」を参照してください。

関連トピック:

トークンバンドルの拡張

コミュニティの標準設計トークン

標準設計トークンのサブセットを使用して、コミュニティビルダーで [ブランド] パネルと互換性のあるコンポーネントを作成します。[ブランド]パネルでは、システム管理者がブランドプロパティを使用してコミュニティ全体のスタイル設定をすばやく行うことができます。[ブランド] パネルの各プロパティは、1 つ以上の標準設計トークンと対応付けられます。システム管理者が[ブランド]パネルのプロパティを更新すると、そのブランドプロパティに関連付けられたトークンを使用する Lightning コンポーネントが自動的に更新されます。



コミュニティに使用できるトークン

カスタマーサービス (Napili) テンプレートを使用するコミュニティには、force:baseから拡張した場合に次の標準トークンを使用できます。

重要: 標準トークン値は SLDS に合わせて変化します。使用可能なトークンとその値は、通知なしで変更される可能性があります。

[ブランド] パネルのプロパティ	対応付けられる標準設計トークン
テキストの色	colorTextDefault
詳細テキストの色	• colorTextLabel • colorTextPlaceholder • colorTextWeak
アクションの色	• colorBackgroundButtonBrand • colorBackgroundHighlight • colorBorderBrand • colorBorderButtonBrand • colorBrand

[ブランド] パネルのプロパティ	対応付けられる標準設計トークン
	• colorTextBrand
リンクの色	colorTextLink
フロート表示テキストの色	• colorTextButtonBrand • colorTextButtonBrandHover • colorTextInverse
境界線の色	• colorBorder • colorBorderButtonDefault • colorBorderInput • colorBorderSeparatorAlt
主要フォント	fontFamily
テキストの大文字/小文字	textTransform

また、次の標準トークンはカスタマーサービス (Napili) テンプレートの派生ブランドプロパティに使用できます。[ブランド]パネルでプロパティを更新すると、派生ブランドプロパティに間接的にアクセスできます。たとえば、[ブランド] パネルで [アクションの色] プロパティを変更した場合、新しい値に基づいて [アクションの濃い色] 値が自動的に再適用されます。

派生ブランドプロパティ	対応付けられる標準設計トークン
アクションの濃い色 ([アクションの色] から派生)	• colorBackgroundButtonBrandActive • colorBackgroundButtonBrandHover
フロート表示の色 ([アクションの色] から派生)	• colorBackgroundButtonDefaultHover • colorBackgroundRowHover • colorBackgroundRowSelected • colorBackgroundShade
リンクの濃い色 ([リンクの色] から派生)	• colorTextLinkActive • colorTextLinkHover

SLDS で使用可能な設計トークンの完全なリストについては、Lightning Design System サイトの「Design Tokens」を参照してください。

関連トピック:

コミュニティのコンポーネントの設定

JavaScript の使用

クライアント側のコードには JavaScript を使用します。`$A` 名前空間は、JavaScript コードのフレームワークを使用するためのエントリポイントです。

`$A` で使用できるすべてのメソッドについては、`https://<myDomain>.lightning.force.com/auradocs/reference.app` の JavaScript API (`<myDomain>` はカスタム Salesforce ドメインの名前) を参照してください。

コンポーネントのバンドルには、クライアント側のコントローラ、ヘルパー、またはレンダラの JavaScript コードを含めることができます。これらの JavaScript リソースで最も使用されるのは、クライアント側のコントローラです。

JavaScript コードの式

JavaScript では、文字列構文を使用して式を評価します。たとえば、次の式ではコンポーネントの `label` 属性を取得します。

```
var theLabel = cmp.get("v.label");
```

メモ: `.app` または `.cmp` リソースのマークアップでは、`{! }` の式の構文のみを使用します。

このセクションの内容:

コンポーネントの初期化時のアクションの呼び出し

`init` イベントを使用して、コンポーネントを構築してから表示するまでの間にコンポーネントを更新したり、イベントを起動したりできます。

コンポーネントのバンドル内の JavaScript コードの共有

再利用する関数をコンポーネントのヘルパーに配置します。ヘルパー関数により、データの処理やサーバ側のアクションの起動などのタスクを特化することもできます。

コンポーネント間の JavaScript コードの共有

完全に独立したシンプルな Lightning コンポーネントを作成できます。ただし、より複雑なアプリケーションを作成する場合は、コンポーネント間でコードやクライアント側データの共有が必要になることがあります。

外部 JavaScript ライブラリの使用

静的リソースとしてアップロードした JavaScript ライブラリを参照するには、`.cmp` または `.app` マークアップで `<ltng:require>` タグを使用します。

JavaScript での属性値の操作

JavaScript で属性値を操作するときに役に立つ、よく使用されるパターンを次に示します。

JavaScript でのコンポーネントのボディの操作

JavaScript でコンポーネントのボディを操作するときに役に立つ、よく使用されるパターンを次に示します。

JavaScript でのイベントの操作

JavaScript でイベントを操作するときに役に立つ、よく使用されるパターンを次に示します。

DOM の変更

ドキュメントオブジェクトモデル (DOM) は、HTML および XML ドキュメントのオブジェクトを表したり、操作したりする、言語に依存しないモデルです。重要なことは、DOM を安全に変更する方法を理解し、変更がフレームワークの表示サービスによって踏みつけられて予想外の結果にならないようにすることです。

コンポーネントの有効性の確認

非同期コードの実行中に UI で他の場所に移動すると、フレームワークは、非同期要求を実行したコンポーネントを非表示にして破棄します。そのコンポーネントを引き続き参照できますが、コンポーネントは無効になっています。`cmp.isValid()` コールは、無効なコンポーネントでは `false` を返します。

フレームワークのライフサイクル外のコンポーネントの変更

`$A.getCallback()` を使用して、`setTimeout()` コールの場合のように通常の表示ライフサイクル外のコンポーネントを変更するコードをラップします。`$A.getCallback()` コールは、フレームワークが変更されたコンポーネントを確実に表示し、すべてのエンキューされたアクションが処理されるようにします。

項目の検証

ユーザ入力を検証してエラーを処理し、入力項目にエラーメッセージを表示します。

エラーの発生および処理

このフレームワークでは、復旧できないアプリケーションエラーおよび復旧できるアプリケーションエラーを JavaScript コードで柔軟に対処できます。たとえば、サーバ側の応答のエラーを処理するときに、これらのエラーをコールバック内に発生させることができます。

コンポーネントメソッドのコール

`<aura:method>` を使用して、コンポーネントの API の一部としてメソッドを定義します。これにより、コンポーネントイベントを起動して処理する代わりに、コンポーネントのクライアント側コントローラからメソッドを直接コールできるようになります。`<aura:method>` を使用すると、親コンポーネントに含まれる子コンポーネントのメソッドをコールする場合に、親コンポーネントに必要なコードが簡略化されます。

JavaScript Promise の使用

JavaScript コードで ES6 Promise を使用できます。プロミスにより、非同期コールの成否を処理するコードや、複数の非同期コールをまとめてチェーニングするコードを簡素化できます。

コンポーネントからの API コールの実行

デフォルトでは、クライアント側のコードからサードパーティの API にコールを実行することはできません。リモートサイトを CSP 信頼済みサイトとして追加して、クライアント側のコードがアセットを読み込み、そのサイトのドメインに API 要求を実行できるようにします。

サードパーティ API にアクセスするための CSP 信頼済みサイトの作成

Lightning コンポーネントフレームワークでは、W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御します。外部 (Salesforce 以外の) サーバに対する要求を実行するサードパーティ API を使用するには、サーバを CSP 信頼済みサイトとして追加します。

関連トピック:

クライアント側コントローラを使用したイベントの処理

コンポーネントの初期化時のアクションの呼び出し

`init` イベントを使用して、コンポーネントを構築してから表示するまでの間にコンポーネントを更新したり、イベントを起動したりできます。

コンポーネントのソース

```
<aura:component>
    <aura:attribute name="setMeOnInit" type="String" default="default value" />
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <p>This value is set in the controller after the component initializes and before
rendering.</p>
    <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

クライアント側コントローラのソース

```
({
    doInit: function(cmp) {
        // Set the attribute value.
        // You could also fire an event here instead.
        cmp.set("v.setMeOnInit", "controller init magic!");
    }
})
```

コンポーネントのソースを見て、どのように機能するのかを確認しましょう。重要なのは次の行です。

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

これで、コンポーネントの `init` イベントハンドラが登録されます。`init` は、すべてのコンポーネントに送信される定義済みイベントです。コンポーネントが初期化されたら、コンポーネントのコントローラで `doInit` アクションがコールされます。このサンプルでは、コントローラアクションで属性値を設定していますが、イベントの起動などの処理を実行することもできます。

`value="{!this}"` を設定すると、これ自体が値のイベントとしてマークされます。`init` イベントでは、常にこの設定を使用する必要があります。

関連トピック:

クライアント側コントローラを使用したイベントの処理
カスタムレンダラの作成
コンポーネントの属性
変更ハンドラを使用したデータ変更の検出

コンポーネントのバンドル内の JavaScript コードの共有

再利用する関数をコンポーネントのヘルパーに配置します。ヘルパー関数により、データの処理やサーバ側のアクションの起動などのタスクを特化することもできます。

ヘルパー関数は、コンポーネントのバンドルの任意の JavaScript コード (クライアント側のコントローラまたはレンダラなど) からコールできます。

ヘルパー関数の形状は、クライアント側のコントローラ関数と似ており、名前-値ペアの対応付けが含まれるオブジェクトリテラル表記法の JavaScript オブジェクトであることを示すために括弧と中括弧で囲まれます。ヘルパー関数は、関数で要求される任意の引数(属するコンポーネント、コールバック、またはその他のオブジェクトなど) を渡すことができます。

```
({
    helperMethod1 : function() {
        // logic here
    },

    helperMethod2 : function(component) {
        // logic here
        this.helperMethod3(var1, var2);
    },

     helperMethod3 : function(var1, var2) {
         // do something with var1 and var2 here
    }
})
```

ヘルパーの作成

ヘルパーリソースは、コンポーネントのバンドルの一部で、*<componentName>*`Helper.js` という命名規則で自動的に結び付けられます。

開発者コンソールを使用してヘルパーを作成するには、コンポーネントのサイドバーで **[HELPER (ヘルパー)]** をクリックします。このヘルパーファイルは、自動的に結び付けられるコンポーネントの範囲で有効です。

コントローラでのヘルパーの使用

`helper` 引数をコントローラ関数に追加して、コントローラ関数でヘルパーを使用できるようにします。コントローラで `(component, event, helper)` を指定します。これらは標準パラメータで、関数でアクセスする必要はありません。また、`createExpense: function(component, expense){...}` などのインスタンス変数をパラメータとして渡すこともできます。ここで、`expense` は、コンポーネントで定義された変数です。

次のコードに、カスタムイベントハンドラで使用できる `updateItem` ヘルパー関数をコントローラでコールする方法を示します。

```
/* controller */
({
    newItemEvent: function(component, event, helper) {
        helper.updateItem(component, event.getParam("item"));
    }
})
```

ヘルパー関数はコンポーネントに対してローカルであり、コードの再利用が促進され、クライアント側のコントローラの JavaScript ロジックの複雑な作業が軽減されます (可能な場合)。次のコードに、コントローラで設定

された value パラメータを item 引数を使用して取得するヘルパー関数を示します。このコードは、サーバ側のアクションをコールし、コールバックを返しますが、ヘルパー関数で他の処理を行うこともできます。

```
/* helper */
({
    updateItem : function(component, item, callback) {
        //Update the items via a server-side action
        var action = component.get("c.saveItem");
        action.setParams({"item" : item});
        //Set any optional callback and enqueue the action
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    }
})
```

レンダラでのヘルパーの使用

helper 引数をレンダラ関数に追加して、レンダラ関数でヘルパーを使用できるようにします。レンダラで、関数の署名のパラメータとして (component, helper) を指定し、関数からコンポーネントのヘルパーにアクセスできるようにします。これらは標準パラメータで、関数でアクセスする必要はありません。次のコード例に、レンダラの afterRender() 関数を上書きして、ヘルパーメソッドの open をコールする方法を示します。

detailsRenderer.js

```
({
    afterRender : function(component, helper){
        helper.open(component, null, "new");
    }
})
```

detailsHelper.js

```
({
    open : function(component, note, mode, sort){
        if(mode === "new") {
            //do something
        }
        // do something else, such as firing an event
    }
})
```

関連トピック:

カスタムレンダラの作成
コンポーネントのバンドル
クライアント側コントローラを使用したイベントの処理

コンポーネント間の JavaScript コードの共有

完全に独立したシンプルなLightningコンポーネントを作成できます。ただし、より複雑なアプリケーションを作成する場合は、コンポーネント間でコードやクライアント側データの共有が必要になることがあります。

`<ltng:require>` タグでは、外部 JavaScript ライブラリを静的リソースとしてアップロードした後で読み込むことができます。`<ltng:require>` を使用して、ユーティリティメソッドの独自の JavaScript ライブラリをインポートすることもできます。

カウンタの現在の値を返す `getValue()` メソッドと、カウンタの値を増分する `increment()` メソッドを提供する簡単なカウンタライブラリを見てみましょう。

JavaScript ライブラリの作成

1. 開発者コンソールで、**[File (ファイル)]** > **[New (新規)]** > **[Static Resource (静的リソース)]** をクリックします。
2. [Name (名前)] 項目に「*counter*」と入力します。
3. [MIME Type (MIME タイプ)] 項目で *[text/javascript]* を選択します。
4. **[登録]** をクリックします。
5. 次のコードを入力し、**[File (ファイル)]** > **[Save (保存)]** をクリックします。

```
window._counter = (function() {
    var value = 0; // private

    return { //public API
        increment: function() {
            value = value + 1;
            return value;
        },

        getValue: function() {
            return value;
        }
    };
}());
```

このコードでは、JavaScriptモジュールパターンを使用しています。このクロージャベースのパターンを使用すると、`value` 変数はライブラリに非公開のままになります。ライブラリを使用するコンポーネントは、`value` には直接アクセスできません。

コードで最も重要なのは、次の行です。

```
window._counter = (function() {
```

LockerServiceで明示的に有効になっているJavaScriptの厳格モードの要件として、`_counter` を `window` オブジェクトに関連付ける必要があります。`window._counter` はグローバル宣言のように見えますが、`_counter` が LockerService のセキュアな window オブジェクトに関連付けられているため、これはグローバル変数ではなく、名前空間変数です。

window._counter の代わりに _counter を使用すると、_counter は利用できません。アクセスしようとすると、次のようなエラーが表示されます。

```
Action failed: ... [_counter is not defined]
```

JavaScript ライブラリの使用

シンプルな UI がある MyCounter コンポーネントのライブラリを使用して、counter メソッドを演習してみましょう。

```
<!--c:MyCounter-->
<aura:component access="global">
    <ltng:require scripts="{!$Resource.counter}"
                  afterScriptsLoaded="{!c.getValue}"/>
    <aura:attribute name="value" type="Integer"/>

    <h1>MyCounter</h1>
    <p>{!v.value}</p>
    <lightning:button label="Get Value" onclick="{!c.getValue}"/>
    <lightning:button label="Increment" onclick="{!c.increment}"/>
</aura:component>
```

<ltng:require> タグがカウンタライブラリを読み込み、ライブラリが読み込まれた後、コンポーネントのクライアント側コントローラで getValue アクションを呼び出します。

クライアント側コントローラを次に示します。

```
/* MyCounterController.js */
({
    getValue : function(component, event, helper) {
        component.set("v.value", _counter.getValue());
    },

    increment : function(component, event, helper) {
        component.set("v.value", _counter.increment());
    }
})
```

window. プレフィックスを入力しなくても、window オブジェクトのプロパティにアクセスできます。したがって、_counter.getValue() を window._counter.getValue() の短縮として使用できます。

ボタンをクリックして、値を取得または増分します。

カウンタ値はカウンタライブラリを使用するコンポーネント間で共有されます。各コンポーネントに個別のカウンタが必要な場合は、カウンタの実装を変更できます。コンポーネントごとのコードおよび詳細は、ブログ投稿「*Modularizing Code in Lightning Components*」を参照してください。

関連トピック:

外部 JavaScript ライブラリの使用

ltng:require

JavaScript ES5 の厳格モードの適用

外部 JavaScript ライブラリの使用

静的リソースとしてアップロードした JavaScript ライブラリを参照するには、.cmp または .app マークアップで <ltng:require> タグを使用します。

このフレームワークのコンテンツセキュリティポリシーでは、外部 JavaScript ライブラリを Salesforce 静的リソースにアップロードすることが義務付けられています。静的リソースについての詳細は、Salesforce オンラインヘルプの「静的リソース」を参照してください。

次に <ltng:require> の使用例を示します。

```
<ltng:require scripts="{!$Resource.resourceName}"
    afterScriptsLoaded="{!c.afterScriptsLoaded}" />
```

resourceName は、静的リソースの [名前] です。管理パッケージでは、リソース名にパッケージ名前空間プレフィックス ($Resource.yourNamespace__resourceName など) を含める必要があります。個々のグラフィックやスクリプトなど、スタンドアロンの静的リソースの場合、必要なのはそれだけです。アーカイブ静的リソース内の項目を参照するには、項目へのパスの残りを文字列の連結を使用して含めます。

スクリプトが読み込まれると、クライアント側コントローラの afterScriptsLoaded アクションがコールされます。<ltng:require> によって読み込まれるスクリプトにアクセスするために、init イベントを使用しないでください。これらのスクリプトは非同期に読み込まれるため、init イベントハンドラがコールされるときに使用できない可能性が高くなります。

スクリプトの読み込みに関する考慮事項は次のとおりです。

スクリプトのセットの読み込み

リソースのセットを読み込むときは、scripts 属性にリソースのカンマ区切りリストを指定します。

メモ: 式で $Resource が解析される方法に予測できない動作があるため、複数の $Resource 参照を 1 つの属性に含めるには join 演算子を使用します。たとえば、コンポーネントに含める JavaScript ライブラリが複数ある場合、scripts 属性は次のようになります。

```
scripts="{!join(',',
    $Resource.jsLibraries + '/jsLibOne.js',
    $Resource.jsLibraries + '/jsLibTwo.js')}"
```

読み込み順序

スクリプトはリストに記載された順に読み込まれます。

1 回のみの読み込み

同じコンポーネントまたは異なるコンポーネントの複数の <ltng:require> タグでスタイルが指定されていても、スクリプトが読み込まれるのは 1 回のみです。

並列読み込み

相互に連動していないスクリプトのセットが複数ある場合は、並列読み込み用の <ltng:require> タグを使用します。

カプセル化

カプセル化および再利用を確実に行うには、JavaScript ライブラリを使用する .cmp または .app リソースのそれぞれに <ltng:require> タグを追加します。

<ltng:require> には、CSS リソースのリストを読み込む styles 属性もあります。scripts と styles 属性は 1 つの <ltng:require> タグで設定できます。

表示後の HTML 要素の操作に外部ライブラリを使用している場合、`afterScriptsLoaded` を使用してクライアント側コントローラに結び付けます。次の例では、静的リソースとしてアップロードされた `Chart.js` ライブラリを使用してグラフを設定しています。

```
<ltng:require scripts="{!$Resource.chart}"
              afterScriptsLoaded="{!c.setup}"/>
<canvas aura:id="chart" id="myChart" width="400" height="400"/>
```

コンポーネントのクライアント側コントローラは、コンポーネントの初期化と表示の後にグラフを設定します。

```
setup : function(component, event, helper) {
    var data = {
        labels: ["January", "February", "March"],
        datasets: [{
            data: [65, 59, 80, 81, 56, 55, 40]
        }]
    };
    var el = component.find("chart").getElement();
    var ctx = el.getContext("2d");
    var myNewChart = new Chart(ctx).Line(data);
}
```

関連トピック:

リファレンスドキュメントアプリケーション
コンテンツセキュリティポリシーの概要
外部 CSS の使用
$Resource

JavaScript での属性値の操作

JavaScript で属性値を操作するときに役に立つ、よく使用されるパターンを次に示します。

`component.get(String key)` および `component.set(String key, Object value)` は、コンポーネントの指定されたキーに関連付けられた値を取得して割り当てます。キーは、属性値を表す式として渡されます。コンポーネント参照の属性値を取得するには、`component.find("cmpId").get("v.value")` を使用します。同様に、コンポーネント参照の属性値を設定するには、`component.find("cmpId").set("v.value", myValue)` を使用します。次の例は、ID `button1` のボタンで表される、コンポーネント参照の属性値を取得して設定する方法を示します。

```
<aura:component>
    <aura:attribute name="buttonLabel" type="String"/>
    <lightning:button aura:id="button1" label="Button 1"/>
    {!v.buttonLabel}
    <lightning:button label="Get Label" onclick="{!c.getLabel}"/>
</aura:component>
```

次のコントローラアクションは、コンポーネントのボタンの label 属性値を取得し、その値を buttonLabel 属性に設定します。

```
({
    getLabel : function(component, event, helper) {
        var myLabel = component.find("button1").get("v.label");
        component.set("v.buttonLabel", myLabel);
    }
})
```

この例では、cmp は、JavaScript コードのコンポーネントへの参照です。

属性値を取得する

コンポーネントの label 属性の値を取得するには、次のように記述します。

```
var label = cmp.get("v.label");
```

属性値を設定する

コンポーネントの label 属性の値を設定するには、次のように記述します。

```
cmp.set("v.label","This is a label");
```

属性値が定義されているかどうかを検証する

コンポーネントの label 属性が定義されているかどうかを判断するには、次のように記述します。

```
var isDefined = !$A.util.isUndefined(cmp.get("v.label"));
```

属性値が空であるかどうかを検証する

コンポーネントの label 属性が空であるかどうかを判断するには、次のように記述します。

```
var isEmpty = $A.util.isEmpty(cmp.get("v.label"));
```

関連トピック:

JavaScript でのコンポーネントのボディの操作

JavaScript でのコンポーネントのボディの操作

JavaScript でコンポーネントのボディを操作するときに役に立つ、よく使用されるパターンを次に示します。

例に含まれる cmp は、JavaScript コードのコンポーネントへの参照です。通常、コンポーネントへの参照は JavaScript コードで簡単に取得できます。body 属性はコンポーネントの配列であるため、その属性に対して JavaScript Array メソッドを使用できます。

メモ: cmp.set("v.body", ...) を使用してコンポーネントのボディを設定するときは、コンポーネントマークアップに {!v.body} を明示的に含める必要があります。

コンポーネントのボディを置き換える

コンポーネントのボディの現在の値を別のコンポーネントで置き換えるには、次のように記述します。

```
// newCmp is a reference to another component
cmp.set("v.body", newCmp);
```

コンポーネントのボディをクリアする

コンポーネントのボディの現在の値をクリアする (空にする) には、次のように記述します。

```
cmp.set("v.body", []);
```

コンポーネントをコンポーネントのボディに追加する

newCmp コンポーネントをコンポーネントのボディに追加するには、次のように記述します。

```
var body = cmp.get("v.body");
// newCmp is a reference to another component
body.push(newCmp);
cmp.set("v.body", body);
```

コンポーネントをコンポーネントのボディの先頭に追加する

newCmp コンポーネントをコンポーネントのボディの先頭に追加するには、次のように記述します。

```
var body = cmp.get("v.body");
body.unshift(newCmp);
cmp.set("v.body", body);
```

コンポーネントをコンポーネントのボディから削除する

インデックス化されたエントリをコンポーネントのボディから削除するには、次のように記述します。

```
var body = cmp.get("v.body");
// Index (3) is zero-based so remove the fourth component in the body
body.splice(3, 1);
cmp.set("v.body", body);
```

関連トピック:

コンポーネントのボディ
JavaScript での属性値の操作

JavaScript でのイベントの操作

JavaScript でイベントを操作するときに役に立つ、よく使用されるパターンを次に示します。

イベントはコンポーネント間でデータをやり取りします。イベントは、イベントの起動前に値が設定された属性を含めて、それをイベントの処理時に読み取ることができます。

イベントの起動

コンポーネントに登録されたコンポーネントイベントまたはアプリケーションイベントを起動します。

```
//Fire a component event
var compEvent = cmp.getEvent("sampleComponentEvent");
compEvent.fire();

//Fire an application event
var appEvent = $A.get("e.c:appEvent");
appEvent.fire();
```

詳細は、以下を参照してください。

- コンポーネントイベントの起動
- アプリケーションイベントの起動

イベント名の取得

起動されたイベントの名前を取得するには:

```
event.getSource().getName();
```

イベントパラメータの取得

イベントに渡された属性を取得するには:

```
event.getParam("value");
```

イベントのパラメータの取得

イベントに渡されたすべての属性を取得するには:

```
event.getParams();
```

`event.getParams()` はすべてのイベントパラメータを含むオブジェクトを返します。

イベントの現在のフェーズの取得

イベントの現在のフェーズを取得するには:

```
event.getPhase();
```

イベントが起動されていない場合、`event.getPhase()` は `undefined` を返します。コンポーネントイベントとアプリケーションイベントの有効な戻り値は `capture`、`bubble`、`default` です。値イベントは `default` を返します。詳細は、以下を参照してください。

- コンポーネントイベント伝達
- アプリケーションイベントの伝達

ソースコンポーネントの取得

イベントを起動したコンポーネントを取得するには:

```
event.getSource();
```

イベントを起動したコンポーネントで属性を取得するには:

```
event.getSource().get("v.myName");
```

イベントの一時停止

起動したイベントを一時停止するには:

```
event.pause();
```

一時停止した場合、`event.resume()` がコールされるまでイベントは処理されません。イベントは `capture` または `bubble` フェーズでのみ一時停止できます。詳細は、以下を参照してください。

- バブルまたはキャプチャのコンポーネントイベントの処理
- アプリケーションのバブルイベントとキャプチャイベントの処理

デフォルトのイベント実行の防止

イベントでデフォルトアクションをキャンセルするには:

```
event.preventDefault();
```

たとえば、クリックしたときに `lightning:button` コンポーネントがフォームを送信するのを防ぐことができます。

一時停止されたイベントの再開

一時停止されたイベントのイベント処理を再開するには:

```
event.resume();
```

一時停止されたイベントは `capture` または `bubble` フェーズでのみ再開できます。詳細は、以下を参照してください。

- バブルまたはキャプチャのコンポーネントイベントの処理
- アプリケーションのバブルイベントとキャプチャイベントの処理

イベントパラメータの値の設定

イベントパラメータの値を設定するには:

```
event.setParam("name", cmp.get("v.myName");
```

イベントがすでに起動されている場合、パラメータ値を設定してもイベントに影響を与えません。

イベントパラメータの値の設定

イベントでパラメータの値を設定するには:

```
event.setParams({
    key : value
});
```

イベントがすでに起動されている場合、パラメータ値を設定してもイベントに影響を与えません。

イベント伝達を停止する

イベントのさらなる伝達を防止するには:

```
event.stopPropagation();
```

イベントの伝達は `capture` または `bubble` フェーズでのみ停止できます。

DOM の変更

ドキュメントオブジェクトモデル (DOM) は、HTML および XML ドキュメントのオブジェクトを表したり、操作したりする、言語に依存しないモデルです。重要なことは、DOM を安全に変更する方法を理解し、変更がフレームワークの表示サービスによって踏みつけられて予想外の結果にならないようにすることです。

このセクションの内容:

Lightning コンポーネントフレームワークによって管理される DOM 要素の変更

フレームワークは、コンポーネントによって所有される DOM 要素の作成と管理を行います。フレームワークによって作成される DOM 要素を変更する場合は、コンポーネントの `render` イベントのハンドラまたはカスタムレンダラで DOM 要素を変更します。このようにしないと、フレームワークはコンポーネントの表示時に変更を無効にします。

外部ライブラリによって管理される DOM 要素の変更

Charting ライブラリなど、さまざまなライブラリを使用して、DOM 要素の作成と管理を行うことができます。これらの DOM 要素は外部ライブラリによって管理されるため、`render` イベントハンドラまたはレンダラ内で変更する必要はありません。

Lightning コンポーネントフレームワークによって管理される DOM 要素の変更

フレームワークは、コンポーネントによって所有される DOM 要素の作成と管理を行います。フレームワークによって作成される DOM 要素を変更する場合は、コンポーネントの `render` イベントのハンドラまたはカスタムレンダラで DOM 要素を変更します。このようにしないと、フレームワークはコンポーネントの表示時に変更を無効にします。

たとえば、クライアント側コントローラから DOM 要素を直接変更すると、コンポーネントの表示時にその変更が上書きされる可能性があります。

`render` イベントハンドラまたはカスタムレンダラの外部で DOM を読み取ることはできます。

最も簡単な方法は、フレームワークに DOM の更新を任せることです。コンポーネントの属性を更新し、マークアップで式を使用します。フレームワークの表示サービスが DOM の更新を行います。

レンダラ外のコンポーネントのCSSクラスを変更するには、`$A.util.addClass()`、`$A.util.removeClass()`、`$A.util.toggleClass()` のメソッドを使用できます。

DOMで後処理を実行したり、コンポーネントの表示または再表示に反応したりすると役立つ場合があります。そのような場合のために、いくつかのオプションが用意されています。

このセクションの内容:

render イベントの処理

コンポーネントが表示または再表示されるときに、`aura:valueRender` イベント (`render` イベントとも呼ばれる) が起動されます。DOM で後処理を実行したり、コンポーネントの表示または再表示に反応したりするには、このイベントが処理されます。このイベントは優先され、カスタムレンダラを作成する代替方法よりも簡単に使用できます。

カスタムレンダラの作成

このフレームワークの表示サービスは、メモリ内のコンポーネントの状態を取得し、コンポーネントによって所有される DOM 要素の作成と管理を行います。コンポーネントのフレームワークで作成された DOM 要素を変更する場合、コンポーネントのレンダラで DOM 要素を変更できます。このようにしないと、フレームワークはコンポーネントの表示時に変更を無効にします。

関連トピック:

外部ライブラリによって管理される DOM 要素の変更

式の使用

マークアップの動的な表示または非表示

`render` イベントの処理

コンポーネントが表示または再表示されるときに、`aura:valueRender` イベント (`render` イベントとも呼ばれる) が起動されます。DOM で後処理を実行したり、コンポーネントの表示または再表示に反応したりするには、このイベントが処理されます。このイベントは優先され、カスタムレンダラを作成する代替方法よりも簡単に使用できます。

`render` イベントは、カスタムレンダラのすべてのメソッドが呼び出された後に起動されます。表示または再表示ライフサイクルの順序についての詳細は、「カスタムレンダラの作成」を参照してください。

`aura:valueRender` イベントの処理は、`init` フックの処理と似ています。コンポーネントのマークアップにハンドラを追加します。

```
<aura:handler name="render" value="{!this}" action="{!c.onRender}"/>
```

この例では、クライアント側コントローラの onRender アクションがコンポーネントの最初の表示と再表示を処理します。action 属性には任意の名前を選択できます。

関連トピック:

コンポーネントの初期化時のアクションの呼び出し

カスタムレンダラの作成

カスタムレンダラの作成

このフレームワークの表示サービスは、メモリ内のコンポーネントの状態を取得し、コンポーネントによって所有される DOM 要素の作成と管理を行います。コンポーネントのフレームワークで作成された DOM 要素を変更する場合、コンポーネントのレンダラで DOM 要素を変更できます。このようにしないと、フレームワークはコンポーネントの表示時に変更を無効にします。

DOM は、HTML および XML ドキュメントのオブジェクトを表したり、操作したりする、言語に依存しないモデルです。コンポーネントの表示はフレームワークによって自動的に行われるため、コンポーネントのデフォルトの表示動作をカスタマイズする必要がなければ、表示に関して詳細に把握する必要はありません。

メモ: カスタムレンダラを作成する代替策よりも、render イベントを処理する方が望ましく簡単です。

基本コンポーネントの表示

フレームワークの基本コンポーネントは aura:component です。どのコンポーネントもこの基本コンポーネントを拡張します。

aura:component のレンダラは、componentRenderer.js にあります。このレンダラには、表示サイクルおよび再表示サイクルの 4 つのフェーズの基本実装があります。

- render()
- rerender()
- afterRender()
- unrender()

フレームワークでは、これらの関数が表示ライフサイクルおよび再表示ライフサイクルの一部としてコールされます。これらの関数についてはすぐに学習します。基本表示関数は、カスタムレンダラで上書きできます。

表示ライフサイクル

表示ライフサイクルは、コンポーネントが明示的に非表示にされない限りコンポーネントの有効期間内に 1 回発生します。コンポーネントを作成すると、次の処理が行われます。

1. フレームワークによって init イベントが起動され、コンポーネントを構築してから表示するまでの間にコンポーネントを更新したり、イベントを起動したりできます。
2. render() メソッドがコールされてコンポーネントのボディが表示されます。
3. afterRender() メソッドがコールされて、フレームワークの表示サービスによって DOM 要素が挿入されたら、DOM ツリーを操作できます。

4. フレームワークの表示サービスによって DOM 要素が挿入されたら、フレームワークによって `render` イベントが起動され、DOM ツリーを操作できるようになります。`render` イベントの処理は、カスタムレンダラの作成や `afterRender()` の上書きよりも優先されます。

再表示ライフサイクル

再表示ライフサイクルにより、基盤となるデータが変更されたときにコンポーネントの再表示が自動的に処理されます。一般的なシーケンスを次に示します。

1. ブラウザイベントによって 1 つの以上の Lightning イベントがトリガされます。
2. 各 Lightning イベントによって、データを更新できる 1 つ以上のアクションがトリガされます。更新されたデータで複数のイベントが起動される場合もあります。
3. 表示サービスによって、起動されたイベントのスタックが追跡されます。
4. フレームワークによって、各コンポーネントの `rerender()` メソッドがコールされることで、更新されたデータを所有するすべてのコンポーネントが再表示されます。
5. フレームワークによって `render` イベントが起動され、コンポーネントが再表示された後に DOM ツリーを操作できます。カスタムレンダラの作成と `rerender()` の上書きには、`render` イベントの処理をお勧めします。

コンポーネントが有効で明示的に非表示にされない限り、基盤となるデータが変更されると常にコンポーネント再表示ライフサイクルが繰り返されます。

詳細は、「表示ライフサイクル中に起動されたイベント 」を参照してください。

カスタムレンダラ

通常はカスタムレンダラを作成する必要はありませんが、フレームワークの表示サービスによって DOM 要素が挿入されたら、DOM ツリーを操作する場合に役立ちます。表示動作をカスタマイズする際に、マークアップでのカスタマイズや、`init` イベントを使用したカスタマイズができない場合、クライアント側レンダラの作成が可能です。

レンダラファイルは、コンポーネントのバンドルの一部で、`<componentName>Renderer.js` という命名規則に従っていれば自動的に結び付けられます。たとえば、`sample.cmp` のレンダラ名は `sampleRenderer.js` の形式になります。

メモ: 表示をカスタマイズする場合、次のガイドラインが重要です。

- コンポーネントの一部であるDOM要素のみを変更してください。親コンポーネントからアクセスする場合でも、別のコンポーネントにアクセスしてそのDOM要素を変更すると、コンポーネントのカプセル化が壊れるため、このような行為は避けてください。
- 新しい表示サイクルがトリガされる可能性があるため、イベントを起動しないでください。代わりに `init` イベントを使用できます。
- 新しい表示サイクルがトリガされる可能性があるため、属性値を他のコンポーネントに設定しないでください。
- UI の多くの懸念事項 (位置設定など) を CSS に移動します。

コンポーネントの表示のカスタマイズ

表示をカスタマイズするには、コンポーネントのレンダラで render() 関数を作成して、基本 render() 関数を上書きします。これにより、DOM が更新されます。

render() 関数では、DOM ノードや DOM ノードの配列が返されるか、何も返されません。HTML の基本コンポーネントでは、コンポーネントを表示するときに DOM ノードが必要になります。

通常、カスタム表示コードを追加する前に render() 関数から superRender() をコールして、デフォルトの表示を拡張します。superRender() をコールすると、マークアップで指定された DOM ノードが作成されます。

次のコードは、カスタム render() 関数の概要を示します。

```
render : function(cmp, helper) {
    var ret = this.superRender();
    // do custom rendering here
    return ret;
},
```

コンポーネントの再表示

イベントが起動されると、影響を受けるコンポーネントでデータを変更して rerender() をコールするアクションがトリガされます。rerender() 関数では、最後の表示以降の他のコンポーネントに対する更新に基づいて、そのコンポーネント自体を更新できます。この関数では、値は返されません。

コンポーネントのデータを更新すると、フレームワークによって自動的に rerender() がコールされます。

通常、カスタム再表示コードを追加する前に renderer() 関数から superRerender() をコールして、デフォルトの再表示を拡張します。superRerender() をコールすると、body 属性のコンポーネントに再表示がチェーニングされます。

次のコードは、カスタム rerender() 関数の概要を示します。

```
rerender : function(cmp, helper){
    this.superRerender();
    // do custom rerendering here
}
```

表示後の DOM へのアクセス

フレームワークの表示サービスによって DOM 要素が挿入されたら、afterRender() 関数で DOM ツリーを操作できます。表示ライフサイクルで最後のコールは必要ありません。render() の後にコールされるだけで、値は返されません。

通常、カスタムコードを追加する前に superAfterRender() 関数をコールして、表示の後にデフォルトを拡張します。

次のコードは、カスタム afterRender() 関数の概要を示します。

```
afterRender: function (component, helper) {
    this.superAfterRender();
    // interact with the DOM here
},
```

コンポーネントの非表示

基本 `unrender()` 関数では、コンポーネントの `render()` 関数によって表示されているすべての DOM ノードが削除されます。この関数は、コンポーネントが破棄されると、フレームワークによってコールされます。この動作をカスタマイズするには、コンポーネントのレンダラで `unrender()` を上書きします。このメソッドは、フレームワークに対してネイティブでないサードパーティライブラリを操作している場合に便利です。

通常、カスタムコードを追加する前に `unrender()` 関数から `superUnrender()` をコールして、デフォルトの非表示を拡張します。

次のコードは、カスタム `unrender()` 関数の概要を示します。

```
unrender: function () {
    this.superUnrender();
    // do custom unrendering here
}
```

関連トピック:

DOM の変更
コンポーネントの初期化時のアクションの呼び出し
コンポーネントのバンドル
フレームワークのライフサイクル外のコンポーネントの変更
コンポーネントのバンドル内の JavaScript コードの共有

外部ライブラリによって管理される DOM 要素の変更

Charting ライブラリなど、さまざまなライブラリを使用して、DOM 要素の作成と管理を行うことができます。これらのDOM要素は外部ライブラリによって管理されるため、`render` イベントハンドラまたはレンダラ内で変更する必要はありません。

`render` イベントハンドラまたはレンダラは、Lightning コンポーネントフレームワークによって作成と管理が行われる DOM 要素のカスタマイズにのみ使用します。

外部ライブラリを使用するには、`<ltng:require>` を使用します。ライブラリが読み込まれて DOM の準備ができた後に、`afterScriptsLoaded` 属性を使用して DOM を操作できます。`<ltng:require>` タグにより、選択したライブラリの読み込みが、Lightningコンポーネントフレームワークの表示サイクルと調整されて、すべてが調和します。

関連トピック:

ltng:require
外部 JavaScript ライブラリの使用
Lightning コンポーネントフレームワークによって管理される DOM 要素の変更

コンポーネントの有効性の確認

非同期コードの実行中に UI で他の場所に移動すると、フレームワークは、非同期要求を実行したコンポーネントを非表示にして破棄します。そのコンポーネントを引き続き参照できますが、コンポーネントは無効になっています。`cmp.isValid()` コールは、無効なコンポーネントでは `false` を返します。

無効なコンポーネントに対して `cmp.get()` をコールすると、`cmp.get()` は `null` を返します。

無効なコンポーネントに対して `cmp.set()` をコールしても、何も起こらず、エラーも発生しません。これは基本的に無演算 (no op) 関数です。

多くのシナリオで `cmp.isValid()` コールは不要です。`cmp.get()` から取得した値の `null` チェックで十分です。`cmp.isValid()` をコールする主な理由は、コンポーネントに対して複数のコールを実行する場合に、結果ごとに `null` チェックを実行しなくて済むことです。

フレームワークのライフサイクルの内部

クライアント側コントローラに関連付けられたコンポーネントを参照する場合、クライアント側コントローラのコールバックで `cmp.isValid()` チェックは必要ありません。コンポーネントが無効であることがフレームワークによって自動的にチェックされます。同様に、イベント処理時や、フレームワークのライフサイクルフック (`init` イベントなど) 内で `cmp.isValid()` チェックは必要ありません。

クライアント側コントローラのサンプルを見てみましょう。

```
({
    "doSomething" : function(cmp) {
        var action = cmp.get("c.serverEcho");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                if (cmp.get("v.displayResult)) {
                    alert("From server: " + response.getReturnValue());
                }
            }
            // other state handling omitted for brevity
        });

        $A.enqueueAction(action);
    }
})
```

クライアント側コントローラに結び付けられているコンポーネントは、`cmp` パラメータとして `doSomething` アクションに渡されます。`cmp.get("v.displayResult)` がコールされたときに `cmp.isValid()` チェックは必要ありません。

ただし、自分のコンポーネントが有効であるにもかかわらず、有効でない可能性のある別のコンポーネントへの参照を保持している場合は、他のコンポーネントの `cmp.isValid()` チェックが必要になることがあります。ローカル ID が `child` の別のコンポーネントへの参照を持つコンポーネントの例を見てみましょう。

```
({
    "doSomething" : function(cmp) {
        var action = cmp.get("c.serverEcho");
        var child = cmp.find("child");
```

```
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                if (child.get("v.displayResult)) {
                    alert("From server: " + response.getReturnValue());
                }
            }
            // other state handling omitted for brevity
        });

        $A.enqueueAction(action);
    }
})
```

子コンポーネントがない前の例の次の行を、

```
if (cmp.get("v.displayResult)) {
```

次のように変更します。

```
if (child.get("v.displayResult)) {
```

子コンポーネントが無効な場合 `child.get("v.displayResult)` は `null` を返すため、ここでは `child.isValid()` コールは不要です。子コンポーネントに対して複数のコールを実行し、結果ごとに `null` チェックを実行するのを回避する場合にのみ、`child.isValid()` チェックを追加します。

フレームワークのライフサイクルの外部

`setTimeout()` や `setInterval()` などの非同期コードでコンポーネントを参照する場合、または Promise を使用する場合、`cmp.isValid()` コールは非同期要求の結果を処理する前にコンポーネントがまだ有効であることをチェックします。多くのシナリオで `cmp.isValid()` コールは不要です。`cmp.get()` から取得した値の `null` チェックで十分です。`cmp.isValid()` をコールする主な理由は、コンポーネントに対して複数のコールを実行する場合に、結果ごとに `null` チェックを実行しなくて済むことです。

たとえば、コンポーネントが無効な場合、`cmp.set()` コールは何もしないため、この `setTimeout()` コール内で `cmp.isValid()` チェックは必要ありません。

```
window.setTimeout(
    $A.getCallback(function() {
        cmp.set("v.visible", true);
    }), 5000
);
```

関連トピック:

[クライアント側コントローラを使用したイベントの処理]

[コンポーネントの初期化時のアクションの呼び出し]

[フレームワークのライフサイクル外のコンポーネントの変更]

フレームワークのライフサイクル外のコンポーネントの変更

$A.getCallback() を使用して、setTimeout() コールの場合のように通常の表示ライフサイクル外のコンポーネントを変更するコードをラップします。$A.getCallback() コールは、フレームワークが変更されたコンポーネントを確実に表示し、すべてのエンキューされたアクションが処理されるようにします。

メモ: $A.run() は廃止されました。代わりに $A.getCallback() を使用します。

コードがフレームワークのコールスタックの一部として実行される場合は、$A.getCallback() を使用する必要はありません。たとえば、コードがイベントを処理している場合や、サーバ側のコントローラアクションのコールバックにある場合です。

$A.getCallback() を使用する必要がある場合の例として、イベントハンドラで window.setTimeout() をコールして、一部のロジックを遅延実行する場合が挙げられます。この場合は、コードがフレームワークのコールスタック外に配置されます。

次のサンプルでは、5 秒の遅延後に、コンポーネントの visible 属性を true に設定します。

```
window.setTimeout(
    $A.getCallback(function() {
        cmp.set("v.visible", true);
    }), 5000
);
```

フレームワークが変更されたコンポーネントを確実に表示するようにする $A.getCallback() で、コンポーネントの属性を更新するコードがどのようにラップされているかに注意します。

メモ: コンポーネントが無効な場合、cmp.set() コールは何もしないため、この setTimeout() コール内で cmp.isValid() チェックは必要ありません。

警告: $A.getCallback() でラップされた関数への参照を保存しないでください。後でこの参照を使用してアクションを送信すると、保存されたトランザクション状態により、アクションが中止されます。

関連トピック:

クライアント側コントローラを使用したイベントの処理
コンポーネントの有効性の確認
非 Lightning コードからの Lightning イベントの起動
イベントとの通信

項目の検証

ユーザ入力を検証してエラーを処理し、入力項目にエラーメッセージを表示します。

クライアント側の入力規則は、次のコンポーネントに使用できます。

- lightning:input
- lightning:select
- lightning:textarea
- ui:input*

lightning 名前空間のコンポーネントでは、エラー条件を定義する属性が提供されて入力規則が簡素化されるため、コンポーネントの有効性の状態を確認してエラーを処理できるようになります。たとえば、項目の最小の長さを設定し、条件が満たされないときにエラーメッセージを表示して、特定の有効性の状態に基づいてエラーを処理できます。詳細は、「コンポーネントの参照」の lightning 名前空間コンポーネントに関する内容を参照してください。

または、ui 名前空間の入力コンポーネントでは、クライアント側コントローラでエラーの定義と処理を行い、エラーのリスト内を反復処理できます。

次のセクションでは、ui:input* コンポーネントのエラー処理について説明します。

デフォルトのエラー処理

フレームワークでは、デフォルトのエラーコンポーネント ui:inputDefaultError を使用して、エラーを処理および表示できます。このコンポーネントは、inputCmp.set("v.errors",[{message:"my error message"}]) 構文を使用してエラーを設定するときに動的に作成されます。次の例で、検証エラーを処理してエラーメッセージを表示する方法を示します。これがそのマークアップです。

```
<!--c:errorHandling-->
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp"/> <br/>
    <lightning:button label="Submit" onclick="{!c.doAction}"/>
</aura:component>
```

これはクライアント側コントローラです。

```
/*errorHandlingController.js*/
{
    doAction : function(component) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // Is input numeric?
        if (isNaN(value)) {
            // Set error
            inputCmp.set("v.errors", [{message:"Input not a number: " + value}]);
        } else {
            // Clear error
            inputCmp.set("v.errors", null);
        }
    }
}
```

値を入力して [送信] をクリックすると、コントローラの doAction アクションによって入力値が検証され、入力値が数値でない場合はエラーメッセージが表示されます。有効な入力値を入力すると、エラーがクリアされます。errors 属性を使用して、エラーメッセージを入力コンポーネントに追加します。

カスタムエラーの処理

ui:input およびその子コンポーネントは、onError および onClearErrors イベントを使用してエラーを処理できます。これらのイベントは、コントローラで定義されたカスタムエラーハンドラに結び付けられてい

ます。onError は ui:validationError イベントに対応付けられ、onClearErrors は ui:clearErrors に対応付けられます。

次の例に、カスタムエラーハンドラを使用して検証エラーを処理し、デフォルトのエラーコンポーネントを使用してエラーメッセージを表示する方法を示します。これがそのマークアップです。

```
<!--c:errorHandlingCustom-->
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/> <br/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

これはクライアント側コントローラです。

```
/*errorHandlingCustomController.js*/
{
    doAction : function(component, event) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // is input numeric?
        if (isNaN(value)) {
            inputCmp.set("v.errors", [{message:"Input not a number: " + value}]);
        } else {
            inputCmp.set("v.errors", null);
        }
    },

    handleError: function(component, event){
        /* do any custom error handling
         * logic desired here */
        // get v.errors, which is an Object[]
        var errorsArr  = event.getParam("errors");
        for (var i = 0; i < errorsArr.length; i++) {
            console.log("error " + i + ": " + JSON.stringify(errorsArr[i]));
        }
    },

    handleClearError: function(component, event) {
        /* do any custom error handling
         * logic desired here */
    }
}
```

値を入力して [送信] をクリックすると、コントローラの doAction アクションが実行されます。ただし、フレームワークにエラーを処理させるのではなく、<ui:inputNumber> の onError イベントを使用するカスタムエラーハンドラを定義します。検証に失敗すると、doAction が errors attribute を使用してエラーメッセージを追加します。これにより、handleError カスタムエラーハンドラが自動的に起動します。

同様に、onClearErrors イベントを使用して、エラーをクリアする方法をカスタマイズできます。例については、コントローラの handleClearError ハンドラを参照してください。

関連トピック:

クライアント側コントローラを使用したイベントの処理
コンポーネントイベント

エラーの発生および処理

このフレームワークでは、復旧できないアプリケーションエラーおよび復旧できるアプリケーションエラーを JavaScript コードで柔軟に対処できます。たとえば、サーバ側の応答のエラーを処理するときに、これらのエラーをコールバック内に発生させることができます。

復旧できないエラー

アプリケーションが正常に起動できないエラーなどの復旧できないエラーには、throw new Error("error message here") を使用します。これでエラーメッセージが表示されます。

メモ: $A.error() は廃止されました。代わりに、throw new Error() を使用することによって、ネイティブ JavaScript Error オブジェクトを発生させます。

次の例では、JavaScript コントローラでの基本的な復旧できないエラーの発生を示します。

```
<!--c:unrecoverableError-->
<aura:component>
    <lightning:button label="throw error" onclick="{!c.throwError}"/>
</aura:component>
```

クライアント側コントローラのソースを次に示します。

```
/*unrecoverableErrorController.js*/
({
    throwError : function(component, event){
        throw new Error("I can't go on. This is the end.");
    }
})
```

復旧できるエラー

復旧できるエラーを処理するには、ui:message などのコンポーネントを使用して、その問題についてユーザに通知します。

次のサンプルでは、JavaScript コントローラでの基本的な復旧できるエラーの発生およびキャッチを示します。

```
<!--c:recoverableError-->
<aura:component>
    <p>Click the button to trigger the controller to throw an error.</p>
    <div aura:id="div1"></div>
```

```
    <lightning:button label="Throw an Error" onclick="{!c.throwErrorForKicks}"/>
</aura:component>
```

クライアント側コントローラのソースを次に示します。

```
/*recoverableErrorController.js*/
({
    throwErrorForKicks: function(cmp) {
        // this sample always throws an error to demo try/catch
        var hasPerm = false;
        try {
            if (!hasPerm) {
                throw new Error("You don't have permission to edit this record.");
            }
        }
        catch (e) {
            $A.createComponents([
                ["ui:message",{
                    "title" : "Sample Thrown Error",
                    "severity" : "error",
                }],
                ["ui:outputText",{
                    "value" : e.message
                }]
                ],
                function(components, status, errorMessage){
                    if (status === "SUCCESS") {
                        var message = components[0];
                        var outputText = components[1];
                        // set the body of the ui:message to be the ui:outputText
                        message.set("v.body", outputText);
                        var div1 = cmp.find("div1");
                        // Replace div body with the dynamic component
                        div1.set("v.body", message);
                    }
                    else if (status === "INCOMPLETE") {
                        console.log("No response from server or client is offline.")
                        // Show offline error
                    }
                    else if (status === "ERROR") {
                        console.log("Error: " + errorMessage);
                        // Show error message
                    }
                }
            );
        }
    }
})
```

この例では、コントローラコードが常にエラーを発生させてキャッチします。エラーのメッセージは、動的に作成される ui:message コンポーネントでユーザに表示されます。ui:message のボディは、エラーテキストを含む ui:outputText コンポーネントです。

関連トピック:

項目の検証

コンポーネントの動的な作成

コンポーネントメソッドのコール

<aura:method> を使用して、コンポーネントの API の一部としてメソッドを定義します。これにより、コンポーネントイベントを起動して処理する代わりに、コンポーネントのクライアント側コントローラからメソッドを直接コールできるようになります。<aura:method> を使用すると、親コンポーネントに含まれる子コンポーネントのメソッドをコールする場合に、親コンポーネントに必要なコードが簡略化されます。

コンポーネント間の通信

コンテインメント階層の下位と通信するには、aura:method を使用します。たとえば親コンポーネントで、その親に含まれる子コンポーネントの aura:method をコールします。

コンテインメント階層の上位と通信するには、子コンポーネントでコンポーネントイベントを起動し、そのイベントを親コンポーネントで処理します。

構文

次の構文を使用して、JavaScript コードのメソッドをコールします。

```
cmp.sampleMethod(arg1, … argN);
```

cmp は、コンポーネントへの参照です。

sampleMethod は、aura:method の名前です。

arg1, … argN は、メソッドに渡される引数の省略可能なカンマ区切りのリストです。各引数は、aura:method マークアップで定義された aura:attribute に対応します。

継承されたメソッドの使用

スーパーコンポーネントを拡張するサブコンポーネントは、スーパーコンポーネントで定義されたメソッドにアクセスできます。

インターフェースに <aura:method> タグを含めることもできます。このインターフェースを実装するコンポーネントは、このメソッドにアクセスできます。

例

アプリケーションの例を見てみましょう。

```
<!-- c:auraMethodCallerWrapper.app -->
<aura:application >
    <c:auraMethodCaller />
</aura:application>
```

c:auraMethodCallerWrapper.app には c:auraMethodCaller コンポーネントが含まれます。

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    ...
</aura:component>
```

c:auraMethodCaller は親コンポーネントです。c:auraMethodCaller には子コンポーネント c:auraMethod が含まれます。

c:auraMethodCaller を使用して c:auraMethod で定義された aura:method をコールする方法について説明します。

c:auraMethodCallerWrapper.app を使用して、同期コードと非同期コードの結果を返す方法を確認します。

このセクションの内容:

同期コードの結果を返す

aura:method は同期して実行されます。同期メソッドは、返す前に実行を終了します。同期 JavaScript コードから値を返すには、return ステートメントを使用します。

非同期コードの結果を返す

aura:method は同期して実行されます。同期 JavaScript コードから値を返すには、return ステートメントを使用します。サーバ側のアクションをコールする JavaScript コードは非同期です。非同期コードは返した後も実行を続けることができます。aura:method は非同期コードが終了する前に返すため、return ステートメントを使用して、非同期コールの結果を返すことはできません。非同期コードの場合は、return ステートメントではなくコールバックを使用します。

関連トピック:

aura:method

コンポーネントイベント

同期コードの結果を返す

aura:method は同期して実行されます。同期メソッドは、返す前に実行を終了します。同期 JavaScript コードから値を返すには、return ステートメントを使用します。

非同期メソッドは返した後も実行を続けることができます。JavaScript コードは、多くの場合コールバックパターンを使用して、非同期コードの終了後に結果を返します。非同期アクションの結果を返す方法については、後で説明します。

ステップ 1: マークアップで `aura:method` を定義する

同期コードを実行する `logParam` `aura:method` を見てみましょう。`c:auraMethodCallerWrapper.app` および「コンポーネントメソッドのコール」に示されたコンポーネントを使用します。`aura:method` を定義するマークアップは次のようになります。

```
<!-- c:auraMethod -->
<aura:component>
    <aura:method name="logParam"
      description="Sample method with parameter">
        <aura:attribute name="message" type="String" default="default message" />
    </aura:method>

    <p>This component has an aura:method definition.</p>
</aura:component>
```

`logParam` `aura:method` には、名前が `message` の `aura:attribute` があります。この属性を使用して、`logParam` メソッドをコールするときに `message` パラメータを設定できます。

`logParam` の名前属性で、クライアント側コントローラの `logParam()` を呼び出すように `aura:method` を設定します。

`aura:method` には、複数の `aura:attribute` タグを使用できます。各 `aura:attribute` は、`aura:method` に渡すことができるパラメータに対応します。構文についての詳細は、aura:method を参照してください。

`aura:method` マークアップでは、戻り値を明示的に宣言しません。JavaScript コントローラで `return` ステートメントを使用します。

ステップ 2: コントローラで `aura:method` ロジックを実装する

`logParam` `aura:method` で、`auraMethodController.js` の `logParam()` を呼び出します。このソースを見てみましょう。

```
/* auraMethodController.js */
({
    logParam : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var message = params.message;
            console.log("message: " + message);
            return message;
        }
    },
})
```

`logParam()` は単に渡されたパラメータを記録し、`return` ステートメントの使用方法を示すパラメータ値を返します。たとえば、非同期のサーバ側アクションコールを行わない同期コードの場合、`return` ステートメントを使用できます。

ステップ 3: 親コントローラから aura:method をコールする

c:auraMethodCaller の callAuraMethod() で、その子コンポーネント c:auraMethod で定義された logParam aura:method をコールします。c:auraMethodCaller のコントローラは次のようになります。

```
/* auraMethodCallerController.js */
({
    callAuraMethod : function(component, event, helper) {
        var childCmp = component.find("child");
        // call the aura:method in the child component
        var auraMethodResult =
          childCmp.logParam("message sent by parent component");
        console.log("auraMethodResult: " + auraMethodResult);
    },
})
```

callAuraMethod() は子コンポーネント c:auraMethod を検索し、aura:method のメッセージパラメータの引数がある logParam aura:method をコールします。

```
childCmp.logParam("message sent by parent component");
```

auraMethodResult は、logParam から返された値です。

ステップ 4: aura:method へのコールを開始するボタンを追加する

c:auraMethodCaller マークアップには、auraMethodCallerController.js の callAuraMethod() を呼び出す lightning:button が含まれます。このボタンを使用して、子コンポーネントの aura:method へのコールを開始します。

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    <lightning:button label="Call aura:method in child component"
        onclick="{! c.callAuraMethod}" />
</aura:component>
```

関連トピック:

非同期コードの結果を返す
コンポーネントメソッドのコール
aura:method

非同期コードの結果を返す

aura:method は同期して実行されます。同期 JavaScript コードから値を返すには、return ステートメントを使用します。サーバ側のアクションをコールする JavaScript コードは非同期です。非同期コードは返した後も実行を続けることができます。aura:method は非同期コードが終了する前に返すため、return ステートメントを使用して、非同期コールの結果を返すことはできません。非同期コードの場合は、return ステートメントではなくコールバックを使用します。

ステップ 1: マークアップで aura:method を定義する

コールバックを使用する echo aura:method を見てみましょう。c:auraMethodCallerWrapper.app および「コンポーネントメソッドのコール」に示されたコンポーネントを使用します。c:auraMethod コンポーネントの echo aura:method は次のようになります。

```
<!-- c:auraMethod -->
<aura:component controller="SimpleServerSideController">
    <aura:method name="echo"
      description="Sample method with server-side call">
        <aura:attribute name="callback" type="Object" />
    </aura:method>

    <p>This component has an aura:method definition.</p>
</aura:component>
```

echo aura:method には、名前が callback の aura:attribute があります。この属性を使用して、SimpleServerSideController でサーバ側のアクションを実行した後に、aura:method によって呼び出されるコールバックを設定できます。

ステップ 2: コントローラで aura:method ロジックを実装する

echo aura:method で、auraMethodController.js の echo() を呼び出します。ソースを見てみましょう。

```
/* auraMethodController.js */
({
    echo : function(cmp, event) {
        var params = event.getParam('arguments');
        var callback;
        if (params) {
            callback = params.callback;
        }

        var action = cmp.get("c.serverEcho");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                console.log("From server: " + response.getReturnValue());
                // return doesn't work for async server action call
                //return response.getReturnValue();
                // call the callback passed into aura:method
                if (callback) callback(response.getReturnValue());
            }
            else if (state === "INCOMPLETE") {
                // do something
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                          errors[0].message);
```

```
                }
            } else {
                console.log("Unknown error");
            }
        }
    });
    $A.enqueueAction(action);
  },
})
```

echo() でサーバ側コントローラアクション serverEcho() をコールします。これは次に作成します。

メモ: return ステートメントで結果を返すことはできません。aura:method は、非同期のサーバ側アクションコールが終了する前に返します。代わりに、aura:method に渡されたコールバックを呼び出して、コールバックのパラメータとして結果を設定します。

ステップ 3: Apex サーバ側コントローラを作成する

echo aura:method で、serverEcho というサーバ側コントローラアクションをコールします。サーバ側コントローラのソースは次のようになります。

```
public with sharing class SimpleServerSideController {
    @AuraEnabled
    public static String serverEcho() {
        return ('Hello from the server');
    }
}
```

serverEcho() メソッドは String を返します。

ステップ 4: 親コントローラから **aura:method** をコールする

c:auraMethodCaller のコントローラは次のようになります。その子コンポーネント c:auraMethod で、echo aura:method をコールします。

```
/* auraMethodCallerController.js */
({
    callAuraMethodServerTrip : function(component, event, helper) {
        var childCmp = component.find("child");
        // call the aura:method in the child component
        childCmp.echo(function(result) {
            console.log("callback for aura:method was executed");
            console.log("result: " + result);
        });
    },
})
```

callAuraMethodServerTrip() は子コンポーネント c:auraMethod を検索し、その echo aura:method をコールします。echo() で、コールバック関数を aura:method に渡します。

auraMethodCallerController.js で設定されたコールバックで結果を記録します。

```
function(result) {
    console.log("callback for aura:method was executed");
```

```
    console.log("result: " + result);
}
```

ステップ 5: `aura:method` へのコールを開始するボタンを追加する

`c:auraMethodCaller` マークアップには、`auraMethodCallerController.js` の `callAuraMethodServerTrip()` を呼び出す `lightning:button` が含まれます。このボタンを使用して、子コンポーネントの `aura:method` へのコールを開始します。

`c:auraMethodCaller` のマークアップは次のようになります。

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    <lightning:button label="Call aura:method (server trip) in child component"
        onclick="{! c.callAuraMethodServerTrip}" />
</aura:component>
```

関連トピック:

同期コードの結果を返す

コンポーネントメソッドのコール

aura:method

JavaScript Promise の使用

JavaScript コードで ES6 Promise を使用できます。プロミスにより、非同期コールの成否を処理するコードや、複数の非同期コールをまとめてチェーニングするコードを簡素化できます。

ブラウザでネイティブバージョンが提供されない場合は、フレームワークがポリフィルを使用するため、プロミスは Lightning Experience でサポートされるすべてのブラウザで機能します。

ここでは、プロミスの基本を十分に理解していることを前提としています。プロミスのわかりやすい概要については、https://developers.google.com/web/fundamentals/getting-started/primers/promisesを参照してください。

プロミスは省略可能な機能です。プロミスを愛用している人もいれば、そうでない人もいます。各自の使用事例にとって有用であれば利用します。

プロミスの作成

次の `firstPromise` 関数は、プロミスを返します。

```
firstPromise : function() {
    return new Promise($A.getCallback(function(resolve, reject) {
      // do something

      if (/* success */) {
        resolve("Resolved");
      }
```

```
      else {
        reject("Rejected");
      }
    }));
}
```

このプロミスのコンストラクタが、プロミスで `resolve()` または `reject()` をコールする条件を決定します。

プロミスのチェーニング

複数のコールバックを連携させたり、まとめてチェーニングしたりする必要があるときはプロミスが役立ちます。汎用的なパターンは次のとおりです。

```
firstPromise()
    .then(
        // resolve handler
        $A.getCallback(function(result) {
            return anotherPromise();
        }),

        // reject handler
        $A.getCallback(function(error) {
            console.log("Promise was rejected: ", error);
            return errorRecoveryPromise();
        })
    )
    .then(
        // resolve handler
        $A.getCallback(function() {
            return yetAnotherPromise();
        })
    );
```

`then()` メソッドは、複数のプロミスをチェーニングします。この例では、各解決ハンドラが別のプロミスを返します。

`then()` は Promise API の一部です。次の 2 つの引数を取ります。

1. 達成されたプロミスのコールバック (解決ハンドラ)
2. 却下されたプロミスのコールバック (却下ハンドラ)

1 つ目のコールバック `function(result)` は、プロミスコンストラクタで `resolve()` がコールされたときにコールされます。コールバックの `result` オブジェクトは、`resolve()` への引数として渡されるオブジェクトです。

2 つ目のコールバック `function(error)` は、プロミスコンストラクタで `reject()` がコールされたときにコールされます。コールバックの `error` オブジェクトは、`reject()` への引数として渡されるオブジェクトです。

メモ: この例では、2 つのコールバックが `$A.getCallback()` でラップされています。どういうことでしょうか? プロミスはその解決関数と却下関数を非同期に実行するため、コードは Lightning イベントループおよび通常の表示ライフサイクルの外側に存在します。解決または却下コードによって、コンポーネ

ント属性の設定など、Lightning コンポーネントフレームワークに何らかのコールを実行する場合は、`$A.getCallback()` を使用してコードをラップします。詳細は、「フレームワークのライフサイクル外のコンポーネントの変更」(ページ 290)を参照してください。

`catch()` または却下ハンドラを常に使用

1 つ目の `then()` メソッドの却下ハンドラは、`errorRecoveryPromise()` が設定されたプロミスを返します。却下ハンドラは多くの場合、プロミスチェーンの「中流」で使用され、エラー回復メカニズムをトリガします。

Promise API には、必要に応じて未処理のエラーを検出する `catch()` メソッドが含まれます。プロミスチェーンには常に拒否ハンドラまたは `catch()` メソッドを含めます。

プロミスでエラーが発生しても、フレームワークがグローバルエラーハンドラを設定する `window.onerror` はトリガされません。`catch()` メソッドがない場合は、開発時に、ブラウザのコンソールにプロミスの未検出エラーに関するレポートがないか注意します。`catch()` メソッドにエラーメッセージを表示するには、`$A.reportError()` を使用します。`catch()` の構文は次のとおりです。

```
promise.then(...)
    .catch(function(error) {
        $A.reportError("error message here", error);
    });
```

`catch()` についての詳細は、「Mozilla Developer Network」を参照してください。

プロミスで保存可能なアクションを使用しない

フレームワークは、保存可能なアクションの応答をクライアント側のキャッシュに保存します。この保存された応答によってアプリケーションのパフォーマンスが大幅に向上し、一時的にネットワークに接続されていないデバイスをオフラインで使用できるようになります。保存可能なアクションが適しているのは参照のみのアクションだけです。

保存可能なアクションのコールバックが複数回呼び出されていることがあります。この場合、初回はキャッシュされたデータが使用され、それ以降はサーバからの更新済みデータが使用されます。プロミスは解決または却下を 1 回のみ行うものであるため、複数の呼び出しはプロミスに適していません。

関連トピック:

保存可能なアクション

コンポーネントからの API コールの実行

デフォルトでは、クライアント側のコードからサードパーティの API にコールを実行することはできません。リモートサイトを CSP 信頼済みサイトとして追加して、クライアント側のコードがアセットを読み込み、そのサイトのドメインに API 要求を実行できるようにします。

Lightning コンポーネントフレームワークでは、W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御します。Lightning アプリケーションは Salesforce API

以外のドメインから提供され、デフォルトの CSP ポリシーにより JavaScript コードからの API コールが許可されません。CSP 信頼済みサイトを追加して、ポリシーと、CSP ヘッダーのコンテンツを変更します。

重要: サードパーティサイトからは、CSP 信頼済みサイトであっても JavaScript リソースを読み込むことはできません。サードパーティサイトの JavaScript ライブラリを使用するには、そのライブラリを静的リソースに追加し、静的リソースをコンポーネントに追加します。ライブラリが静的リソースから読み込まれたら、通常どおり使用できます。

API コールを、クライアント側のコードではなく、サーバ側のコントローラから実行しなければならないことがあります。特に、クライアント側の Lightning コンポーネントコードから Salesforce API にコールを実行することはできません。サーバ側のコントローラからの API コールの実行についての詳細は、「Apex からの API コールの実行」 (ページ 344)を参照してください。

関連トピック:

コンテンツセキュリティポリシーの概要
サードパーティ API にアクセスするための CSP 信頼済みサイトの作成

サードパーティ API にアクセスするための CSP 信頼済みサイトの作成

Lightning コンポーネントフレームワークでは、W3C 標準のコンテンツセキュリティポリシー (CSP) を使用して、ページに読み込むことができるコンテンツのソースを制御します。外部(Salesforce以外の)サーバに対する要求を実行するサードパーティ API を使用するには、サーバを CSP 信頼済みサイトとして追加します。

CSP は、Web アプリケーションセキュリティに関する W3C ワーキンググループの勧告候補です。このフレームワークでは、W3C が推奨する `Content-Security-Policy` HTTP ヘッダーを使用しています。デフォルトでは、フレームワークのヘッダーに読み込むことができるコンテンツは安全な (HTTPS) URL からのみで、JavaScript からの XHR 要求は禁止されています。

CSP 信頼済みサイトを定義すると、そのサイトの URL が CSP ヘッダーの次に示すディレクティブの許可サイト一覧に追加されます。

- `connect-src`
- `frame-src`
- `img-src`
- `style-src`
- `font-src`
- `media-src`

このCSPヘッダーディレクティブへの変更によって、Lightning コンポーネントがサイトから画像、スタイル、フォントなどのリソースを読み込むことができます。また、クライアント側のコードでサイトへの要求を行うこともできます。

エディション

使用可能なエディション: Salesforce Classic および Lightning Experience

使用可能なエディション: **Developer** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition

ユーザ権限

作成、参照、更新、および削除する

- 「アプリケーションのカスタマイズ」または「すべてのデータの編集」

重要: サードパーティサイトからは、CSP 信頼済みサイトであっても JavaScript リソースを読み込むことはできません。サードパーティサイトの JavaScript ライブラリを使用するには、そのライブラリを静的リソースに追加し、静的リソースをコンポーネントに追加します。ライブラリが静的リソースから読み込まれたら、通常どおり使用できます。

1. [設定] から、[クイック検索] ボックスに「*CSP*」と入力し、**[CSP 信頼済みサイト]** を選択します。
   このページには、すでに登録済みの CSP 信頼済みサイトの一覧が表示され、サイト名や URL など、各サイトの付加情報があります。
2. **[新規信頼済みサイト]** を選択します。
3. 信頼済みサイトに名前を付けます。
   たとえば、「*Google マップ*」と入力します。
4. 信頼済みサイトの URL を入力します。
   この URL は `http://` または `https://` で開始し、ドメイン名を含める必要があります。また、ポートを含めることもできます。

   警告: デフォルトの CSP では、外部リソース用のセキュアな (HTTPS) 接続が要求されます。セキュアでない (HTTP) URL を使用して CSP 信頼済みサイトを設定しないでください。組織のセキュリティを侵害することになります。

5. オプション: 信頼済みサイトの説明を入力します。
6. オプション: 信頼済みサイトを実際に削除するのではなく一時的に無効にするには、[有効] チェックボックスをオフにします。
7. **[保存]** を選択します。

メモ: CSP 信頼済みサイトは、Lightning コンポーネントフレームワーク要求でのみ CSP ヘッダーに影響します。Visualforce または Apex での対応するアクセス権を有効にするには、リモートサイトを作成します。

CSP が適用されないブラウザもあります。CSP が適用されるブラウザのリストについては、caniuse.com を参照してください。

IE11 では CSP がサポートされていないため、サポートされている他のブラウザを使用してセキュリティを強化することをお勧めします。

関連トピック:

コンテンツセキュリティポリシーの概要

コンポーネントからの API コールの実行

Lightning コンポーネントのブラウザサポートの考慮事項

JavaScript Cookbook

このセクションには、さまざまな JavaScript ファイルで使用できるコードスニペットとサンプルがあります。

このセクションの内容:

コンポーネントの動的な作成

`$A.createComponent()` メソッドを使用して、クライアント側の JavaScript コードでコンポーネントを動的に作成します。複数のコンポーネントを作成するには、`$A.createComponents()` を使用します。

変更ハンドラを使用したデータ変更の検出

コンポーネントのいずれかの属性の値が変更されたときに、変更ハンドラを自動的に呼び出す (クライアント側コントローラのアクション) ようにコンポーネントを設定します。

ID によるコンポーネントの検索

JavaScript コードに ID を使用してコンポーネントを取得します。

コンポーネントへのイベントハンドラの動的な追加

コンポーネントから起動されるイベントのハンドラを動的に追加できます。

マークアップの動的な表示または非表示

マークアップの表示の切り替えに CSS を使用できます。ただし、`<aura:if>` は必要になるまで囲まれた要素ツリーを保留するため、この使用をお勧めします。

スタイルの追加と削除

実行時にコンポーネントまたは要素の CSS スタイルを追加または削除できます。

押下されたボタンの確認

複数のボタンがあるコンポーネントで押下されたボタンを確認するには、`Component.getLocalId()` を使用します。

JavaScript での日付の書式設定

`AuraLocalizationService` JavaScript API では、日付の書式設定とローカライズを処理するメソッドを使用できます。

コンポーネントの動的な作成

`$A.createComponent()` メソッドを使用して、クライアント側の JavaScript コードでコンポーネントを動的に作成します。複数のコンポーネントを作成するには、`$A.createComponents()` を使用します。

メモ: 廃止された `$A.newCmp()` および `$A.newCmpAsync()` メソッドの代わりに、`$A.createComponent()` メソッドを使用します。

構文は次のとおりです。

```
$A.createComponent(String type, Object attributes, function callback)
```

1. `type` — 作成するコンポーネントの種類 (`"ui:button"` など)。
2. `attributes` — コンポーネントの属性の対応付け。ローカル ID (`aura:id`) を含みます。
3. `callback(cmp, status, errorMessage)` — コンポーネントの作成後に呼び出すコールバック。コールバックには 3 つのパラメータがあります。
   a. `cmp` — 作成されたコンポーネント。このパラメータを作成するコンポーネントのボディへの追加など、新しいコンポーネントによって何らかの動作を実行することができます。エラーが生じた場合は、`cmp` が `null` になります。

b. status — コールの状況。有効な値は、SUCCESS、INCOMPLETE、ERROR です。コンポーネントを使用する前に必ず状況が SUCCESS であることを確認します。

c. errorMessage — 状況が ERROR の場合のエラーメッセージ。

動的に作成されたボタンを次のサンプルコンポーネントに追加してみましょう。

```
<!--c:createComponent-->
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <p>Dynamically created button</p>
    {!v.body}
</aura:component>
```

クライアント側のコントローラは $A.createComponent() をコールして、ローカル ID と press イベントのハンドラが設定された ui:button を作成します。function(newButton, ...) コールバックによって、c:createComponent の body にボタンが追加されます。$A.createComponent() によって動的に作成された newButton は、1 つ目の引数としてコールバックに渡されます。

```
/*createComponentController.js*/
({
    doInit : function(cmp) {
        $A.createComponent(
            "lightning:button",
            {
                "aura:id": "findableAuraId",
                "label": "Press Me",
                "onclick": cmp.getReference("c.handlePress")
            },
            function(newButton, status, errorMessage){
                //Add the new button to the body array
                if (status === "SUCCESS") {
                    var body = cmp.get("v.body");
                    body.push(newButton);
                    cmp.set("v.body", body);
                }
                else if (status === "INCOMPLETE") {
                    console.log("No response from server or client is offline.")
                    // Show offline error
                }
                else if (status === "ERROR") {
                    console.log("Error: " + errorMessage);
                    // Show error message
                }
            }
        );
    },

    handlePress : function(cmp) {
        console.log("button pressed");
    }
})
```

メモ: c:createComponent には、{!v.body} 式が含まれています。cmp.set("v.body", ...) を使用してコンポーネントのボディを設定するときは、コンポーネントマークアップに {!v.body} を明示的に含める必要があります。

ネストしたコンポーネントの作成

別のコンポーネントのボディにコンポーネントを動的に作成するには、$A.createComponents() を使用してコンポーネントを作成します。関数コールバックで、外部コンポーネントの body に内部コンポーネントを設定して、コンポーネントをネストします。次の例では、ui:message コンポーネントの body に ui:outputText コンポーネントを作成します。

```
$A.createComponents([
    ["ui:message",{
        "title" : "Sample Thrown Error",
        "severity" : "error",
    }],
    ["ui:outputText",{
        "value" : e.message
    }]
    ],
    function(components, status, errorMessage){
        if (status === "SUCCESS") {
            var message = components[0];
            var outputText = components[1];
            // set the body of the ui:message to be the ui:outputText
            message.set("v.body", outputText);
        }
        else if (status === "INCOMPLETE") {
            console.log("No response from server or client is offline.")
            // Show offline error
        }
        else if (status === "ERROR") {
            console.log("Error: " + errorMessage);
            // Show error message
        }
    }
);
```

動的に作成されたコンポーネントの廃棄

マークアップで宣言されたコンポーネントが使用されなくなると、フレームワークで自動的に廃棄され、メモリが解放されます。

コンポーネントが JavaScript で動的に作成され、そのコンポーネントがファセット (v.body または Aura.Component[] 種別の他の属性) に追加されていない場合は、メモリリークを回避するために、Component.destroy() を使用して手動で廃棄する必要があります。

サーバとの往復の回避

`createComponent()` メソッドおよび `createComponents()` メソッドでは、クライアント側のコンポーネントの作成とサーバ側のコンポーネントの作成の両方がサポートされています。パフォーマンス上やその他の理由により、クライアント側の作成をお勧めします。サーバ側の連動関係が見つからない場合、メソッドはクライアント側で実行されます。コンポーネントの作成にサーバ要求が必要かどうかは、最上位コンポーネントで判別されます。

このフレームワークでは、マークアップで定義されたコンポーネントなどの定義間の連動関係が自動的に追跡されます。これらの連動関係は、コンポーネントと共に読み込まれます。ただし、フレームワークで簡単に検出できない連動関係もあります。たとえば、コンポーネントのマークアップで直接参照されていないコンポーネントを動的に作成する場合などがこれに該当します。こうした動的な連動関係をフレームワークが把握できるようにするには、`<aura:dependency>` タグを使用します。この宣言により、コンポーネントとその*連動関係*がクライアントに送信されます。

サーバ側の連動関係があるコンポーネントは、そのサーバで作成する必要があります。サーバ側の連動関係には、動的に作成されたコンポーネント定義、動的に読み込まれた表示ラベル、および静的マークアップ分析によって事前に定義できる他の要素が含まれます。

メモ: コントローラアクションがコールされるのはコンポーネントが作成された後のみであるため、コンポーネントの作成では、サーバ側のコントローラはサーバ側の連動関係にはなりません。

`createComponent()` または `createComponents()` への単一のコールによって、多数のコンポーネントが作成される可能性があります。このコールでは、要求されたコンポーネントとそのすべての子コンポーネントが作成されます。パフォーマンス上の考慮事項に加えて、サーバ側のコンポーネント作成には、1 つの要求で作成可能なコンポーネント数が 10,000 個という制限があります。この制限に達する場合は、代わりにクライアント側でコンポーネントが作成されるように、`<aura:dependency>` タグでコンポーネントの連動関係を明示的に宣言するか、連動要素を事前に読み込みます。

クライアント側でのコンポーネント作成には制限はありません。

メモ: 最上位コンポーネントにサーバ側の連動関係はないが、ネストされた内部コンポーネントに連動関係があるコンポーネントの作成は、現在サポートされていません。

関連トピック:

リファレンスドキュメントアプリケーション
aura:dependency
コンポーネントの初期化時のアクションの呼び出し
コンポーネントへのイベントハンドラの動的な追加

変更ハンドラを使用したデータ変更の検出

コンポーネントのいずれかの属性の値が変更されたときに、変更ハンドラを自動的に呼び出す (クライアント側コントローラのアクション) ようにコンポーネントを設定します。

値が変更されると、`valueChange.evt` イベントが自動的に起動します。イベントには `type="VALUE"` が設定されています。

コンポーネントで、name="change" のあるハンドラを定義します。

```
<aura:handler name="change" value="{!v.numItems}" action="{!c.itemsChange}"/>
```

value 属性は、変更ハンドラが追跡するコンポーネント属性を設定します。

action 属性は、属性値が変更されたときに呼び出すクライアント側コントローラのアクションを設定します。

コンポーネントに複数の <aura:handler name="change"> タグを設定して、さまざまな属性の変更を検出できます。

コントローラで、ハンドラのアクションを定義します。

```
({
    itemsChange: function(cmp, evt) {
        console.log("numItems has changed");
        console.log("old value: " + evt.getParam("oldValue"));
        console.log("current value: " + evt.getParam("value"));
    }
})
```

valueChange イベントは、ハンドラのアクションで以前の値 (oldValue) と現在の値 (value) にアクセスできるようにします。

change ハンドラで表されている値が変更された場合、フレームワークによってイベントの起動とコンポーネントの再表示が処理されます。

関連トピック:

コンポーネントの初期化時のアクションの呼び出し

aura:valueChange

ID によるコンポーネントの検索

JavaScript コードに ID を使用してコンポーネントを取得します。

aura:id を使用して lightning:button コンポーネントに button1 というローカル ID を追加します。

```
<lightning:button aura:id="button1" label="button1"/>
```

cmp.find("button1") をコールすれば、このコンポーネントを検索できます。この cmp は、ボタンを含むコンポーネントへの参照です。find() 関数には、1 つのパラメータがあり、それはマークアップ内のコンポーネントのローカル ID です。

find() は、結果によって異なる種別を返します。

- ローカル ID が一意である場合、find() はコンポーネントを返します。
- 同じローカル ID のコンポーネントが複数ある場合、find() はコンポーネントの配列を返します。

- 一致するローカル ID がない場合、`find()` は `undefined` を返します。

関連トピック:

コンポーネントの ID
値プロバイダ

コンポーネントへのイベントハンドラの動的な追加

コンポーネントから起動されるイベントのハンドラを動的に追加できます。

`Component` コンポーネントの `addEventHandler()` メソッドは、廃止された `addHandler()` メソッドの代わりに使用されます。

コンポーネントにイベントハンドラを動的に追加するには、`addEventHandler()` メソッドを使用します。

```
addEventHandler(String event, Function handler, String phase, String includeFacets)
```

event

最初の引数は、ハンドラをトリガするイベントの名前です。コンポーネントから起動しないイベントの起動開始をコンポーネントに強制することはできないため、コンポーネントから起動するイベントに、この引数が対応することを確認してください。コンポーネントのマークアップの `<aura:registerEvent>` タグにより、コンポーネントから起動するイベントが公開されます。

- コンポーネントイベントの場合、`<aura:registerEvent>` タグの `name` 属性と一致するようにこの引数を設定します。
- アプリケーションイベントの場合、`namespace:eventName` 形式のイベント記述子と一致するようにこの引数を設定します。

handler

2 つ目の引数は、イベントを処理するアクションです。ハンドラをマークアップで静的に定義した場合、`<aura:handler>` タグの `action` 属性に指定する値と似ています。この引数には 2 つのオプションがあります。

- コントローラアクションを使用するには、`cmp.getReference("c.actionName")` 形式を使用します。
- 匿名関数を使用するには、次の形式を使用します。

```
function(auraEvent) {
    // handling logic here
}
```

その他の引数の説明については、ドキュメント参照アプリケーションの JavaScript API を参照してください。

`$A.createComponent()` のコールバック関数で動的に作成されたコンポーネントにイベントハンドラを追加することもできます。詳細は、「コンポーネントの動的な作成」を参照してください。

例

次のコンポーネントには、コンポーネントイベントとアプリケーションイベントを起動および処理するボタンがあります。

```
<!--c:dynamicHandler-->
<aura:component >
    <aura:registerEvent name="compEvent" type="c:sampleEvent"/>
    <aura:registerEvent name="appEvent" type="c:appEvent"/>
    <h1>Add dynamic handler for event</h1>
    <p>
        <lightning:button label="Fire component event" onclick="{!c.fireEvent}" />
        <lightning:button label="Add dynamic event handler for component event"
onclick="{!c.addEventHandler}" />
    </p>
    <p>
        <lightning:button label="Fire application event" onclick="{!c.fireAppEvent}" />
        <lightning:button label="Add dynamic event handler for application event"
onclick="{!c.addAppEventHandler}" />
    </p>

</aura:component>
```

クライアント側コントローラを次に示します。

```
/* dynamicHandlerController.js */
({
    fireEvent : function(cmp, event) {
        // Get the component event by using the
        // name value from <aura:registerEvent> tag
        var compEvent = cmp.getEvent("compEvent");
        compEvent.fire();
        console.log("Fired a component event");
    },

    addEventHandler : function(cmp, event) {
        // First param matches name attribute in <aura:registerEvent> tag
        cmp.addEventHandler("compEvent", cmp.getReference("c.handleEvent"));
        console.log("Added handler for component event");
    },

    handleEvent : function(cmp, event) {
        alert("Handled the component event");
    },

    fireAppEvent : function(cmp, event) {
        var appEvent = $A.get("e.c:appEvent");
        appEvent.fire();
        console.log("Fired an application event");
    },

    addAppEventHandler : function(cmp, event) {
        // Can use cmp.getReference() or anonymous function for handler
        // First param is event descriptor, "c:appEvent", for application events
        cmp.addEventHandler("c:appEvent", cmp.getReference("c.handleAppEvent"));
```

```
        // Can alternatively use anonymous function for handler
        //cmp.addEventHandler("c:appEvent", function(auraEvent) {
            // console.log("Handled the application event in anonymous function");
        //});
        console.log("Added handler for application event");
    },

    handleAppEvent : function(cmp, event) {
        alert("Handled the application event");
    }
})
```

`addEventHandler()` コールの最初のパラメータに注目します。コンポーネントイベントの構文は次のようになっています。

```
cmp.addEventHandler("compEvent", cmp.getReference("c.handleEvent"));
```

アプリケーションイベントの構文は次のようになっています。

```
cmp.addEventHandler("c:appEvent", cmp.getReference("c.handleAppEvent"));
```

コンポーネントイベントまたはアプリケーションイベントでは、コントローラアクションの `cmp.getReference()` を使用する代わりに、匿名関数をハンドラとして使用できます。

たとえば、アプリケーションイベントハンドラは次のようになります。

```
cmp.addEventHandler("c:appEvent", function(auraEvent) {
    // add handler logic here
    console.log("Handled the application event in anonymous function");
});
```

関連トピック:

- クライアント側コントローラを使用したイベントの処理
- コンポーネントイベントの処理
- リファレンスドキュメントアプリケーション

マークアップの動的な表示または非表示

マークアップの表示の切り替えに CSS を使用できます。ただし、`<aura:if>` は必要になるまで囲まれた要素ツリーを保留するため、この使用をお勧めします。

`<aura:if>` の使用例については、「条件付きマークアップのベストプラクティス」を参照してください。

次の例では、`$A.util.toggleClass(cmp, 'class')` を使用してマークアップの表示を切り替えます。

```
<!--c:toggleCss-->
<aura:component>
    <lightning:button label="Toggle" onclick="{!c.toggle}"/>
    <p aura:id="text">Now you see me</p>
</aura:component>
```

```
/*toggleCssController.js*/
({
```

```
    toggle : function(component, event, helper) {
        var toggleText = component.find("text");
        $A.util.toggleClass(toggleText, "toggle");
    }
})
```

```
/*toggleCss.css*/
.THIS.toggle {
    display: none;
}
```

[切り替え] ボタンをクリックすると、CSS クラスが切り替えられ、テキストが表示または非表示になります。

関連トピック:

クライアント側コントローラを使用したイベントの処理

コンポーネントの属性

スタイルの追加と削除

スタイルの追加と削除

実行時にコンポーネントまたは要素の CSS スタイルを追加または削除できます。

コンポーネントのクラス名を取得するには、`component.find('myCmp').get('v.class')` (`myCmp` は `aura:id` 属性値) を使用します。

コンポーネントまたは要素の CSS クラスを追加または削除するには、`$A.util.addClass(cmpTarget, 'class')` および `$A.util.removeClass(cmpTarget, 'class')` メソッドを使用します。

コンポーネントのソース

```
<aura:component>
    <div aura:id="changeIt">Change Me!</div><br />
    <lightning:button onclick="{!c.applyCSS}" label="Add Style" />
    <lightning:button onclick="{!c.removeCSS}" label="Remove Style" />
</aura:component>
```

CSS ソース

```
.THIS.changeMe {
    background-color:yellow;
    width:200px;
}
```

クライアント側コントローラのソース

```
{
    applyCSS: function(cmp, event) {
        var cmpTarget = cmp.find('changeIt');
        $A.util.addClass(cmpTarget, 'changeMe');
    },

    removeCSS: function(cmp, event) {
        var cmpTarget = cmp.find('changeIt');
```

```
        $A.util.removeClass(cmpTarget, 'changeMe');
    }
}
```

このデモのボタンは、CSS スタイルを追加または削除するコントローラアクションに結び付けられています。CSS スタイルをコンポーネントに追加するには、`$A.util.addClass(cmpTarget, 'class')` を使用します。同様に、クラスを削除するには、コントローラで `$A.util.removeClass(cmpTarget, 'class')` を使用します。`cmp.find()` でローカル ID (このデモでは `aura:id="changeIt"`) を使用してコンポーネントを特定します。

クラスの切り替え

クラスを切り替えるには、クラスを追加または削除する `$A.util.toggleClass(cmp, 'class')` を使用します。

`cmp` パラメータは、コンポーネントまたは DOM 要素の場合があります。

メモ: DOM 要素ではなくコンポーネントを使用することをお勧めします。`afterRender()` または `rerender()` 内でユーティリティ関数が使用されていない場合に、`cmp.getElement()` を渡すと、コンポーネントの表示時にクラスが適用されないことがあります。詳細は、「表示ライフサイクル中に起動されたイベント 」 (ページ 217)を参照してください。

マークアップを動的に表示または非表示にする場合は、「マークアップの動的な表示または非表示」 (ページ 313)を参照してください。

コンポーネントの配列のクラスを条件に応じて設定するには、配列を `$A.util.toggleClass()` に渡します。

```
mapClasses: function(arr, cssClass) {
    for(var cmp in arr) {
        $A.util.toggleClass(arr[cmp], cssClass);
    }
}
```

関連トピック:

クライアント側コントローラを使用したイベントの処理
コンポーネント内の CSS
コンポーネントのバンドル

押下されたボタンの確認

複数のボタンがあるコンポーネントで押下されたボタンを確認するには、`Component.getLocalId()` を使用します。

このフレームワークは、`ui:button` と `lightning:button`の 2 つのボタンコンポーネントを提供します。

メモ: Lightning Design System のスタイル設定に付属するボタンコンポーネント、`lightning:button` を使用することをおすすめします。

それでは、複数の ui:button コンポーネントの例を見てみましょう。各ボタンには、aura:id 属性によって設定された一意のローカル ID があります。

```
<!--c:buttonPressed-->
<aura:component>
    <aura:attribute name="whichButton" type="String" />

    <p>You clicked: {!v.whichButton}</p>

    <ui:button aura:id="button1" label="Click me" press="{!c.nameThatButton}"/>
    <ui:button aura:id="button2" label="Click me too" press="{!c.nameThatButton}"/>
</aura:component>
```

クライアント側コントローラで event.getSource() を使用して、クリックされたボタンコンポーネントを取得します。getLocalId() をコールして、クリックされたボタンの aura:id を取得します。

```
/* buttonPressedController.js */
({
    nameThatButton : function(cmp, event, helper) {
        var whichOne = event.getSource().getLocalId();
        console.log(whichOne);
        cmp.set("v.whichButton", whichOne);
    }
})
```

lightning:button を使用している場合は、press イベントハンドラではなく onclick イベントハンドラを使用します。

```
<aura:component>
    <aura:attribute name="whichButton" type="String" />

    <p>You clicked: {!v.whichButton}</p>

    <lightning:button aura:id="button1" name="buttonname1" label="Click me"
onclick="{!c.nameThatButton}"/>
    <lightning:button aura:id="button2" name="buttonname2" label="Click me"
onclick="{!c.nameThatButton}"/>
</aura:component>
```

クライアント側コントローラでは、次のいずれかのメソッドを使用して、どのボタンがクリックされたかを明らかにできます。

- event.getSource().getLocalId() はクリックされたボタンの aura:id を返します。
- event.getSource().get("v.name") はクリックされたボタンの name を返します。

関連トピック:

コンポーネントの ID
ID によるコンポーネントの検索

JavaScript での日付の書式設定

`AuraLocalizationService` JavaScript API では、日付の書式設定とローカライズを処理するメソッドを使用できます。

たとえば、`formatDate()` メソッドは、2 つ目の引数として設定された `formatString` パラメータに基づいて日付を書式設定します。

```
formatDate (String | Number | Date date, String formatString)
```

`date` パラメータには、String、Number、または最も一般的な JavaScript の Date を使用できます。String 値を指定する場合、解析エラーを避けるために ISO 8601 形式を使用します。

`formatString` パラメータには、日時を書式設定するトークンが含まれます。たとえば、`"YYYY-MM-DD"` では `15th January, 2017` が `"2017-01-15"` と書式設定されます。デフォルトの形式文字列は、`$Locale` 値プロバイダから提供されます。

次の表に、`formatString` でサポートされているトークンのリストを示します。

説明	トークン	出力
日付	d	1 ... 31
月	M	1 ... 12
月 (短縮名)	MMM	Jan ... Dec
月 (完全名)	MMMM	January ... December
年	y	2017
年 (y と同一)	Y	2017
年 (2 桁)	YY	17
年 (4 桁)	YYYY	2017
時 (1 〜 12)	h	1 ... 12
時 (0 〜 23)	H	0 ... 23
時 (1 〜 24)	k	1 ... 24
分	m	0 ... 59
秒	s	0 ... 59
ミリ秒	SSS	000 ... 999
AM または PM	a	AM または PM
AM または PM (a と同一)	A	AM または PM
UTC からのゾーンオフセット	Z	-12:00 ... +14:00
四半期	Q	1 ... 4

説明	トークン	出力
年の通算週	w	1 ... 53
年の通算週 (ISO)	W	1 ... 53

デフォルトの出力値が異なる類似したメソッドもあります。

- `formatDateTime()` — デフォルトの formatString は、日付の代わりに日時を出力します。
- `formatDateTimeUTC()` — 日時を UTC 標準時で書式設定します。
- `formatDateUTC()` — 日付を UTC 標準時で書式設定します。

`AuraLocalizationService` のすべてのメソッドについての詳細は、「リファレンスドキュメントアプリケーション」の JavaScript API を参照してください。

例: `$A.localizationService` を使用して、`AuraLocalizationService` のメソッドを使用します。

```
var now = new Date();
var dateString = "2017-01-15";

// Returns date in the format "Jun 8, 2017"
console.log($A.localizationService.formatDate(now));

// Returns date in the format "Jan 15, 2017"
console.log($A.localizationService.formatDate(dateString));

// Returns date in the format "2017 01 15"
console.log($A.localizationService.formatDate(dateString, "YYYY MM DD"));

// Returns date in the format "June 08 2017, 01:45:49 PM"
console.log($A.localizationService.formatDate(now, "MMMM DD YYYY, hh:mm:ss a"));

// Returns date in the format "Jun 08 2017, 01:48:26 PM"
console.log($A.localizationService.formatDate(now, "MMM DD YYYY, hh:mm:ss a"));
```

関連トピック:

ローカライズ

Apex の使用

Apex を使用して、コントローラやテストクラスなどのサーバ側コードを作成します。

サーバ側コントローラは、クライアント側コントローラからの要求を処理します。 たとえば、クライアント側コントローラでイベントを処理し、サーバ側コントローラアクションをコールしてレコードを保持する場合などがあります。サーバ側コントローラでは、レコードデータを読み込むこともできます。

このセクションの内容:

コントローラのサーバ側ロジックの作成

フレームワークは、クライアント側コントローラとサーバ側コントローラをサポートします。イベントは常にクライアント側コントローラのアクションに結び付けられ、このアクションがサーバ側コントローラのアクションをコールします。たとえば、クライアント側コントローラでイベントを処理し、サーバ側コントローラアクションをコールしてレコードを保持する場合などがあります。

Salesforce レコードの操作

Apex では、Salesforce レコードを簡単に操作できます。

Apex コードのテスト

管理パッケージをアップロードする前に、Apex コードのテストを作成および実行して、最小コードカバー率要件を満たす必要があります。また、パッケージを AppExchange にアップロードするときには、すべてのテストがエラーなしで実行される必要があります。

Apex からの API コールの実行

Apex コントローラから API コールを行います。JavaScript コードから Salesforce API コールを行うことはできません。

Apex でのコンポーネントの作成

サーバ側で Apex の `Cmp.<myNamespace>.<myComponent>` 構文を使用してコンポーネントを作成することができなくなりました。代わりに、クライアント側の JavaScript コードで `$A.createComponent()` を使用してください。

コントローラのサーバ側ロジックの作成

フレームワークは、クライアント側コントローラとサーバ側コントローラをサポートします。イベントは常にクライアント側コントローラのアクションに結び付けられ、このアクションがサーバ側コントローラのアクションをコールします。たとえば、クライアント側コントローラでイベントを処理し、サーバ側コントローラアクションをコールしてレコードを保持する場合などがあります。

サーバ側のアクションは、クライアントからサーバ、その後サーバからクライアントに往復させる必要があるため、通常はクライアント側のアクションよりも完了に時間がかかります。

サーバ側のアクションをコールするプロセスについての詳細は、「サーバ側のアクションのコール」(ページ 325)を参照してください。

このセクションの内容:

Apex サーバ側コントローラの概要

サーバ側コントローラを Apex で作成し、`@AuraEnabled` アノテーションを使用して、コントローラメソッドにアクセスできるようにします。

Apex サーバ側コントローラの作成

開発者コンソールを使用して、Apex サーバ側コントローラを作成します。

Apex サーバ側コントローラからデータを返す

`return` ステートメントを使用して、サーバ側コントローラからクライアント側コントローラに結果を返します。結果データは、JSON 形式に逐次化できる必要があります。

Apex サーバ側コントローラからエラーを返す

サーバ側コントローラから `System.AuraHandledException` を作成してスローし、カスタムエラーメッセージを返します。

AuraEnabled アノテーション

`AuraEnabled` アノテーションは、Lightning コンポーネントフレームワークで使用する Apex メソッドとプロパティをサポートします。

サーバ側のアクションのコール

クライアント側コントローラからサーバ側コントローラのアクションをコールします。クライアント側コントローラにコールバックを設定し、サーバ側のアクションが完了したときにコールされるようにします。サーバ側のアクションは、逐次化可能な JSON データを含む任意のオブジェクトを返すことができます。

サーバ側のアクションのキュー配置

フレームワークは、アクションをサーバに送信する前にキューに配置します。コードの記述時のこのメカニズムの大半は透過的ですが、複数のアクションを 1 つの要求 (XHR) にまとめて、フレームワークがネットワークトラフィックを最小限に抑えることができます。

フォアグラウンドアクションおよびバックグラウンドアクション

フォアグラウンドアクションがデフォルトです。アクションをバックグラウンドアクションとしてマークできます。ユーザに対するアプリケーションの応答性を維持しながら、優先度が低く実行時間が長いアクションをアプリケーションで実行する場合、これが役立ちます。大まかなガイドラインとして、応答がサーバから戻るまでに 5 秒以上かかる場合は、バックグラウンドアクションを使用します。

保存可能なアクション

アクションを保存可能としてマークすると、サーバとの往復を待たずにクライアント側ストレージのキャッシュデータをすばやく表示できるようになり、コンポーネントのパフォーマンスが向上します。キャッシュデータが古くなっている場合、フレームワークによってサーバから最新データが取得されます。特に、待ち時間の長い接続、低速の接続、信頼性の低い接続(3G ネットワークなど)のユーザの場合には、キャッシュが役立ちます。

中止可能なアクション

アクションを中止可能とマークして、サーバへの送信キューに入っているときに中止可能にすることができます。キュー内の中止可能なアクションは、そのアクションを作成したコンポーネントが無効になった場合 (`cmp.isValid() == false`)、サーバに送信されません。コンポーネントは自動的に破棄され、表示されないときにフレームワークによって無効とマークされます。

Apex サーバ側コントローラの概要

サーバ側コントローラを Apex で作成し、`@AuraEnabled` アノテーションを使用して、コントローラメソッドにアクセスできるようにします。

`@AuraEnabled` を使用して明示的にアノテーションを付加したメソッドのみが公開されます。サーバ側アクションのコールは、組織の API 制限に対してカウントされません。ただし、サーバ側コントローラアクションは Apex で記述されているため、Apex の通常の制限がすべて適用されます。

次の Apex コントローラには、渡される値の先頭に文字列を付加する `serverEcho` アクションが含まれます。

```
public with sharing class SimpleServerSideController {
```

```
    //Use @AuraEnabled to enable client- and server-side access to the method
    @AuraEnabled
    public static String serverEcho(String firstName) {
        return ('Hello from the server, ' + firstName);
    }
}
```

@AuraEnabled アノテーションを使用することに加えて、Apex コントローラは次の要件も満たす必要があります。

- メソッドは static で、かつ public または global とマークされている必要があります。非静的メソッドはサポートされていません。
- メソッドがオブジェクトを返す場合、オブジェクトのインスタンス項目の値を取得するインスタンスメソッドは public である必要があります。
- コンポーネント内のクライアント側アクションとサーバ側アクションには一意の名前を使用します。JavaScript 関数 (クライアント側アクション) と Apex メソッド (サーバ側アクション) が同じ名前だと、問題が発生したときにデバッグしにくくなるおそれがあります。デバッグモードでは、フレームワークによって、クライアント側アクション名とサーバ側アクション名の競合に関するブラウザコンソールの警告が記録されます。

ヒント: コントローラ (クライアント側またはサーバ側) にコンポーネントの状態を保存しないでください。代わりにコンポーネントのクライアント側属性で状態を保存します。

詳細は、*『Apex 開発者ガイド』*の「クラス」を参照してください。

関連トピック:

- サーバ側のアクションのコール
- Apex サーバ側コントローラの作成
- AuraEnabled アノテーション

Apex サーバ側コントローラの作成

開発者コンソールを使用して、Apex サーバ側コントローラを作成します。

1. 開発者コンソールを開きます。
2. **[File (ファイル)]** > **[New (新規)]** > **[Apex Class (Apex クラス)]** をクリックします。
3. サーバ側コントローラの名前を入力します。
4. **[OK]** をクリックします。
5. クラスのボディにサーバ側の各アクションのメソッドを入力します。

   @AuraEnabled アノテーションをメソッドに追加して、メソッドをサーバ側アクションとして公開します。また、サーバ側アクションは static メソッドで、global または public でなければなりません。
6. **[File (ファイル)]** > **[保存]** をクリックします。
7. 新しいコントローラクラスに結び付けるコンポーネントを開きます。

8. `controller` システム属性を `<aura:component>` タグに追加して、コンポーネントをコントローラに結び付けます。次に例を示します。

```
<aura:component controller="SimpleServerSideController">
```

関連トピック:

Salesforce ヘルプ: 開発者コンソールを開く

Apex サーバ側コントローラからデータを返す

AuraEnabled アノテーション

Apex サーバ側コントローラからデータを返す

`return` ステートメントを使用して、サーバ側コントローラからクライアント側コントローラに結果を返します。結果データは、JSON 形式に逐次化できる必要があります。

返されるデータ型には次のいずれかを使用できます。

- 単純 — String、Integer など。詳細は、「基本の型」を参照してください。
- sObject — 標準およびカスタムの sObjects がサポートされます。「標準オブジェクト型とカスタムオブジェクト型」を参照してください。
- Apex — Apex クラスのインスタンス (多くの場合、カスタムクラス)。「カスタム Apex クラス型」を参照してください。
- コレクション — 他のいずれかの型のコレクション。「コレクション型」を参照してください。

Apex オブジェクトを返す

次に、カスタム Apex オブジェクトのコレクションを返すコントローラの例を示します。

```
public with sharing class SimpleAccountController {

    @AuraEnabled
    public static List<SimpleAccount> getAccounts() {

        // Perform isAccessible() check here

        // SimpleAccount is a simple "wrapper" Apex class for transport
        List<SimpleAccount> simpleAccounts = new List<SimpleAccount>();

        List<Account> accounts = [SELECT Id, Name, Phone FROM Account LIMIT 5];
        for (Account acct : accounts) {
            simpleAccounts.add(new SimpleAccount(acct.Id, acct.Name, acct.Phone));
        }

        return simpleAccounts;
    }
}
```

サーバ側アクションから Apex クラスのインスタンスが返されると、インスタンスはフレームワークによって JSON に逐次化されます。`@AuraEnabled` でアノテーションされた `public` インスタンスのプロパティとメソッドの値のみが逐次化されて返されます。

たとえば、取引先レコードのいくつかの詳細を含む簡単な「ラッパー」Apex クラスを次に示します。このクラスを使用して、取引先レコードのいくつかの詳細を逐次化可能な形式にパッケージ化します。

```
public class SimpleAccount {

    @AuraEnabled public String Id { get; set; }
    @AuraEnabled public String Name { get; set; }
    public String Phone { get; set; }

    // Trivial constructor, for server-side Apex -> client-side JavaScript
    public SimpleAccount(String id, String name, String phone) {
        this.Id = id;
        this.Name = name;
        this.Phone = phone;
    }

    // Default, no-arg constructor, for client-side -> server-side
    public SimpleAccount() {}

}
```

リモート Apex コントローラアクションから返されたときに、Id および Name プロパティがクライアント側で定義されます。ただし、`@AuraEnabled` アノテーションがないため、Phone プロパティはサーバ側で逐次化されず、結果データの一部として返されません。

関連トピック:

AuraEnabled アノテーション
カスタム Apex クラス型

Apex サーバ側コントローラからエラーを返す

サーバ側コントローラから `System.AuraHandledException` を作成してスローし、カスタムエラーメッセージを返します。

エラーの発生は避けられません。ユーザによる無効な入力、データベース内の重複レコードなど、一部のエラーは予期できます。一方、予期しないエラーが発生する場合もあります。少しでもプログラミングの経験があれば、予期せぬエラーの範囲はほぼ無限であることをご存じでしょう。

サーバ側コントローラのコードでエラーが発生した場合、2 つの対処方法が考えられます。そこでエラーをキャッチして、Apex でエラーを処理できます。それ以外の場合、エラーはコントローラの応答で戻されます。

Apex でエラーを処理するときにも、2 つの対処方法が考えられます。エラーを処理し、可能であれば復旧し、通常の応答をクライアントに返すことができます。または、`AuraHandledException` を作成してスローできます。

システム例外を返す代わりに `AuraHandledException` をスローする利点は、クライアントのコードで例外をより適切に処理できるという点です。システム例外はセキュリティ上の理由により重要な詳細情報が取り除かれるため、不安を煽る「内部サーバエラーが発生しました...」というメッセージになります。このメッセージ

は誰も好みません。AuraHandledException を使用すると、クライアント側のコードに返される応答にいくつかの詳細情報を追加できます。さらに良いことには、より適切なメッセージをユーザに表示できます。

不正入力に対する応答で AuraHandledException を作成してスローする例を次に示します。ただし、AuraHandledException を使用する真価は、システム例外への応答で使用するときに発揮されます。たとえば、DML 例外のキャッチへの応答では、クライアントコンポーネントのコードにその例外を伝搬する代わりに AuraHandledException をスローします。

```
public with sharing class SimpleErrorController {

    static final List<String> BAD_WORDS = new List<String> {
        'bad',
        'words',
        'here'
    };

    @AuraEnabled
    public static String helloOrThrowAnError(String name) {

        // Make sure we're not seeing something naughty
        for(String badWordStem : BAD_WORDS) {
            if(name.containsIgnoreCase(badWordStem)) {
                // How rude! Gracefully return an error...
                throw new AuraHandledException('NSFW name detected.');
            }
        }

        // No bad word found, so...
        return ('Hello ' + name + '!');
    }

}
```

AuraEnabled アノテーション

AuraEnabled アノテーションは、Lightning コンポーネントフレームワークで使用する Apex メソッドとプロパティをサポートします。

AuraEnabled アノテーションはオーバーロードされ、2 つの別々の異なる目的で使用されます。

- Apex クラスの静的メソッドに対して @AuraEnabled を使用すると、その静的メソッドをリモートコントローラアクションとして Lightning コンポーネントで使用できます。
- Apex インスタンスのメソッドおよびプロパティに対して @AuraEnabled を使用すると、クラスのインスタンスがサーバ側アクションからデータとして返されたときにそれらのメソッドおよびプロパティを逐次化できます。

重要:

- @AuraEnabled のこの異なる使用法を同じ Apex クラス内で組み合わせないでください。

- 静的 @AuraEnabled Apex メソッドのみをクライアント側コードからコールできます。Visualforce スタイルのインスタンスのプロパティと getter/setter メソッドは使用できません。代わりに、クライアント側コンポーネントの属性を使用します。

関連トピック:

Apex サーバ側コントローラからデータを返す
カスタム Apex クラス型

サーバ側のアクションのコール

クライアント側コントローラからサーバ側コントローラのアクションをコールします。クライアント側コントローラにコールバックを設定し、サーバ側のアクションが完了したときにコールされるようにします。サーバ側のアクションは、逐次化可能な JSON データを含む任意のオブジェクトを返すことができます。

クライアント側コントローラは、名前-値のペアの対応付けを含む、オブジェクトリテラル表記の JavaScript オブジェクトです。

コンポーネントからサーバコールをトリガするとします。次のコンポーネントには、クライアント側コントローラの echo アクションに接続されるボタンが含まれます。SimpleServerSideController には、クライアント側コントローラから渡される文字列を返すメソッドが含まれます。

```
<aura:component controller="SimpleServerSideController">
    <aura:attribute name="firstName" type="String" default="world"/>
    <lightning:button label="Call server" onclick="{!c.echo}"/>
</aura:component>
```

このクライアント側コントローラには、サーバ側コントローラで serverEcho メソッドを実行する echo アクションが含まれます。

ヒント: コンポーネント内のクライアント側アクションとサーバ側アクションには一意の名前を使用します。JavaScript 関数 (クライアント側アクション) と Apexメソッド (サーバ側アクション) が同じ名前だと、問題が発生したときにデバッグしにくくなるおそれがあります。デバッグモードでは、フレームワークによって、クライアント側アクション名とサーバ側アクション名の競合に関するブラウザコンソールの警告が記録されます。

```
({
    "echo" : function(cmp) {
        // create a one-time use instance of the serverEcho action
        // in the server-side controller
        var action = cmp.get("c.serverEcho");
        action.setParams({ firstName : cmp.get("v.firstName") });

        // Create a callback that is executed after
        // the server-side action returns
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                // Alert the user with the value returned
                // from the server
                alert("From server: " + response.getReturnValue());
```

```
                // You would typically fire a event here to trigger
                // client-side notification that the server-side
                // action is complete
            }
            else if (state === "INCOMPLETE") {
                // do something
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });

        // optionally set storable, abortable, background flag here

        // A client-side action could cause multiple events,
        // which could trigger other events and
        // other server-side action calls.
        // $A.enqueueAction adds the server-side action to the queue.
        $A.enqueueAction(action);
    }
})
```

クライアント側コントローラでは、c の値プロバイダを使用してサーバ側コントローラのアクションを呼び出します。また、マークアップでこの c 構文を使用して、クライアント側コントローラのアクションも呼び出します。

`cmp.get("c.serverEcho")` コールは、サーバ側コントローラで `serverEcho` メソッドをコールしていることを示します。サーバ側コントローラのメソッド名は、クライアント側のコールの `c.` に続く内容と完全に一致している必要があります。この場合、それは `serverEcho` です。

`action.setParams()` を使用して、サーバ側コントローラに渡されるデータを設定します。次のコールは、`firstName` 属性値に基づいて、サーバ側コントローラの `serverEcho` メソッドで `firstName` 引数の値を設定します。

```
action.setParams({ firstName : cmp.get("v.firstName") });
```

`action.setCallback()` は、サーバ側のアクションが返されたら呼び出されるコールバックアクションを設定します。

```
action.setCallback(this, function(response) { ... });
```

サーバ側のアクションの結果は、コールバックの引数である `response` 変数に格納されます。

`response.getState()` は、サーバから返されたアクションの状態を取得します。

メモ: クライアント側コントローラに関連付けられたコンポーネントを参照する場合、クライアント側コントローラのコールバックで `cmp.isValid()` チェックは必要ありません。コンポーネントが無効であることがフレームワークによって自動的にチェックされます。

`response.getReturnValue()` は、サーバから返された値を取得します。この例では、コールバック関数がユーザにサーバから返された値を含むアラートを表示します。

`$A.enqueueAction(action)` により、サーバ側コントローラのアクションがアクション実行キューに追加されます。キューに追加されたアクションはすべて、イベントループの最後に実行されます。フレームワークでは、個々のアクションごとに個別の要求を送信するのではなく、イベントチェーンを処理し、キューのアクションを 1 つの要求にまとめます。これらのアクションは非同期で、コールバックが設定されます。

ヒント: アクションが実行されない場合、フレームワークの通常の表示ライフサイクル外のコードを実行していないことを確認してください。たとえば、イベントハンドラで `window.setTimeout()` を使用して一部のロジックを遅延実行する場合は、コードを `$A.getCallback()` でラップします。コードがフレームワークのコールスタックの一部として実行される場合は、`$A.getCallback()` を使用する必要はありません。たとえば、コードがイベントを処理している場合や、サーバ側のコントローラアクションのコールバックにある場合です。

クライアントペイロードデータ制限

サーバ側コントローラに渡すアクションのデータを設定するには、`action.setParams()` を使用します。

フレームワークでは、キュー内にあるアクションを1つのサーバ要求にまとめます。要求ペイロードには、すべてのアクションとそのデータが JSON として逐次化されます。要求ペイロードの制限は 4 MB です。

このセクションの内容:

アクションの状態
クライアント側コントローラからサーバ側コントローラのアクションをコールします。アクションは処理中にさまざまな状態になる可能性があります。

関連トピック:

クライアント側コントローラを使用したイベントの処理

サーバ側のアクションのキュー配置

アクションの状態

コンポーネントの有効性の確認

アクションの状態

クライアント側コントローラからサーバ側コントローラのアクションをコールします。アクションは処理中にさまざまな状態になる可能性があります。

アクションの有効な状態は次のとおりです。

NEW
アクションが作成されていますが、まだ処理されていません。

RUNNING
アクションを処理中です。

SUCCESS
アクションが正常に実行されました。

ERROR
サーバからエラーが返されました。

INCOMPLETE
サーバから応答が返されませんでした。サーバがダウンしているか、クライアントがオフラインである可能性があります。フレームワークは、コンポーネントが有効である限り、アクションのコールバックが常に呼び出されることを保証します。サーバへのソケットが一度も正常に開いていない場合や突然閉じた場合、または他のネットワークエラーが生じた場合は、XHR が解決され、状態が `INCOMPLETE` のコールバックが呼び出されます。

ABORTED
アクションが中止されました。このアクション状態は非推奨です。中止されたアククションのコールバックが実行されることはないため、この状態を処理する術はありません。

関連トピック:

サーバ側のアクションのコール

サーバ側のアクションのキュー配置

フレームワークは、アクションをサーバに送信する前にキューに配置します。コードの記述時のこのメカニズムの大半は透過的ですが、複数のアクションを 1 つの要求 (XHR) にまとめて、フレームワークがネットワークトラフィックを最小限に抑えることができます。

アクションの一括処理は、*ボックスカーイング(貨車の連結)*とも呼ばれ、貨車を連結する列車に似ています。

フレームワークはスタックを使用して、サーバに送信するアクションを追跡します。クライアントでブラウザがイベントと JavaScript の処理を終了すると、スタック上でキューに追加されたアクションが 1 つのバッチでサーバに送信されます。

ヒント: アクションが実行されない場合、フレームワークの通常の表示ライフサイクル外のコードを実行していないことを確認してください。たとえば、イベントハンドラで `window.setTimeout()` を使用して一部のロジックを遅延実行する場合は、コードを `$A.getCallback()` でラップします。

アクションがサーバへの送信を待機するキューにある間、フレームワークがそのアクションをどのように管理するかに影響を与えるプロパティをアクションに設定できます。詳細は、以下を参照してください。

- フォアグラウンドアクションおよびバックグラウンドアクション (ページ 329)
- 保存可能なアクション (ページ 330)
- 中止可能なアクション (ページ 334)

関連トピック:

フレームワークのライフサイクル外のコンポーネントの変更

フォアグラウンドアクションおよびバックグラウンドアクション

フォアグラウンドアクションがデフォルトです。アクションをバックグラウンドアクションとしてマークできます。ユーザに対するアプリケーションの応答性を維持しながら、優先度が低く実行時間が長いアクションをアプリケーションで実行する場合、これが役立ちます。大まかなガイドラインとして、応答がサーバから戻るまでに 5 秒以上かかる場合は、バックグラウンドアクションを使用します。

アクションの一括処理

キューに入れられた複数のフォアグラウンドアクションが 1 つの要求 (XHR) にまとめられ、ネットワークトラフィックが最小限に抑えられます。アクションの一括処理は、*ボックスカーイング(貨車の連結)*とも呼ばれ、貨車を連結する列車に似ています。

サーバですべてのアクションが処理されると、サーバはXHR応答をクライアントに送信します。実行時間の長いアクションがボックスカーの中にある場合、その実行時間の長いアクションが完了するまでXHR応答は保留されます。アクションをバックグラウンドとしてマークすると、そのアクションはフォアグラウンドアクションとは別に送信されます。このため、バックグラウンドアクションはフォアグラウンドアクションの応答時間に影響しません。

キュー内のサーバ側アクションが実行されると、まずフォアグラウンドアクションが実行されてから、バックグラウンドアクションが実行されます。バックグラウンドアクションはフォアグラウンドアクションと並行して実行され、フォアグラウンドアクションとバックグラウンドアクションの応答は順序に関係なく戻ることができます。

アクションのコールバックの実行順序を保証することはできません。サーバの処理時間のために、XHR要求の送信順序とは異なる順序で XHR 応答が返される場合があります。

メモ: 独自の要求内で送信された各バックグラウンドアクションに依存しないでください。その動作は保証されず、パフォーマンスの問題が生じる可能性があります。バックグラウンドアクションの目的は、実行時間の長い要求を別の要求に分離することで、フォアグラウンドアクションの応答が遅くなることを回避することです。

2 つのアクションを順次実行する必要がある場合、コンポーネントでその順序を統制する必要があります。コンポーネントで最初のアクションをキューに追加できます。最初のアクションのコールバックで、コンポーネントは次に 2 番目のアクションをキューに追加できます。

フレームワークで管理される要求の調整

フレームワークではフォアグランド要求とバックグラウンド要求が別々に調整されます。つまり、フレームワークでは、任意の時点で実行されるフォアグラウンド要求の数とバックグラウンドアクションの数を制御できます。フレームワークでは要求が自動的に調整され、ユーザが制御することはありません。フレームワークではフォアグラウンドとバックグラウンドのXHRの数が管理されますが、その数は使用可能なリソースによって異なります。

別々の調整が行われても、サーバへの過剰な数の要求など、条件によってはバックグラウンドアクションがパフォーマンスに影響する場合があります。

バックグラウンドアクションの設定

アクションをバックグラウンドアクションとしてマークするには、JavaScriptでアクションオブジェクトに対して setBackground() メソッドをコールします。

```
// set up the server-action action
var action = cmp.get("c.serverEcho");
// optionally set actions params
//action.setParams({ firstName : cmp.get("v.firstName") });
// set as a background action
action.setBackground();
```

メモ: バックグラウンドアクションをフォアグラウンドアクションに戻すことはできません。つまり、setBackground をコールして false を設定しても、それは無効です。

関連トピック:

サーバ側のアクションのキュー配置

サーバ側のアクションのコール

保存可能なアクション

アクションを保存可能としてマークすると、サーバとの往復を待たずにクライアント側ストレージのキャッシュデータをすばやく表示できるようになり、コンポーネントのパフォーマンスが向上します。キャッシュデータが古くなっている場合、フレームワークによってサーバから最新データが取得されます。特に、待ち時間の長い接続、低速の接続、信頼性の低い接続 (3G ネットワークなど) のユーザの場合には、キャッシュが役立ちます。

警告:

- 保存可能なアクションは、サーバにコールされない可能性があります。データを更新または削除するアクションは保存可能とマークしないでください。
- キャッシュに保存可能なアクションについては、フレームワークがキャッシュされた応答をただちに返し、データが古い場合には更新も行います。そのため、保存可能なアクションのコールバックが複数回呼び出されていることがあります。この場合、初回はキャッシュされたデータが使用され、それ以降はサーバからの更新済みデータが使用されます。

大部分のサーバ要求は参照のみで冪等です。つまり、データの変更が発生することなく、必要に応じて何度も要求の繰り返しまたは再試行を行うことができます。冪等なアクションに対する応答はキャッシュできるため、後続の同一アクションですばやく再利用できます。保存可能なアクションで同一アクションを判断する場合、次の組み合わせが重要になります。

- Apex コントローラ名
- メソッド名
- メソッドのパラメータ値

アクションを保存可能としてマーク

サーバ側のアクションを保存可能とマークするには、JavaScript コードで次のようにアクションの setStorable() をコールします。

```
action.setStorable();
```

メモ: 保存可能なアクションは、常に暗黙的に中止可能としてもマークされます。

setStorable 関数は、省略可能な引数を取ります。この引数は、ストレージオプションを表すキー - 値ペアと設定値の設定対応付けです。次のプロパティのみを設定できます。

ignoreExisting

キャッシュをスキップするには、true に設定します。デフォルト値は、false です。

このプロパティは、キャッシュデータが無効であるとわかっている場合 (レコードの変更後など) に便利です。ほとんどありませんが、明示的にキャッシュが無効になっているためにこのプロパティを使用することが必要な場合があります。

アクション応答のストレージオプションを設定するには、この設定対応付けを setStorable(***configObj***) に渡します。

このセクションの内容:

保存可能なアクションのライフサイクル

次の画像では、保存可能なアクションのコールバックの実行シーケンスについて説明します。

アプリケーション内での保存可能なアクションの有効化

Lightning Experience と Salesforce1 では、保存可能なアクションは自動的に設定されます。スタンドアロンアプリケーション (.app リソース) で保存可能なアクションを使用するには、キャッシュされたアクション応答のクライアント側ストレージを設定する必要があります。

ストレージサービスアダプタ

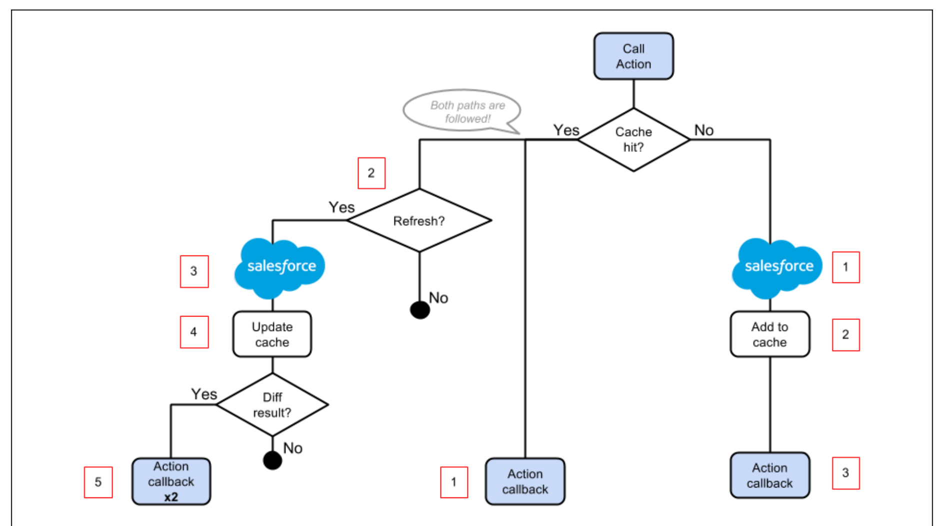
Storage Service は、ストレージの複数の実装をサポートし、ブラウザサポートと、永続性やセキュリティの指定された特性に基づいて、実行時に 1 つのアダプタを選択します。ストレージには、永続的なストレージおよびセキュアなストレージがあります。永続的なストレージでは、キャッシュされたデータはブラウザのユーザセッション間で保持されます。セキュアなストレージでは、キャッシュされたデータは暗号化されます。

保存可能なアクションのライフサイクル

次の画像では、保存可能なアクションのコールバックの実行シーケンスについて説明します。

メモ: アクションでコールバックが複数回呼び出されることがあります。

- 1 回目はキャッシュされた応答で呼び出されます (ストレージにある場合)。
- 2 回目はサーバの更新されたデータで呼び出されます (保存されている応答がエントリの更新時間を超えた場合)。



キャッシュの欠落

アクションがストレージエントリに一致せずに、キャッシュヒットでない場合、次の処理が行われます。

1. アクションがサーバ側コントローラに送信されます。
2. 応答が SUCCESS の場合、応答がストレージに追加されます。
3. クライアント側コントローラのコールバックが実行されます。

キャッシュヒット

アクションがストレージエントリに一致して、キャッシュヒットである場合、次の処理が行われます。

1. キャッシュされたアクション応答でクライアント側コントローラのコールバックが実行されます。
2. 更新時間よりも長く応答がキャッシュされると、ストレージエントリが更新されます。

   アプリケーションで保存可能なアクションが有効になると、更新時間が設定されます。更新時間は、ストレージのエントリが更新されるまでの期間 (秒数) です。Lightning Experience と Salesforce1 では、更新時間は自動的に設定されます。

3. アクションがサーバ側コントローラに送信されます。
4. 応答が SUCCESS の場合、応答がストレージに追加されます。
5. 更新された応答がキャッシュされた応答と異なる場合、クライアント側コントローラの2回目のコールバックが実行されます。

関連トピック:

保存可能なアクション

アプリケーション内での保存可能なアクションの有効化

アプリケーション内での保存可能なアクションの有効化

Lightning Experience と Salesforce1 では、保存可能なアクションは自動的に設定されます。スタンドアロンアプリケーション (.app リソース) で保存可能なアクションを使用するには、キャッシュされたアクション応答のクライアント側ストレージを設定する必要があります。

スタンドアロンアプリケーションのクライアント側ストレージを設定するには、アプリケーションのテンプレートの auraPreInitBlock 属性の <auraStorage:init> を使用します。この例を次に示します。

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="auraPreInitBlock">
        <auraStorage:init
          name="actions"
          persistent="false"
          secure="true"
          maxSize="1024"
          defaultExpiration="900"
          defaultAutoRefreshInterval="30" />
    </aura:set>
</aura:component>
```

name
ストレージ名は actions にする必要があります。現在サポートされているストレージ種別は、保存可能なアクションのみです。

persistent
ブラウザのユーザセッション間でキャッシュデータを保持するには、true に設定します。

secure
キャッシュデータを暗号化するには、true に設定します。

maxsize
ストレージの最大サイズ (KB)。

defaultExpiration
エントリがストレージに保持される期間 (秒数)。

defaultAutoRefreshInterval
ストレージのエントリが更新されるまでの期間 (秒数)。

詳細は、「リファレンスドキュメントアプリケーション」を参照してください。

保存可能なアクションは、Storage Service を使用します。Storage Service は、ストレージの複数の実装をサポートし、ブラウザサポートと、永続性やセキュリティの指定された特性に基づいて、実行時に 1 つのアダプタを選択します。

関連トピック:

ストレージサービスアダプタ

ストレージサービスアダプタ

Storage Service は、ストレージの複数の実装をサポートし、ブラウザサポートと、永続性やセキュリティの指定された特性に基づいて、実行時に 1 つのアダプタを選択します。ストレージには、永続的なストレージおよびセキュアなストレージがあります。永続的なストレージでは、キャッシュされたデータはブラウザのユーザセッション間で保持されます。セキュアなストレージでは、キャッシュされたデータは暗号化されます。

ストレージアダプタ名	永続的	セキュア
IndexedDB	`true`	`false`
Memory	`false`	`true`

IndexedDB

(永続的だがセキュアではない) クライアント側ストレージおよび構造化されたデータの検索のために API へのアクセスを提供します。詳細については、「Indexed Database API」を参照してください。

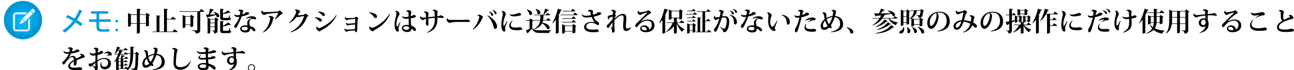
Memory

(永続的ではないがセキュア) データをキャッシュするために JavaScript メモリへのアクセスを提供します。保存されたキャッシュはブラウザページごとにのみ保持されます。新しいページに移動すると、キャッシュはリセットされます。

Storage Service では、サービスの初期化の際に指定した永続性とセキュリティのオプションに一致するストレージアダプタが選択されます。たとえば、永続的だがセキュアではないストレージサービスを要求すると、ブラウザでサポートされていれば Storage Service は IndexedDB ストレージを返します。

中止可能なアクション

アクションを中止可能とマークして、サーバへの送信キューに入っているときに中止可能にすることができます。キュー内の中止可能なアクションは、そのアクションを作成したコンポーネントが無効になった場合 (`cmp.isValid() == false`)、サーバに送信されません。コンポーネントは自動的に破棄され、表示されないときにフレームワークによって無効とマークされます。



メモ: 中止可能なアクションはサーバに送信される保証がないため、参照のみの操作にだけ使用することをお勧めします。

中止可能なアクションは、サーバにアクションが送信されるまでにそのアクションを作成したコンポーネントが無効にならなければ、通常どおりサーバに送信されて実行されます。

中止不能のアクションは常にサーバに送信され、キュー内で中止することはできません。

サーバからアクション応答が返され、この時点で関連付けられているコンポーネントが無効な場合、サーバでロジックは実行されていますが、アクションのコールバックは行われていません。これは、アクションが中止可能としてマークされているかどうかに関係なく true になります。

アクションを中止可能としてマーク

サーバ側アクションを中止可能とマークするには、JavaScript で Action オブジェクトに対して setAbortable() メソッドを使用します。次に例を示します。

```
var action = cmp.get("c.serverEcho");
action.setAbortable();
```

関連トピック:

コントローラのサーバ側ロジックの作成

サーバ側のアクションのキュー配置

サーバ側のアクションのコール

Salesforce レコードの操作

Apex では、Salesforce レコードを簡単に操作できます。

sObject という用語は、Force.com に保存可能なオブジェクトを意味します。これは、標準オブジェクト (Account など) でも、ユーザが作成するカスタムオブジェクト (Merchandise オブジェクトなど) でもかまいません。

sObject 変数は、1 行のデータを表し、レコードとも呼ばれます。Apex でオブジェクトを操作するには、オブジェクトの SOAP API 名を使用して宣言します。次に例を示します。

```
Account a = new Account();
MyCustomObject__c co = new MyCustomObject__c();
```

Apex でのレコードの操作についての詳細は、「Apex でのデータの操作」を参照してください。

次のコントローラ例では、更新された Account レコードを保持します。update メソッドには、サーバ側コントローラアクションとしてコールできるように @AuraEnabled アノテーションが付加されています。

```
public with sharing class AccountController {

    @AuraEnabled
    public static void updateAnnualRevenue(String accountId, Decimal annualRevenue) {
        Account acct = [SELECT Id, Name, BillingCity FROM Account WHERE Id = :accountId];

        acct.AnnualRevenue = annualRevenue;

        // Perform isAccessible() and isUpdateable() checks here
        update acct;
    }
}
```

JavaScript から Apex コードをコールする例については、「クイックスタート」 (ページ 8)を参照してください。

標準オブジェクトからのレコードデータの読み込み

サーバ側コントローラの標準オブジェクトからレコードを読み込みます。次のサーバ側コントローラには、商談レコードのリストと個々の商談レコードを返すメソッドがあります。

```
public with sharing class OpportunityController {

    @AuraEnabled
    public static List<Opportunity> getOpportunities() {
        List<Opportunity> opportunities =
                [SELECT Id, Name, CloseDate FROM Opportunity];
        return opportunities;
    }

    @AuraEnabled
    public static Opportunity getOpportunity(Id id) {
        Opportunity opportunity = [
                SELECT Id, Account.Name, Name, CloseDate,
                       Owner.Name, Amount, Description, StageName
            FROM Opportunity
            WHERE Id = :id
         ];

        // Perform isAccessible() check here
        return opportunity;
    }
}
```

次のコンポーネント例では、ボタンを押したときに上記のサーバ側コントローラを使用して商談レコードのリストを表示します。

```
<aura:component controller="OpportunityController">
    <aura:attribute name="opportunities" type="Opportunity[]"/>

    <ui:button label="Get Opportunities" press="{!c.getOpps}"/>
    <aura:iteration var="opportunity" items="{!v.opportunities}">
     <p>{!opportunity.Name} : {!opportunity.CloseDate}</p>
    </aura:iteration>
</aura:component>
```

ボタンを押すと、次のクライアント側コントローラで、サーバ側コントローラの getOpportunities() をコールして、コンポーネントの opportunities 属性を設定します。サーバ側コントローラメソッドのコール方法についての詳細は、「サーバ側のアクションのコール」(ページ 325) を参照してください。

```
({
    getOpps: function(cmp){
        var action = cmp.get("c.getOpportunities");
        action.setCallback(this, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                cmp.set("v.opportunities", response.getReturnValue());
            }
        });
  $A.enqueueAction(action);
```

```
    }
})
```

メモ: コンポーネントの初期化時にレコードデータを読み込むには、init ハンドラを使用します。

カスタムオブジェクトからのレコードデータの読み込み

Apexコントローラを使用し、コンポーネントの属性にデータを設定して、レコードデータを読み込みます。次のサーバ側コントローラは、カスタムオブジェクト myObj__c のレコードを返します。

```
public with sharing class MyObjController {

    @AuraEnabled
    public static List<MyObj__c> getMyObjects() {

        // Perform isAccessible() checks here
        return [SELECT Id, Name, myField__c FROM MyObj__c];
    }
}
```

次のコンポーネント例では、上記のコントローラを使用して myObj__c カスタムオブジェクトからレコードのリストを返します。

```
<aura:component controller="MyObjController"/>
<aura:attribute name="myObjects" type="namespace.MyObj__c[]"/>
<aura:iteration items="{!v.myObjects}" var="obj">
    {!obj.Name}, {!obj.namespace__myField__c}
</aura:iteration>
```

次のクライアント側コントローラでは、サーバ側コントローラの getMyObjects() メソッドをコールして、myObjects コンポーネントの属性をレコードデータで設定します。次の手順は、init ハンドラを使用したコンポーネントの初期化時に行うこともできます。

```
getMyObjects: function(cmp){
    var action = cmp.get("c.getMyObjects");
    action.setCallback(this, function(response){
        var state = response.getState();
        if (state === "SUCCESS") {
            cmp.set("v.myObjects", response.getReturnValue());
        }
    });
    $A.enqueueAction(action);
}
```

コントローラを使用したレコードの読み込みと更新の例については、「クイックスタート」(ページ 8)を参照してください。

このセクションの内容:

CRUD および項目レベルセキュリティ (FLS)

Lightning コンポーネントでは、オブジェクトを参照したり、Apex コントローラからオブジェクトを取得したりするときに、CRUD および FLS が自動的に適用されることはありません。つまり、このフレームワークでは、ユーザに CRUD アクセス権および FLS 表示権限がないレコードと項目は引き続き表示されます。CRUD と FLS は、Apex コントローラで手動によって適用する必要があります。

レコードの保存

Salesforce1 に組み込まれたレコードの作成および編集ページを利用して、Lightning コンポーネントからレコードを作成または編集することができます。

レコードの削除

Lightning コンポーネントを介してビューとデータベースの両方からレコードを削除できます。

関連トピック:

CRUD および項目レベルセキュリティ (FLS)

CRUD および項目レベルセキュリティ (FLS)

Lightning コンポーネントでは、オブジェクトを参照したり、Apex コントローラからオブジェクトを取得したりするときに、CRUD および FLS が自動的に適用されることはありません。つまり、このフレームワークでは、ユーザに CRUD アクセス権および FLS 表示権限がないレコードと項目は引き続き表示されます。CRUD と FLS は、Apex コントローラで手動によって適用する必要があります。

たとえば、`with sharing` キーワードを Apex コントローラに含めることで、ユーザには Lightning コンポーネントでアクセス権を持つレコードのみが表示されます。さらに、レコードまたはオブジェクトに対する操作を実行する前に、`isAccessible()`、`isCreateable()`、`isDeletable()`、`isUpdateable()` があることを明示的にチェックする必要があります。

次の例は、カスタム経費オブジェクトに対する推奨される操作実行方法を示します。

```
public with sharing class ExpenseController {

    // ns refers to namespace; leave out ns__ if not needed
    // This method is vulnerable.
    @AuraEnabled
    public static List<ns__Expense__c> get_UNSAFE_Expenses() {
        return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
     }

    // This method is recommended.
    @AuraEnabled
    public static List<ns__Expense__c> getExpenses() {
        String [] expenseAccessFields = new String [] {'Id',
                                                       'Name',
                                                       'ns__Amount__c',
                                                       'ns__Client__c',
                                                       'ns__Date__c',
                                                       'ns__Reimbursed__c',
```

```
                                                      'CreatedDate'
                                                      };


    // Obtain the field name/token map for the Expense object
    Map<String,Schema.SObjectField> m = Schema.SObjectType.ns__Expense__c.fields.getMap();


    for (String fieldToCheck : expenseAccessFields) {

        // Check if the user has access to view field
        if (!m.get(fieldToCheck).getDescribe().isAccessible()) {

            // Pass error to client
            throw new System.NoAccessException();

           // Suppress editor logs
           return null;
        }
    }

    // Query the object safely
    return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
    }
}
```

メモ: 詳細は、CRUD および FLS の適用と Lightning セキュリティに関する記事を参照してください。

レコードの保存

Salesforce1 に組み込まれたレコードの作成および編集ページを利用して、Lightning コンポーネントからレコードを作成または編集することができます。

次のコンポーネントには、クライアント側コントローラをコールしてレコードの編集ページを表示するボタンがあります。

```
<aura:component>
    <lightning:button label="Edit Record" onclick="{!c.edit}"/>
</aura:component>
```

クライアント側コントローラから、指定された取引先責任者 ID を持つレコードの編集ページを表示する `force:recordEdit` イベントが起動されます。このイベントを正しく処理するには、コンポーネントが Salesforce1 に含まれている必要があります。

```
edit : function(component, event, helper) {
    var editRecordEvent = $A.get("e.force:editRecord");
    editRecordEvent.setParams({
        "recordId": component.get("v.contact.Id")
    });
    editRecordEvent.fire();
}
```

`force:recordEdit` イベントを使用して更新されたレコードは、デフォルトにより保持されます。

Lightning コンポーネントを使用したレコードの保存

または、ユーザがレコードを追加できるカスタムフォームを提供する Lightning コンポーネントを指定することもできます。新しいレコードを保存するには、クライアント側コントローラを Apex コントローラに結び付けます。次のリストに、コンポーネントおよび Apex コントローラを使用してレコードを保持する方法を示します。

メモ: レコード更新を処理するカスタムフォームを作成する場合は、独自の項目検証を指定する必要があります。

`upsert` 操作で行う更新を保存する Apex コントローラを作成する。次の例に、レコードデータを更新/挿入する Apex コントローラを示します。

```
@AuraEnabled
public static Expense__c saveExpense(Expense__c expense) {
    // Perform isUpdateable() check here
    upsert expense;
    return expense;
}
```

コンポーネントからクライアント側コントローラをコールする。たとえば、「`<lightning:button label="Submit" onclick="{!c.createExpense}"/>`」などです。

クライアント側コントローラで、項目入力規則を提供し、レコードデータをヘルパー関数に渡す。

```
createExpense : function(component, event, helper) {
    // Validate form fields
    // Pass form data to a helper function
    var newExpense = component.get("v.newExpense");
    helper.createExpense(component, newExpense);
}
```

コンポーネントヘルパーで、サーバ側コントローラのインスタンスを取得し、コールバックを設定する。次の例では、カスタムオブジェクトでレコードを更新/挿入します。`setParams()` は、サーバ側コントローラの `saveExpense()` メソッドで `expense` 引数の値を設定します。

```
createExpense: function(component, expense) {
    //Save the expense and update the view
    this.upsertExpense(component, expense, function(a) {
        var expenses = component.get("v.expenses");
        expenses.push(a.getReturnValue());
        component.set("v.expenses", expenses);
    });
},
upsertExpense : function(component, expense, callback) {
  var action = component.get("c.saveExpense");
  action.setParams({
      "expense": expense
  });
  if (callback) {
      action.setCallback(this, callback);
  }
```

```
  $A.enqueueAction(action);
}
```

関連トピック:

CRUD および項目レベルセキュリティ (FLS)

レコードの削除

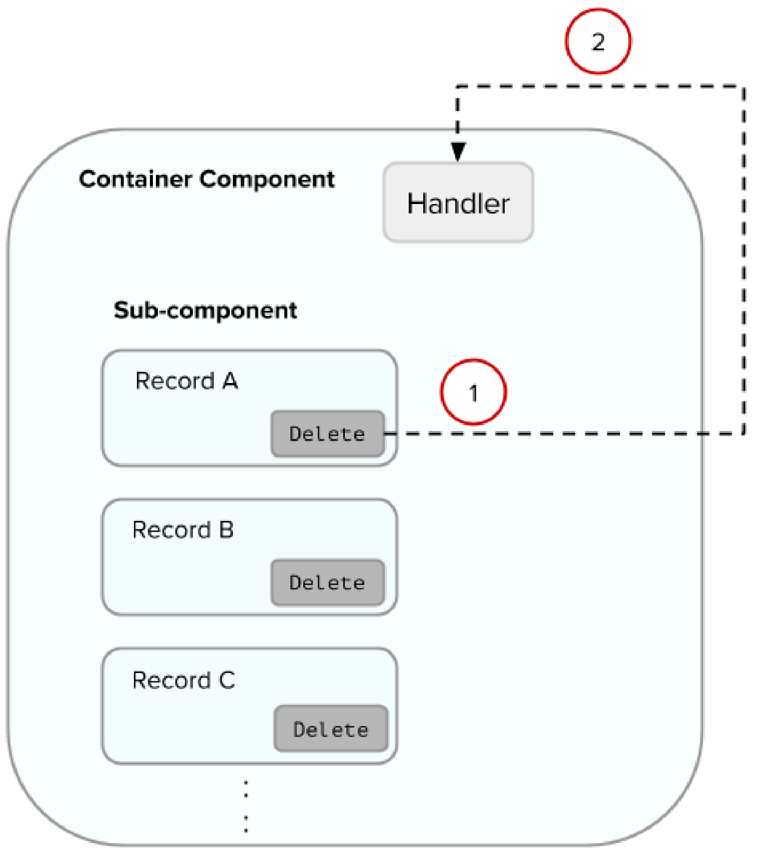
Lightning コンポーネントを介してビューとデータベースの両方からレコードを削除できます。

`delete` 操作で指定されたレコードを削除する Apex コントローラを作成します。次の Apex コントローラは経費オブジェクトレコードを削除します。

```
@AuraEnabled
public static Expense__c deleteExpense(Expense__c expense) {
    // Perform isDeletable() check here
    delete expense;
    return expense;
}
```

コンピュータの設定方法に応じて、別のコンポーネントにレコードが削除されたことを伝えるイベントの作成が必要になる場合があります。たとえば、サブコンポーネントを反復処理してレコードを表示するコンポーネントがあるとします。サブコンポーネントに含まれるボタン (1) は、押されるとイベントを起動し、そのイベントを処理するコンテナコンポーネント (2) がクリックされたレコードを削除します。

```
<aura:registerEvent name="deleteExpenseItem" type="c:deleteExpenseItem"/>
<lightning:button label="Delete" onclick="{!c.delete}"/>
```



コンポーネントイベントを作成して、削除するレコードを取得して渡します。イベントに deleteExpenseItem という名前を付けます。

```
<aura:event type="COMPONENT">
    <aura:attribute name="expense" type="Expense__c"/>
</aura:event>
```

次に、削除するレコードを渡し、クライアント側コントローラでイベントを起動します。

```
delete : function(component, evt, helper) {
    var expense = component.get("v.expense");
    var deleteEvent = component.getEvent("deleteExpenseItem");
    deleteEvent.setParams({ "expense": expense }).fire();
}
```

コンテナコンポーネントに、イベントのハンドラを含めます。この例では、c:expenseList が、レコードを表示するサブコンポーネントです。

```
<aura:handler name="deleteExpenseItem" event="c:deleteExpenseItem" action="c:deleteEvent"/>
<aura:iteration items="{!v.expenses}" var="expense">
    <c:expenseList expense="{!expense}"/>
</aura:iteration>
```

さらに、コンテナコンポーネントのクライアント側コントローラでイベントを処理します。

```
deleteEvent : function(component, event, helper) {
    // Call the helper function to delete record and update view
    helper.deleteExpense(component, event.getParam("expense"));
}
```

最後に、コンテナコンポーネントのヘルパー関数でApexコントローラをコールしてレコードを削除し、ビューを更新します。

```
deleteExpense : function(component, expense, callback) {
    // Call the Apex controller and update the view in the callback
    var action = component.get("c.deleteExpense");
    action.setParams({
        "expense": expense
    });
    action.setCallback(this, function(response) {
        var state = response.getState();
        if (state === "SUCCESS") {
            // Remove only the deleted expense from view
            var expenses = component.get("v.expenses");
            var items = [];
            for (i = 0; i < expenses.length; i++) {
                if(expenses[i]!==expense) {
                    items.push(expenses[i]);
                }
            }
            component.set("v.expenses", items);
            // Other client-side logic
        }
    });
    $A.enqueueAction(action);
}
```

ヘルパー関数がApexコントローラをコールしてデータベースのレコードを削除します。コールバック関数で、`component.set("v.expenses", items)` が、更新されたレコードの配列を使用してビューを更新します。

関連トピック:

CRUD および項目レベルセキュリティ (FLS)

コンポーネントイベント

サーバ側のアクションのコール

Apex コードのテスト

管理パッケージをアップロードする前に、Apexコードのテストを作成および実行して、最小コードカバー率要件を満たす必要があります。また、パッケージを AppExchange にアップロードするときには、すべてのテストがエラーなしで実行される必要があります。

Apexコードを使用するアプリケーションとコンポーネントをパッケージ化するには、次の条件を満たす必要があります。

- Apex コードの少なくとも 75% が単体テストでカバーされており、かつすべてのテストが成功している。

次の点に注意してください。

- 本番組織に Apex をリリースするときに、組織の名前空間内の各単体テストがデフォルトで実行されます。
- System.debug へのコールは、Apex コードカバー率の対象とはみなされません。
- テストメソッドとテストクラスは、Apex コードカバー率の対象とはみなされません。
- Apexコードの75%が単体テストでカバーされている必要がありますが、カバー率を上げることだけに集中すべきではありません。アプリケーションのすべての使用事例 (正・誤両方の場合や単一データだけでなく複数データの場合) の単体テストを作成するようにしてください。このような多様な使用事例のテストコードを実装することが 75% 以上のカバー率につながります。

- すべてのトリガについて何らかのテストを行う。
- すべてのクラスとトリガが正常にコンパイルされる。

次のサンプルは、コンポーネントに接続されたカスタムオブジェクトの Apex テストクラスを示します。

```
@isTest
class TestExpenseController {
    static testMethod void test() {
        //Create new expense and insert it into the database
        Expense__c exp = new Expense__c(name='My New Expense',
                            amount__c=20, client__c='ABC',
                            reimbursed__c=false, date__c=null);
         ExpenseController.saveExpense(exp);

        //Assert the name field and saved expense
        System.assertEquals('My New Expense',
                           ExpenseController.getExpenses()[0].Name,
                          'Name does not match');
        System.assertEquals(exp, ExpenseController.saveExpense(exp));
    }
}
```

メモ: Apex クラスは手動でパッケージに追加する必要があります。

Apex コードの配布についての詳細は、『Apex コード開発者ガイド』を参照してください。

関連トピック:

アプリケーションとコンポーネントの配布

Apex からの API コールの実行

Apex コントローラから API コールを行います。JavaScript コードから Salesforce API コールを行うことはできません。

セキュリティ上の理由により、Lightning コンポーネントフレームワークでは、JavaScript コードからの API コールの実行が制限されています。コンポーネントの JavaScript コードからサードパーティ API をコールするには、API エンドポイントを CSP 信頼済みサイトとして追加します。

Salesforce API をコールするには、コンポーネントの Apex コントローラから API コールを行います。指定ログイン情報を使用して、Salesforce に対して認証します。

メモ: セキュリティポリシーにより、Lightning コンポーネントによって作成されたセッションでは API アクセスが有効になっていません。これにより、Apex コードも Salesforce への API コールを行うことができなくなります。特定の API コールに値して指定ログイン情報を使用することによって、このセキュリティ制限を慎重かつ選択的にスキップすることができます。

API が有効なセッションが制限されているのは偶然ではありません。指定ログイン情報を使用するコードを慎重に確認し、脆弱性が生じないようにしてください。

Apex からの API コールの実行については、*『Apex 開発者ガイド』*を参照してください。

関連トピック:

Apex 開発者ガイド: コールアウトエンドポイントとしての指定ログイン情報

コンポーネントからの API コールの実行

サードパーティ API にアクセスするための CSP 信頼済みサイトの作成

コンテンツセキュリティポリシーの概要

Apex でのコンポーネントの作成

サーバ側で Apex の `Cmp.<myNamespace>.<myComponent>` 構文を使用してコンポーネントを作成することができなくなりました。代わりに、クライアント側の JavaScript コードで `$A.createComponent()` を使用してください。

関連トピック:

コンポーネントの動的な作成

Lightning データサービス

Lightning Data Service を使用すると、Apex コードを必要とせずに、コンポーネントでレコードの読み込み、作成、編集、削除ができます。Lightning データサービスは、共有ルールと項目レベルセキュリティを処理します。Apex を要しないだけでなく、Lightning データサービスはパフォーマンスとユーザインターフェースの一貫性を改善します。

端的に言うと、Lightning データサービスは、Visualforce 標準コントローラの Lightning コンポーネントバージョンと考えることができます。こう言うと単純化しすぎですが、要点は伝わるでしょう。可能な場合は、Lightning データサービスを使用して、コンポーネントで Salesforce データの参照と変更を行います。

Lightning Data Service を使用したデータアクセスは、サーバ側の Apex コントローラを使用した同等の処理よりも簡単です。参照のみアクセスは、コンポーネントのマークアップ内ですべて宣言型にできます。データを変更するコードの場合、コンポーネントの JavaScript コントローラはほぼ同じ量のコードになり、Apex をすべてなくすことができます。データアクセスコードのすべてがコンポーネントに統合され、複雑さが大幅に緩和されます。

Lightning データサービスには、コード以外にも利点があります。非常に効率のよいローカルストレージ上に構築されます。このローカルストレージはそれを使用するすべてのコンポーネントで共有されます。Lightning データサービスに読み込まれたレコードはキャッシュされ、コンポーネント間で共有されます。同じレコード

にアクセスするコンポーネントでは、パフォーマンスが大幅に改善されます。これはレコードが、それを使用するコンポーネントの数に関係なく1回だけ読み込まれるためです。共有レコードもユーザインターフェースの一貫性を高めます。あるコンポーネントがレコードを更新すると、そのレコードを使用する他のコンポーネントに通知され、ほとんどの場合は自動的に更新されます。

このセクションの内容:

レコードの読み込み
レコードの読み込みは、Lightning データサービスの操作の中でも最も簡単です。すべてマークアップで達成できます。

レコードの保存
Lightning データサービスを使用してレコードを保存するには、`force:recordData` コンポーネントの `saveRecord` をコールして、保存操作の完了後に呼び出されるコールバック関数を渡します。

レコードの作成
Lightning データサービスを使用してレコードを作成する場合は、`recordId` を割り当てずに、`force:recordData` を宣言します。次に、`force:recordData` の `getNewRecord` 関数をコールして、レコードテンプレートを読み込みます。最後に、新しいレコードに値を適用し、`force:recordData` の `saveRecord` 関数をコールしてレコードを保存します。

レコードの削除
Lightning データサービスを使用してレコードを削除するには、`force:recordData` コンポーネントの `deleteRecord` をコールして、削除操作の完了後に呼び出されるコールバック関数を渡します。

レコードの変更
レコードの変更時にレコードの再表示以外のタスクを実行するには、`recordUpdated` イベントを処理します。レコードの読み込み、更新、削除による変更を処理して、変更種別ごとにアクションを適用できます。

エラー
エラーの発生時に動作を実行するには、`recordUpdated` イベントを処理して、`changeType` が「ERROR」のケースを処理します。

考慮事項
Lightning データサービスは、強力で使いやすいサービスです。けれども、独自のデータアクセスコードの記述に完全に取って代わるものではありません。以下に、使用上の留意事項をいくつか示します。

Lightning データサービスの例
ここでは、Lightning データサービスを使用して「Quick Contact (取引先責任者のクイック作成)」アクションパネルを作成する詳細な例を示します。

SaveRecordResult
レコードデータに永続的な変更をもたらす Lightning データサービス操作の結果を表します。

レコードの読み込み

レコードの読み込みは、Lightning データサービスの操作の中でも最も簡単です。すべてマークアップで達成できます。

Lightning データサービスを使用してレコードを読み込むには、コンポーネントに force:recordData タグを追加します。force:recordData タグで、読み込むレコードの ID、項目のリスト、読み込まれたレコードを割り当てる属性を指定します。force:recordData で次の事項を指定する必要があります。

- 読み込むレコードの ID
- 読み込んだレコードを割り当てるコンポーネント属性
- 読み込む項目のリスト

fields 属性で、読み込む項目のリストを明示的に指定できます。たとえば、fields="Name,BillingCity,BillingState" です。

または、layoutType 属性を使用してレイアウトを指定することもできます。そのレイアウトのすべての項目がレコードに読み込まれます。レイアウトは通常、システム管理者が変更します。layoutType を使用してレコードデータを読み込むと、コンポーネントをこれらのレイアウト定義に適合させることができます。layoutType の有効な値は、FULL および COMPACT です。

例: レコードの読み込み

次の例は、Lightning データサービスを使用したレコードの読み込みの要点を示しています。このコンポーネントは、Lightning アプリケーションビルダーのレコードホームページに追加するか、カスタムアクションとして追加することができます。レコード ID は、force:hasRecordId インターフェースによって追加される暗黙的な recordId 属性によって指定されます。

ldsLoad.cmp

```
<aura:component
implements="flexipage:availableForRecordHome,force:lightningQuickActionWithoutHeader,
 force:hasRecordId">

    <aura:attribute name="record" type="Object"/>
    <aura:attribute name="simpleRecord" type="Object"/>
    <aura:attribute name="recordError" type="String"/>

    <force:recordData aura:id="recordLoader"
      recordId="{!v.recordId}"
      layoutType="FULL"
      targetRecord="{!v.record}"
      targetFields="{!v.simpleRecord}"
      targetError="{!v.recordError}"
      recordUpdated="{!c.handleRecordUpdated}"
      />

    <!-- Display a header with details about the record -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading--label">{!v.simpleRecord.Name}</p>
        <h1 class="slds-page-header__title slds-m-right--small
            slds-truncate slds-align-left">{!v.simpleRecord.BillingCity},
{!v.simpleRecord.BillingState}</h1>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
```

```
                {!v.recordError}
            </ui:message>
        </div>
    </aura:if>
</aura:component>
```

ldsLoadController.js

```
({
    handleRecordUpdated: function(component, event, helper) {
        var eventParams = event.getParams();
        if(eventParams.changeType === "LOADED") {
           // record is loaded (render other component which needs record data value)

            console.log("Record is loaded successfully.");
        } else if(eventParams.changeType === "CHANGED") {
            // record is changed
        } else if(eventParams.changeType === "REMOVED") {
            // record is deleted
        } else if(eventParams.changeType === "ERROR") {
            // there’s an error while loading, saving, or deleting the record
        }
    }
})
```

関連トピック:

Lightning Experience のレコードホームページのコンポーネントの設定

レコード固有のアクションのコンポーネントの設定

force:recordPreview

レコードの保存

Lightning データサービスを使用してレコードを保存するには、force:recordData コンポーネントの saveRecord をコールして、保存操作の完了後に呼び出されるコールバック関数を渡します。

Lightning データサービスの保存操作は、次の 2 つの状況で行われます。

- 変更を既存のレコードに保存する場合
- 新しいレコードを作成して保存する場合

変更を既存のレコードに保存するには、レコードを EDIT モードで読み込み、force:recordData コンポーネントの saveRecord をコールします。

新しいレコードを保存するには作成する必要があるため、「レコードの作成」に記載のとおり、レコードテンプレートからレコードを作成します。次に、force:recordData コンポーネントの saveRecord をコールします。

EDIT モードのレコードの読み込み

更新される可能性のあるレコードを読み込むには、`force:recordData` タグの `mode` 属性を「EDIT」に設定します。`mode` を明示的に設定すること以外は、編集するレコードの読み込みも他の目的の読み込みと同じです。

メモ: Lightning データサービスのレコードは複数のコンポーネントで共有されるため、レコードを読み込むと、直接参照ではなくレコードのコピーを使用してコンポーネントが読み込まれます。コンポーネントがレコードを VIEW モードで読み込む場合、そのレコードが変更されると、Lightning データサービスはそのコピーをレコードの新しいコピーで自動的に上書きします。レコードが EDIT モードで読み込みれる場合、レコードが変更されてもそのレコードは更新されません。この動作は、レコードの編集中にそのレコードを参照するコンポーネントで未保存の変更が表示されることを防ぎ、処理中の編集が上書きされることを防ぎます。ただし、通知は両方のモードで送信されます。

`saveRecord` のコールによるレコードの変更の保存

保存操作を実行するには、適切なコントローラアクションハンドラから、`force:recordData` コンポーネントの `saveRecord` をコールします。`saveRecord` は、操作の完了時に呼び出されるコールバック関数という 1 つの引数を取ります。このコールバック関数は、その唯一のパラメータとして `SaveRecordResult` オブジェクトを受け取ります。`SaveRecordResult` には、操作の成否や、結果の処理に使用できるその他の詳細を示す `state` 属性が含まれます。

例: レコードの保存

次の例は、Lightning データサービスを使用したレコードの保存の要点を示しています。この例は、レコードページで使用されることを意図しています。レコード ID は、`force:hasRecordId` インターフェースによって追加される暗黙的な `recordId` 属性によって指定されます。

ldsSave.cmp

```
<<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">

    <aura:attribute name="record" type="Object"/>
    <aura:attribute name="simpleRecord" type="Object"/>
    <aura:attribute name="recordError" type="String"/>

    <force:recordData aura:id="recordHandler"
      recordId="{!v.recordId}"
      layoutType="FULL"
      targetRecord="{!v.record}"
      targetFields="{!v.simpleRecord}"
      targetError="{!v.recordError}"
      mode="EDIT"
      recordUpdated="{!c.handleRecordUpdated}"
      />

    <!-- Display a header with details about the record -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading--label">Edit Record</p>
        <h1 class="slds-page-header__title slds-m-right--small
            slds-truncate slds-align-left">{!v.simpleRecord.Name}</h1>
    </div>
```

```
    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
                {!v.recordError}
            </ui:message>
        </div>
    </aura:if>

    <!-- Display an editing form -->
    <lightning:input aura:id="recordName" name="recordName" label="Name"
                  value="{!v.simpleRecord.Name}" required="true"/>

     <lightning:button label="Save Record" onclick="{!c.handleSaveRecord}"
               variant="brand" class="slds-m-top--medium"/>
</aura:component>
```

メモ: 姓と名のあるオブジェクト (取引先責任者など) でこのコンポーネントを使用する場合、`{!v.simpleRecord.FirstName}` と `{!v.simpleRecord.LastName}` の個別の `lightning:input` コンポーネントを作成します。

このコンポーネントは、EDIT モードに設定された `force:recordData` を使用してレコードを読み込み、レコードの値を編集するためのフォームを提供します(この単純な例では、レコード名項目のみです)。

ldsSaveController.js

```
({
    handleSaveRecord: function(component, event, helper) {
       component.find("recordHandler").saveRecord($A.getCallback(function(saveResult)
 {
            // NOTE: If you want a specific behavior(an action or UI behavior) when
this action is successful
            // then handle that in a callback (generic logic when record is changed
should be handled in recordUpdated event handler)
            if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                // handle component related logic in event handler
            } else if (saveResult.state === "INCOMPLETE") {
                console.log("User is offline, device doesn't support drafts.");
            } else if (saveResult.state === "ERROR") {
                console.log('Problem saving record, error: ' +
JSON.stringify(saveResult.error));
            } else {
                console.log('Unknown problem, state: ' + saveResult.state + ', error:
 ' + JSON.stringify(saveResult.error));
            }
        }));
    },

    /**
     * Control the component behavior here when record is changed (via any component)

     */
    handleRecordUpdated: function(component, event, helper) {
```

```
        var eventParams = event.getParams();
        if(eventParams.changeType === "CHANGED") {
            // get the fields that changed for this record
            var changedFields = eventParams.changedFields;
            console.log('Fields that are changed: ' + JSON.stringify(changedFields));

            // record is changed, so refresh the component (or other component logic)

            var resultsToast = $A.get("e.force:showToast");
            resultsToast.setParams({
                "title": "Saved",
                "message": "The record was updated."
            });
            resultsToast.fire();

        } else if(eventParams.changeType === "LOADED") {
            // record is loaded in the cache
        } else if(eventParams.changeType === "REMOVED") {
            // record is deleted and removed from the cache
        } else if(eventParams.changeType === "ERROR") {
            // there's an error while loading, saving or deleting the record
        }
    }
})
```

この例の `handleSaveRecord` アクションは最小限のものです。フォームの検証や実際のエラー処理はありません。フォームに入力された内容はすべて、レコードへの保存の対象になります。

関連トピック:

SaveRecordResult

Lightning Experience のレコードホームページのコンポーネントの設定

レコード固有のアクションのコンポーネントの設定

force:recordPreview

レコードの作成

Lightning データサービスを使用してレコードを作成する場合は、`recordId` を割り当てずに、`force:recordData` を宣言します。次に、`force:recordData` の `getNewRecord` 関数をコールして、レコードテンプレートを読み込みます。最後に、新しいレコードに値を適用し、`force:recordData` の `saveRecord` 関数をコールしてレコードを保存します。

1. `getNewRecord` をコールして、レコードテンプレートから空のレコードを作成します。このレコードは、フォームのバッキングストアとして使用できます。あるいは、その値を保存用のデータに設定することもできます。
2. `saveRecord` をコールして、レコードを確定させます。この点は、「レコードの保存」に記載されています。

レコードテンプレートから空のレコードの作成

レコードテンプレートから空のレコードを作成する場合、`force:recordData` タグに `recordId` は設定できません。`recordId` がなければ、Lightning データサービスが既存のレコードを読み込むことがありません。

コンポーネントの `init` または別のハンドラで、`force:recordData` の `getNewRecord` をコールします。`getNewRecord` は次の引数を取ります。

属性名	型	説明
objectApiName	String	新しいレコードのオブジェクト API 参照名。
recordTypeId	String	新しいレコードのレコードタイプの 18 文字の ID。 指定されていない場合は、ユーザのプロファイルで定義したオブジェクトのデフォルトのレコードタイプが使用されます。
skipCache	Boolean	レコードテンプレートを、クライアント側の Lightning データサービスキャッシュではなく、サーバから読み込むかどうか。デフォルトは `false` です。
callback	Function	空のレコードの作成後に呼び出される関数。この関数は引数を受け取りません。

`getNewRecord` は結果を返しません。空のレコードを準備して、`targetRecord` 属性に割り当てるだけです。

例: レコードの作成

次の例は、Lightning データサービスを使用したレコードの作成の要点を示しています。この例は、取引先レコードの Lightning ページに追加されることを意図しています。

ldsCreate.cmp

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">

    <aura:attribute name="newContact" type="Object"/>
    <aura:attribute name="simpleNewContact" type="Object"/>
    <aura:attribute name="newContactError" type="String"/>

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <force:recordData aura:id="contactRecordCreator"
                      layoutType="FULL"
                      targetRecord="{!v.newContact}"
                      targetFields="{!v.simpleNewContact}"
                      targetError="{!v.newContactError}" />

    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading--label">Create Contact</p>
    </div>

    <!-- Display Lightning Data Service errors -->
    <aura:if isTrue="{!not(empty(v.newContactError))}">
```

```
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
                {!v.newContactError}
            </ui:message>
        </div>
    </aura:if>

    <!-- Display the new contact form -->
    <div class="slds-form--stacked">
        <lightning:input aura:id="contactField" name="firstName" label="First Name"
                         value="{!v.simpleNewContact.FirstName}" required="true"/>

        <lightning:input aura:id="contactField" name="lastname" label="Last Name"
                      value="{!v.simpleNewContact.LastName}" required="true"/>

        <lightning:input aura:id="contactField" name="title" label="Title"
                      value="{!v.simpleNewContact.Title}" />

        <lightning:button label="Save contact" onclick="{!c.handleSaveContact}"
                   variant="brand" class="slds-m-top--medium"/>
   </div>

</aura:component>
```

このコンポーネントは、`force:recordData` の `recordId` 属性を設定しません。Lightning データサービスに、レコードの新規作成が見込まれることを伝達します。以下は、コンポーネントの `init` ハンドラにこのレコードが作成されています。

ldsCreateController.js

```
({
    doInit: function(component, event, helper) {
        // Prepare a new record from template
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // sObject type (objectApiName)
            null,      // recordTypeId
            false,     // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                    return;
                }
                console.log("Record template initialized: " + rec.sobjectType);
            })
        );
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
          component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));

            component.find("contactRecordCreator").saveRecord(function(saveResult) {
```

```
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {

                    // record is saved successfully
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Saved",
                        "message": "The record was saved."
                    });
                    resultsToast.fire();

                } else if (saveResult.state === "INCOMPLETE") {
                    // handle the incomplete state
                    console.log("User is offline, device doesn't support drafts.");
                } else if (saveResult.state === "ERROR") {
                    // handle the error state
                    console.log('Problem saving contact, error: ' +
JSON.stringify(saveResult.error));
                } else {
                    console.log('Unknown problem, state: ' + saveResult.state + ',
error: ' + JSON.stringify(saveResult.error));
                }
            });
        }
    }
})
```

doInit init ハンドラが、force:recordData コンポーネントの getNewRecord() をコールして、簡単なコールバックハンドラを渡します。このコールによって、コンポーネントのマークアップで取引先責任者フォームが使用する新しい空の取引先責任者レコードが作成されます。

メモ: getNewRecord() に渡されるコールバックは、コールバックが呼び出されるときのアクセスコンテキストが適切なものになるように、$A.getCallback() でラップする必要があります。コールバックが $A.getCallback() でラップされないまま渡されると、コンポーネントの非公開属性にアクセスしようとしたときにアクセスチェックに失敗します。

アクセス先が非公開属性でなくても、$A.getCallback() の getNewRecord() のコールバック関数を常にラップすることをお勧めします。コンテキストを混同することさえなければ大丈夫です。

[取引先責任者を保存] ボタンがクリックされると、handleSaveContact ハンドラがコールされます。これは、「レコードの保存」に記載のとおり、取引先責任者を保存して、ユーザインターフェースを更新する単純明快なアプリケーションです。

メモ: ヘルパー関数 validateContactForm は示されていません。この関数は単にフォームの値を検証するものです。この検証例については、「Lightning データサービスの例」を参照してください。

関連トピック:

レコードの保存

Lightning Experience のレコードホームページのコンポーネントの設定

レコード固有のアクションのコンポーネントの設定

アクセスの制御

force:recordPreview

レコードの削除

Lightning データサービスを使用してレコードを削除するには、force:recordData コンポーネントの deleteRecord をコールして、削除操作の完了後に呼び出されるコールバック関数を渡します。

Lightning データサービスの削除操作は単純明快です。force:recordData タグに最小限の詳細を含めることができます。レコードデータが不要になる場合は、fields 属性を単に Id に設定します。削除が唯一の操作であることがわかっている場合は、任意の mode を使用できます。

削除操作を実行するには、適切なコントローラアクションハンドラから、force:recordData コンポーネントの deleteRecord をコールします。deleteRecord は、操作の完了時に呼び出されるコールバック関数という 1 つの引数を取ります。このコールバック関数は、その唯一のパラメータとして SaveRecordResult オブジェクトを受け取ります。SaveRecordResult には、操作の成否や、結果の処理に使用できるその他の詳細を示す state 属性が含まれます。

例: レコードの削除

次の例は、Lightning データサービスを使用したレコードの削除の要点を示しています。このコンポーネントは、[レコードを削除] ボタンをレコードページに追加して、表示されているレコードを削除できるようにします。レコード ID は、force:hasRecordId インターフェースによって追加される暗黙的な recordId 属性によって指定されます。

ldsDelete.cmp

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">

    <aura:attribute name="recordError" type="String" access="private"/>

    <force:recordData aura:id="recordHandler"
        recordId="{!v.recordId}"
        fields="Id"
        targetError="{!v.recordError}"
        recordUpdated="{!c.handleRecordUpdated}" />

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
                {!v.recordError}
```

```
            </ui:message>
        </div>
    </aura:if>

    <div class="slds-form-element">
        <lightning:button
            label="Delete Record"
            onclick="{!c.handleDeleteRecord}"
            variant="brand" />
    </div>
</aura:component>
```

force:recordData タグには、recordId と、最小限絶対必要なものだけを記載したほぼ空の fields リストのみが含まれます。たとえば、確認メッセージの一部として、ユーザインターフェースにレコード値を表示したい場合は、この最小限の削除例ではなく、読み込み操作の場合と同様に、force:recordData タグを定義します。

ldsDeleteController.js

```
({
    handleDeleteRecord: function(component, event, helper) {

component.find("recordHandler").deleteRecord($A.getCallback(function(deleteResult) {
            // NOTE: If you want a specific behavior(an action or UI behavior) when
this action is successful
            // then handle that in a callback (generic logic when record is changed
should be handled in recordUpdated event handler)
            if (deleteResult.state === "SUCCESS" || deleteResult.state === "DRAFT") {

                // record is deleted
                console.log("Record is deleted.");
            } else if (deleteResult.state === "INCOMPLETE") {
                console.log("User is offline, device doesn't support drafts.");
            } else if (deleteResult.state === "ERROR") {
                console.log('Problem deleting record, error: ' +
JSON.stringify(deleteResult.error));
            } else {
               console.log('Unknown problem, state: ' + deleteResult.state + ', error:
 ' + JSON.stringify(deleteResult.error));
            }
        }));
    },

    /**
     * Control the component behavior here when record is changed (via any component)

     */
    handleRecordUpdated: function(component, event, helper) {
        var eventParams = event.getParams();
        if(eventParams.changeType === "CHANGED") {
           // record is changed
        } else if(eventParams.changeType === "LOADED") {
            // record is loaded in the cache
        } else if(eventParams.changeType === "REMOVED") {
```

```
            // record is deleted, show a toast UI message
            var resultsToast = $A.get("e.force:showToast");
            resultsToast.setParams({
                "title": "Deleted",
                "message": "The record was deleted."
            });
            resultsToast.fire();

        } else if(eventParams.changeType === "ERROR") {
            // there's an error while loading, saving, or deleting the record
        }
    }
})
```

レコードが削除された時点で、別のページに移動します。移動しないと、コンポーネントが更新されたときに「レコードが見つかりません」というエラーが表示されます。ここでは、コールバック関数に提供された `SaveRecordResult` の `objectApiName` プロパティをコントローラが使用して、オブジェクトのホームページに移動します。

関連トピック:

SaveRecordResult

Lightning Experience のレコードホームページのコンポーネントの設定

レコード固有のアクションのコンポーネントの設定

force:recordPreview

レコードの変更

レコードの変更時にレコードの再表示以外のタスクを実行するには、`recordUpdated` イベントを処理します。レコードの読み込み、更新、削除による変更を処理して、変更種別ごとにアクションを適用できます。

コンポーネントがレコードデータに固有のロジックを実行する場合は、レコードの変更時にそのロジックをもう一度実行する必要があります。よくある例は、レコードの値に応じてレコードに適用されるアクションが変化するビジネスプロセスです。たとえば、営業サイクルのフェーズごとに商談に適用されるアクションが異なります。

メモ: 変更された項目がリスナーの項目またはレイアウトと同じ場合にのみ、Lightning データサービスからリスナーにデータの変更について通知されます。

例: コンポーネントで recordUpdated イベントを処理することを宣言します。

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    layoutType="FULL"
    targetRecord="{!v._record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    recordUpdated="{!c.recordUpdated}" />
```

変更を処理するアクションハンドラを実装します。

```
({
  recordUpdated: function(component, event, helper) {

    var changeType = event.getParams().changeType;

    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") { /* handle record change */ }
})
```

編集モードでレコードを読み込む場合は、現在進行中の編集が上書きされないように、レコードが自動的に更新されません。レコードを更新するには、アクションハンドラで `reloadRecord` メソッドを使用します。

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    layoutType="FULL"
    targetRecord="{!v._record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    mode="EDIT"
    recordUpdated="{!c.recordUpdated}" />
```

```
({
  recordUpdated : function(component, event, helper) {

    var changeType = event.getParams().changeType;

    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") {
      /* handle record change; reloadRecord will cause you to lose your current record,
 including any changes you've made */
      component.find("forceRecord").reloadRecord();}
    }
})
```

エラー

エラーの発生時に動作を実行するには、`recordUpdated` イベントを処理して、`changeType` が「ERROR」のケースを処理します。

例: コンポーネントで `recordUpdated` イベントを処理することを宣言します。

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    layoutType="FULL"
    targetRecord="{!v._record}"
```

```
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    recordUpdated="{!c.recordUpdated}" />
```

エラーを処理するアクションハンドラを実装します。

```
({
  recordUpdated: function(component, event, helper) {

    var changeType = event.getParams().changeType;

    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") { /* handle record change */ }
})
```

レコードの読み込み開始時にエラーが発生すると、`targetError` がローカライズされたエラーメッセージに設定されます。エラーが発生するのは次の場合です。

- 属性値または属性値の組み合わせが無効なため、入力が無効の場合。たとえば、`recordId` が無効な場合、`layoutType` 属性と `fields` 属性の両方を省略した場合などです。
- レコードがキャッシュになく、サーバにアクセスできない (オフライン) 場合

サーバ上でレコードがアクセス不能になると、`changeType` が「REMOVED」に設定された `recordUpdated` イベントが起動されます。レコードにアクセス不能になることが操作の予期される結果であることもあるため、`targetError` ではエラーが設定されません。たとえば、リードの取引開始後はリードレコードがアクセス不能になります。

次のことが原因でレコードがアクセス不能になることがあります。

- レコードまたはエンティティの共有または表示設定
- レコードまたはエンティティの削除

サーバ上でレコードがアクセス不能になった場合、`targetRecord` に割り当てられているレコードの JavaScript オブジェクトは変更されません。

考慮事項

Lightning データサービスは、強力で使いやすいサービスです。けれども、独自のデータアクセスコードの記述に完全に取って代わるものではありません。以下に、使用上の留意事項をいくつか示します。

Lightning データサービスは、Lightning Experience および Salesforce1 のみで使用できます。Visualforce 用 Lightning コンポーネント、Lightning Out、コミュニティなど、他のコンテナでの Lightning データサービスの使用はサポートされていません。Lightning Experience に追加された Visualforce ページなど、Lightning Experience や Salesforce1 内でこれらのコンテナにアクセスした場合も同じです。

Lightning データサービスは、プリミティブな DML 操作 (作成、参照、更新、削除) をサポートします。これらの操作は一度に 1 つのレコードで行われ、レコード ID を使用してレコードを取得または変更します。Lightning データサービスでは、最大深度が 5 レベルに及ぶ項目がサポートされています。レコードのコレクションの処

理や、レコード ID 以外によるレコードのクエリはサポートされていません。高度な操作や、1 回のトランザクションでの複数の操作をサポートする必要がある場合は、標準の `@AuraEnabled` Apex メソッドを使用します。

Lightning データサービス共有データストレージでは、コンポーネントによってレコードが変更されるたびに、そのレコードを使用するすべてのコンポーネントに通知されます。たとえば他のユーザがレコードを変更した場合など、レコードがサーバ上で変更された場合はコンポーネントに通知されません。レコードがサーバ上で変更された場合、再読み込みされるまでローカルで更新されることはありません。変更された項目がリスナーの項目またはレイアウトと同じ場合にのみ、Lightning データサービスからリスナーにデータの変更について通知されます。

サポートされるオブジェクト

Lightning データサービスは、カスタムオブジェクトと次のオブジェクトをサポートします。

- Account
- AccountTeamMember
- Asset
- AssetRelationship
- AssignedResource
- AttachedContentNote
- BusinessAccount
- Campaign
- CampaignMember
- Case
- Contact
- ContentDocument
- ContentNote
- ContentVersion
- ContentWorkspace
- Contract
- ContractContactRole
- ContractLineItem
- Entitlement
- EnvironmentHubMember
- Lead
- LicensingRequest
- MaintenanceAsset
- MaintenancePlan
- MarketingAction
- MarketingResource
- Note
- OperatingHours
- Opportunity

- OpportunityLineItem
- OpportunityTeamMember
- Order
- OrderItem
- PersonAccount
- Pricebook2
- PricebookEntry
- ProcessInstanceHistory
- Product2
- Quote
- QuoteDocument
- QuoteLineItem
- ResourceAbsence
- ServiceAppointment
- ServiceContract
- ServiceCrew
- ServiceCrewMember
- ServiceResource
- ServiceResourceCapacity
- ServiceResourceSkill
- ServiceTerritory
- ServiceTerritoryLocation
- ServiceTerritoryMember
- Shipment
- SkillRequirement
- SocialPost
- Tenant
- TimeSheet
- TimeSheetEntry
- TimeSlot
- UsageEntitlement
- UsageEntitlementPeriod
- User
- WorkOrder
- WorkOrderLineItem
- WorkType

Lightning データサービスの例

ここでは、Lightning データサービスを使用して「Quick Contact (取引先責任者のクイック作成)」アクションパネルを作成する詳細な例を示します。

例: この例は、Lightning アクションとして取引先オブジェクトに追加されることを意図しています。取引先レイアウトのアクションのボタンをクリックすると、取引先責任者を新規作成するパネルが開きます。

この例は、「レコード固有のアクションのコンポーネントの設定」に記載の例と似ています。この2つの例を比較すると、`@AuraEnabled` Apex コントローラを使用した場合と Lightning データサービスを使用した場合の違いがよくわかります。

`ldsQuickContact.cmp`

```
<aura:component implements="force:lightningQuickActionWithoutHeader,force:hasRecordId">


    <aura:attribute name="account" type="Object"/>
    <aura:attribute name="simpleAccount" type="Object"/>
    <aura:attribute name="accountError" type="String"/>
    <force:recordData aura:id="accountRecordLoader"
        recordId="{!v.recordId}"
        fields="Name,BillingCity,BillingState"
        targetRecord="{!v.account}"
        targetFields="{!v.simpleAccount}"
        targetError="{!v.accountError}"
    />

    <aura:attribute name="newContact" type="Object" access="private"/>
    <aura:attribute name="simpleNewContact" type="Object" access="private"/>
```

```
    <aura:attribute name="newContactError" type="String" access="private"/>
    <force:recordData aura:id="contactRecordCreator"
        layoutType="FULL"
        targetRecord="{!v.newContact}"
        targetFields="{!v.simpleNewContact}"
        targetError="{!v.newContactError}"
        />

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <!-- Display a header with details about the account -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading--label">{!v.simpleAccount.Name}</p>
        <h1 class="slds-page-header__title slds-m-right--small
            slds-truncate slds-align-left">Create New Contact</h1>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.accountError))}">
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
                {!v.accountError}
            </ui:message>
        </div>
    </aura:if>
    <aura:if isTrue="{!not(empty(v.newContactError))}">
        <div class="recordError">
            <ui:message title="Error" severity="error" closable="true">
                {!v.newContactError}
            </ui:message>
        </div>
    </aura:if>

    <!-- Display the new contact form -->
     <lightning:input aura:id="contactField" name="firstName" label="First Name"
                     value="{!v.simpleNewContact.FirstName}" required="true"/>

    <lightning:input aura:id="contactField" name="lastname" label="Last Name"
                  value="{!v.simpleNewContact.LastName}" required="true"/>

    <lightning:input aura:id="contactField" name="title" label="Title"
                  value="{!v.simpleNewContact.Title}" />

    <lightning:input aura:id="contactField" type="phone" name="phone" label="Phone
Number"
                     pattern="^(1?(-?\d{3})-?)?(\d{3})(-?\d{4})$"
                     messageWhenPatternMismatch="The phone number must contain 7, 10,
 or 11 digits. Hyphens are optional."
                   value="{!v.simpleNewContact.Phone}" required="true"/>

    <lightning:input aura:id="contactField" type="email" name="email" label="Email"
                value="{!v.simpleNewContact.Email}" />

    <lightning:button label="Cancel" onclick="{!c.handleCancel}"
```

```
class="slds-m-top--medium" />
    <lightning:button label="Save Contact" onclick="{!c.handleSaveContact}"
               variant="brand" class="slds-m-top--medium"/>


</aura:component>
```

ldsQuickContactController.js

```
({
    doInit: function(component, event, helper) {
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // objectApiName
            null, // recordTypeId
            false, // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                }
                else {
                    console.log("Record template initialized: " + rec.sobjectType);
                }
            })
        );
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
          component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));

            component.find("contactRecordCreator").saveRecord(function(saveResult) {
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {


                    // Success! Prepare a toast UI message
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Contact Saved",
                        "message": "The new contact was created."
                    });

                    // Update the UI: close panel, show toast, refresh account page
                    $A.get("e.force:closeQuickAction").fire();
                    resultsToast.fire();

                    // Reload the view so components not using force:recordData
                    // are updated
                    $A.get("e.force:refreshView").fire();
                }
                else if (saveResult.state === "INCOMPLETE") {
                    console.log("User is offline, device doesn't support drafts.");
                }
                else if (saveResult.state === "ERROR") {
```

```
                    console.log('Problem saving contact, error: ' +
                                 JSON.stringify(saveResult.error));
                }
                else {
                    console.log('Unknown problem, state: ' + saveResult.state +
                                ', error: ' + JSON.stringify(saveResult.error));
                }
            });
        }
    },

    handleCancel: function(component, event, helper) {
        $A.get("e.force:closeQuickAction").fire();
    },
})
```

メモ: `getNewRecord()` に渡されるコールバックは、コールバックが呼び出されるときのアクセスコンテキストが適切なものになるように、`$A.getCallback()` でラップする必要があります。コールバックが `$A.getCallback()` でラップされないまま渡されると、コンポーネントの非公開属性にアクセスしようとしたときにアクセスチェックに失敗します。

アクセス先が非公開属性でなくても、`$A.getCallback()` の `getNewRecord()` のコールバック関数を常にラップすることをお勧めします。コンテキストを混同することさえなければ大丈夫です。

ldsQuickContactHelper.js

```
({
    validateContactForm: function(component) {
        var validContact = true;

         // Show error messages if required fields are blank
        var allValid = component.find('contactField').reduce(function (validFields,
inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validFields && inputCmp.get('v.validity').valid;
        }, true);

        if (allValid) {
            // Verify we have an account to attach it to
            var account = component.get("v.account");
            if($A.util.isEmpty(account)) {
                validContact = false;
                console.log("Quick action context doesn't have a valid account.");
            }
         return(validContact);

        }
 }
```

```
})
```

関連トピック:

レコード固有のアクションのコンポーネントの設定

アクセスの制御

force:recordPreview

SaveRecordResult

レコードデータに永続的な変更をもたらす Lightning データサービス操作の結果を表します。

SaveRecordResult オブジェクト

`saveRecord` および `deleteRecord` 関数のコールバック関数は、`SaveRecordResult` オブジェクトを唯一の引数として受け取ります。

属性名	型	説明
`objectApiName`	String	レコードのオブジェクト API 参照名。
`entityLabel`	String	レコードの sObject の名前の表示ラベル。
`error`	String	エラーは次のいずれかになります。 • どこで問題が生じたかを示すローカライズ済みメッセージ。 • エラーの配列(どこで問題が生じたかを示すローカライズ済みメッセージを含む)。また、項目またはページレベルのエラーなど、エラーの処理に役立つ詳しいデータが含まれていることもあります。 保存の `state` が SUCCESS または DRAFT の場合は、`error` が定義されません。
`recordId`	String	影響を受けるレコードの 18 文字の ID。
`state`	String	操作結果の状態。値は次のとおりです。 • SUCCESS — 操作がサーバで正常に完了しました。 • DRAFT — サーバに到達できなかったため、操作がローカルにドラフトとして保存されました。到達可能になった時点で変更がサーバに適用されます。 • INCOMPLETE — サーバに到達できず、デバイスがドラフトをサポートしていません(ドラフトは Salesforce1 アプリケーションのみでサポートされています)。しばらくしてからこの操作をやり直してください。

属性名	型	説明
		• ERROR — 操作を完了できませんでした。詳細は、error 属性を参照してください。

Lightning コンテナ

サードパーティフレームワークで開発したアプリケーションを静的リソースとしてアップロードし、lightning:container を使用して Lightning コンポーネントのコンテンツをホストします。

lightning:container を使用して、サードパーティフレームワーク (AngularJS、React など) を Lightning ページ内で使用します。

lightning:container コンポーネントは iframe 内のコンテンツをホストします。フレーム化されたアプリケーションとの通信を実装でき、Lightning コンポーネントとのやりとりが可能になります。lightning:container は message() メソッドを提供します。ユーザはこれを JavaScript コントローラ内で使用してアプリケーションにメッセージを送信できます。コンポーネントでは、onmessage 属性を使用して、メッセージを処理するメソッドを指定します。

このセクションの内容:

Lightning コンテナコンポーネントの制限

lightning:container の制限を理解します。

Lightning Realty アプリケーション

Lightning Realty (不動産) アプリケーションは、Lightning コンテナコンポーネントと Salesforce 間のメッセージングに関する堅牢な例です。

lightning-container NPM モジュールリファレンス

JavaScript コードで lightning-container NPM モジュールに含まれるメソッドを使用して、カスタム Lightning コンポーネントに対するメッセージの送受信を行ったり、Salesforce REST API とやりとりしたりします。

サードパーティフレームワークの使用

lightning:container を使用すると、サードパーティフレームワーク (AngularJS、React など) で開発されたアプリケーションを Lightning コンポーネントで使用できます。このアプリケーションを静的リソースとしてアップロードします。

アプリケーションには起動ページが必要であり、このページは、lightning:container src 属性で指定します。通常、起動ページは index.html ですが、マニフェストファイルを静的リソースに追加することで別の起動ページを指定できます。次の例は、index.html の起動ページを使用する myApp (静的リソースとしてアップロードされたアプリケーション) を参照する簡単な Lightning コンポーネントを示しています。

```
<aura:component>
    <lightning:container src="{!$Resource.myApp + '/index.html'}" />
</aura:component>
```

静的リソースのコンテンツはユーザ次第です。静的リソースには、アプリケーションを構成する JavaScript、関連付けられたすべてのアセット、および起動ページが含まれている必要があります。

他の Lightning コンポーネントと同様に、カスタム属性を指定できます。この例は、同じ静的リソース (myApp) を参照し、3 つの属性 (messageToSend、messageReceived、error) を使用します。このコンポーネントには implements="flexipage:availableForAllPageTypes" が含まれるため、Lightning アプリケーションビルダでこのコンポーネントを使用して Lightning ページに追加できます。

メモ: このセクションの例は、Developerforce Github リポジトリで入手できます。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >
    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/>
        <lightning:button label="Send" onclick="{!c.sendMessage}"/>
        <br/>
       <lightning:textarea value="{!v.messageReceived}" label="Message received from React
 app: "/>
        <br/>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>
</aura:component>
```

コンポーネントには lightning:input 要素が含まれており、これによりユーザは messageToSend の値を入力できます。ユーザが [Send (送信)] を押すと、コンポーネントはコントローラメソッド sendMessage をコールします。このコンポーネントは、メッセージとエラーを処理するためのメソッドも提供します。

このスニペットにコンポーネントのコントローラや他のコードは含まれませんが、心配はいりません。メッセージとエラー処理の詳細と実装方法については、「Lightning コンテナコンポーネントからのメッセージの送信」および「コンテナのエラーの処理」で説明します。

関連トピック:

Lightning コンテナ

Lightning コンテナコンポーネントからのメッセージの送信

コンテナのエラーの処理

Lightning コンテナコンポーネントからのメッセージの送信

lightning:container の onmessage 属性を使用して、コンポーネントのコンテンツ (つまり、組込みアプリケーション) との間のメッセージを処理するメソッドを指定します。lightning:container のコンテンツは iframe 内でラップされます。この方法によって、フレームの境界を超える通信ができます。

この例は、lightning:container を含み、3 つの属性 (messageToSend、messageReceived、error) を持つ Lightning コンポーネントを示しています。

この例は、「サードパーティフレームワークの使用」の例と同じコードを使用します。この例の完全版は、Developerforce GitHub リポジトリからダウンロードできます。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >
    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/>
        <lightning:button label="Send" onclick="{!c.sendMessage}"/>
        <br/>
       <lightning:textarea value="{!v.messageReceived}" label="Message received from React
 app: "/>
        <br/>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>
</aura:component>
```

messageToSend は、フレーム化されたアプリケーションに Salesforce から送信されたメッセージを表し、messageReceived は、アプリケーションが Lightning コンポーネントに送信したメッセージを表します。lightning:container には、必須の src 属性と、aura:id および onmessage 属性が含まれます。onmessage 属性には、JavaScript コントローラのメッセージ処理メソッドを指定し、aura:id を使用してこのメソッドでコンポーネントを参照できます。

この例は、コンポーネントの JavaScript コントローラを示しています。

```
({
    sendMessage : function(component, event, helper) {

        var msg = {
            name: "General",
            value: component.get("v.messageToSend")
        };
        component.find("ReactApp").message(msg);
    },
```

```
    handleMessage: function(component, message, helper) {
        var payload = message.getParams().payload;
        var name = payload.name;
        if (name === "General") {
            var value = payload.value;
            component.set("v.messageReceived", value);
        }
        else if (name === "Foo") {
            // A different response
        }
    },

    handleError: function(component, error, helper) {
        var e = error;
    }
})
```

このコードはいくつかの異なる処理を実行します。sendMessage アクションでは、囲んでいる Lightning コンポーネントから組み込みアプリケーションにメッセージを送信します。これは、name と value を含む JSON 定義を持つ変数 msg を作成します。このメッセージの定義はユーザが定義します。つまり、メッセージのペイロードを値や、構造化された JSON 応答などにすることができます。Lightning コンポーネントの messageToSend 属性はメッセージの value に設定されます。次に、メソッドはコンポーネントの aura:id と message() 関数を使用して、メッセージを Lightning コンポーネントに送り返します。

handleMessage メソッドは、組み込みアプリケーションからメッセージを受信して適切に処理します。これは、コンポーネント、メッセージ、およびヘルパーを引数として取ります。このメソッドは条件ロジックを使用してメッセージを解析します。これが、name と value を持つ予期されたメッセージの場合、メソッドは Lightning コンポーネントの messageReceived 属性をメッセージの value に設定します。このコードは 1 つのメッセージのみを定義しますが、条件ステートメントを使用すると、sendMessage メソッドで定義されたさまざまな種別のメッセージを処理できます。

メッセージを送受信するためのハンドラコードは複雑になる可能性があります。Lightning コンポーネント、そのコントローラ、およびアプリケーション間のメッセージのフローを理解することをお勧めします。ユーザが messageToSend 属性としてメッセージを入力すると、処理が開始されます。ユーザが **[Send (送信)]** をクリックすると、コンポーネントは sendMessage をコールします。sendMessage はメッセージペイロードを定義し、message() メソッドを使用してメッセージペイロードをアプリケーションに送信します。アプリケーションを定義する静的リソース内で、指定されたメッセージハンドラ関数がメッセージを受信します。lightning-container モジュールの addMessageHandler() メソッドを使用して、JavaScript コード内でメッセージ処理関数を指定します。詳細は、「lightning-container NPM モジュールリファレンス」を参照してください。

lightning:container は、フレーム化されたアプリケーションからメッセージを受信すると、lightning:container の onmessage 属性に設定された、コンポーネントのコントローラの handleMessage メソッドをコールします。handleMessage メソッドはメッセージを取得し、その値を messageReceived 属性として設定します。最後に、コンポーネントは messageReceived を lightning:textarea に表示します。

これは、コンテナを横断するメッセージ処理の簡単な例です。アプリケーションのコントローラ側のコードと機能を実装するため、Salesforce と、`lightning:container` に埋め込まれたアプリケーション間のあらゆる種類の通信でこの機能を使用できます。

関連トピック:

Lightning コンテナ

サードパーティフレームワークの使用

コンテナのエラーの処理

Lightning コンテナコンポーネントへのメッセージの送信

lightning-container NPM モジュール内のメソッドを使用して、`lightning:container` によりフレーム化された JavaScript コードからメッセージを送信します。

Lightning-container NPM モジュールは、JavaScript アプリケーションと Lightning コンテナコンポーネント間でメッセージを送受信するためのメソッドを提供します。lightning-container モジュールについては、NPM の Web サイトを参照してください。

lightning-container モジュールを連動関係としてコードに追加し、アプリケーションにメッセージングフレームワークを実装します。

```
import LCC from 'lightning-container';
```

アプリケーションの `package.json` ファイルにも `lightning-container` を連動関係として含める必要があります。

アプリケーションから `lightning:container` にメッセージを送信するコードは簡単です。このコードは、「Lightning コンテナコンポーネントからのメッセージの送信」および「コンテナのエラーの処理」のコードサンプルに対応し、Developerforce Github リポジトリからダウンロードできます。

```
sendMessage() {
  LCC.sendMessage({name: "General", value: this.state.messageToSend});
}
```

静的リソースの一部であるこのコードは、ユーザにより定義された名前と値を含むオブジェクトとしてメッセージを送信します。

アプリケーションがメッセージを受信すると、メッセージは、`addMessageHandler()` メソッドによりマウントされた関数で処理されます。React アプリケーションでは、関数をドキュメント-オブジェクトモデルの一部としてマウントし、出力に表示する必要があります。

lightning-container モジュールでは、メッセージフレームワーク内のエラーを処理する関数を定義するための同様のメソッドが提供されます。詳細は、「lightning-container NPM モジュールリファレンス」を参照してください。

コンテナのエラーの処理

コンポーネントのコントローラでメソッドを使用して Lightning コンテナのエラーを処理します。

この例は、「サードパーティフレームワークの使用」および「Lightning コンテナコンポーネントからのメッセージの送信」の例と同じコードを使用します。

このコンポーネントでは、lightning:container の onerror 属性で handleError をエラー処理メソッドとして指定します。エラーを表示するために、コンポーネントのマークアップは条件ステートメントと、エラーメッセージを保持するための別の属性 (error) を使用します。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >

    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/><lightning:button label="Send" onclick="{!c.sendMessage}"/>

        <br/>

        <lightning:textarea name="messageReceived" value="{!v.messageReceived}"
label="Message received from React app: "/>

        <br/>

        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorMessage" value="{!v.error}" label="Error: "/>
        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>

</aura:component>
```

コンポーネントのコントローラを次に示します。

```
({
    sendMessage : function(component, event, helper) {

        var msg = {
            name: "General",
            value: component.get("v.messageToSend")
        };
        component.find("ReactApp").message(msg);
    },

    handleMessage: function(component, message, helper) {
        var payload = message.getParams().payload;
        var name = payload.name;
        if (name === "General") {
            var value = payload.value;
            component.set("v.messageReceived", value);
        }
        else if (name === "Foo") {
            // A different response
        }
```

```
    },

    handleError: function(component, error, helper) {
        var description = error.getParams().description;
        component.set("v.error", description);
    }
})
```

Lightning コンテナアプリケーションがエラーが発生すると、エラー処理関数は `error` 属性を設定します。次に、コンポーネントのマークアップ内の条件式で error 属性が空かどうかを確認します。空でない場合、コンポーネントは、`error` に保存されたエラーメッセージを `lightning:textarea` 要素に設定します。

関連トピック:

Lightning コンテナ

サードパーティフレームワークの使用

Lightning コンテナコンポーネントからのメッセージの送信

CSP レベルの指定

`lightning:container` のコンテンツセキュリティポリシー (CSP) を指定して、アプリケーション開発時の柔軟性を高めます。

CSPは、クロスサイトスクリプティングやデータインジェクション攻撃など、特定の攻撃の防止に役立つセキュリティの追加レイヤです。CSPヘッダーでは、Webページの特定の要素(画像、動画、またはその他のメディアなど) が指定の外部ドメインセットから読み込まれることを許可するポリシーを指定します。

`lightning:container` によって参照される静的リソースに `manifest.json` ファイルを追加して、アプリケーションの CSP レベルとランディングページを指定します。`manifest.json` ファイルは、アプリケーション内のページのJSON配列です(省略可能)。マニフェストファイルを静的リソースに含めない場合、アプリケーションのランディングページの名前を `index.html` にする必要があります。

次の例では、`manifest.json` に `index.html`、`foo.html`、`bar.html` の 3 つのページが含まれています。

```
{
    "landing-pages" : [
        {
            "path": "index.html",
            "content-security-policy-type": "high"
        },
        {
            "path": "foo.html",
            "content-security-policy-type": "low"
        },
        {
            "path": "bar.html",
            "content-security-policy-type": "custom"
            "content-security-policy": "default-src *;"
        },
    ]
 }
```

アプリケーションの各ページには、high、low、custom の CSP レベルが設定されています。

CSP レベル	説明
High (高)	Lightning ドメインのコンテンツのみを読み込むことができます。この値では、セキュリティが最も高くなります。
Low (低)	`default-src self 'unsafe-eval'; style-src self 'unsafe-inline';` デフォルト。この最小 CSP には、Lightning コンテナの上位 iframe を制限する frame-ancestors 属性も含まれます。
Custom (カスタム)	ユーザが指定します。CSP ヘッダーでは、default-src、img-src、media-src、script-src、およびその他のディレクティブを指定できます。CSP ヘッダーの構文や使用についての詳細は、『Content Security Policy Reference (コンテンツセキュリティポリシーリファレンス)』を参照してください。

lightning:container のデフォルト CSP は low です。manifest.json ファイルを静的リソースに含めない場合、Lightning の CSP レベルが適用されます。manifest.json ファイルを含めても、いずれかのページに content-security-policy-type を指定しないと、Lightning の CSP レベルが適用されます。

コンテンツの CSP ヘッダーには、クリックジャック保護を提供するために上位 iframe を制限する frame-ancestors 属性もあります。

関連トピック:

https://content-security-policy.com/

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

コンテナからの Apex サービスの使用

lightning-container NPM モジュールを使用して、Lightning コンテナコンポーネントから Apex メソッドをコールします。

lightning:container から Apex メソッドをコールするには、manifest.json ファイルで CSP レベルを low に設定します。CSP レベルが low になっていると、Lightning コンテナコンポーネントは Lightning ドメイン外のリソースを読み込むことができます。

次に、Apex サービスを使用する Lightning コンテナコンポーネントが含まれる Lightning コンポーネントを示します。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes">

    <aura:attribute access="private" name="error" type="String" default=""/>
```

```
    <div>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="/ApexController/index.html"
                             onerror="{!c.handleError}"/>
    </div>

</aura:component>
```

次に、コンポーネントのコントローラを示します。

```
({
    handleError: function(component, error, helper) {
        var description = error.getParams().description;
        component.set("v.error", description);
    }
})
```

メモ: この例の完全版は、Developerforce GitHub リポジトリからダウンロードできます。

コンポーネントの JavaScript コントローラではそれほど多くのアクションは実行されません。実際のアクションは JavaScript アプリケーションにあり、Lightning コンテナで参照される静的リソースとしてアップロードされます。

```
import React, { Component } from 'react';
import LCC from "lightning-container";
import logo from './logo.svg';
import './App.css';

class App extends Component {

  callApex() {
    LCC.callApex("lcc1.ApexController.getAccount",
                 this.state.name,
                 this.handleAccountQueryResponse,
                 {escape: true});
  }

  handleAccountQueryResponse(result, event) {
    if (event.status) {
      this.setState({account: result});
    }
    else if (event.type === "exception") {
      console.log(event.message + " : " + event.where);
    }
  }

  render() {
    var account = this.state.account;
```

```
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to LCC</h2>
        </div>
        <p className="App-intro">
          Account Name: <input type="text" id="accountName" value={this.state.name}
onChange={e => this.onAccountNameChange(e)}/><br/>
         <input type="submit" value="Call Apex Controller" onClick={this.callApex}/><br/>

          Id: {account.Id}<br/>
          Phone: {account.Phone}<br/>
          Type: {account.Type}<br/>
          Number of Employees: {account.NumberOfEmployees}<br/>
        </p>
      </div>
    );
  }

  constructor(props) {
    super(props);
    this.state = {
      name: "",
      account: {}
    };

    this.handleAccountQueryResponse = this.handleAccountQueryResponse.bind(this);
    this.onAccountNameChange = this.onAccountNameChange.bind(this);
    this.callApex = this.callApex.bind(this);
  }

  onAccountNameChange(e) {
    this.setState({name: e.target.value});
  }
}

export default App;
```

最初の関数 `callApex()` は、`LCC.callApex` メソッドを使用して、取引先の情報を取得して表示する Apex メソッド `getAccount` をコールします。

Lightning コンテナコンポーネントの制限

`lightning:container` の制限を理解します。

`lightning:container` には既知の制限があります。iframe の使用に関連する、パフォーマンスとスクロールの問題が発生する場合があります。このコンポーネントは複数ページモデル用に設計されていません。また、ブラウザのナビゲーション履歴とは統合されません。

`lightning:container` コンポーネントを表示しているページから別のページに移動した場合、コンポーネントはその状態を自動的に記憶しません。iframe 内のコンテンツは、Lightning Experience の残りと同じオフラインスキームおよびキャッシュスキームを使用しません。

関連トピック:

Lightning コンテナ

Lightning Realty アプリケーション

Lightning Realty (不動産) アプリケーションは、Lightning コンテナコンポーネントと Salesforce 間のメッセージングに関する堅牢な例です。

Lightning Realty アプリケーションのメッセージングフレームワークは、Lightning コンポーネントのコード、コンポーネントのハンドラ、および `lightning:container` で参照する静的リソースに依存します。Lightning コンテナコンポーネントは、Lightning コンポーネントの JavaScript コントローラーのメッセージ処理関数を参照します。メッセージ処理関数はソース JavaScript から送信されたメッセージを受け入れ、lightning-container NPM モジュールにより提供されるメソッドを使用します。

開発組織にこの例をインストールする手順については、「サンプルの Lightning Realty アプリケーションをインストールする」を参照してください。

まずは、Lightning コンポーネントを見てみましょう。Realty コンポーネントを定義するこのコードは単純ですが、これを使用して不動産アプリケーションの JavaScript で Salesforce と通信し、サンプルデータを読み込むことができます。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >

    <aura:attribute access="global" name="mainTitle" type="String" required="true"
default="My Properties"/>

    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <aura:if isTrue="{! !empty(v.messageReceived) }">
          <lightning:textarea name="messageReceivedTextArea" value="{!v.messageReceived}"
 label=" "/>
        </aura:if>

        <aura:if isTrue="{! !empty(v.error) }">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.Realty + '/index.html?mainTitle=' +
v.mainTitle}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>
```

```
</aura:component>
```

このコードは、「Lightning コンテナコンポーネントからのメッセージの送信」および「コンテナのエラーの処理」のコード例に似ています。

また、Lightning コンポーネントのコントローラおよびソース JavaScript にあるコードを使用して iframe アプリケーションで Salesforce と通信することもできます。ソースに含まれる PropertyHome.js 内で、Realty アプリケーションは LCC.sendMessage をコールします。このコードセグメントは、物件のリストを絞り込んだ後、選択された物件の住所、価格、市区町村、都道府県、郵便番号、および説明を含むメッセージを作成してコンテナに返します。

```
saveHandler(property) {
    let filteredProperty = propertyService.filterProperty(property);
    propertyService.createItem(filteredProperty).then(() => {
        propertyService.findAll(this.state.sort).then(properties => {
            let filteredProperties = propertyService.filterFoundProperties(properties);
            this.setState({addingProperty: false, properties:filteredProperties});
        });
        let message = {};
        message.address = property.address;
        message.price = property.price;
        message.city = property.city;
        message.state = property.state;
        message.zip = property.zip;
        message.description = property.description;
        LCC.sendMessage({name: "PropertyCreated", value: message});
    });
},
```

次に、JavaScript は名前-値のペアを使用して LCC.sendMessage をコールします。このコードは sendMessage メソッドを使用します。これは、lightning-container NPM モジュールにより提供されるメッセージング API の一部です。詳細は、「Lightning コンテナコンポーネントへのメッセージの送信」を参照してください。

アクションの最後の部分がコンポーネントのコントローラで handleMessage() 関数内で実行されます。

```
handleMessage: function(component, message, helper) {
        var payload = message.getParams().payload;
        var name = payload.name;
        if (name === "PropertyCreated") {
            var value = payload.value;
            var messageToUser;
            if (value.price > 1000000) {
                messageToUser = "Big Real Estate Opportunity in " + value.city + ", " +
value.state + " : $" + value.price;
            }
            else {
                messageToUser = "Small Real Estate Opportunity in " + value.city + ", " +
 value.state + " : $" + value.price;
            }
            var log = component.get("v.log");
            log.push(messageToUser);
            component.set("v.log", log);
```

```
        }
    },
```

この関数はメッセージを引数として取り、名前が "PropertyCreated"であることを確認します。これは、アプリケーションの JavaScript 内で LCC.sendMessage により設定された name と同じです。

この関数はメッセージペイロード(この場合は、物件を記述する JSON 配列)を取り、物件の価値を確認します。価値が 100 万ドルを超える場合、ユーザにメッセージを送信し、大規模な不動産商談が存在することを知らせます。それ以外の場合、ユーザにメッセージを返し、小規模の不動産商談が存在することを知らせます。

このセクションの内容:

サンプルの Lightning Realty アプリケーションをインストールする
lightning:container のその他の例が Developerforce Git リポジトリにあります。

サンプルの Lightning Realty アプリケーションをインストールする

lightning:container のその他の例が Developerforce Git リポジトリにあります。

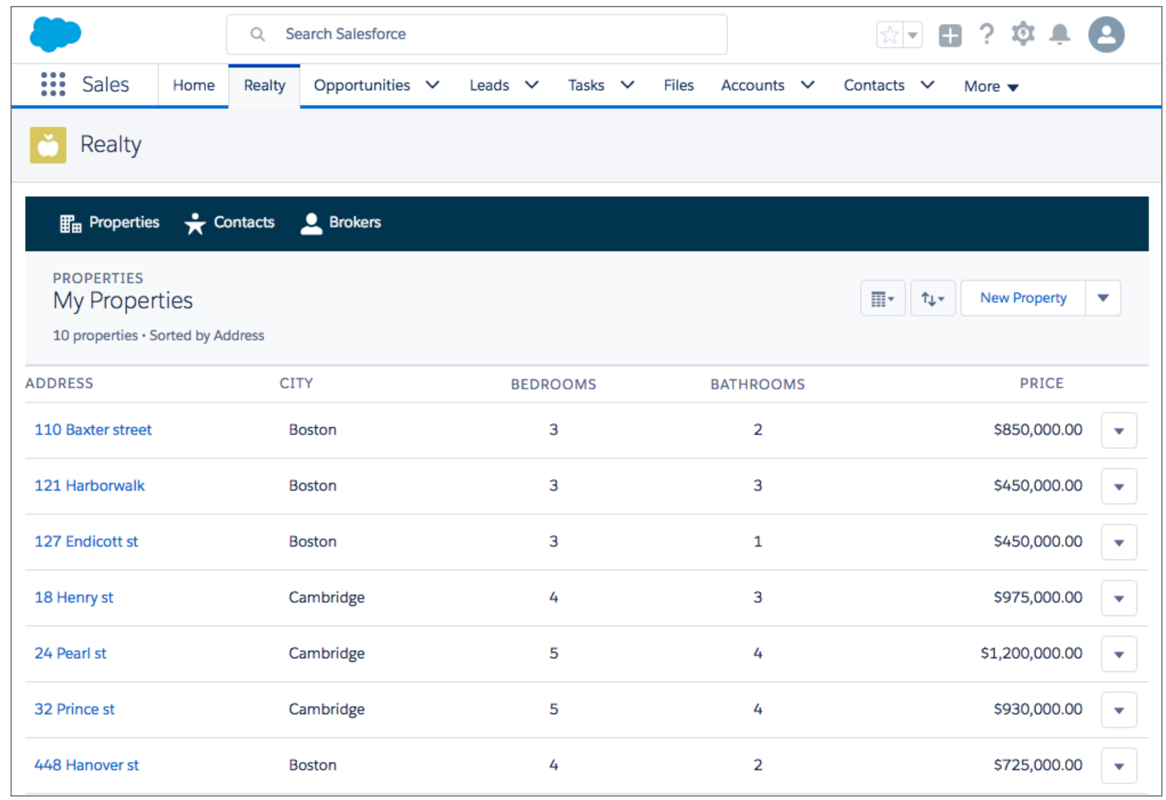
https://github.com/developerforce/LightningContainerExamples に含まれるコードを使用して lightning:container のより詳細な例を実装します。この例は、React と lightning:container を使用して、不動産リストアプリケーションを Lightning ページに表示します。

この例を実装するには、npm を使用します。npm の最も簡単なインストール方法は、node.js をインストールすることです。npm をインストールしたら、コマンドラインから npm install --save latest-version を実行して最新バージョンをインストールします。

カスタム Lightning コンポーネントを作成するには、組織で「私のドメイン」も有効にする必要があります。「私のドメイン」の詳細は、Salesforce ヘルプの「私のドメイン」を参照してください。

1. Git リポジトリをコピーします。コマンドラインから *git clone https://github.com/developerforce/LightningContainerExamples* と入力します。
2. コマンドラインから LightningContainerExamples/ReactJS/Javascript/Realty に移動し、npm install と入力してプロジェクトの連動関係を構築します。
3. コマンドラインから *npm run build* と入力してアプリケーションをビルドします。
4. package.json を編集し、指定された Salesforce ログイン情報を追加します。
5. コマンドラインから npm run deploy と入力します。
6. Salesforce にログインし、新しい Realty Lightning ページを Lightning アプリケーションに追加して、Lightning アプリケーションビルダー内でそのページを有効にします。
7. 組織にサンプルデータをアップロードするには、コマンドラインから npm run load と入力します。

組織で動作している Lightning Realty アプリケーションを確認します。このアプリケーションは lightning:container を使用して React アプリケーションを Lightning ページに埋め込んで、サンプルの不動産リストデータを表示します。



コンポーネントとハンドラコードは、「Lightning コンテナコンポーネントからのメッセージの送信」および「コンテナのエラーの処理」の例に似ています。

lightning-container NPM モジュールリファレンス

JavaScript コードで lightning-container NPM モジュールに含まれるメソッドを使用して、カスタム Lightning コンポーネントに対するメッセージの送受信を行ったり、Salesforce REST API とやりとりしたりします。

このセクションの内容:

addMessageErrorHandler()
メッセージングフレームワークでエラーが発生したときにコールするエラー処理関数をマウントします。

addMessageHandler()
メッセージ処理関数をマウントします。この関数を使用して Lightning コンポーネントから、フレーム化された JavaScript アプリケーションに送信されたメッセージを処理します。

callApex()
Apex コールを行います。

getRESTAPISessionKey()
Salesforce REST API セッションキーを返します。

removeMessageErrorHandler()
エラー処理関数をマウント解除します。

removeMessageHandler()
メッセージ処理関数をマウント解除します。

sendMessage()
フレーム化された JavaScript コードから Lightning コンポーネントにメッセージを送信します。

addMessageErrorHandler()

メッセージングフレームワークでエラーが発生したときにコールするエラー処理関数をマウントします。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、メッセージエラー処理関数をマウントします。React アプリケーションでは、関数をドキュメント-オブジェクトモデルの一部としてマウントし、出力に表示する必要があります。

```
componentDidMount() {
  LCC.addMessageErrorHandler(this.onMessageError);
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
handler: (errorMsg: string) => void)	function	メッセージングフレームワークで発生したエラーメッセージを処理する関数。

応答

なし。

addMessageHandler()

メッセージ処理関数をマウントします。この関数を使用して Lightning コンポーネントから、フレーム化された JavaScript アプリケーションに送信されたメッセージを処理します。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、メッセージ処理関数をマウントします。React アプリケーションでは、関数をドキュメント-オブジェクトモデルの一部としてマウントし、出力に表示する必要があります。

```
componentDidMount() {
    LCC.addMessageHandler(this.onMessage);
}
```

```
onMessage(msg) {
  let name = msg.name;
  if (name === "General") {
    let value = msg.value;
    this.setState({messageReceived: value});
  }
  else if (name === "Foo") {
    // A different response
  }
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
handler: (userMsg: any) => void	function	Lightning コンポーネントから送信されたメッセージを処理する関数。

応答

なし。

callApex()

Apex コールを行います。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、Apex メソッド getAccount をコールします。

```
  callApex() {
    LCC.callApex("lcc1.ApexController.getAccount",
                 this.state.name,
                 this.handleAccountQueryResponse,
                 {escape: true});
  }
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
fullyQualifiedApexMethodName	string	Apex メソッドの名前。
apexMethodParameters	array	Apex メソッドの引数の JSON 配列。

名前	型	説明
callbackFunction	function	コールバック関数。
apexCallConfiguration	array	Apex コールの設定パラメータ。

応答

なし。

getRESTAPISessionKey()

Salesforce REST API セッションキーを返します。

SOQL クエリを実行するなど、組み込みアプリケーションで Salesforce REST API とやり取りする必要がある場合、このメソッドを使用します。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、REST API セッションキーを取得し、それを使用して SOQL クエリを実行します。

```
componentDidMount() {
  let sid = LCC.getRESTAPISessionKey();
  let conn = new JSForce.Connection({accessToken: sid});
  conn.query("SELECT Id, Name from Account LIMIT 50", this.handleAccountQueryResponse);
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

なし。

応答

名前	型	説明
key	string	REST API セッションキー。

removeMessageErrorHandler()

エラー処理関数をマウント解除します。

React を使用する場合、関数をマウント解除して DOM から削除し、必要なクリーンアップを実行する必要があります。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、メッセージエラー処理関数をマウント解除します。React アプリケーションでは、関数をドキュメント-オブジェクトモデルの一部としてマウントし、出力に表示する必要があります。

```
componentWillUnmount() {
  LCC.removeMessageErrorHandler(this.onMessageError);
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
handler: (errorMsg: string) => void	function	メッセージングフレームワークで発生したエラーメッセージを処理する関数。

応答

なし。

removeMessageHandler()

メッセージ処理関数をマウント解除します。

React を使用する場合、関数をマウント解除して DOM から削除し、必要なクリーンアップを実行する必要があります。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、メッセージ処理関数をマウント解除します。

```
componentWillUnmount() {
  LCC.removeMessageHandler(this.onMessage);
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
handler: (userMsg: any) => void	function	Lightning コンポーネントから送信されたメッセージを処理する関数。

応答

なし。

sendMessage()

フレーム化された JavaScript コードから Lightning コンポーネントにメッセージを送信します。

サンプル

静的リソースとしてアップロードされ、lightning:container により参照される JavaScript アプリケーション内でこの例を使用して、アプリケーションから lightning:container にメッセージを送信します。

```
sendMessage() {
  LCC.sendMessage({name: "General", value: this.state.messageToSend});
}
```

この例は、Developerforce Github リポジトリで参照およびダウンロードできます。

引数

名前	型	説明
userMsg	any	メッセージ内で送信されるデータをユーザは完全に制御できますが、通常、そのデータは、名前項目と値項目を持つオブジェクトです。

応答

なし。

アクセスの制御

このフレームワークでは、access システム属性を介して、アプリケーション、属性、コンポーネント、イベント、インターフェース、メソッドへのアクセスを制御できます。access システム属性は、リソースをその名前空間外で使用できるかどうかを示します。

access システム属性は次のタグで使用します。

- <aura:application>
- <aura:attribute>
- <aura:component>
- <aura:event>
- <aura:interface>
- <aura:method>

アクセス値

access システム属性には、次の値を指定できます。

private
コンポーネント、アプリケーション、インターフェース、イベント、メソッド内では使用できますが、リソース外からは参照できません。この値は、<aura:attribute> または <aura:method> にのみ使用できます。

属性を非公開とマークすると、属性をリソース内でのみ使用可能になるため、その後属性のリファクタリングが容易になります。

非公開属性にアクセスすると、宣言したコンポーネントから参照した場合を除き、undefined が返されます。非公開属性が含まれるコンポーネントを拡張するサブコンポーネントから非公開属性にアクセスすることはできません。

public
組織内でのみ使用できます。これはデフォルトのアクセス値です。

global
すべての組織で使用できます。

メモ: コンポーネントなどのリソースを access="global" としてマークし、リソースを自分の組織外で使用できるようにします。たとえば、インストール済みパッケージで、または他の組織のLightning アプリケーションビルダーユーザまたはコミュニティビルダーユーザが、コンポーネントを使用できるようにする場合などです。

例

次のサンプルコンポーネントにはグローバルアクセス権があります。

```
<aura:component access="global">
    ...
</aura:component>
```

アクセス違反

リソースへのアクセスを許可する access システム属性のないコンポーネントなどのリソースにコードがアクセスすると、次のようになります。

- クライアント側のコードは実行されないか、undefined を返します。デバッグモードを有効にした場合は、ブラウザコンソールにエラーメッセージが表示されます。
- サーバ側のコードは、コンポーネントの読み込みに失敗します。デバッグモードを有効にした場合は、ポップアップエラーメッセージが表示されます。

アクセス権チェックエラーメッセージの構造

アクセス違反のアクセス権チェックエラーメッセージのサンプルを次に示します。

```
Access  Check  Failed ! ComponentService.getDef():'markup://c:targetComponent' is not
visible to 'markup://c:sourceComponent'.
```

エラーメッセージは、次の 4 つの部分で構成されます。

1. コンテキスト (誰がリソースにアクセスしようとしているか)。この例では、`markup://c:sourceComponent` です。
2. 対象 (アクセスされているリソース)。この例では、`markup://c:targetComponent` です。
3. エラーの種類。この例では、`not visible` です。
4. エラーをトリガしたコード。通常はクラスメソッドです。この例では、対象定義 (コンポーネント) にアクセスできなかったことを示す `ComponentService.getDef()` です。定義には、コンポーネントなどのリソースのメタデータが記述されます。

アクセス権チェックエラーの修正

ヒント: コードが期待どおりに動作しない場合は、デバッグモードを有効にしてより詳細なエラーレポートを取得します。

アクセス権チェックエラーは、次の 1 つ以上の方法を使用して修正できます。

- 所有するリソースに適切な `access` システム属性を追記する。
- 使用できないリソースへの参照をコードから削除する。前の例では `markup://c:targetComponent` に `markup://c:sourceComponent` からのアクセスを許可するアクセス値はありません。
- `<aura:attribute>` 定義を参照して、アクセスしている属性が存在することを確認する。`name` の大文字小文字の区別も含め正しいスペルを使用していることを確認します。

  未定義属性にアクセスした場合も非公開属性などの範囲外の属性にアクセスした場合も、同じアクセス違反メッセージがトリガされます。アクセスコンテキストでは属性が未定義であるかアクセスできないかを判断できません。

例: `is not visible to 'undefined'`

```
ComponentService.getDef():'markup://c:targetComponent' is not visible to 'undefined'
```

このエラーメッセージのキーワードは、フレームワークがコンテキストを失ったことを示す `undefined` です。これは、`setTimeout()` または `setInterval()` コールや ES6 Promise など、通常のフレームワークライフサイクル外でコードがコンポーネントにアクセスすると発生します。

このエラーを修正するには、コードを `$A.getCallback()` コールでラップトップします。詳細は、「フレームワークのライフサイクル外のコンポーネントの変更」を参照してください。

例: Cannot read property 'Yb' of undefined

```
Action failed: c$sourceComponent$controller$doInit [Cannot read property 'Yb' of undefined]
```

このエラーメッセージは、値が undefined である変数のプロパティを参照した場合に表示されます。このエラーはさまざまな状況で発生する可能性がありますが、その一例がアクセス権チェックに失敗した場合の副次的影響です。たとえば、JavaScript で未定義属性である imaginaryAttribute にアクセスしようとするとどうなるか見てみましょう。

```
var whatDoYouExpect = cmp.get("v.imaginaryAttribute");
```

これはアクセス権チェックエラーで、whatDoYouExpect が undefined に設定されています。この場合に、whatDoYouExpect のプロパティにアクセスしようとすると、エラーが表示されます。

```
Action failed: c$sourceComponent$controller$doInit [Cannot read property 'Yb' of undefined]
```

エラーメッセージの c$sourceComponent$controller$doInit の部分は、エラーが c 名前空間にある sourceComponent コンポーネントのコントローラの doInit メソッドにあることを示しています。

このセクションの内容:

アプリケーションのアクセス制御

aura:application タグの access 属性は、アプリケーションをその名前空間外で使用できるかどうかを制御します。

インターフェースのアクセス制御

aura:interface タグの access 属性は、インターフェースをその名前空間外で使用できるかどうかを制御します。

コンポーネントのアクセス制御

aura:component タグの access 属性は、コンポーネントをその名前空間外で使用できるかどうかを制御します。

属性のアクセス制御

aura:attribute タグの access 属性は、属性をその名前空間外で使用できるかどうかを制御します。

イベントのアクセス制御

aura:event タグの access 属性は、イベントをその名前空間外で使用できるかどうかを制御します。

関連トピック:

Lightning コンポーネントのデバッグモードの有効化

アプリケーションのアクセス制御

aura:application タグの access 属性は、アプリケーションをその名前空間外で使用できるかどうかを制御します。

使用できる値は次のとおりです。

修飾子	説明
public	組織内でのみ使用できます。これはデフォルトのアクセス値です。
global	すべての組織で使用できます。

インターフェースのアクセス制御

aura:interface タグの access 属性は、インターフェースをその名前空間外で使用できるかどうかを制御します。

使用できる値は次のとおりです。

修飾子	説明
public	組織内でのみ使用できます。これはデフォルトのアクセス値です。
global	すべての組織で使用できます。

コンポーネントは aura:component タグの implements 属性を使用してインターフェースを実装できます。

コンポーネントのアクセス制御

aura:component タグの access 属性は、コンポーネントをその名前空間外で使用できるかどうかを制御します。

使用できる値は次のとおりです。

修飾子	説明
public	組織内でのみ使用できます。これはデフォルトのアクセス値です。
global	すべての組織で使用できます。

メモ: コンポーネントを URL で直接アドレス指定することはできません。コンポーネントの出力を確認するには、コンポーネントを .app リソースに埋め込みます。

属性のアクセス制御

aura:attribute タグの access 属性は、属性をその名前空間外で使用できるかどうかを制御します。

使用できる値は次のとおりです。

アクセス	説明
private	コンポーネント、アプリケーション、インターフェース、イベント、メソッド内では使用できますが、リソース外からは参照できません。 メモ: 非公開属性にアクセスすると、宣言したコンポーネントから参照した場合を除き、undefined が返されます。非公開属性が含まれるコンポーネントを拡張するサブコンポーネントから非公開属性にアクセスすることはできません。
public	組織内でのみ使用できます。これはデフォルトのアクセス値です。
global	すべての組織で使用できます。

イベントのアクセス制御

aura:event タグの access 属性は、イベントをその名前空間外で使用できるかどうかを制御します。

使用できる値は次のとおりです。

修飾子	説明
public	組織内でのみ使用できます。これはデフォルトのアクセス値です。
global	すべての組織で使用できます。

オブジェクト指向開発の使用

フレームワークは、オブジェクト指向プログラミングから継承およびカプセル化の基本概念を提供し、それをプレゼンテーションレイヤの開発に適用します。

たとえば、コンポーネントはカプセル化され、その内部は非公開のまま保持されます。コンポーネントのコンシューマは、コンポーネントの公開形状(属性および登録済みイベント)にアクセスできますが、コンポーネントバンドルの他の実装の詳細にはアクセスできません。この強固な分離により、コンポーネント作成者は自由に内部実装の詳細を変更することができ、コンポーネントのコンシューマはこうした変更から隔離されます。

コンポーネント、アプリケーション、インターフェースを拡張したり、コンポーネントインターフェースを実装したりできます。

継承とは?

このトピックでは、コンポーネントなど定義を拡張したときに継承されるものについて説明します。

このドキュメントでは、コンポーネントに別のコンポーネントが含まれている場合に、コンテインメント階層の親コンポーネントおよび子コンポーネントについて説明します。コンポーネントで別のコンポーネントを拡張する場合には、継承階層のサブコンポーネントおよびスーパーコンポーネントについて説明します。

コンポーネントの属性

スーパーコンポーネントを拡張するサブコンポーネントは、スーパーコンポーネントの属性を継承します。サブコンポーネントのマークアップで `<aura:set>` を使用して、スーパーコンポーネントから継承された属性の値を設定します。

イベント

スーパーコンポーネントを拡張するサブコンポーネントは、スーパーコンポーネントによって起動されたイベントを処理できます。サブコンポーネントは、スーパーコンポーネントからイベントハンドラを自動的に継承します。

サブコンポーネントに `<aura:handler>` タグを追加すると、スーパーコンポーネントとサブコンポーネントが同じイベントを異なる方法で処理できます。フレームワークは、イベント処理の順序を保証しません。

ヘルパー

サブコンポーネントのヘルパーは、そのスーパーコンポーネントのヘルパーからメソッドを継承します。サブコンポーネントは、メソッドを継承されたメソッドと同じ名前で定義して、スーパーコンポーネントのヘルパーメソッドを上書きできます。

コントローラ

スーパーコンポーネントを拡張するサブコンポーネントは、スーパーコンポーネントのクライアント側のコントローラでアクションをコールできます。たとえば、スーパーコンポーネントに `doSomething` というアクションがある場合、サブコンポーネントは、`{!c.doSomething}` 構文を使用してこのアクションを直接コールできます。

メモ: クライアント側のコントローラの継承は、コンポーネントのカプセル化の改良を維持するために今後廃止される可能性があるため、この機能の使用はお勧めしません。代わりに、ヘルパーに一般的なコードを配置することをお勧めします。

関連トピック:

コンポーネントの属性
イベントとの通信
コンポーネントのバンドル内の JavaScript コードの共有
クライアント側コントローラを使用したイベントの処理
aura:set

継承されるコンポーネントの属性

スーパーコンポーネントを拡張するサブコンポーネントは、スーパーコンポーネントの属性を継承します。

属性値は、どの拡張レベルでも同じです。`body` 属性の場合はこのルールに例外がありますが、これについては後で説明します。

簡単な例から始めましょう。c:super には、値が「Default description」の description 属性があります。

```
<!--c:super-->
<aura:component extensible="true">
    <aura:attribute name="description" type="String" default="Default description" />

    <p>super.cmp description: {!v.description}</p>

    {!v.body}
</aura:component>
```

{!v.body} 式についてはまだ心配しないでください。これについては、body 属性を取り扱うときに説明します。

c:sub は、<aura:component> タグで extends="c:super" を設定することによって c:super を拡張します。

```
<!--c:sub-->
<aura:component extends="c:super">
    <p>sub.cmp description: {!v.description}</p>
</aura:component
```

sub.cmp には、継承される description 属性へのアクセス権があり、その値は sub.cmp および super.cmp と同じです。

継承される属性の値を設定するには、サブコンポーネントのマークアップで <aura:set> を使用します。

継承される body 属性

すべてのコンポーネントは <aura:component> から body 属性を継承します。body の継承動作は、他の属性とは異なります。コンポーネントの拡張レベルごとに異なる値を指定して、継承チェーンのコンポーネントごとに異なる出力が可能です。例を見てみると、この点が明確になります。

別のタグで囲まれていない独立したマークアップは、body の一部とみなされます。これは、その独立したマークアップを <aura:set attribute="body"> 内にラップするのと同じです。

コンポーネントのデフォルトのレンダラは、その body 属性を反復処理し、すべてを表示し、表示データをスーパーコンポーネントに渡します。スーパーコンポーネントは、{!v.body} をマークアップに含めることによって、渡されたデータを出力できます。スーパーコンポーネントが存在しない場合は、ルートコンポーネントに達しているため、データが document.body に挿入されています。

簡単な例を使用して、body 属性がコンポーネントのさまざまな拡張レベルでどのように動作するかを確認してみましょう。次の 3 つのコンポーネントがあります。

c:superBody はスーパーコンポーネントです。これは本質的に <aura:component> を拡張します。

```
<!--c:superBody-->
<aura:component extensible="true">
    Parent body: {!v.body}
</aura:component>
```

この時点では、{!v.body} が c:superBody を拡張するコンポーネントによって渡されるデータのプレースホルダにすぎないため、c:superBody では何も出力されません。

c:subBody は、<aura:component> タグで extends="c:superBody" を設定することによって c:superBody を拡張します。

```
<!--c:subBody-->
<aura:component extends="c:superBody">
    Child body: {!v.body}
</aura:component>
```

c:subBody の出力は、次のようになります。

```
Parent body: Child body:
```

つまり、c:subBody は、{!v.body} の値をスーパーコンポーネント c:superBody で設定します。

c:containerBody には、c:subBody への参照が含まれます。

```
<!--c:containerBody-->
<aura:component>
    <c:subBody>
        Body value
    </c:subBody>
</aura:component>
```

c:containerBody で、c:subBody の body 属性を Body value に設定します。c:containerBody の出力は、次のようになります。

```
Parent body: Child body: Body value
```

関連トピック:

aura:set

コンポーネントのボディ

コンポーネントのマークアップ

抽象コンポーネント

Java などのオブジェクト指向言語では、オブジェクトを部分的に実装して残りの実装を具体的なサブクラスに残すという抽象クラスの概念がサポートされています。Java では抽象クラスを直接インスタンス化することはできませんが、非抽象サブクラスをインスタンス化することができます。

同様に、Lightning コンポーネントフレームワークでも、部分的に実装して残りの実装を具体的なサブコンポーネントに残すという抽象コンポーネントの概念がサポートされています。

抽象コンポーネントを使用するには、それを拡張して残りの実装を入力する必要があります。抽象コンポーネントをマークアップで直接使用することはできません。

<aura:component> タグには、Boolean の abstract 属性があります。abstract="true" に設定して、コンポーネントを抽象にします。

関連トピック:

インターフェース

インターフェース

Javaなどのオブジェクト指向言語では、一連のメソッド署名を定義するインターフェースという概念がサポートされています。インターフェースを実装するクラスでは、メソッドの実装を提供する必要があります。Javaのインターフェースを直接インスタンス化することはできませんが、そのインターフェースを実装するクラスをインスタンス化することはできます。

同様に、Lightningコンポーネントフレームワークでは、属性を定義することでコンポーネントの形状を定義するインターフェースの概念がサポートされています。

インターフェースは、`<aura:interface>` タグで始まります。次のタグのみを含めることができます。

- インターフェースの属性を定義する `<aura:attribute>` タグ。
- 起動できるイベントを定義する `<aura:registerEvent>` タグ。

インターフェースでは、マークアップ、レンダラ、コントローラなどを使用できません。

インターフェースを使用するには、インターフェースを実装する必要があります。実装しないと、インターフェースをマークアップで直接使用できません。`<aura:component>` タグの `implements` システム属性を、実装するインターフェースの名前に設定します。次に例を示します。

```
<aura:component implements="mynamespace:myinterface" >
```

コンポーネントは、インターフェースを実装し、別のコンポーネントを拡張できます。

```
<aura:component extends="ns1:cmp1" implements="ns2:intf1" >
```

インターフェースは、カンマ区切りのリストを使用した複数のインターフェースを拡張できます。

```
<aura:interface extends="ns:intf1,ns:int2" >
```

メモ: スーパーコンポーネントから継承する属性の値を設定するには、サブコンポーネントで `<aura:set>` を使用します。この使用方法は、コンポーネントと抽象コンポーネントでは動作しますが、インターフェースでは動作しません。インターフェースから継承する属性の値を設定するには、サブコンポーネントで `<aura:attribute>` を使用して属性を再定義し、そのデフォルト属性の値を設定します。

抽象コンポーネントのコンテンツには制限が少ないため、インターフェースより一般的です。コンポーネントでは、複数のインターフェースを実装できますが、抽象コンポーネントは1つしか拡張できないため、一部の設計パターンではインターフェースのほうが便利です。

関連トピック:

インターフェースから継承される属性の設定

抽象コンポーネント

マーカーインターフェース

インターフェースを一連のコンポーネントでマーカーインターフェースとして実装することで、アプリケーションの特定の用途に合っているかどうかを簡単に識別できます。

JavaScriptで `myCmp.isInstanceOf("mynamespace:myinterface")` を使用して、コンポーネントでインターフェースを実装しているかどうかを判断できます。

継承ルール

次の表に、さまざまな要素の継承ルールを示します。

要素	拡張	実装	デフォルトの基本要素
component	拡張可能な 1 つのコンポーネント	複数のインターフェース	`<aura:component>`
app	拡張可能な 1 つのアプリケーション	N/A	`<aura:application>`
interface	カンマ区切りのリストを使用した複数のインターフェース (`extends="ns:intf1,ns:int2"`)	N/A	なし

関連トピック:

インターフェース

AppCache の使用

AppCacheのサポートは終了する予定です。ブラウザベンダーがAppCacheを廃止したため、Salesforceもそれに従いました。スタンドアロンアプリケーション(`.app` リソース)の `<aura:application>` タグの `useAppcache` 属性を削除してください。ブラウザベンダによる廃止が原因で生じるブラウザ間サポートの問題を回避するためです。

`<aura:application>` タグに `useAppcache` を現在設定していない場合、`useAppcache` のデフォルト値は `false` のため何もする必要はありません。

メモ: 詳細は、AppCache の概要を参照してください。

関連トピック:

aura:application

アプリケーションとコンポーネントの配布

ISV または Salesforce パートナーは、アプリケーションとコンポーネントをパッケージ化して、社外を含む、他の Salesforce ユーザおよび組織に配布できます。

アプリケーションとコンポーネントの公開とインストールは AppExchange で行います。アプリケーションまたはコンポーネントをパッケージに追加すると、他のコンポーネント、イベント、インターフェースなど、アプリケーションまたはコンポーネントで参照されるすべての定義バンドルは自動的に含まれます。アプリケーションまたはコンポーネントで参照されるカスタム項目、カスタムオブジェクト、リストビュー、ページレイアウト、Apexクラスも含まれます。ただし、カスタムオブジェクトをパッケージに追加する場合、そのカスタムオブジェクトを参照するアプリケーションおよびその他の定義バンドルは、明示的にパッケージに追加する必要があります。明示的にパッケージに追加する必要があるその他の連動関係には、次のものがあります。

- CSP 信頼済みサイト
- リモートサイトの設定

管理パッケージを使用すると、アプリケーションとその他のリソースが完全にアップグレード可能になります。管理パッケージを作成して操作するには、Developer Edition 組織を使用して名前空間プレフィックスを登録する必要があります。管理パッケージでは、コンポーネント名に名前空間プレフィックスを追加して、アプリケーションをインストールする組織での名前の競合を回避します。組織は、他の組織でダウンロードおよびインストールできる単一の管理パッケージを作成できます。管理パッケージからのインストール後、アプリケーションまたはコンポーネント名はロックされますが、次の属性は編集できます。

- API バージョン
- 説明
- 表示ラベル
- 言語
- マークアップ

定義バンドルの一部として含まれている Apex はいずれも、累積テストカバー率が少なくとも 75% である必要があります。パッケージを AppExchange にアップロードすると、すべてのテストが実行され、エラーがない状態で実行されていることが確認されます。テストは、パッケージがインストールされている場合にも実行されます。

パッケージ化と配布についての詳細は、『*ISVforce ガイド*』を参照してください。

関連トピック:

Apex コードのテスト

第 7 章　デバッグ

トピック:

- Lightning コンポーネントのデバッグモードの有効化
- Salesforce Lightning Inspector Chrome 拡張機能
- ログメッセージ

アプリケーションのデバッグに役立つと思われる基本的なツールと手法がいくつかあります。

クライアント側のコードをデバッグするには、Chrome デベロッパーツールを使用します。

- Windows および Linux でデベロッパーツールを開くには、Google Chrome ブラウザで Ctrl キーと Shift キーを押しながら「I」を押します。Mac の場合は、Option キーと Command キーを押しながら「I」を押します。
- コードのどの行で失敗しているのかをすばやく見つけるには、コードを実行する前に **[Pause on all exceptions (すべての例外で一時停止)]** オプションを有効にします。

Google Chrome での JavaScript のデバッグについての詳細は、「Google Chrome DevTools」の Web サイトを参照してください。

Lightning コンポーネントのデバッグモードの有効化

エディション

使用可能なエディション: Salesforce Classic および Lightning Experience

使用可能なエディション: **Contact Manager** Edition、**Group** Edition、**Professional** Edition、**Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、および **Developer** Edition

UI を使用して Lightning コンポーネントを作成するエディション: **Enterprise** Edition、**Performance** Edition、**Unlimited** Edition、**Developer** Edition、または Sandbox。

デバッグモードを有効化すると、Lightning コンポーネントで JavaScript コードをデバッグしやすくなります。

モードには、本番とデバッグの2種類があります。デフォルトでは、Lightning コンポーネントフレームワークは本番モードで実行されます。このモードはパフォーマンス向上のために最適化されています。Google Closure Compiler を使用して JavaScript コードを最適化し、サイズを最小化します。メソッド名とコードは大幅に難読化されます。

デバッグモードを有効化すると、JavaScript コードは最小化されず、容易にコードを読んでデバッグを行えるようになります。また、デバッグモードでは、一部の警告やエラーでより詳細な出力が追加されます。

重要: デバッグモードは、パフォーマンスに大きな影響があります。この設定は、組織内のすべてのユーザに影響します。そのため、Sandbox 組織と Developer Edition 組織でのみ使用することをお勧めします。本番組織で永続的にデバッグモードのままにしないでください。

組織のデバッグモードを有効化する手順は、次のとおりです。

1. [設定] から、[クイック検索] ボックスに「`Lightning コンポーネント`」と入力し、**[Lightning コンポーネント]** を選択します。
2. [デバッグモードを有効化] チェックボックスをオンにします。
3. **[保存]** をクリックします。

メモ: [管理パッケージに対して `LockerService` を有効にする] 設定では、管理パッケージからインストールされたコンポーネントで LockerService が適用されるかどうかを制御できます。このチェックボックスは、LockerService の重要な更新が有効化されている場合にのみ表示されます。

Salesforce Lightning Inspector Chrome 拡張機能

Salesforce Lightning Inspector は、コンポーネントツリーを移動して、コンポーネントの属性を調べ、コンポーネントのパフォーマンスのプロファイルを作成できる Google Chrome デベロッパーツール拡張機能です。この拡張機能を使用すると、イベントの起動や処理のシーケンスを容易に知ることもできます。

この拡張機能により、次のことが可能になります。

- アプリケーションのコンポーネントツリー内を移動し、コンポーネントおよび関連する DOM 要素を調べる。
- コンポーネント作成時間のグラフを見ることにより、パフォーマンスのボトルネックを識別する。
- 応答を監視および変更することにより、サーバとのやりとりをデバッグする。
- エラー状態やアクション応答の欠落をシミュレーションすることにより、アプリケーションのフォールトトレランスをテストする。
- イベント起動とアクション処理のシーケンスを追跡する。

このドキュメントでは、Google Chrome デベロッパーツールを十分に理解していることを前提としています。

このセクションの内容:

Salesforce Lightning Inspector のインストール
コンポーネントのパフォーマンスをデバッグしたり、そのプロファイルを作成したりできるように、Google Chrome デベロッパーツール拡張機能をインストールします。

Salesforce Lightning Inspector
Chrome拡張機能により、デベロッパーツールメニューに[Lightning] タブが追加されます。このタブを使用して、アプリケーションのさまざまな側面を調べることができます。

Salesforce Lightning Inspector のインストール

コンポーネントのパフォーマンスをデバッグしたり、そのプロファイルを作成したりできるように、Google Chrome デベロッパーツール拡張機能をインストールします。

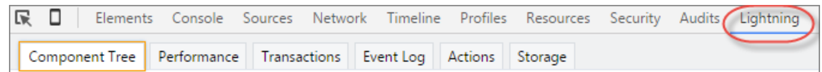
1. Google Chrome で、Chrome ウェブストアの Salesforce Lightning Inspector 拡張機能ページに移動します。
2. **[Chrome に追加]** ボタンをクリックします。

Salesforce Lightning Inspector

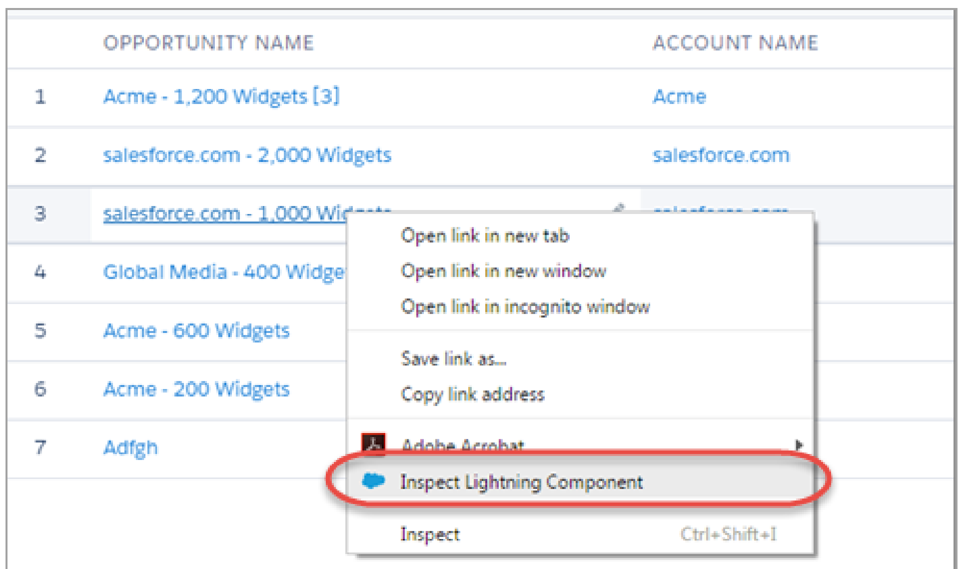
Chrome拡張機能により、デベロッパーツールメニューに[Lightning] タブが追加されます。このタブを使用して、アプリケーションのさまざまな側面を調べることができます。

1. Lightning Experience (`one.app`) などの Lightning コンポーネントを含むページに移動します。
2. Chrome デベロッパーツールを開きます (Chrome のコントロールメニューの [その他のツール] > [デベロッパーツール])。

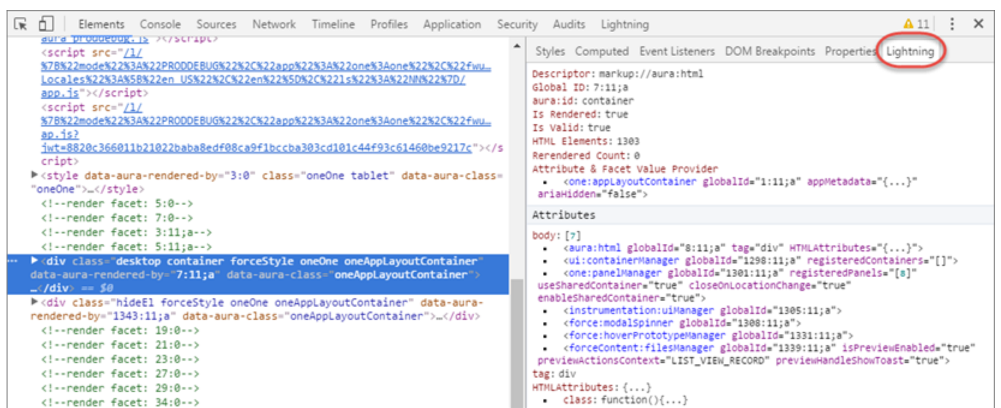
   デベロッパーツールメニューに [Lightning] タブが表示されます。

   Lightning ページの要素に関する情報をすばやく取得するには、要素を右クリックして **[Lightning コンポーネントを検査]** を選択します。



また、[DevTools Elements (デベロッパーツール要素)] タブの [Lightning] コンポーネント、または `data-aura-rendered-by` 属性が設定された要素をクリックして、説明および属性を確認することもできます。



次のサブタブを使用して、アプリケーションのさまざまな側面を調べることができます。

このセクションの内容:

[Component Tree (コンポーネントツリー)] タブ

このタブには、ネストされたコンポーネントのツリーが含まれるコンポーネントのマークアップが表示されます。

[Performance (パフォーマンス)] タブ

[Performance (パフォーマンス)] タブには、コンポーネントの作成時間のフレームグラフが表示されます。グラフの長い部分や深い部分を確認して、パフォーマンスの潜在的なボトルネックを特定します。

[トランザクション] タブ

Salesforce が提供するアプリケーションには、トランザクションマーカーが含まれているものがあります。トランザクションマーカーを使用すると、トランザクション内のアクションの細かい総計値を表示できます。独自のトランザクションを作成することはできません。

[Event Log (イベントログ)] タブ

このタブには、起動されたすべてのイベントが表示されます。イベントグラフでは、1つ以上のアクションのイベントおよびハンドラのシーケンスを分かりやすく表示します。

[Actions (アクション)] タブ

このタブには、実行されたサーバ側のアクションが表示されます。このリストはページの更新時に自動的に更新されます。

[Storage (ストレージ)] タブ

このタブには、Lightning アプリケーションのクライアント側のストレージが表示されます。保存可能としてマークされているアクションは、`actions` ストアに保存されます。このタブを使用して、Salesforce1 および Lightning Experience のストレージを分析します。

[Component Tree (コンポーネントツリー)] タブ

このタブには、ネストされたコンポーネントのツリーが含まれるコンポーネントのマークアップが表示されます。

Component Tree | Performance | Transactions | Event Log | Actions | Storage

Expand All Show Global IDs

```
▼<one:one allowFraming="false" Browser_S1Features_isEncryptedStorageEnabled="{!$Browser.S1Features.isEncryptedStorageEnabled}">
  <ui:asyncComponentManager maxConcurrency="1">
  ▼<force:style>
    ▼{!v.body}
      ▼<force:access>
        ▼<aura:if isTrue="{!m.hasAccess}" else="[<i class="component-array-length">1]" template="[<i class="component-array-length">1]">
          ▼{!v.body}
            ▼<one:appLayoutContainer>
              ▼<div aura:id="container" class="function (cmp, fn) { return fn.add(cmp.get("m.formFactorClass")," container"); }">
                ▶<div aura:id="viewport" class="viewport">
                ▶<aura:if isTrue="{!$Browser.isDesktop}" else="[]" template="[<i class="component-array-length">1]">
                ▶<ui:containerManager registeredContainers="[]">
                ▶<one:panelManager class="one" cancelButtonLabel="{!$Label.MobileWebRecordActions.Cancel}">
                ▶<one:panelManagerDesktop aura:id="pmd" registeredPanels="[<i class="component-array-length">4]" useSharedContainer="true">
                ▶<aura:if isTrue="{!$Browser.isDesktop}" else="[]" template="[<i class="component-array-length">1]">
                ▶<force:modalSpinner>
                 <forceContent:modalPreviewManager isEnabled="{!$Browser.isDesktop}" actionsContext="LIST_VIEW_RECORD" handleShowToast="true"
                 recordIdsToRefresh="[]">
                 <forceContent:filesManager>
                ▶<aura:if isTrue="{!$Browser.isDesktop}" else="[]" template="[<i class="component-array-length">1]">
              ▶<div aura:id="hidden" class="hideEl">
```

マークアップの折りたたみまたは展開

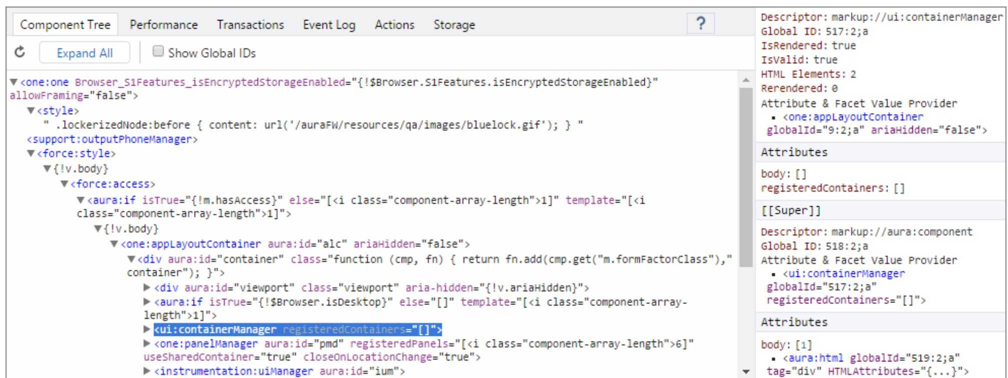
コンポーネント階層の展開または折りたたみを行うには、行頭の三角形をクリックします。

データの更新

コンポーネントツリーは、逐次化にコストがかかるため、コンポーネントの更新には応答しません。必要に応じて、パネルの上部にスクロールして [Refresh (更新)] アイコンをクリックし、ツリーを手動で更新する必要があります。

コンポーネントの詳細の表示

ノードをクリックすると、サイドバーに選択したコンポーネントの詳細が表示されます。コンポーネントツリーは手動で更新する必要がありますが、サイドバーのコンポーネントの詳細は自動的に更新されます。



サイドバーには、次のセクションがあります。

トップパネル

- **Descriptor (記述子)** — `prefix://namespace:name` 形式のコンポーネントの記述。
- **Global ID (グローバル ID)** — アプリケーションのライフサイクルにおけるコンポーネントの一意の識別子。
- **aura:id** — コンポーネントのローカル ID (定義されている場合)。
- **IsRendered** — コンポーネントはコンポーネントツリーに表示できますが、アプリケーションには表示できません。コンポーネントは、`v.body` や式 (`{!v.myCmp}` など) に含まれる場合に表示されます。
- **IsValid** — コンポーネントが破棄されると、無効になります。無効なコンポーネントへの参照は保持できますが、使用しないでください。
- **HTML Elements (HTML 要素)** — コンポーネント (子コンポーネントを含む) の HTML 要素の数。
- **Rerendered (再表示)** — Inspector を開いてからのコンポーネントの再表示回数。コンポーネントのプロパティを変更するとダーティになり、再表示がトリガされます。再表示はコストのかかる操作になる可能性があるため、可能であれば避けることが一般的です。
- **Attribute & Facet Value Provider (属性値プロバイダとファセット値プロバイダ)** — 通常、属性値プロバイダとファセット値プロバイダは同じコンポーネントです。同じである場合、これらは 1 つのエントリに統合されます。

  属性値プロバイダは、式の属性値を提供するコンポーネントです。次の例では、`<c:myComponent>` の名前属性で属性値プロバイダの `avpName` 属性の値を取得しています。

  ```
  <c:myComponent name="{!v.avpName}" />
  ```

  ファセット値プロバイダは、ファセット属性 (`Aura.Component[]` 型の属性) の値プロバイダです。ファセット値プロバイダは、コンポーネント属性値プロバイダとは異なる可能性があります。これは複雑なため、ここでは扱いません。ただし、ファセットの式では属性値プロバイダではなくファセット値プロバイダが使用されることを知っておくことは重要です。

属性

コンポーネントの属性値を表示します。式またはコードで属性を参照する場合、`v.attributeName` を使用します。

[[Super]] ([[スーパー]])

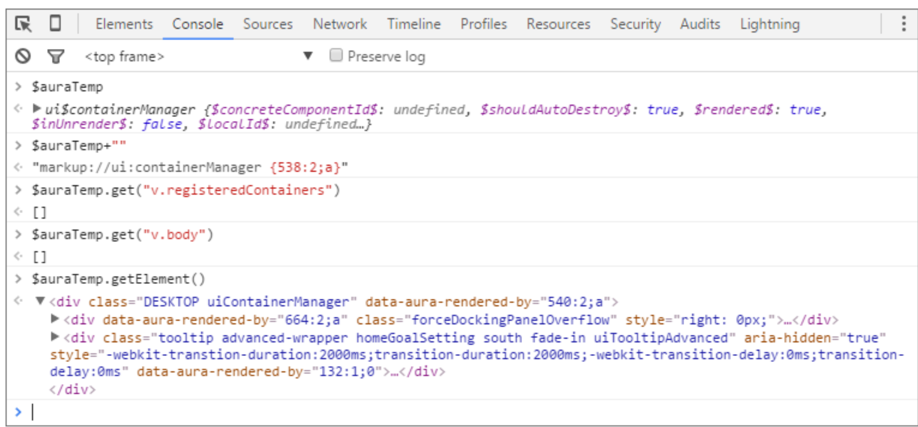
コンポーネントで別のコンポーネントを拡張する場合、サブコンポーネントの作成時にスーパーコンポーネントのインスタンスが作成されます。各スーパーコンポーネントには、独自のプロパティセットがあります。スーパーコンポーネントには独自の属性セクションがありますが、このスーパーコンポーネントには `body` 属性のみがあります。他のすべての属性値は、拡張階層で共有されます。

Model (モデル)

[Model (モデル)] セクションのあるコンポーネントが表示されることがあります。モデルは廃止される予定の機能で、デバッグのためにのみ含まれています。コードが壊れるため、モデルは参照しないでください。

[Console (コンソール)] でのコンポーネントへの参照の取得

Inspector の任意の場所でコンポーネントの参照をクリックして、そのコンポーネントを指し示す `$auraTemp` 変数を生成します。[Console (コンソール)] タブで `$auraTemp` を参照して、コンポーネントをさらに調査できます。



次のコマンドは、`$auraTemp` 変数を使用してコンポーネントのコンテンツを調査するときに便利です。

`$auraTemp+""`

コンポーネント記述子を返します。

`$auraTemp.get("v.attributeName")`

`attributeName` 属性の値を返します。

`$auraTemp.getElement()`

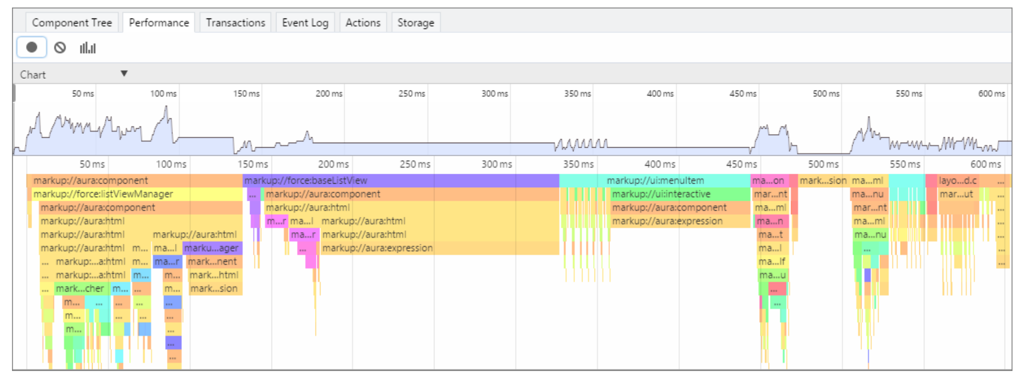
対応する DOM 要素を返します。

`inspect($auraTemp.getElement())`

[Elements (要素)] タブを開き、コンポーネントの DOM 要素を調査します。

[Performance (パフォーマンス)] タブ

[Performance (パフォーマンス)] タブには、コンポーネントの作成時間のフレームグラフが表示されます。グラフの長い部分や深い部分を確認して、パフォーマンスの潜在的なボトルネックを特定します。



パフォーマンスデータの記録

[Record (記録)] 、[Clear (クリア)] 、および [Show current collected (現在収集しているデータを表示)] ボタンを使用して、特定のユーザアクションまたはユーザアクションのコレクションに関するパフォーマンスデータを収集します。

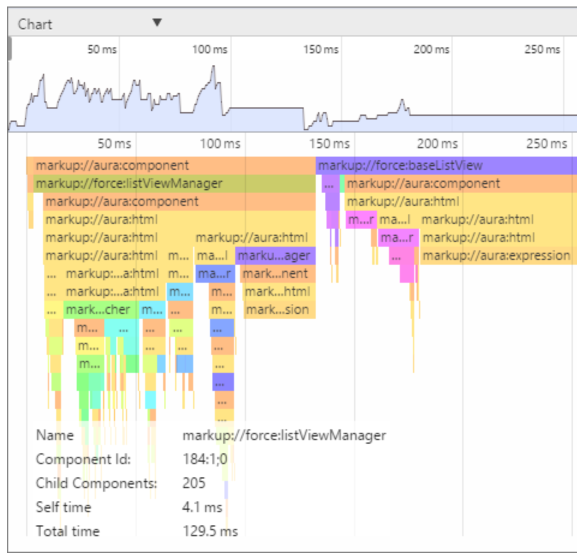
1. パフォーマンスデータの収集を開始するには、 を押します。
2. アプリケーションの 1 つ以上のアクションを取得します。
3. パフォーマンスデータの収集を停止するには、 を押します。

アクションのフレームグラフが表示されます。記録を停止する前にグラフを表示するには、 ボタンを押します。

コンポーネントのパフォーマンスの詳細の表示

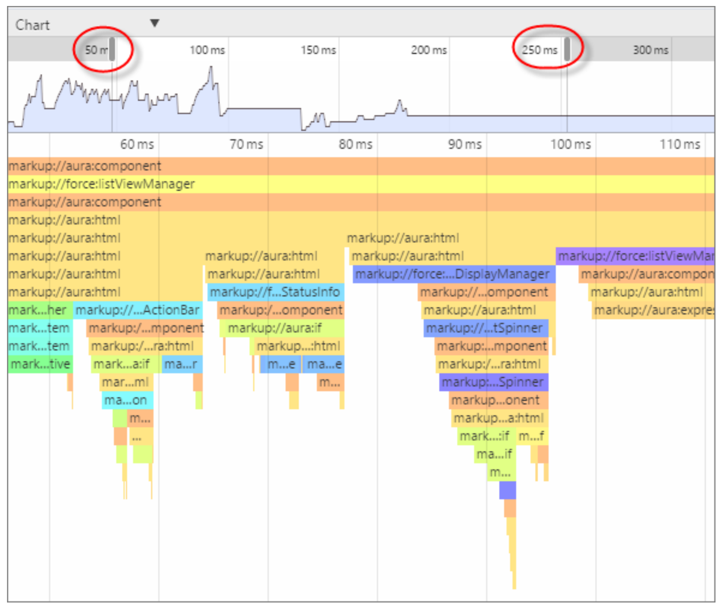
フレームグラフのコンポーネントの上にマウスを置くと、そのコンポーネントに関する詳細情報が左下に表示されます。コンポーネントの複雑性やタイミングの情報は、パフォーマンスの問題を診断に役立ちます。

基準	次の事項の完了に要した時間
セルフ時間	現在の関数。この関数が呼び出した関数の完了時間は含まれません。
通算セルフ時間	記録されたタイムライン全体の関数のすべての呼び出し。この関数が呼び出した関数の完了時間は含まれません。
合計時間	現在の関数とこの関数が呼び出したすべての関数。
通算合計時間	記録されたタイムライン全体の関数のすべての呼び出し。この関数が呼び出した関数の完了時間が含まれます。



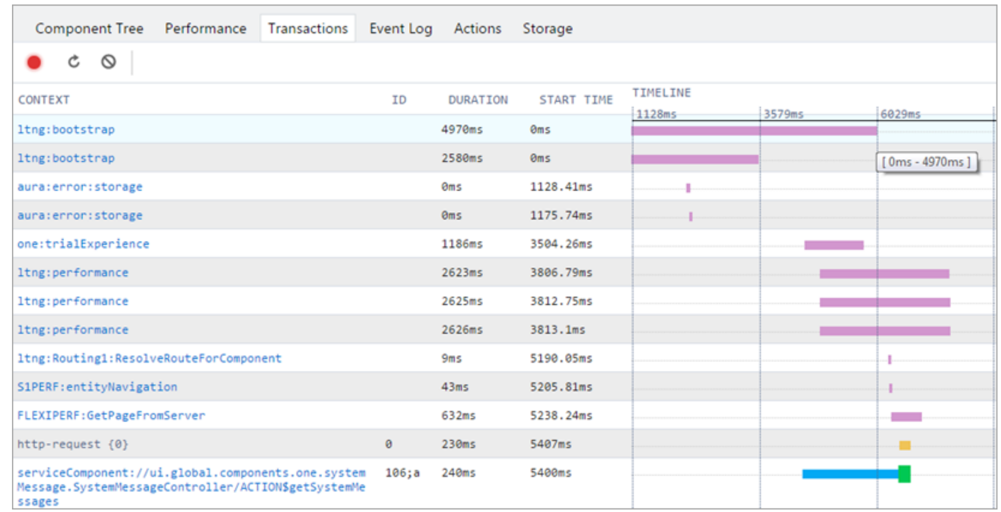
タイムラインの絞り込み

タイムラインの垂直ハンドルをドラッグして、対象となる時間枠を選択します。時間枠を絞り込んでコンポーネントの作成時間を調査し、潜在的なパフォーマンスのホットスポットを特定します。



[トランザクション] タブ

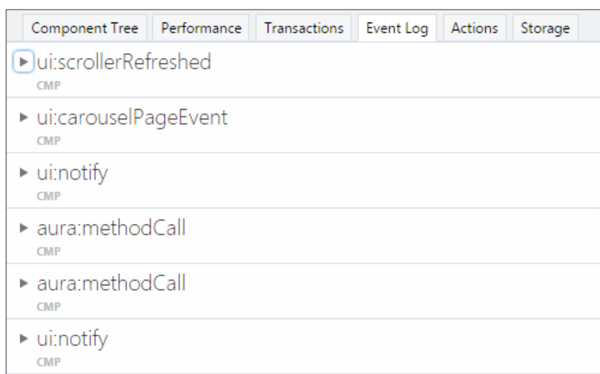
Salesforceが提供するアプリケーションには、トランザクションマーカーが含まれているものがあります。トランザクションマーカーを使用すると、トランザクション内のアクションの細かい総計値を表示できます。独自のトランザクションを作成することはできません。



基準	説明
期間	ページの開始時刻以降のページの期間 (ミリ秒単位)。
開始時刻	ページが最後に読み込みまたは更新されたときの開始時刻 (ミリ秒単位)。
タイムライン	トランザクションの開始時間および終了時間。色付きのバーで表示されます。 • 緑色 - サーバでアクションが実行されている時間 • 黄色 - XMLHttpRequest トランザクション • 青色 - XMLHttpRequest が送信されるまでキューに入っている時間 • 紫色 - カスタムトランザクション

[Event Log (イベントログ)] タブ

このタブには、起動されたすべてのイベントが表示されます。イベントグラフでは、1 つ以上のアクションのイベントおよびハンドラのシーケンスを分かりやすく表示します。



イベントの記録

[Toggle recording (記録を切り替え)] ● と [Clear (クリア)] ⃠ ボタンを使用して、特定のユーザアクションまたはユーザアクションのコレクションをキャプチャします。

1. イベントデータの収集を開始するには、● を押します。
2. アプリケーションの 1 つ以上のアクションを取得します。
3. イベントデータの収集を停止するには、● を押します。

行動の詳細の表示

イベントを展開して詳細を表示します。コールスタックで、イベントハンドラ (`c.handleDataChange` など) をクリックして、コードのどこで定義されているのかを見つけます。黄色の行は最新のハンドラです。

```
ui:dataChanged
CMP

Parameters   ▸{...}

Caller       function(dataProvider, data, currentPage) {
                 var dataChangeEvent = dataProvider.getEvent("onchange");
                 dataChangeEvent
                     .setComponentEvent()
                     .setParams({
                         data: data,
                         currentPage: currentPage
                     }).fire();
             }

Source       <force:listViewPickerDataProvider globalId="546:0" columns="[]" items="[]" currentPage="1" endIndex="-1"
             pageCount="0" pageSize="50" startIndex="-1" totalItems="0" canLoadMore="false" updatedInfiniteLoadingIdleLabel=""
             scope="Opportunity" listIdOrApiName="Recent" listViewTitle="Recently Viewed">

Duration     72.7350ms

Call Stack   Event Fired                                          Handled By
             markup://ui:dataChanged                              c.handleDataChange
               <force:listViewPickerDataProvider globalId="546:0">   <force:virtualAutocompleteMenuList globalId="12:1678;a">
                                                                  c.handleDataChange
                                                                    <force:virtualAutocompleteMenu globalId="1:1678;a">
```

イベントのリストの絞り込み

デフォルトでは、アプリケーションイベントとコンポーネントイベントの両方が表示されます。アプリケーションイベント種別は **[App Events (アプリケーションイベント)]** ボタン、コンポーネントイベント種別は **[Cmp Events (Cmp イベント)]** ボタンを切り替えて非表示または表示を行うことができます。

`[Filter (フィルタ)]` 項目に、サブ文字列に一致する検索文字列を入力します。

`!` で検索文字列を開始すると、絞り込みが反転します。たとえば、「`!aura`」と入力すると、文字列 `aura` を含まないすべてのイベントが返されます。

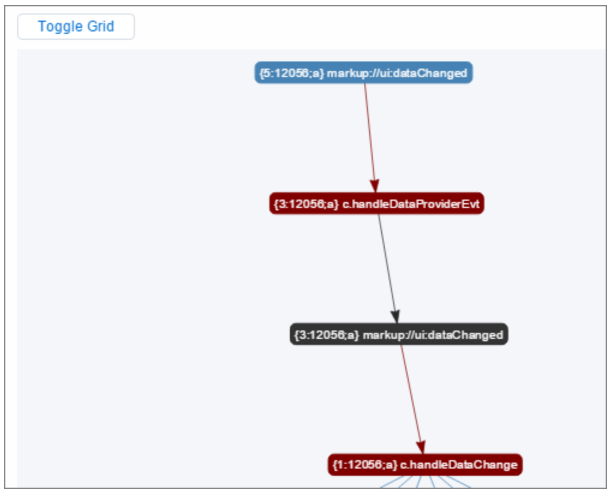
未処理のイベントの表示

起動されたが処理されていないイベントを表示します。未処理のイベントは、デフォルトではリストされていませんが、開発中に参照すると便利です。

イベントのグラフの表示

イベントを展開して詳細を表示します。**[Toggle Grid (グリッドを切り替え)]** ボタンをクリックして、このイベントの前後に起動されたイベントや、これらのイベントを処理するコンポーネントが表示されるネットワーク

グラフを生成します。イベント駆動型プログラミングは、イベントが爆発的に増えると混乱を引き起こす可能性があります。イベントグラフでは、各点を結んで、イベントおよびハンドラのシーケンスを分かりやすく表示します。



グラフは色分けされます。

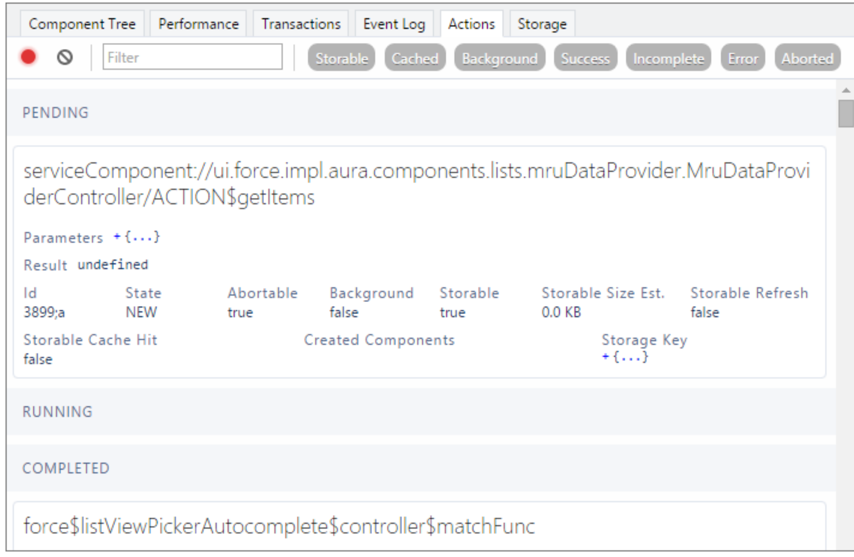
- 黒色 — 現在のイベント
- 茶色 — コントローラアクション
- 青色 — 現在のイベントの前後に起動された別のイベント

関連トピック:

イベントとの通信

[Actions (アクション)] タブ

このタブには、実行されたサーバ側のアクションが表示されます。このリストはページの更新時に自動的に更新されます。



アクションのリストの絞り込み

アクションのリストを絞り込むには、各種アクション種別または状態に関連するボタンを切り替えます。

- **Storable (保存可能)** — 応答をキャッシュできる保存可能なアクション。
- **Cached(キャッシュ)**—応答がキャッシュされる保存可能なアクション。このボタンをオフに切り替えると、キャッシュの欠落と保存不可能なアクションが表示されます。この情報は、パフォーマンスのボトルネックを調査する場合に重要です。
- **Background (バックグラウンド)** — Lightning コンポーネントではサポートされていません。オープンソースの Aura フレームワークで使用できます。
- **Success (成功)** — 正常に実行されたアクション。
- **Incomplete (未完了)** — サーバの応答がないアクション。サーバがダウンしているか、クライアントがオフラインである可能性があります。
- **Error (エラー)** — サーバエラーを返したアクション。
- **Aborted (中止)** — 中止されたアクション。

[`Filter` (検索条件)] 項目に、サブ文字列に一致する検索文字列を入力します。

`!` で検索文字列を開始すると、絞り込みが反転します。たとえば、「`!aura`」と入力すると、文字列 *aura* を含まないすべてのアクションが返され、多くのフレームワークレベルのアクションが除外されます。

このセクションの内容:

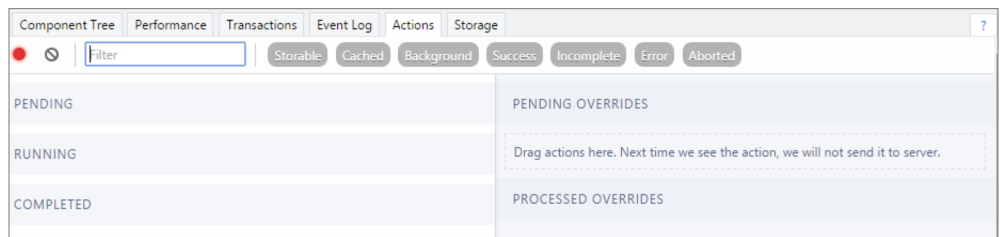
手動によるサーバ応答の上書き
[Actions (アクション)] タブの右側にある [Overrides (上書き)] パネルでは、手動でサーバ応答を調整し、アプリケーションのフォールトトレランスを調査します。

関連トピック:

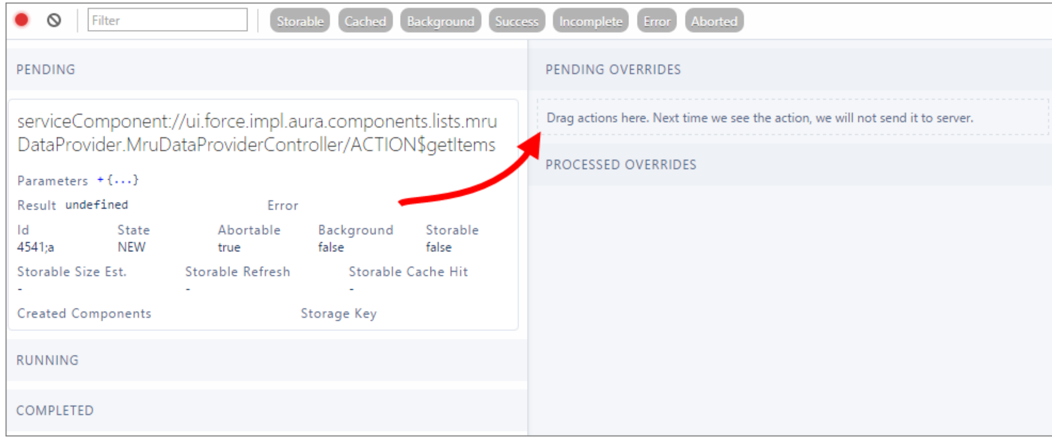
サーバ側のアクションのコール

手動によるサーバ応答の上書き

[Actions (アクション)] タブの右側にある [Overrides (上書き)] パネルでは、手動でサーバ応答を調整し、アプリケーションのフォールトトレランスを調査します。



[PENDING OVERRIDES (保留中の上書き)] セクションの左側にあるリストからアクションをドラッグします。



サーバに送信するために次に同じアクションがキューに追加されても、このフレームワークでは送信されません。代わりに、このフレームワークでは選択した上書きオプションに基づいて疑似応答が行われます。次に、上書きオプションを示します。

- Override the Result (結果を上書き)
- Error Response Next Time (次回にエラー応答)
- Drop the Action (アクションを削除)

メモ: 同じアクションは、同じ名前のアクションを意味します。アクションのパラメータは同じである必要はありません。

このセクションの内容:

アクション応答の変更

JSON オブジェクト値のいずれかを変更して Salesforce Lightning Inspector のアクション応答を変更し、UI の影響を確認します。サーバ側のアクションをコールすると、サーバから JSON オブジェクトが返されます。

エラー応答の設定

エラーが発生した場合、ユーザが状況を理解して処理方法を把握できるように、アプリケーションのパフォーマンスが適切に低下する必要があります。エラー状況をシミュレーションして、ユーザエクスペリエンスの影響を確認するには、Salesforce Lightning Inspector を使用します。

アクション応答の削除

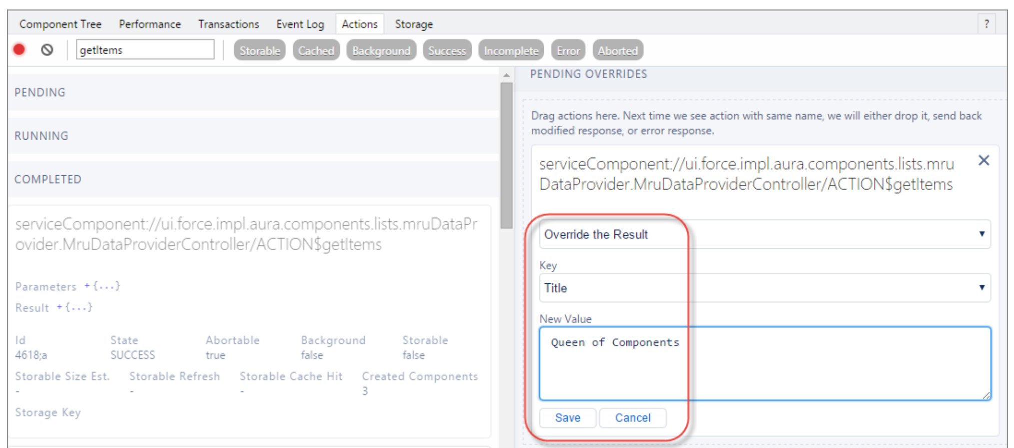
サーバ側のアクションがタイムアウトしたり、応答が削除されたりした場合、アプリケーションのパフォーマンスが適切に低下する必要があります。削除されたアクション応答をシミュレーションして、ユーザエクスペリエンスの影響を確認するには、Salesforce Lightning Inspector を使用します。

アクション応答の変更

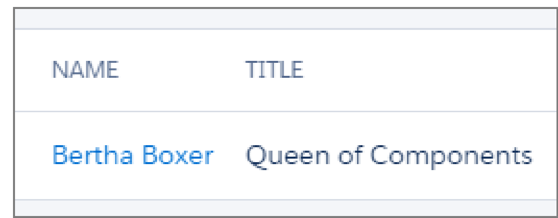
JSON オブジェクト値のいずれかを変更して Salesforce Lightning Inspector のアクション応答を変更し、UI の影響を確認します。サーバ側のアクションをコールすると、サーバから JSON オブジェクトが返されます。

1. 応答を変更するアクションをドラッグして、[PENDING OVERRIDES (保留中の上書き)] セクションに移動します。
2. ドロップダウンリストから [Override the Result (結果を上書き)] を選択します。
3. [Key (キー)] 項目で変更する応答キーを選択します。

4. [New Value (新しい値)] 項目にキーの変更後の値を入力します。



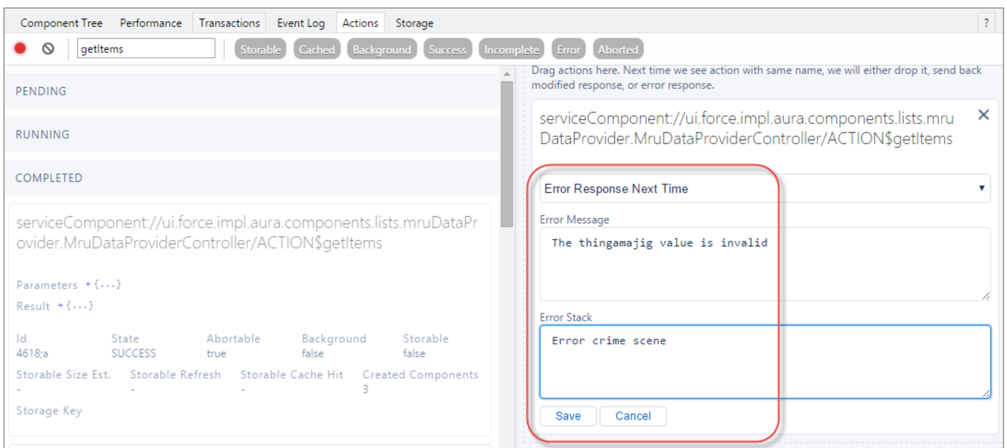
5. [保存] をクリックします。
6. アクションの実行をトリガするには、ページを更新します。
   変更されたアクション応答が [PENDING OVERRIDES (保留中の上書き)] セクションから [PROCESSED OVERRIDES (処理済みの上書き)] セクションに移動します。
7. 行った変更に関連する UI の変更があれば、それを確認します。



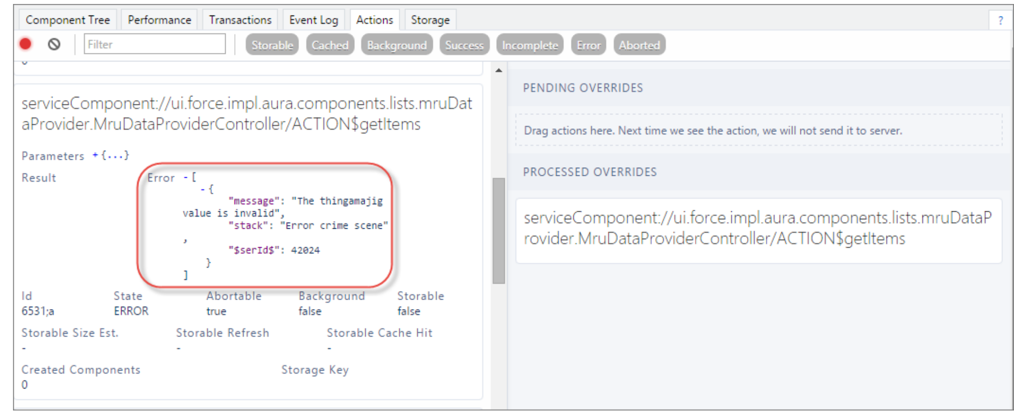
エラー応答の設定

エラーが発生した場合、ユーザが状況を理解して処理方法を把握できるように、アプリケーションのパフォーマンスが適切に低下する必要があります。エラー状況をシミュレーションして、ユーザエクスペリエンスの影響を確認するには、Salesforce Lightning Inspector を使用します。

1. 応答を変更するアクションをドラッグして、[PENDING OVERRIDES (保留中の上書き)] セクションに移動します。
2. ドロップダウンリストから [Error Response Next Time (次回にエラー応答)] を選択します。
3. [Error Message (エラーメッセージ)] を追加します。
4. [Error Stack (エラースタック)] 項目にテキストを追加します。



5. [保存] をクリックします。
6. アクションの実行をトリガするには、ページを更新します。
   - 変更されたアクション応答が [PENDING OVERRIDES (保留中の上書き)] セクションから [PROCESSED OVERRIDES (処理済みの上書き)] セクションに移動します。
   - 左パネルの [COMPLETED (完了)] セクションに `State` が `ERROR` になっているアクション応答が表示されます。



7. 行った変更に関連する UI の変更があれば、それを確認します。UI では、ユーザにアラートが表示されたり、ユーザがアプリケーションを継続して使用したりしている間にエラーが処理されています。

   パフォーマンスが適切に低下するように、アクション応答のコールバックでエラー応答 (`response.getState() === "ERROR"`) が処理されていることを確認します。

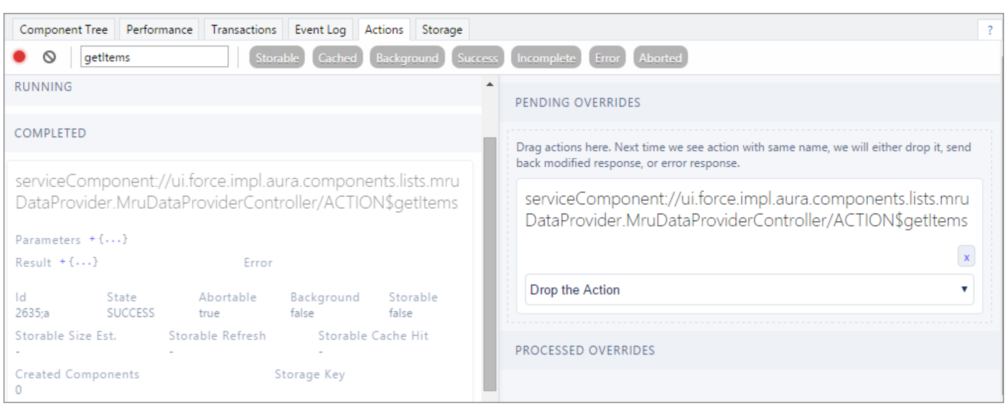
関連トピック:

サーバ側のアクションのコール

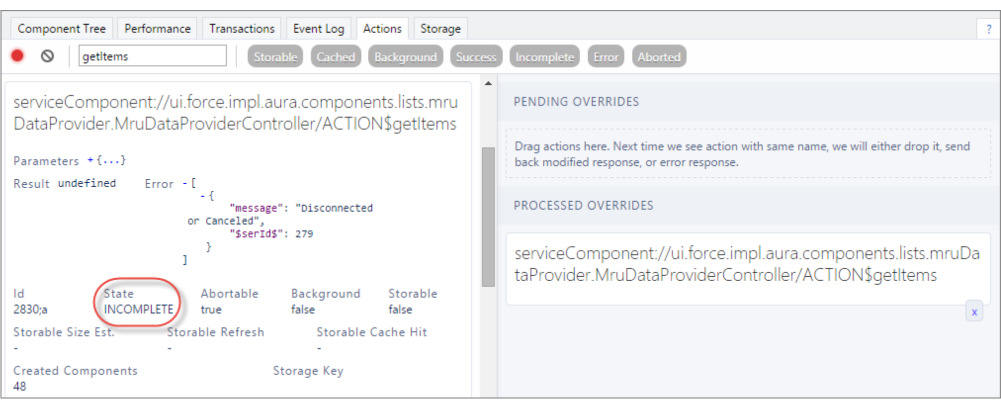
アクション応答の削除

サーバ側のアクションがタイムアウトしたり、応答が削除されたりした場合、アプリケーションのパフォーマンスが適切に低下する必要があります。削除されたアクション応答をシミュレーションして、ユーザエクスペリエンスの影響を確認するには、Salesforce Lightning Inspector を使用します。

1. 応答を変更するアクションをドラッグして、[PENDING OVERRIDES (保留中の上書き)] セクションに移動します。
2. ドロップダウンリストから [Drop the Action (アクションを削除)] を選択します。



3. アクションの実行をトリガするには、ページを更新します。
   - 変更されたアクション応答が [PENDING OVERRIDES (保留中の上書き)] セクションから [PROCESSED OVERRIDES (処理済みの上書き)] セクションに移動します。
   - 左パネルの [COMPLETED (完了)] セクションに `State` が `INCOMPLETE` になっているアクション応答が表示されます。



4. 行った変更に関連する UI の変更があれば、それを確認します。UI では、ユーザにアラートが表示されたり、ユーザがアプリケーションを継続して使用できるようになったりして削除されたアクションが処理されています。

パフォーマンスが適切に低下するように、アクション応答のコールバックで未完了の応答 (`response.getState() === "INCOMPLETE"`) が処理されていることを確認します。

関連トピック:

サーバ側のアクションのコール

[Storage (ストレージ)] タブ

このタブには、Lightning アプリケーションのクライアント側のストレージが表示されます。保存可能としてマークされているアクションは、`actions` ストアに保存されます。このタブを使用して、Salesforce1 および Lightning Experience のストレージを分析します。

Component Tree | Performance | Transactions | Event Log | Actions | Storage


```
- {
    "actions": - {
        "Adapter": "crypto",
        "sizeEstimate": "1017.4 KB (10% of 10240 KB)",
        "Version": "37.0",
        "Items": + {...}
    },
    "objectHomeStateManager": - {
        "Adapter": "indexeddb",
        "sizeEstimate": "0.0 KB (0% of 2048 KB)",
        "Version": "1.0",
        "Items": {}
    },
    "ComponentDefStorage": - {
        "Adapter": "indexeddb",
        "sizeEstimate": "1199.7 KB (30% of 4000 KB)",
        "Version": "37.0",
        "Items": + {...}
    }
}
```


ログメッセージ

クライアント側のコードをデバッグしやすくするには、Webブラウザでサポートされていれば、`console.log()` を使用してブラウザの JavaScript コンソールに出力を書き出すことができます。

JavaScript コンソールの使用手順については、Web ブラウザの説明を参照してください。

第 8 章 パフォーマンスの警告の修正

トピック:

- <aura:if> — 表示されない内容のクリーンアップ
- <aura:iteration> — 複数の items 設定

コード内のいくつかの一般的なパフォーマンスアンチパターンは、ブラウザコンソールに警告メッセージを記録するようにフレームワークに促します。警告メッセージを修正することでコンポーネントが高速化されます。

警告は、デバッグモードを有効にした場合にのみブラウザコンソールに表示されます。

関連トピック:

Lightning コンポーネントのデバッグモードの有効化

<aura:if> — 表示されない内容のクリーンアップ

この警告は、<aura:if> タグの isTrue 属性を true から false に変更した場合に発生します。<aura:if> の表示されない内容は破棄する必要があり、これによりフレームワークで表示にかかる時間が短縮されます。

例

次のコンポーネントは、アンチパターンを示しています。

```
<!--c:ifCleanUnrendered-->
<aura:component>
    <aura:attribute name="isVisible" type="boolean" default="true"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

コンポーネントのクライアント側コントローラを次に示します。

```
/* c:ifCleanUnrenderedController.js */
({
    init: function(cmp) {
        /* Some logic */
        cmp.set("v.isVisible", false); // Performance warning trigger
    }
})
```

コンポーネントが作成されるときに、<aura:if> タグの isTrue 属性が評価されます。isVisible 属性の値はデフォルトで true なので、フレームワークによって <aura:if> タグのボディが作成されます。コンポーネントが作成されてから表示されるまでの間に、init イベントがトリガされます。

クライアント側コントローラの init() 関数で isVisible の値が trueから false に切り替えられます。<aura:if> タグの isTrue 属性が false になったので、フレームワークは <aura:if> タグのボディを破棄する必要があります。この警告は、デバッグモードを有効にした場合にのみブラウザコンソールに表示されます。

```
WARNING: [Performance degradation] markup://aura:if ["5:0"] in c:ifCleanUnrendered ["3:0"]
needed to clear unrendered body.
```

警告のスタック追跡を表示するには、警告の横の展開ボタンをクリックします。

```
AuraInstance.$run$                                 @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                        @ aura_proddebug.js:8324
Component.$fireChangeEvent$                        @ aura_proddebug.js:6203
Component.set                                      @ aura_proddebug.js:6161
init                                               @ ifCleanUnrendered.js:13
Action.$runDeprecated$                             @ aura_proddebug.js:8666
Component$getActionCaller                          @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$      @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$             @ aura_proddebug.js:8274
(anonymous)                                        @ aura_proddebug.js:8326
```

ブラウザコンソールの [ソース] ペインに問題を起こしているコードの行を表示するには、スタック追跡で ifCleanUnrendered エントリのリンクをクリックします。

警告を修正する方法

isTrue 式のロジックを逆にします。isTrue 属性をデフォルトで true に設定せずに、false に設定します。必要に応じて、init() メソッドで isTrue 式を true に設定します。

次に、修正したコンポーネントを示します。

```
<!--c:ifCleanUnrenderedFixed-->
<aura:component>
    <!-- FIX: Change default to false.
         Update isTrue expression in controller instead. -->
    <aura:attribute name="isVisible" type="boolean" default="false"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

次に、修正したコントローラを示します。

```
/* c:ifCleanUnrenderedFixedController.js */
({
    init: function(cmp) {
        // Some logic
        // FIX: set isVisible to true if logic criteria met
        cmp.set("v.isVisible", true);
    }
})
```

関連トピック:

aura:if

Lightning コンポーネントのデバッグモードの有効化

<aura:iteration> — 複数の items 設定

この警告は、<aura:iteration> タグの items 属性を同じ表示サイクルで複数回設定した場合に発生します。

2 つのコレクションが JavaScript で同じであるかどうかを簡単に効率よく確認する方法はありません。items の古い値が新しい値と同じであったとしても、以前に作成された <aura:iteration> タグのボディはフレームワークによって削除され、置き換えられます。

例

次のコンポーネントは、アンチパターンを示しています。

```
<!--c:iterationMultipleItemsSet-->
<aura:component>
    <aura:attribute name="groceries" type="List"
                  default="[ 'Eggs', 'Bacon', 'Bread' ]"/>

    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

コンポーネントのクライアント側コントローラを次に示します。

```
/* c:iterationMultipleItemsSetController.js */
({
    init: function(cmp) {
        var list = cmp.get('v.groceries');
        // Some logic
        cmp.set('v.groceries', list); // Performance warning trigger
    }
})
```

コンポーネントが作成されるときに、`<aura:iteration>` タグの `items` 属性が `groceries` 属性のデフォルト値に設定されます。コンポーネントが作成されてから表示されるまでの間に、`init` イベントがトリガされます。

クライアント側コントローラの `init()` 関数で `groceries` 属性が設定され、それによって `<aura:iteration>` タグの `items` 属性がリセットされます。この警告は、デバッグモードを有効にした場合にのみブラウザコンソールに表示されます。

```
WARNING: [Performance degradation] markup://aura:iteration [id:5:0] in
c:iterationMultipleItemsSet ["3:0"]
had multiple items set in the same Aura cycle.
```

警告のスタック追跡を表示するには、警告の横の展開ボタンをクリックします。

```
AuraInstance.$run$                              @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                     @ aura_proddebug.js:8324
Component.$fireChangeEvent$                     @ aura_proddebug.js:6203
Component.set                                   @ aura_proddebug.js:6161
init                                            @ iterationMultipleItemsSet.js:14
Action.$runDeprecated$                          @ aura_proddebug.js:8666
Component$getActionCaller                       @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$   @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$          @ aura_proddebug.js:8274
(anonymous)                                     @ aura_proddebug.js:8326
```

ブラウザコンソールの [ソース] ペインに問題を起こしているコードの行を表示するには、スタック追跡で `iterationMultipleItemsSet` エントリのリンクをクリックします。

警告を修正する方法

`<aura:iteration>` タグの `items` 属性は何度も変更しないようにしてください。最も簡単な解決策は、マークアップの `groceries` 属性のデフォルト値を削除することです。代わりに、コントローラの `groceries` 属性の値を設定します。

別の解決策は、デフォルト値の保存のみを目的とする 2 番目の属性を作成することです。コントローラのロジックが完成したら、`groceries` 属性を設定します。

次に、修正したコンポーネントを示します。

```
<!--c:iterationMultipleItemsSetFixed-->
<aura:component>
    <!-- FIX: Remove the default from the attribute -->
    <aura:attribute name="groceries" type="List" />
    <!-- FIX (ALTERNATE): Create a separate attribute containing the default -->
    <aura:attribute name="groceriesDefault" type="List"
                  default="[ 'Eggs', 'Bacon', 'Bread' ]"/>

    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

次に、修正したコントローラを示します。

```
/* c:iterationMultipleItemsSetFixedController.js */
({
    init: function(cmp) {
        // FIX (ALTERNATE) if need to set default in markup
        // use a different attribute
        // var list = cmp.get('v.groceriesDefault');
        // FIX: Set the value in code
        var list = ['Eggs', 'Bacon', 'Bread'];
        // Some logic
        cmp.set('v.groceries', list);
    }
})
```

関連トピック:

aura:iteration

Lightning コンポーネントのデバッグモードの有効化

第 9 章 リファレンス

トピック:

- リファレンスドキュメントアプリケーション
- サポートされる aura:attribute の型
- aura:application
- aura:component
- aura:dependency
- aura:event
- aura:interface
- aura:method
- aura:set
- コンポーネントの参照
- メッセージングコンポーネントの参照
- インターフェースの参照
- イベントの参照
- システムイベントの参照
- サポートされる HTML タグ

このセクションには、フレームワークで使用できるさまざまなタグの詳細を示すリファレンスドキュメントが含まれています。

Lightning コンポーネントフレームワークは、オープンソースの Aura フレームワークで使用できるもののサブセットと、Salesforce に固有のコンポーネントおよびイベントを提供します。

リファレンスドキュメントアプリケーション

https://<myDomain>.lightning.force.com/componentReference/suite.app のコンポーネントライブラリ (ベータ) で、Lightning コンポーネントのデザインを確認できます。<myDomain> は、カスタム Salesforce ドメインの名前です。

また、参照情報と JavaScript API を含むリファレンスドキュメントアプリケーションは、引き続き使用できます。アプリケーションには、次の場所からアクセスします。

https://<myDomain>.lightning.force.com/auradocs/reference.app。<myDomain> は、カスタム Salesforce ドメインの名前です。

サポートされる aura:attribute の型

aura:attribute は、アプリケーション、インターフェース、コンポーネント、イベントで使用できる属性を記述します。

属性名	型	説明
access	String	属性が独自の名前空間の外側で使用できるかどうかを示します。有効な値は、public (デフォルト)、global、private です。
name	String	必須。属性の名前。たとえば、aura:newCmp というコンポーネントで <aura:attribute name="isTrue" type="Boolean" /> を設定する場合は、<aura:newCmp isTrue="false" /> のようにコンポーネントをインスタンス化するときに、この属性を設定できます。
type	String	必須。属性の型。サポートされている基本のデータ型のリストについては、「基本の型」を参照してください。
default	String	属性のデフォルト値。必要に応じて上書きできます。デフォルト値を設定するときに、$Label、$Locale、および $Browser グローバル値プロバイダを使用する式を指定できます。または、init イベントを使用して動的なデフォルトを設定できます。「コンポーネントの初期化時のアクションの呼び出し」(ページ 271)を参照してください。
required	Boolean	属性が必須かどうかを指定します。デフォルトは、false です。
description	String	属性およびその用途の概要。

すべての <aura:attribute> タグには、名前とデータ型の値があります。次に例を示します。

```
<aura:attribute name="whom" type="String" />
```

メモ: データ型の値では大文字と小文字は区別されませんが、マークアップで JavaScript、CSS、および Apex とやりとりするときには大文字と小文字の区別に注意する必要があります。

関連トピック:

コンポーネントの属性

基本の型

次に、サポートされている基本の型の値を示します。一部の型は、Java のプリミティブのラッパーオブジェクトに対応します。フレームワークは Java で作成されているため、このような基本の型のデフォルト (数値の最大サイズなど) は、対応付けられる Java オブジェクトで定義されます。

型	例	説明
`Boolean`	`<aura:attribute name="showDetail" type="Boolean" />`	有効な値は、`true` または `false` です。デフォルト値を `true` に設定するには、`default="true"` を追加します。
`Date`	`<aura:attribute name="startDate" type="Date" />`	カレンダー日に対応する yyyy-mm-dd 形式の日付。日付の hh:mm:ss 部分は保存されません。時刻項目を含めるには、`DateTime` を代わりに使用します。
`DateTime`	`<aura:attribute name="lastModifiedDate" type="DateTime" />`	タイムスタンプに対応する日付。日時の詳細がミリ秒の精度で含まれます。
`Decimal`	`<aura:attribute name="totalPrice" type="Decimal" />`	`Decimal` の値には、小数点以下の値 (小数点の右側の桁) を含めることができます。java.math.BigDecimal に対応付けられます。 浮動小数点数計算の精度を保持するには、`Double` より `Decimal` のほうが適切です。これは通貨項目に適しています。
`Double`	`<aura:attribute name="widthInchesFractional" type="Double" />`	`Double` の値には、小数点以下の値を含めることができます。java.lang.Double に対応付けられます。通貨項目には、代わりに `Decimal` を使用します。
`Integer`	`<aura:attribute name="numRecords" type="Integer" />`	`Integer` の値には、小数点以下の値がない数値を含めることができます。最大サイズなどの制限を定義する java.lang.Integer に対応付けられます。

型	例	説明
Long	<aura:attribute name="numSwissBankAccount" type="Long" />	Long の値には、小数点以下の値がない数値を含めることができます。最大サイズなどの制限を定義する java.lang.Long に対応付けられます。 Integer が提供するよりも広範囲の値が必要な場合に、このデータ型を使用します。
String	<aura:attribute name="message" type="String" />	一連の文字。

基本の型のそれぞれには配列を使用できます。次に例を示します。

```
<aura:attribute name="favoriteColors" type="String[]" default="['red','green','blue']" />
```

Apex コントローラからのデータの取得

Apexコントローラから文字列配列を取得するには、コンポーネントをコントローラにバインドします。次のコンポーネントは、ボタンをクリックしたときに文字列配列を取得します。

```
<aura:component controller="namespace.AttributeTypes">
    <aura:attribute name="favoriteColors" type="String[]" default="cyan, yellow, magenta"/>

    <aura:iteration items="{!v.favoriteColors}" var="s">
        {!s}
    </aura:iteration>
    <lightning:button onclick="{!c.getString}" label="Update"/>
</aura:component>
```

List<String> オブジェクトが返されるように Apex コントローラを設定します。

```
public class AttributeTypes {
    private final String[] arrayItems;

 @AuraEnabled
    public static List<String> getStringArray() {
        String[] arrayItems = new String[]{ 'red', 'green', 'blue' };
        return arrayItems;
    }

}
```

次のクライアント側のコントローラは、Apexコントローラから文字列配列を取得し、{!v.favoriteColors} 式を使用してそれを表示します。

```
({
    getString : function(component, event) {
    var action = component.get("c.getStringArray");
     action.setCallback(this, function(response) {
            var state = response.getState();
```

```
            if (state === "SUCCESS") {
                var stringItems = response.getReturnValue();
                component.set("v.favoriteColors", stringItems);
            }
        });
        $A.enqueueAction(action);
    }
})
```

オブジェクト型

属性には、オブジェクトに対応する型を指定できます。

```
<aura:attribute name="data" type="Object" />
```

たとえば、JavaScript 配列をイベントパラメータとして渡すために、`Object` 型の属性を作成する場合があります。コンポーネントイベントで、`aura:attribute` を使用してイベントパラメータを宣言します。

```
<aura:event type="COMPONENT">
    <aura:attribute name="arrayAsObject" type="Object" />
<aura:event>
```

JavaScript コードで、`Object` 型の属性を設定できます。

```
// Set the event parameters
var event = component.getEvent(eventType);
event.setParams({
    arrayAsObject:["file1", "file2", "file3"]
});
event.fire();
```

変数種別の確認

変数種別を判断する場合は、`typeof` を使用するか、JavaScript の標準メソッドを使用します。`instanceof` 演算子は、複数のウィンドウまたはフレームが存在する可能性があるため信頼性が低下します。

関連トピック:

Salesforce レコードの操作

標準オブジェクト型とカスタムオブジェクト型

属性には、標準オブジェクトまたはカスタムオブジェクトに対応する型を指定できます。次の例は、標準 `Account` オブジェクトの属性です。

```
<aura:attribute name="acct" type="Account" />
```

次の例は、Expense__c カスタムオブジェクトの属性です。

```
<aura:attribute name="expense" type="Expense__c" />
```

関連トピック:

Salesforce レコードの操作

コレクション型

次に、サポートされているコレクション型の値を示します。

型	例	説明
type[] (配列)	`<aura:attribute name="colorPalette" type="String[]" default="['red', 'green', 'blue']" />`	定義された種別の項目の配列。
List	`<aura:attribute name="colorPalette" type="List" default="['red', 'green', 'blue']" />`	順序付けされた項目のコレクション。
Map	`<aura:attribute name="sectionLabels" type="Map" default="{ a: 'label1', b: 'label2' }" />`	キーを値に対応付けるコレクション。対応付けに重複キーを含めることはできません。各キーを複数の値に対応付けることはできません。デフォルトは空のオブジェクト {} です。値を取得するには、cmp.get("v.sectionLabels")['a'] を使用します。
Set	`<aura:attribute name="collection" type="Set" default="['red', 'green', 'blue']" />`	重複する要素を含まないコレクション。セット項目の順序は保証されません。たとえば、"red,green,blue" が "blue,green,red" と返される可能性があります。

変数種別の確認

変数種別を判断する場合は、代わりに typeof を使用するか、JavaScript の標準メソッド (Array.isArray() など) を使用します。instanceof 演算子は、複数のウィンドウまたはフレームが存在する可能性があるため信頼性が低下します。

リスト項目の設定

リスト内の項目を設定するには、いくつかの方法があります。クライアント側のコントローラを使用するには、List 型の属性を作成し、component.set() を使用して項目を設定します。

次の例では、ボタンをクリックしたときにクライアント側のコントローラから数値のリストを取得します。

```
<aura:attribute name="numbers" type="List"/>
<lightning:button onclick="{!c.getNumbers}" label="Display Numbers" />
<aura:iteration var="num" items="{!v.numbers}">
  {!num.value}
</aura:iteration>
```

```
/** Client-side Controller **/
({
  getNumbers: function(component, event, helper) {
    var numbers = [];
    for (var i = 0; i < 20; i++) {
      numbers.push({
        value: i
      });
    }
    component.set("v.numbers", numbers);
    }
})
```

リストデータをコントローラから取得するには、aura:iteration を使用します。

対応付け項目の設定

キーと値のペアを対応付けに追加するには、構文 myMap['myNewKey'] = myNewValue を使用します。

```
var myMap = cmp.get("v.sectionLabels");
myMap['c'] = 'label3';
```

次の例では、対応付けからデータを取得します。

```
for (var key in myMap){
     //do something
}
```

カスタム Apex クラス型

属性には、Apex クラスに対応する型を指定できます。次の例は、Color Apex クラスの属性です。

```
<aura:attribute name="color" type="docSampleNamespace.Color" />
```

サーバ側アクションから Apex クラスのインスタンスが返されると、インスタンスはフレームワークによって JSON に逐次化されます。@AuraEnabled でアノテーションされた public インスタンスのプロパティとメソッドの値のみが逐次化されて返されます。

配列の使用

属性に複数の要素が含まれている場合は、配列を使用します。

次の `aura:attribute` タグは、Apex オブジェクトの配列の構文を示します。

```
<aura:attribute name="colorPalette" type="docSampleNamespace.Color[]" />
```

関連トピック:

Apex サーバ側コントローラからデータを返す
AuraEnabled アノテーション
Salesforce レコードの操作

フレームワーク固有の型

次に、フレームワーク固有でサポートされている型の値を示します。

型	例	説明
`Aura.Component`	N/A	単一のコンポーネント。代わりに `Aura.Component[]` を使用することをお勧めします。

`Aura.Component[]`	`<aura:attribute name="detail" type="Aura.Component[]"/>`
	`type="Aura.Component[]"` のデフォルト値を設定するには、`aura:attribute` のボディにデフォルトのマークアップを挿入します。次に例を示します。
	`<aura:component>`
	`    <aura:attribute name="detail" type="Aura.Component[]">`
	`    <p>default paragraph1</p>`
	`    </aura:attribute>`
	`    Default value is: {!v.detail}`
	`</aura:component>`
		マークアップのブロックを設定するには、この型を使用します。`Aura.Component[]` という型の属性は facet と呼ばれます。

関連トピック:

コンポーネントのボディ
コンポーネントのファセット

aura:application

アプリケーションは、.app リソース内にマークアップが含まれている特殊な最上位コンポーネントです。

マークアップはHTMLに似ており、コンポーネントおよびサポートされる一連のHTMLタグを含めることができます。.app リソースは、アプリケーションのスタンドアロンのエントリポイントであり、アプリケーションの全体的なレイアウト、スタイルシート、グローバルなJavaScriptインクルードを定義できます。このリソースは、省略可能なシステム属性を含む、最上位レベルの <aura:application> タグで開始します。システム属性によって、アプリケーションの設定方法がフレームワークに指示されます。

システム属性	型	説明
access	String	名前空間の外側にある別のアプリケーションによって、アプリケーションを拡張できるかどうかを示します。使用できる値は、public (デフォルト) と global です。
controller	String	アプリケーションのサーバ側のコントローラクラス。形式は namespace.myController となります。
description	String	アプリケーションの簡単な説明。
extends	Component	拡張するアプリケーション (該当する場合)。たとえば、「extends="namespace:yourApp"」などです。
extensible	Boolean	アプリケーションが別のアプリケーションによって拡張可能かどうかを示します。デフォルトは false です。
implements	String	アプリケーションで実装するインターフェースのカンマ区切りのリスト。
template	Component	フレームワークとアプリケーションの読み込みのブートストラップに使用されるテンプレートの名前。デフォルト値は、aura:template です。デフォルトのテンプレートを拡張する独自のコンポーネントを作成して、テンプレートをカスタマイズできます。以下に例を示します。 `<aura:component extends="aura:template" ... >`
tokens	String	アプリケーション用のトークンバンドルのカンマ区切りリスト。たとえば、「tokens="ns:myAppTokens"」などです。トークンを使用することで、設計の一貫性を確保しやすくなり、設計の変化に伴った更新がさらに簡単になります。トークンの値を一度定義すると、アプリケーション全体で再利用できます。
useAppcache	Boolean	非推奨。ブラウザベンダーが AppCache を廃止したため、Salesforce もそれに従いました。スタンドアロンアプリケーション (.app リソース) の <aura:application> タグの useAppcache 属性を削除してください。ブラウザベンダによる廃止が原因で生じるブラウザ間サポートの問題を回避するためです。

システム属性	型	説明
		`<aura:application>` タグに `useAppcache` を現在設定していない場合、`useAppcache` のデフォルト値は `false` のため何もする必要はありません。

`aura:application` には、`<aura:attribute>` タグで定義された `body` 属性も含まれます。属性は通常、コンポーネントの出力または動作を制御しますが、システム属性の設定情報は制御しません。

属性	型	説明
body	Component[]	アプリケーションのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。

関連トピック:

アプリケーションの作成

AppCache の使用

アプリケーションのアクセス制御

aura:component

コンポーネント階層のルート。デフォルトの表示を行います。

コンポーネントは、モジュール形式で再利用可能なUIのセクションをカプセル化する、Auraの機能単位です。他のコンポーネントまたは HTML マークアップを含めることができます。コンポーネントの属性とイベントは公開部分です。Aura は、`aura` および `ui` 名前空間で標準コンポーネントを提供します。

すべてのコンポーネントは、名前空間の一部です。たとえば、`ui` 名前空間に `button.cmp` として保存される `button` コンポーネントは、構文 `<ui:button label="Submit"/>` を使用して別のコンポーネントで参照できます。`label="Submit"` は、属性設定です。

コンポーネントを作成するには、次の構文に従います。

```
<aura:component>
    <!-- Optional coponent attributes here -->
    <!-- Optional HTML markup -->
    <div class="container">
        Hello world!
        <!-- Other components -->
    </div>
</aura:component>
```

コンポーネントには次の省略可能な属性があります。

属性	型	説明
access	String	コンポーネントが独自の名前空間の外側で使用できるかどうかを示します。使用できる値は、public (デフォルト) と global です。
controller	String	コンポーネントのサーバ側のコントローラクラス。形式は namespace.myController となります。
description	String	コンポーネントの説明。
extends	Component	拡張するコンポーネント。
extensible	Boolean	コンポーネントを拡張できる場合は true に設定します。デフォルトは false です。
implements	String	コンポーネントで実装するインターフェースのカンマ区切りのリスト。
isTemplate	Boolean	コンポーネントがテンプレートの場合、true に設定されます。デフォルトは、false です。テンプレートの <aura:component> タグで isTemplate="true" と設定されている必要があります。 `<aura:component isTemplate="true" extends="aura:template">`
template	Component	このコンポーネントのテンプレート。テンプレートは、フレームワークとアプリケーションの読み込みのブートストラップを行います。デフォルトのテンプレートは aura:template です。デフォルトのテンプレートを拡張する独自のコンポーネントを作成して、テンプレートをカスタマイズできます。以下に例を示します。 `<aura:component extends="aura:template" ... >`

aura:component には、<aura:attribute> タグで定義された body 属性が含まれます。属性は通常、コンポーネントの出力または動作を制御しますが、システム属性の設定情報は制御しません。

属性	型	説明
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。

aura:dependency

<aura:dependency> タグでは、連動関係を宣言できるため、フレームワークによる連動関係の検出が向上します。

このフレームワークでは、マークアップで定義されたコンポーネントなどの定義間の連動関係が自動的に追跡されます。これにより、フレームワークが定義をブラウザに送信できるようになります。ただし、コンポーネントの Javascript コードによって、別のコンポーネントが動的にインスタンス化されたり、コンポーネントのマークアップで直接参照されないコンポーネントが起動されたりする場合は、コンポーネントのマークアップで <aura:dependency> を使用して、連動関係についてフレームワークに明示的に指示します。
<aura:dependency> タグを追加することで、コンポーネントなどの定義とその連動関係が必要に応じてクライアントに送信されます。

たとえば、このタグをコンポーネントに追加すると、sampleNamespace:sampleComponent コンポーネントが連動関係としてマークされます。

```
<aura:dependency resource="markup://sampleNamespace:sampleComponent" />
```

コンポーネントのマークアップにこのタグを追加して、イベントを連動関係としてマークします。

```
<aura:dependency resource="markup://force:navigateToComponent" type="EVENT"/>
```

JavaScript コードでイベントを起動し、コンポーネントのマークアップで <aura:registerEvent> を使用してイベントを登録していない場合は、<aura:dependency> タグを使用します。推奨される方法は、<aura:registerEvent> タグの使用です。

<aura:dependency> タグには、次のシステム属性があります。

システム属性	説明
resource	コンポーネントやイベントなど、コンポーネントが連動するリソース。たとえば、resource="markup://sampleNamespace:sampleComponent" は、sampleNamespace 名前空間の sampleComponent を指します。 メモ: ワイルドカード照合でのアスタリスク (*) の使用は非推奨です。代わりに、コンポーネントのマークアップで直接参照されない各リソースに <aura:dependency> タグを追加します。ワイルドカード照合では、一致するリソースがない場合に保存規則エラーが発生します。また、ワイルドカード照合ではクライアントに必要以上の定義を送信するため、ページ読み込み時間が長くなることがあります。
type	コンポーネントが依存するリソースの種別。デフォルト値は、COMPONENT です。 メモ: ワイルドカード照合でのアスタリスク (*) の使用は非推奨です。代わりに、コンポーネントのマークアップで直接参照されない各リソースに <aura:dependency> タグを追加します。クライアントに送信する定義の種別はできるだけ慎重に選択してください。 最も一般的に使用される値は、次のとおりです。

システム属性	説明
	• COMPONENT • EVENT • INTERFACE • APPLICATION 複数の種別には、COMPONENT,APPLICATION のようにカンマ区切りのリストを使用します。

関連トピック:

コンポーネントの動的な作成
コンポーネントイベントの起動
アプリケーションイベントの起動

aura:event

イベントは、次の属性を持つ aura:event タグで表されます。

属性	型	説明
access	String	イベントが独自の名前空間の外側で拡張または使用できるかどうかを示します。使用できる値は、public (デフォルト) と global です。
description	String	イベントの説明。
extends	Component	拡張するイベント。たとえば、extends="namespace:myEvent" です。
type	String	必須。有効な値は、COMPONENT または APPLICATION です。

関連トピック:

イベントとの通信
イベントのアクセス制御

aura:interface

aura:interface タグには、省略可能な次の属性があります。

属性	型	説明
access	String	インターフェースが独自の名前空間の外側で拡張または使用できるかどうかを示します。使用できる値は、public (デフォルト) と global です。
description	String	インターフェースの説明。
extends	Component	拡張するインターフェースのカンマ区切りのリスト。たとえば、extends="namespace:intfB" です。

関連トピック:

インターフェース

インターフェースのアクセス制御

aura:method

<aura:method> を使用して、コンポーネントの API の一部としてメソッドを定義します。これにより、コンポーネントイベントを起動して処理する代わりに、コンポーネントのクライアント側コントローラからメソッドを直接コールできるようになります。<aura:method> を使用すると、親コンポーネントに含まれる子コンポーネントのメソッドをコールする場合に、親コンポーネントに必要なコードが簡略化されます。

<aura:method> タグには、次のシステム属性があります。

属性	型	説明
name	String	メソッド名。メソッド名を使用して、JavaScript コードのメソッドをコールします。以下に例を示します。 `cmp.sampleMethod(param1);`
action	Expression	実行するクライアント側のコントローラアクション。以下に例を示します。 `action="{!c.sampleAction}"` sampleAction はクライアント側コントローラのアクションです。action の値を指定しない場合、コントローラアクションは、デフォルトのメソッドの name の値に設定されます。
access	String	メソッドのアクセス制御。有効な値は、次のとおりです。 • **public** — 同じ名前空間のコンポーネントはメソッドをコールできます。これはデフォルトのアクセスレベルです。

属性	型	説明
		• **global** — どの名前空間のコンポーネントもメソッドをコールできます。
description	String	メソッドの説明。

パラメータの宣言

<aura:method> には、必要に応じてパラメータを含めることができます。<aura:method> 内で <aura:attribute> タグを使用して、メソッドのパラメータを宣言します。以下に例を示します。

```
<aura:method name="sampleMethod" action="{!c.doAction}"
  description="Sample method with parameters">
    <aura:attribute name="param1" type="String" default="parameter 1"/>
    <aura:attribute name="param2" type="Object" />
</aura:method>
```

メモ: パラメータの <aura:attribute> タグに access システム属性は必要ありません。

ハンドラアクションの作成

このハンドラアクションでは、メソッドに渡される引数へのアクセス方法を示します。

```
({
    doAction : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var param1 = params.param1;
            // add your code here
        }
    }
})
```

event.getParam('arguments') を使用して引数を取得します。引数がある場合はオブジェクト、引数がない場合は空の配列が返されます。

戻り値

aura:method は同期して実行されます。

- 同期メソッドは、返す前に実行を終了します。同期 JavaScript コードから値を返すには、return ステートメントを使用します。「同期コードの結果を返す」を参照してください。

- 非同期メソッドは返した後も実行を続ける場合があります。非同期JavaScriptコードから値を返すには、コールバックを使用します。「非同期コードの結果を返す」を参照してください。

関連トピック:

コンポーネントメソッドのコール

コンポーネントイベント

aura:set

スーパーコンポーネント、イベント、またはインターフェースから継承される属性の値を設定するには、マークアップで `<aura:set>` を使用します。

詳細は、次のセクションを参照してください。

- スーパーコンポーネントから継承される属性の設定
- コンポーネント参照での属性の設定
- インターフェースから継承される属性の設定

スーパーコンポーネントから継承される属性の設定

継承される属性の値を設定するには、サブコンポーネントのマークアップで `<aura:set>` を使用します。

例を見てみましょう。これは `c:setTagSuper` コンポーネントです。

```
<!--c:setTagSuper-->
<aura:component extensible="true">
    <aura:attribute name="address1" type="String" />
    setTagSuper address1: {!v.address1}<br/>
</aura:component>
```

`c:setTagSuper` の出力は、次のようになります。

```
setTagSuper address1:
```

`address1` 属性は設定されていないため、まだ値は出力されません。

これは `c:setTagSuper` を拡張する `c:setTagSub` コンポーネントです。

```
<!--c:setTagSub-->
<aura:component extends="c:setTagSuper">
    <aura:set attribute="address1" value="808 State St" />
</aura:component>
```

`c:setTagSub` の出力は、次のようになります。

```
setTagSuper address1: 808 State St
```

`sampleSetTagExc:setTagSub` は、スーパーコンポーネント `c:setTagSuper` から継承される `address1` 属性の値を設定します。

警告: この <aura:set> の使用はコンポーネントおよび抽象コンポーネントで有効ですが、インターフェースでは無効です。詳細は、「インターフェースから継承される属性の設定」(ページ 437)を参照してください。

使用コンポーネント内で参照することによってコンポーネントを使用している場合、マークアップでその属性値を直接設定できます。たとえば、c:setTagSuperRef は c:setTagSuper を参照し、aura:set を使用せずに address1 属性を直接設定します。

```
<!--c:setTagSuperRef-->
<aura:component>
    <c:setTagSuper address1="1 Sesame St" />
</aura:component>
```

c:setTagSuperRef の出力は、次のようになります。

```
setTagSuper address1: 1 Sesame St
```

関連トピック:

コンポーネントのボディ

継承されるコンポーネントの属性

コンポーネント参照での属性の設定

コンポーネント参照での属性の設定

コンポーネントに <ui:button> などの別のコンポーネントを含める場合、それを <ui:button> へのコンポーネント参照と呼びます。<aura:set> を使用して、コンポーネント参照に属性を設定できます。たとえば、<ui:button> への参照がコンポーネントに含まれているとします。

```
<ui:button label="Save">
    <aura:set attribute="buttonTitle" value="Click to save the record"/>
</ui:button>
```

これは、次のステートメントと同等です。

```
<ui:button label="Save" buttonTitle="Click to save the record" />
```

この単純な例では、aura:set がない後者の構文のほうが適切です。コンポーネント参照でこの単純な構文を使用して、親コンポーネントから継承される属性の値を設定することもできます。

aura:set は、マークアップを属性値として設定する場合に効果的です。たとえば、このサンプルでは、aura:if タグの else 属性にマークアップを指定します。

```
<aura:component>
    <aura:attribute name="display" type="Boolean" default="true"/>
    <aura:if isTrue="{!v.display}">
        Show this if condition is true
        <aura:set attribute="else">
           <ui:button label="Save" press="{!c.saveRecord}" />
        </aura:set>
```

```
    </aura:if>
</aura:component>
```

関連トピック:

スーパーコンポーネントから継承される属性の設定

インターフェースから継承される属性の設定

インターフェースから継承される属性の値を設定するには、コンポーネントで属性を再定義し、デフォルト値を設定します。`c:myIntf` インターフェースの例を見てみましょう。

```
<!--c:myIntf-->
<aura:interface>
    <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:interface>
```

このコンポーネントはインターフェースを実装し、`myBoolean` を `false` に設定します。

```
<!--c:myIntfImpl-->
<aura:component implements="c:myIntf">
    <aura:attribute name="myBoolean" type="Boolean" default="false" />

    <p>myBoolean: {!v.myBoolean}</p>
</aura:component>
```

コンポーネントの参照

Lightning Experience や Salesforce1 用、または Lightning アプリケーション用の標準搭載コンポーネントを使用します。これらのコンポーネントは、次をはじめとするさまざまな名前空間に属します。

aura

フレームワークのビルディングブロックとなるコンポーネントを提供します。

force

項目およびレコード固有の実装用のコンポーネントを提供します。

forceChatter

Chatter フィード用のコンポーネントを提供します。

forceCommunity

コミュニティ用のコンポーネントを提供します。

lightning

Lightning Design System スタイル設定のコンポーネントを提供します。スタンドアロンの Lightning アプリケーションで使用されるこの名前空間のコンポーネントは、`force:slds` を拡張して Lightning Design System スタイル設定を実装します。一致する `ui` と `lightning` 名前空間コンポーネントがあるインスタンスでは、`lightning` 名前空間コンポーネントを使用することをお勧めします。`lightning` 名前空間コンポーネントは、一般的な使用事例に合うように最適化されています。`lightning` 名前空間コンポーネントのイベント処理は、標準的な HTML の手法に従っており、`ui` 名前空間コンポーネントのイベント処理よりも簡単です。詳細は、「Lightning 基本コンポーネントでのイベント処理」を参照してください。

ui

Lightning Experience および Salesforce1 のデザインと一致しないユーザインターフェースコンポーネントの旧式の実装を行います。この名前空間のコンポーネントは、複数のスタイル設定方式をサポートし、通常はより複雑です。

aura:expression

式の評価後の値を表示します。参照される「プロパティ参照値」を表示するこのコンポーネントのインスタンスを作成します。参照値は、フリーテキストまたはマークアップで式が検出されたときに value 属性に設定されます。

式はリテラル値、変数、サブ式、演算子などで構成され、1 つの値に解決されます。式は、動的出力や、値を属性に割り当ててコンポーネントに渡す場合に使用します。

式の構文は `{!expression}` です。コンポーネントが表示されるとき、またはコンポーネントが値を使用するときに、`expression` が評価され、動的に置換されます。評価の結果、プリミティブ (整数、文字列など)、boolean、JavaScript または Aura オブジェクト、Aura コンポーネントまたはコレクション、コントローラメソッド (アクションメソッドなど)、その他の有益な値が得られます。

式では値プロバイダを使用してデータにアクセスでき、複雑な式の場合は演算子や関数も使用できます。値プロバイダには、`m` (モデルのデータ)、`v` (コンポーネントの属性データ)、`c` (コントローラアクション) があります。次の例に、値が属性 `num` で解決される式 `{!v.num}` を示します。

```
<aura:attribute name="num" type="integer" default="10"/>
<ui:inputNumber label="Enter age" aura:id="num" value="{!v.num}"/>
```

属性

属性名	属性型	説明	必須項目
value	String	評価および表示する式。	

aura:html

すべての html 要素を表すメタコンポーネント。マークアップで html が検出されると、いずれか 1 つの html 要素が作成されます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
HTMLAttributes	Map	html 要素で設定される属性の対応付け。	
tag	String	表示する html 要素の名前。	

aura:if

else 属性のボディまたはコンポーネントのいずれかを条件付きでインスタンス化し、表示します。

aura:if は、サーバで isTrue 式を評価し、その body または else 属性のいずれかでコンポーネントをインスタンス化します。作成および表示されるのは1つのブランチのみです。条件を切り替えると、現在のブランチが非表示となって破棄され、他のブランチが生成されます。

```
<aura:component>
    <aura:if isTrue="{!v.truthy}">
    True
    <aura:set attribute="else">
      False
    </aura:set>
  </aura:if>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	ComponentDefRef[]	isTrue が true と評価されたときに表示するコンポーネント。	はい
else	ComponentDefRef[]	isTrue が false と評価されたときに表示する代替内容で、ボディは表示されません。常に aura:set タグを使用して設定する必要があります。	
isTrue	Boolean	ボディを表示するために true と評価される必要がある式。	はい

aura:iteration

項目のコレクションのビューを表示します。クライアント側で排他的に作成できるコンポーネントを含む反復がサポートされます。

aura:iteration は、項目のコレクションを反復し、項目ごとにタグのボディを表示します。コレクションのデータの変更は、ページに自動的に再表示されます。また、クライアント側で排他的に作成できるコンポーネント、またはサーバ側の連動関係があるコンポーネントを含む反復もサポートされます。

次の例に、クライアント側で aura:iteration を排他的に使用する基本的な方法を示します。

```
<aura:component>

  <aura:iteration items="1,2,3,4,5" var="item">
        <meter value="{!item / 5}"/><br/>
    </aura:iteration>

</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	ComponentDefRef[]	反復ごとにコンポーネントを作成する場合に使用するテンプレート。	はい
end	Integer	コレクションの停止インデックス (含まない)。	
indexVar	String	反復内の各項目のインデックスに使用する変数の名前。	
items	List	反復処理されるデータのコレクション。	はい
loaded	Boolean	反復処理でテンプレートのセットの読み込みが終了した場合は True。	
start	Integer	コレクションの開始インデックス (含む)。	
template	ComponentDefRef[]	コンポーネントの生成に使用されるテンプレート。デフォルトでは、最初の読み込みのボディマークアップから設定されます。	
var	String	反復内の各項目に使用する変数の名前。	はい

aura:renderIf

非推奨。代わりに aura:if を使用します。このコンポーネントでは、内容を条件付きで表示できます。isTrue が true と評価されたときにのみボディを表示します。else 属性では、isTrue が false と評価されたときの代替内容を表示できます。

式で使用する値が変更されるたびに、isTrue 内の式が再評価されます。式の結果が変更されると、コンポーネントの再表示がトリガされます。true および false の両方の状態に対してコンポーネントを表示する場合は、aura:renderIf を使用します。最初に表示されないコンポーネントをインスタンス化するには、サーバへの往復処理が必要です。条件を切り替えると、現在のブランチが非表示となり、他のブランチが表示されます。

body または else 属性のいずれか (両方ではない) でコンポーネントをインスタンス化する場合は、代わりに aura:if を使用します。

```
<aura:component>
    <aura:renderIf isTrue="{!v.truthy}">
    True
    <aura:set attribute="else">
      False
    </aura:set>
  </aura:renderIf>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
else	Component[]	isTrue が false と評価されたときに表示する代替内容で、ボディは表示されません。<aura:set> タグを使用して設定します。	
isTrue	Boolean	コンポーネントのボディを表示するために true と評価される必要がある式。	はい

aura:template

Aura フレームワークのブートストラップに使用されるデフォルトテンプレート。別のテンプレートを使用するには、aura:template を拡張し、aura:set を使用して属性を設定します。

属性

属性名	属性型	説明	必須項目
auraPreInitBlock	Component[]	Aura 初期化の前に表示されるコンテンツブロック。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
bodyClass	String	追加ボディの CSS スタイル。	
defaultBodyClass	String	デフォルトボディの CSS スタイル。	
doctype	String	テンプレートの DOCTYPE 宣言。	
errorMessage	String	エラーの読み込みテキスト。	

属性名	属性型	説明	必須項目
errorTitle	String	エラーが発生した場合のエラータイトル。	
loadingText	String	読み込みテキスト。	
title	String	テンプレートのタイトル。	

aura:text

プレーンテキストを表示します。マークアップでフリーテキスト (タグまたは属性値ではない) が検出されると、マークアップで検出されたテキストに設定された value 属性を使用して、このコンポーネントのインスタンスが作成されます。

属性

属性名	属性型	説明	必須項目
value	String	表示する文字列。	

aura:unescapedHtml

このコンポーネントに割り当てられた値は、内容が変更されず、そのまま表示されます。たとえば、書式設定が任意の場合や、計算に手間がかかるなどの場合に、書式設定済みの HTML を出力するために使用します。このコンポーネントのボディは無視され、表示されません。警告: このコンポーネントの出力値はエスケープ解除された HTML であるため、コードにセキュリティの脆弱性が生じる可能性があります。エスケープ解除された状態で表示する前に、ユーザ入力の不要部分を削除する必要があります。このようにしないと、クロスサイトスクリプト (XSS) の脆弱性が生じます。<aura:unescapedHtml> は、信頼できるか不要部分が削除されたデータソースでのみ使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	<aura:unescapedHtml> のボディは無視され、表示されません。	
value	String	エスケープ解除された HTML として表示する文字列。	

auraStorage:init

指定された条件を満たすアダプタを使用してストレージインスタンスを初期化します。

auraStorage:init は、サーバ側のアクション応答値をキャッシュするためにアプリケーションのテンプレートでストレージを初期化する場合に使用します。

この例では、テンプレートを使用してサーバ側のアクション応答値のストレージを初期化します。テンプレートには、ストレージの初期化プロパティを指定する auraStorage:init タグが含まれています。

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="auraPreInitBlock">
        <!-- Note that the maxSize attribute in auraStorage:init is in KB -->
        <auraStorage:init name="actions" persistent="false" secure="false"
             maxSize="1024" />
    </aura:set>
</aura:component>
```

ストレージを初期化するときに、名前、最大キャッシュサイズ、デフォルトの有効期限など、いくつかのオプションを設定できます。

サーバ側アクションのストレージには、アクション応答値がキャッシュされます。ストレージ名は actions にする必要があります。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
clearStorageOnInit	Boolean	初期化時に以前のすべてのデータを削除する場合は True に設定します (永続ストレージにのみ関連します)。この値のデフォルトは true です。	
debugLoggingEnabled	Boolean	$A.log() でのデバッグログを有効にするには True に設定します。この値のデフォルトは false です。	
defaultAutoRefreshInterval	Integer	自動更新要求が開始されるまでのデフォルト期間 (秒数)。Action.setStorable() を使用してエントリ単位にアクションがこれを上書きする場合があります。この値のデフォルトは 30 です。	
defaultExpiration	Integer	オブジェクトがストレージに保持されるデフォルト期間 (秒数)。Action.setStorable() を使用してエントリ単位にアクションがこれを上書きする場合があります。この値のデフォルトは 10 です。	
maxSize	Integer	ストレージインスタンスの最大サイズ (KB)。既存の項目は、新規項目用の領域を確保するために強制削除されます。アルゴリズムはアダプタ固有です。この値のデフォルトは 1000 です。	

属性名	属性型	説明	必須項目
name	String	ストレージインスタンスのプログラムでの名前。	はい
persistent	Boolean	このストレージに永続性が必要な場合は true に設定します。この値のデフォルトは false です。	
secure	Boolean	このストレージにセキュアなストレージサポートが必要な場合は true に設定します。この値のデフォルトは false です。	
version	String	保存されているすべての項目に関連付けるバージョン。	

force:canvasApp

Lightning コンポーネントに Force.com Canvas アプリケーションを含めることができます。

force:canvasApp コンポーネントは、Lightning コンポーネントに埋め込まれたキャンバスアプリケーションを表します。任意の言語でWebアプリケーションを作成し、Salesforceでキャンバスアプリケーションとして公開できます。Lightningコンポーネントに埋め込む前にキャンバスアプリケーションをテストおよびデバッグするにはキャンバスアプリケーションのプレビューアを使用します。

名前空間プレフィックスがある場合は、namespacePrefix 属性を使用して指定します。developerName または applicationName 属性のいずれかが必要です。次の例では、キャンバスアプリケーションを Lightning コンポーネントに埋め込みます。

```
<aura:component>
    <force:canvasApp developerName="MyCanvasApp" namespacePrefix="myNamespace" />
</aura:component />
```

キャンバスアプリケーション作成についての詳細は、『Force.com Canvas 開発者ガイド』を参照してください。

属性

属性名	属性型	説明	必須項目
applicationName	String	キャンバスアプリケーションの名前。applicationName または developerName が必要です。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
border	String	キャンバスアプリケーションの境界線の幅(ピクセル単位)。指定されていない場合、デフォルトの 0 ピクセルに設定されます。	

属性名	属性型	説明	必須項目
canvasId	String	キャンバスアプリケーションウィンドウのページ内の一意の表示ラベル。イベントをこのキャンバスアプリケーションの対象にする場合に使用します。	
containerId	String	キャンバスアプリケーションが表示される html 要素 ID。canvasApp コンポーネントを使用する前に、コンテナを定義する必要があります。	
developerName	String	キャンバスアプリケーションの開発者名。この名前は、キャンバスアプリケーションが作成されたときに定義され、キャンバスアプリケーションのプレビューアで表示されます。developerName または applicationName が必要です。	
displayLocation	String	現在キャンバスアプリケーションをコールしているアプリケーションの場所。	
height	String	キャンバスアプリケーションウィンドウの高さ (ピクセル単位)。指定されていない場合、デフォルトの 900 ピクセルに設定されます。	
maxHeight	String	キャンバスアプリケーションウィンドウの高さの最大値 (ピクセル)。デフォルトは 2000 ピクセルで、「infinite」も有効な値です。	
maxWidth	String	キャンバスアプリケーションウィンドウの幅の最大値 (ピクセル)。デフォルトは 1000 ピクセルで、「infinite」も有効な値です。	
namespacePrefix	String	キャンバスアプリケーションが作成された Developer Edition 組織の名前空間の値。キャンバスアプリケーションが Developer Edition 組織で作成されていない場合は省略できます。指定されていない場合、デフォルトの null に設定されます。	
onCanvasAppError	String	キャンバスアプリケーションがレンダリングできないときにコールされる JavaScript 関数の名前。	
onCanvasAppLoad	String	キャンバスアプリケーションがロードした後にコールされる JavaScript 関数の名前。	
onCanvasSubscribed	String	キャンバスアプリケーションを親に登録した後にコールされる JavaScript 関数の名前。	
parameters	String	キャンバスアプリケーションに渡されるパラメータのオブジェクト表現。JSON 形式または JavaScript オブジェクトリテラルとして指定する必要があります。JavaScript オブジェクトリテラルとしてのパラメータの例:	

属性名	属性型	説明	必須項目
		{param1:'value1',param2:'value2'}。指定されていない場合、デフォルトの null に設定されます。	
referenceId	String	キャンバスアプリケーションの参照ID。設定されている場合、developerName、applicationName、および namespacePrefix の代わりに使用されます。	
scrolling	String	キャンバスウィンドウのスクロール。	
sublocation	String	sublocation は、現在キャンバスアプリケーションをコールしているアプリケーション内の場所です。たとえば、displayLocation が PageLayout の場合、sublocation は S1MobileCardPreview または S1MobileCardFullview などのようになります。	
title	String	リンクのタイトル。	
watermark	Boolean	true に設定されている場合、リンクが表示されます。	
width	String	キャンバスアプリケーションウィンドウの幅 (ピクセル単位)。指定されていない場合、デフォルトの 800 ピクセルに設定されます。	

force:inputField

バインドされるデータに基づいて型固有の具体的な入力コンポーネントの実装を提供するコンポーネント。

Salesforce オブジェクトの項目に対応する入力項目を表します。このコンポーネントは、関連付けられた項目の属性を考慮します。たとえば、コンポーネントが小数点以下の桁数が 2 桁の数値項目である場合、デフォルトの入力値には同じ小数点以下の桁数が含まれます。データ型に応じて入力項目を読み込みます。コンポーネントが日付項目に対応している場合、項目に日付ピッカーが表示されます。連動選択リストとリッチテキスト項目はサポートされていません。必須項目はクライアント側では強制されません。

次の例では、取引先責任者名のデータを表示する入力項目を作成します。`value` 属性を使用して項目をバインドし、デフォルト値を指定してオブジェクトを初期化します。

```
<aura:component controller="ContactController">
    <aura:attribute name="contact" type="Contact"
              default="{ 'sobjectType': 'Contact' }"/>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />
    <force:inputField value="{!v.contact.Name}"/>
</aura:component>
```

この例では、`v.contact.Name` 式が値を取引先責任者の [名前] 項目にバインドしています。レコードデータを読み込むには、取引先責任者を返す Apex コントローラにコンテナコンポーネントを接続します。

```
public with sharing class ContactController {
    @AuraEnabled
```

```
    public static Contact getContact() {
        return [select Id, Name from Contact Limit 1];
    }
}
```

クライアント側コントローラを介して取引先責任者データをコンポーネントに渡します。

```
({
     doInit : function(component, event, helper) {
        var action = component.get("c.getContact");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                component.set("v.contact", response.getReturnValue());
                console.log(response.getReturnValue());
            }
         });
         $A.enqueueAction(action);
    }
})
```

このコンポーネントは、Lightning Design System スタイル設定を使用しません。Lightning Design System スタイル設定を継承する入力項目が必要な場合は、`lightning:input` を使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	項目の表示に使用される CSS スタイル。	
errorComponent	Component[]	エラーメッセージの表示を行うコンポーネント。	
required	Boolean	この項目が必須かどうかを指定します。	
value	Object	バインドする Salesforce 項目のデータ値。	

イベント

イベント名	イベントタイプ	説明
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。

force:outputField

バインドされるデータに基づいて型固有の具体的な出力コンポーネントの実装を提供するコンポーネント。

Salesforce オブジェクトの項目値の参照のみ表示を表します。このコンポーネントは、関連項目およびその表示方法の属性を考慮します。たとえば、コンポーネントに日時値が含まれる場合、デフォルトの出力値にはユーザのロケールでの日時が含まれます。

次の例では、取引先責任者名のデータを表示します。`value` 属性を使用して項目をバインドし、デフォルト値を指定してオブジェクトを初期化します。

```
<aura:component controller="ContactController">
    <aura:attribute name="contact" type="Contact"
               default="{ 'sobjectType': 'Contact' }"/>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />
    <force:outputField value="{!v.contact.Name}"/>
</aura:component>
```

レコードデータを読み込むには、取引先責任者を返す Apex コントローラにコンテナコンポーネントを接続します。

```
public with sharing class ContactController {
    @AuraEnabled
    public static Contact getContact() {
        return [select Id, Name from Contact Limit 1];
    }
}
```

クライアント側コントローラを介して取引先責任者データをコンポーネントに渡します。

```
({
    doInit : function(component, event, helper) {
        var action = component.get("c.getContact");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                component.set("v.contact", response.getReturnValue());
                console.log(response.getReturnValue());
            }
         });
         $A.enqueueAction(action);
    }
})
```

このコンポーネントは、Lightning Design System スタイル設定を使用しません。Lightning Design System スタイル設定を継承する入力項目が必要な場合は、`lightning:input` を使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
value	Object	バインドする Salesforce 項目のデータ値。	

force:recordData

Lightning で Salesforce レコードの作成、読み取り、更新、削除を行うことができます。

force:recordData コンポーネントでは、Lightning データサービスによるレコードへのアクセス、変更、作成のパラメータを定義します。

```
<aura:component>
    <force:recordData aura:id="forceRecordCmp"
        recordId="{!v.recordId}"
        layoutType="{!v.layout}"
        fields="{!v.fieldsToQuery}"
        mode="VIEW"
        targetRecord="{!v.record}"
        targetFields="{!v.simpleRecord}"
        targetError="{!v.error}" />
</aura:component>
```

メソッド

このコンポーネントは、次のメソッドをサポートします。

- getNewRecord: レコードテンプレートを読み込んで、targetRecord 属性 (オブジェクトおよびレコードタイプに定義済みの値を含む) に設定します。
- reloadRecord: 現在の設定値 (recordId、layoutType、mode など) を使用して、初期化時と同じ読み込み関数を実行します。必要な場合を除き、サーバとの往復は強制されません。
- saveRecord: レコードを保存します。
- deleteRecord: レコードを削除します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
fields	ArrayList	クエリするレコードの項目を指定します。	

属性名	属性型	説明	必須項目
layoutType	String	クエリするレイアウトの名前。含まれる項目が決まります。有効な値は、FULL または COMPACT です。layoutType または fields 属性の指定は必須です。	
mode	String	レコードを読み込むモード: VIEW (デフォルト) または EDIT。	
recordId	String	レコード ID	
targetError	String	レコードを提供できない場合にローカライズされたエラーメッセージに設定されます。	
targetFields	Object	コンポーネントマークアップでレコード項目を参照するための targetRecord の項目の簡易ビュー。	
targetRecord	Object	提供されたレコード。この属性には、要求されたlayoutType または fields 属性に関連する項目のみが含まれます。	

イベント

イベント名	イベントタイプ	説明
recordUpdated	COMPONENT	レコードが変更されたときに起動したイベント。

force:recordEdit

指定された Salesforce レコードの編集可能なビューを生成します。

force:recordEdit コンポーネントは、指定された recordId のレコード編集 UI を表します。

次の例に、レコード編集 UI と、押したときにレコードが保存されるボタンを示します。

```
<force:recordEdit aura:id="edit" recordId="a02D0000006V8Ni"/>
<ui:button label="Save" press="{!c.save}"/>
```

このクライアント側のコントローラは、レコードを保存する recordSave イベントを起動します。

```
save : function(component, event, helper) {
component.find("edit").get("e.recordSave").fire();
}
```

recordId 属性には、{!v.myObject.recordId} 形式で動的IDを指定できます。レコードデータを読み込むには、データを返すApexコントローラにコンテナコンポーネントを接続します。詳細は、『Lightningコンポーネント開発者ガイド』の「Salesforce レコードの操作」を参照してください。

レコードが正常に保存されたことを示すには、force:recordSaveSuccess イベントを処理します。

スタンドアロンアプリケーションでこのコンポーネントを使用するには、コンポーネントのスタイルが適切に設定されるように force:slds を拡張します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
recordId	String	読み込むレコードの ID。record 属性が指定されている場合は省略可能です。	

イベント

イベント名	イベントタイプ	説明
recordSave	COMPONENT	レコード保存の要求を示すためにユーザが起動するイベント。
onSaveSuccess	COMPONENT	レコードの保存が正常に行われたときに起動されます。

force:utilityBarAPI

utilitybar の公開 API

このコンポーネントでは、Lightningアプリケーションのユーティリティバー内のユーティリティをプログラムで制御するメソッドにアクセスできます。ユーティリティバーは、ユーザが頻繁に使用するツールやコンポーネントにすばやくアクセスできるようにするためのフッターです。各ユーティリティは、標準またはカスタム Lightning コンポーネントが含まれる 1 列の Lightning ページです。

メソッドにアクセスするには、ユーティリティ内の force:utilityBarAPI コンポーネントのインスタンスを作成し、aura:id 属性を割り当てます。

```
<force:utilityBarAPI aura:id="utilitybar"/>
```

次の例では、ボタンをクリックすると、ユーティリティのアイコンがSLDSの「insert tag field (タグ項目を挿入)」アイコンに設定されます。

```
<aura:component implements="flexipage:availableForAllPageTypes" access="global" >
    <force:utilityBarAPI aura:id="utilitybar" />
    <lightning:button label="Set Utility Icon" onclick="{! c.setUtilityIcon }" />
</aura:component>
```

コンポーネントのボタンにより、次のクライアント側コントローラがコールされます。

```
({
    setUtilityIcon : function(component, event, helper) {
        var utilityAPI = component.find("utilitybar");
        utilityAPI.setUtilityIcon({sldsKey: 'insert_tag_field'});
    }
})
```

メソッド

このコンポーネントは、次のメソッドをサポートします。大部分のメソッドは 1 つの引数 (パラメータの JSON 配列) のみを取ります。utilityId パラメータは、ユーティリティ自体内の場合にのみ省略可能です。これらのメソッドについての詳細は、『コンソール開発者ガイド』を参照してください。

`getEnclosingUtilityId()`

`getUtilityInfo({utilityId})`

- `utilityId` (string): 省略可能。情報を取得するユーティリティの ID。

`getAllUtilityInfo()`

`minimizeUtility({utilityId})`

- `utilityId` (string): 省略可能。最小化するユーティリティの ID。

`openUtility({utilityId})`

- `utilityId` (string): 省略可能。開くユーティリティの ID。

`setPanelHeaderIcon({sldsKey, utilityId})`

- `sldsKey` (string): SLDS ユーティリティアイコンキー。ユーティリティパネルに表示されます。SLDS リファレンスサイトでユーティリティアイコンキーの完全なリストを確認してください。
- `utilityId` (string): 省略可能。パネルヘッダーアイコンを設定するユーティリティの ID。

`setPanelHeaderLabel({label, utilityId})`

- `label` (string): パネルヘッダーに表示されるユーティリティの表示ラベル。
- `utilityId` (string): 省略可能。パネルヘッダー表示ラベルを設定するユーティリティの ID。

`setPanelHeight({heightPX, utilityId})`

- `heightPX` (integer): ユーティリティパネルの高さ (ピクセル単位)。
- `utilityId` (string): 省略可能。パネルの高さを設定するユーティリティの ID。

`setPanelWidth({widthPX, utilityId})`

- `widthPX` (integer): ユーティリティパネルの幅 (ピクセル単位)。
- `utilityId` (string): 省略可能。パネルの幅を設定するユーティリティの ID。

`setUtilityHighlighted({highlighted, utilityId})`

- `highlighted` (boolean): ユーティリティを強調表示するかどうか。異なる背景色を設定してユーティリティを強調表示します。
- `utilityId` (string): 省略可能。強調表示するユーティリティの ID。

`setUtilityIcon({sldsKey, utilityId})`

- `sldsKey` (string): SLDS ユーティリティアイコンキー。ユーティリティバーに表示されます。SLDS リファレンスサイトでユーティリティアイコンキーの完全なリストを確認してください。
- `utilityId` (string): 省略可能。アイコンを設定するユーティリティの ID。

`setUtilityLabel({label, utilityId})`

- `label` (string): ユーティリティの表示ラベル。ユーティリティバーに表示されます。
- `utilityId` (string): 省略可能。表示ラベルを設定するユーティリティの ID。

`toggleModalMode({enableModalMode, utilityId})`

- `enableModalMode` (boolean): ユーティリティのモーダルモードを有効にするかどうか。モーダルモードの場合、オーバーレイがアプリケーション全体に表示されて、ユーティリティパネルが表示されていても使用できなくなります。
- `utilityId` (string): 省略可能。モーダルモードを切り替えるユーティリティの ID。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

force:workspaceAPI

これは、ワークスペース (タブおよびサブタブ) へのアクセスやその操作を行う公開 API です。

このコンポーネントでは、Lightningコンソールアプリケーションのワークスペースタブおよびサブタブをプログラムで制御するメソッドにアクセスできます。Lightningコンソールアプリケーションでは、レコードはワークスペースタブとして開き、関連レコードはサブタブとして開きます。

メソッドにアクセスするには、`force:workspaceAPI` コンポーネントのインスタンスを作成し、`aura:id` 属性を割り当てます。

```
<force:workspaceAPI aura:id="workspace"/>
```

次の例では、ボタンをクリックすると、`url` 属性に指定された相対URLでレコードを表示する新しいタブが開きます。

```
<aura:component implements="flexipage:availableForAllPageTypes" access="global" >
    <force:workspaceAPI aura:id="workspace" />
    <lightning:button label="Open Tab" onclick="{! c.openTab }" />
</aura:component>
```

コンポーネントのボタンにより、次のクライアント側コントローラがコールされます。

```
({
    openTab : function(component, event, helper) {
        var workspaceAPI = component.find("workspace");
        workspaceAPI.openTab({
            url: '#/sObject/001R0000003HgssIAC/view',
            focus: true
        });
    },
})
```

メソッド

このコンポーネントは、次のメソッドをサポートします。大部分のメソッドは 1 つの引数 (パラメータの JSON 配列) のみを取ります。これらのメソッドについての詳細は、『コンソール開発者ガイド』を参照してください。

`closeTab({tabId})`

- `tabId` (string): 閉じるワークスペースタブまたはサブタブの ID。

`focusTab({tabId})`

- `tabId` (string): フォーカスするワークスペースタブまたはサブタブの ID。

`getAllTabInfo()`

`getFocusedTabInfo()`

`getTabInfo({tabId})`

- `tabId` (string): 情報を取得するタブの ID。

`getTabURL({tabId})`

- `tabId` (string): URL を取得するタブの ID。

`isSubtab({tabId})`

- `tabId` (string): タブの ID。

`isConsoleNavigation()`

`getEnclosingTabId()`

`openSubtab({parentTabId, url, recordId, focus})`

- `parentTabId` (string): 新しいサブタブを開くワークスペースタブの ID。
- `url` (string): 省略可能。新しいサブタブのコンテンツを表す URL。URL は、相対 URL または絶対 URL になります。
- `recordId` (string): 省略可能。新しいサブタブのコンテンツを表すレコード ID。
- `focus` (boolean): 省略可能。新しいサブタブにフォーカスするかどうかを指定します。

`openTab({url, recordId, focus})`

- `url` (string): 省略可能。新しいタブのコンテンツを表す URL。URL は、相対 URL または絶対 URL になります。
- `recordId` (string): 省略可能。新しいタブのコンテンツを表すレコード ID。
- `focus` (boolean): 省略可能。新しいタブにフォーカスするかどうかを指定します。
- `overrideNavRules` (boolean): 省略可能。新しいタブを開いたときにナビゲーションルールを上書きするかどうかを指定します。

`setTabIcon({tabId, icon, iconAlt})`

- `tabId` (string): アイコンを設定するタブの ID。
- `icon` (string): SLDS アイコンキー。SLDS リファレンスサイトでアイコンキーの完全なリストを確認してください。
- `iconAlt` (string): 省略可能。アイコンの代替テキスト。

`setTabLabel({tabId, label})`

- `tabId` (string): 表示ラベルを設定するタブの ID。
- `label` (string): ワークスペースタブまたはサブタブの表示ラベル。

`setTabHighlighted({tabId, highlighted})`

- `tabId` (string): 強調表示するタブの ID。

- highlighted (boolean): 新しいタブを強調表示するかどうかを指定します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

force:recordPreview

force:recordPreview は廃止される予定です。代わりに force:recordData を使用します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

getNewRecord: レコードテンプレートを読み込んで、force:recordPreview の targetRecord 属性 (エンティティおよびレコードタイプに定義済みの値を含む) に設定します。

reloadRecord:現在の設定値(recordId、layoutType、mode など)を使用して、初期化時と同じ読み込み関数を実行します。必要な場合を除き、サーバとの往復は強制されません。

saveRecord: レコードを保存します。

deleteRecord: レコードを削除します。

属性

属性名	属性型	説明	必須項目?
fields	String []	クエリする項目のリスト。 この属性または layoutType を指定する必要があります。両方を指定した場合は、必須項目のリストに、fields の項目と layoutType の項目がまとめられます。	
ignoreExistingAction	Boolean	キャッシュを省略してサーバ要求を強制するかどうか。デフォルトは false です。 この属性を true に設定すると、「プルして更新」などユーザによってトリガされるアクションの処理に役立ちます。	
layoutType	String	クエリするレイアウトの名前。含まれる項目が決まります。有効な値は次のとおりです。 • FULL	

属性名	属性型	説明	必須項目?
		• COMPACT この属性または fields を指定する必要があります。両方を指定した場合は、必須項目のリストに、fields の項目と layoutType の項目がまとめられます。	
mode	String	レコードにアクセスするモード。有効な値は次のとおりです。 • VIEW • EDIT デフォルトは VIEW です。	
recordId	String	読み込み、変更、削除するレコードの 15 文字または 18 文字の ID。デフォルトは null で、レコードを作成します。	
targetError	String	必要に応じて、ローカライズされたエラーメッセージが割り当てられるコンポーネント属性への参照。	
targetRecord	Record	読み込まれたレコードが割り当てられるコンポーネント属性への参照。 レコードへの変更もこの値に割り当てられ、変更ハンドラや再表示などがトリガされます。	

イベント

イベント名	イベントタイプ	説明
recordUpdated	COMPONENT	レコードが読み込み、変更、更新、削除されたときに起動されるイベント。

force:recordView

指定された Salesforce レコードのビューを生成します。

force:recordView コンポーネントは、レコードの参照のみのビューを表します。異なるレイアウト種別を使用して、レコードを表示できます。デフォルトでは、レコードビューでフルレイアウトを使用して、レコードのすべての項目が表示されます。ミニレイアウトには、コンパクトレイアウトに対応する項目が表示されます。項目およびコンポーネントに表示される順序を変更する場合は、特定のオブジェクトの [設定] にある [コンパクトレイアウト] に移動します。

次の例に、ミニレイアウトを使用したレコードビューを示します。

```
<force:recordView recordId="a02D0000006V8Ov" type="MINI"/>
```

`recordId` 属性には、`{!v.myObject.recordId}` 形式で動的IDを指定できます。レコードデータを読み込むには、データを返すApexコントローラにコンテナコンポーネントを接続します。詳細は、『Lightningコンポーネント開発者ガイド』の「Salesforce レコードの操作」を参照してください。

スタンドアロンアプリケーションでこのコンポーネントを使用するには、コンポーネントのスタイルが適切に設定されるように `force:slds` を拡張します。

属性

属性名	属性型	説明	必須項目
`body`	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
`record`	SObjectRow	読み込むレコード (SObject)。recordId 属性が指定されている場合は省略可能です。	
`recordId`	String	読み込むレコードの ID。record 属性が指定されている場合は省略可能です。	
`type`	String	レコードの表示に使用されるレイアウトの種別。使用可能な値: FULL、MINI。デフォルトは FULL です。	

`forceChatter:feed`

Chatter フィードを表します。

`forceChatter:feed` コンポーネントは、種別で指定されたフィードを表します。`type` 属性を使用して、特定のフィード種別を表示します。たとえば、コンテキストユーザが所有するか、メンバーであるすべてのグループのフィードを表示するには、`type="groups"` を設定します。

```
<aura:component implements="force:appHostable">
    <forceChatter:feed type="groups"/>
</aura:component>
```

また、選択した種別に応じてフィードを表示することもできます。次の例は、表示するフィードの種別を制御するドロップダウンメニューを表示します。

```
<aura:component implements="force:appHostable">
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <aura:attribute name="options" type="List" />
   <aura:attribute name="type" type="String" default="News" description="The type of feed"
 access="GLOBAL"/>
    <aura:attribute name="types" type="String[]"

default="Bookmarks,Company,DirectMessages,Feeds,Files,Filter,Groups,Home,Moderation,Mute,News,PendingReview,Record,Streams,To,Topics,UserProfile"

                    description="A list of feed types"/>
   <h1>My Feeds</h1>
<lightning:select aura:id="typeSelect" onchange="{!c.onChangeType}" label="Type"
```

```
name="typeSelect">
        <aura:iteration items="{!v.options}" var="item">
          <option text="{!item.label}" value="{!item.value}" selected="{!item.selected}"/>

        </aura:iteration>
    </lightning:select>
    <div aura:id="feedContainer" class="feed-container">
        <forceChatter:feed />
    </div>
</aura:component>
```

types 属性は、コンポーネントの初期化時に lightning:select コンポーネントで設定されるフィード種別を指定します。ユーザがフィード種別を選択すると、フィードが動的に作成され、表示されます。

```
({
    // Handle component initialization
    doInit : function(component, event, helper) {
        var type = component.get("v.type");
        var types = component.get("v.types");
        var opts = new Array();

        // Set the feed types on the lightning:select component
        for (var i = 0; i < types.length; i++) {
            opts.push({label: types[i], value: types[i], selected: types[i] === type});
        }
        component.set("v.options", opts);
    },

 onChangeType : function(component, event, helper) {
        var typeSelect = component.find("typeSelect");
        var type = typeSelect.get("v.value");
        component.set("v.type", type);

        // Dynamically create the feed with the specified type
        $A.createComponent("forceChatter:feed", {"type": type}, function(feed) {
            var feedContainer = component.find("feedContainer");
            feedContainer.set("v.body", feed);
        });
    }
})
```

フィードコンポーネントは、Lightning Experience と、カスタマーサービステンプレートに基づくコミュニティでサポートされます。

フィード種別のリストは、『Chatter REST API 開発者ガイド』を参照してください。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
feedDesign	String	有効な値には、DEFAULT (インラインコメントをデスクトップに多少詳しく表示) または BROWSE (主にフィード項目の概要) が含まれます。	
subjectId	String	エンティティに関連付けられているほとんどのフィードの場合、目的のエンティティを指定するために使用されます。指定されていない場合、デフォルトの現在のユーザに設定されます。	
type	String	件名に関連付けられている項目の検索に使用される方法。有効な値は、Bookmarks、Company、DirectMessages、Feeds、Files、Filter、Groups、Home、Moderation、Mute、News、PendingReview、Record、Streams、To、Topics、UserProfile です。	

forceChatter:fullFeed

完全な長さの Chatter フィード。

fullFeed コンポーネントはまだベータと見なされており、本番環境への準備ができていません。

fullFeed コンポーネントは、Lightning Out か、その他の Salesforce1 および Lightning Experience 以外のアプリケーションで使用することを目的としています。

現時点で Lightning Experience に fullFeed コンポーネントを追加すると、UI で投稿が一時的に重複するなど、予期しない動作が発生します。Lightning Experience で Chatter フィードを実装するには、`forceChatter:publisher` および `forceChatter:feed` を使用してください。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
handleNavigationEvents	Boolean	このコンポーネントがエンティティと URL のナビゲーションイベントを処理するかどうか。trueの場合、ナビゲーションイベントによって新しいウィンドウでエンティティまたは URL が開かれます。	
subjectId	String	エンティティに関連付けられているほとんどのフィードの場合、目的のエンティティを指定するために使用されます。指定されていない場合、デフォルトの現在のユーザに設定されます。	

属性名	属性型	説明	必須項目
type	String	件名に関連付けられている項目の検索に使用される方法。有効な値は、News、Home、Record、To です。	

forceChatter:publisher

ユーザは、レコードまたはグループに対して投稿を作成したり、デスクトップの Lightning Experience やコミュニティおよびモバイルデバイスのコミュニティから添付ファイルをアップロードしたりできます。このコンポーネントは、モバイルデバイスの Lightning Experience では使用できません。

forceChatter:publisher コンポーネントは、レコードページに配置できるスタンドアロンのパブリッシャーコンポーネントです。Lightning アプリケーションビルダーで使用可能な forceChatter:feed コンポーネントと連動して、完全なChatter操作環境を提供します。パブリッシャーとフィードのコンポーネントを別個にすると、ページコンポーネントを柔軟に配置できるという利点があります。パブリッシャーとフィード間の接続は自動的に行われ、追加のコーディングは不要です。

forceChatter:publisher コンポーネントには、表示されるフィード種別を決定する context 属性が含まれます。レコードフィードには RECORD、その他すべてのフィード種別には GLOBAL を使用します。

```
<aura:component implements="flexipage:availableForAllPageTypes" description="Sample
Component">
    <forceChatter:publisher context="GLOBAL" />
    <forceChatter:feed type="Company" />
</aura:component>
```

このコンポーネントは、Lightning Experience と、カスタマーサービステンプレートに基づくコミュニティでサポートされます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
context	String	コンポーネントが表示されているコンテキスト (RECORD または GLOBAL)。レコードフィードの場合は RECORD、その他すべてのフィード種別の場合は GLOBAL です。この属性は大文字と小文字を区別します。	はい
recordId	String	レコード ID	

forceCommunity:appLauncher

Lightning コミュニティにアプリケーションランチャーを表示して、ユーザがコミュニティと Salesforce 組織の間を容易に移動できるようにします。このコンポーネントは、コミュニティの任意のカスタム Lightning コンポーネントに追加できます。

`forceCommunity:appLauncher` コンポーネントはアプリケーションランチャーアイコンを表します。このアイコンをクリックすると、コミュニティ、接続アプリケーション、Salesforce アプリケーション、およびオンプレミス型アプリケーションにリンクされたタイルが表示されます。メンバーに表示されるのは、プロファイルや権限セットに従って表示が許可されるコミュニティとアプリケーションのみです。メンバーがアプリケーションランチャーを表示できるようにするには、[設定] のユーザプロファイルで [コミュニティでアプリケーションランチャーを表示] 権限も有効にする必要があります。このコンポーネントは、Salesforce1 モバイルアプリケーションまたは Salesforce タブ + Visualforce コミュニティでは使用できません。

```
<aura:component>
    <forceCommunity:appLauncher/>
</aura:component>
```

アプリケーションランチャーをカスタムテーマレイアウトに追加すると、そのカスタムテーマレイアウトを使用するすべてのページに表示されます。

デフォルトのナビゲーションメニューを使用し、`forceCommunity:appLauncher` を含むカスタムテーマレイアウトコンポーネントの例を次に示します。

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample
 Custom Theme Layout">
    <aura:attribute name="search" type="Aura.Component[]" required="false"/>
    <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
    <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
    <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
    <div>
        <div class="appLauncher">
           <forceCommunity:appLauncher/>
        </div>
        <div class="searchRegion">
           {!v.search}
        </div>
        <div class="profileMenuRegion">
            {!v.profileMenu}
        </div>
        <div class="navigation">
            {!v.navBar}
        </div>
        <div class="newHeader">
            {!v.newHeader}
        </div>
        <div class="mainContentArea">
            {!v.body}
        </div>
    </div>
</aura:component>
```

アプリケーションランチャーは、デフォルトのナビゲーションメニューに含まれているものを使用することも、カスタムテーマレイアウトに含めてデフォルトのナビゲーションメニューのアプリケーションランチャー

を非表示にすることもできます。デフォルトのナビゲーションメニューに含まれるアプリケーションランチャーを削除するには、コミュニティビルダーの[ナビゲーションメニュー]プロパティエディタで[コミュニティヘッダーにアプリケーションランチャーを表示しない] を選択します。

または、forceCommunity:appLauncher コンポーネントを含むカスタムナビゲーションメニューを作成できます。そして、カスタムテーマレイアウトでこのメニューを使用します。

forceCommunity:appLauncher コンポーネントを含むカスタムナビゲーションメニューコンポーネントの例を次に示します。

```
<aura:component extends="forceCommunity:navigationMenuBase"
implements="forceCommunity:availableForAllPageTypes">
    <ul onclick="{!c.onClick}">
        <li><forceCommunity:appLauncher/></li>
        <aura:iteration items="{!v.menuItems}" var="item">
            <aura:if isTrue="{!item.subMenu}">
                <li>{!item.label}</li>
                    <ul>
                        <aura:iteration items="{!item.subMenu}" var="subItem">
                            <li><a data-menu-item-id="{!subItem.id}"
href="">{!subItem.label}</a></li>
                        </aura:iteration>
                    </ul>
                 <aura:set attribute="else">
                     <li><a data-menu-item-id="{!item.id}" href="">{!item.label}</a></li>

                 </aura:set>
             </aura:if>
        </aura:iteration>
    </ul>
</aura:component>
```

forceCommunity:appLauncher コンポーネントを含むカスタムナビゲーションメニューを使用するカスタムテーマレイアウトコンポーネントの例を次に示します。この例では、カスタムコンポーネント c:CustomNavMenu によってカスタムナビゲーションメニューが提供されています。

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample
 Custom Theme Layout">
    <aura:attribute name="search" type="Aura.Component[]" required="false"/>
    <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
    <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
    <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
    <div>
        <div class="searchRegion">
            {!v.search}
        </div>
        <div class="profileMenuRegion">
            {!v.profileMenu}
        </div>
        <div class="navigation">
            <c:CustomNavMenu/>
        </div>
        <div class="newHeader">
            {!v.newHeader}
        </div>
```

```
        <div class="mainContentArea">
            {!v.body}
        </div>
    </div>
</aura:component>
```

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

forceCommunity:navigationMenuBase

コミュニティのナビゲーションメニューをカスタマイズするための抽象コンポーネントで、メニューデータの読み込みとナビゲーションの処理を行います。メニューのデザインは、このメニューを拡張しているコンポーネントによって制御されます。

forceCommunity:navigationMenuBase コンポーネントを拡張して、カスタマーサービス (Napili) またはカスタムコミュニティテンプレート用にカスタマイズされたナビゲーションコンポーネントを作成します。コミュニティビルダーのメニューエディタまたは NavigationMenuItem エンティティを使用して、ナビゲーションメニューデータを指定します。

menuItems 属性には、それぞれ次のプロパティがある最上位のメニュー項目の配列が自動的に入力されます。

- id: navigate メソッドで使用されます。
- label: メニュー項目の表示ラベル。
- subMenu: メニュー項目の配列 (省略可能)。

次に、カスタムナビゲーションメニューコンポーネントの例を示します。

```
<aura:component extends="forceCommunity:navigationMenuBase"
implements="forceCommunity:availableForAllPageTypes">
    <ul onclick="{!c.onClick}">
        <aura:iteration items="{!v.menuItems}" var="item" >
            <aura:if isTrue="{!item.subMenu}">
                <li>{!item.label}</li>
                <ul>
                    <aura:iteration items="{!item.subMenu}" var="subItem">
                        <li><a data-menu-item-id="{!subItem.id}"
href="">{!subItem.label}</a></li>
                    </aura:iteration>
                </ul>
            <aura:set attribute="else">
                <li><a data-menu-item-id="{!item.id}" href="">{!item.label}</a></li>
            </aura:set>
            </aura:if>
        </aura:iteration>
```

```
    </ul>
</aura:component>
```

次に、コントローラの例を示します。

```
({
    onClick : function(component, event, helper) {
        var id = event.target.dataset.menuItemId;
        if (id) {
            component.getSuper().navigate(id);
         }
   }
})
```

メソッド

navigate(menuItemId): メニュー項目が指し示すページに移動します。メニュー項目の id をパラメータに取ります。

属性

属性名	属性型	説明	必須項目?
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
menuItems	Object	メニュー項目のデータが自動的に入力されます。この属性は参照のみです。	

forceCommunity:notifications

通知ツールを使用すると、メンバーはコミュニティやアプリケーションのどこで作業をしていても通知を受け取ることができます。メンバーは、モバイル、タブレット、デスクトップのすべての画面で通知を受け取ることができます。通知をトリガするすべての行動 (@メンションとグループ投稿) がサポートされています。メンバーが通知をクリックすると、元の詳細ページまたは他の該当する場所が表示され、コミュニティやアプリケーションを越えてシームレスにコラボレーションできます。

forceCommunity:notifications コンポーネントは通知アイコンを表します。通知は、Chatter 投稿でメンションされたときなどの重要なイベントが発生した場合のアラートとしてユーザに送信されます。このコンポーネントは、Lightning Experience、Salesforce1 モバイルアプリケーション、Lightning コミュニティでサポートされます。

```
<aura:component>
   <forceCommunity:notifications/>
</aura:component>
```

通知によって、ユーザはコミュニティやアプリケーションのどこで作業をしていても通知を受け取ることができます。通知をトリガするすべての行動 (@メンションとグループ投稿) がサポートされています。ユーザはレコードフィードで通知をトリガすることもできます。たとえば、内部ユーザはリードまたは商談で外部ユーザに @メンションして Salesforce 組織から通知をトリガできます。

forceCommunity:notifications を含むカスタムテーマレイアウトコンポーネントの例を次に示します。

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample
 Custom Theme Layout">
    <aura:attribute name="search" type="Aura.Component[]" required="false"/>
    <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
    <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
    <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
    <div>
        <div class="notifications">
               <forceCommunity:notifications/>
        </div>
        <div class="searchRegion">
            {!v.search}
        </div>
        <div class="profileMenuRegion">
            {!v.profileMenu}
        </div>
        <div class="navigation">
            {!v.navBar}
        </div>
        <div class="newHeader">
            {!v.newHeader}
        </div>
        <div class="mainContentArea">
            {!v.body}
        </div>
    </div>
</aura:component>
```

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

forceCommunity:routeLink

指定されたレコード ID から自動的に生成される href 属性を定義した HTML アンカータグを設定します。このタグは、テンプレートに基づくコミュニティで SEO リンクエクイティを改善するために使用します。

href 属性は指定されたレコード ID から自動的に生成されるため、forceCommunity:routeLink は、記事の詳細ページやケースの詳細ページなど、コミュニティの recordId ベースのページへの内部リンクの作成にのみ適しています。

内部リンクは、SEOに対応するサイト階層を確立し、リンクエクイティ(またはリンクジュース)をコミュニティのページに分散させるうえで役立ちます。

次に、`forceCommunity:routeLink` コンポーネントの例を示します。

```
<aura:component implements="forceCommunity:availableForAllPageTypes">
    <aura:attribute name="recordId" type="String" default="500xx000000YkvU" />
    <aura:attribute name="routeInput" type="Map"/>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <forceCommunity:routeLink id="myCaseId" class="caseClass" title="My Case Tooltip"
label="My Case Link Text" routeInput="{!v.routeInput}" onClick="{!c.onClick}"/>
</aura:component>
```

リンクを作成するには、初期化中にクライアント側コントローラで `routeInput` 属性のレコード ID を設定します。リンクをクリックすると、レコードページに移動できます。

```
({
    doInit : function(component, event, helper) {
    component.set('v.routeInput', {recordId: component.get('v.recordId')});
    },

    onClick : function(component, event, helper) {
           var navEvt = $A.get("e.force:navigateToSObject");
           navEvt.setParams({
             "recordId": component.get('v.recordId')
           });
           navEvt.fire();
   }
})
```

前の例によって次のアンカータグが表示されます。

```
<a class="caseClass" href="/myCommunity/s/case/500xx000000YkvU/mycase"
   id="myCaseId" title="My Case Tooltip">My Case Link Text</a>
```

属性

属性名	属性型	説明	必須項目?
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	アンカータグの CSS クラス。	
id	String	アンカータグの ID。	
label	String	リンクに表示されるテキスト。	

属性名	属性型	説明	必須項目?
onClick	Action	アンカーがクリックされたときにトリガされるアクション。	
routeInput	HashMap	リンクを作成する動的パラメータの対応付け。recordIdベースのルートのみがサポートされます。	はい
title	String	リンクツールチップに表示するテキスト。	

forceCommunity:waveDashboard

このコンポーネントは、Salesforce Analytics ダッシュボードをコミュニティページに追加する場合に使用します。

Analytics Wave ダッシュボードコンポーネントをコミュニティページに追加すれば、データの対話型の視覚化が可能になります。ユーザはコミュニティページまたは Analytics のウィンドウのフレーム内でダッシュボードのドリルインや探索を行うことができます。

Wave ダッシュボードコンポーネントは、カスタマーサービス (Napili) テンプレートでドラッグアンドドロップコンポーネントとして使用できますが、forceCommunity:waveDashboard を使用して独自の Wave ダッシュボードコンポーネントを作成することもできます。

次に、forceCommunity:waveDashboard コンポーネントの例を示します。

```
<aura:component implements="forceCommunity:availableForAllPageTypes">
        <forceCommunity:waveDashboard dashboardId="0FKxx000000000uGAA" />
        </aura:component>
```

認証

属性名	属性型	説明	必須
accessToken	String	Salesforce にログインすることによって取得される有効なアクセストークン。コンポーネントが非 Salesforce ドメインで Lightning Out によって使用される場合に有用です。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
dashboardId	String	ダッシュボードの一意の ID。ダッシュボードの URL からダッシュボードの ID (0FK から始まる 18 文字のコード) を取得できます。また、API を使用してこの ID を要求することもできます。開発者名の代わりにこの属性を使用できますが、名前が設定されている場合にこれを含めることはできません。2 つのいずれかは必要です。	

属性名	属性型	説明	必須
developerName	String	ダッシュボードの一意の開発者名。開発者名は API で要求できます。ダッシュボードIDの代わりにこの属性を使用できますが、IDが設定されている場合にこれを含めることはできません。2 つのいずれかは必要です。	
filter	String	実行時に埋め込みダッシュボードに選択または条件を追加します。filter属性はJSONを使用して設定されます。ディメンションで絞り込む場合、{'datasets' : {'dataset1': [{'fields': ['field1'], 'selection': ['$value1', '$value2']}, {'fields': ['field2'], 'filter': { 'operator': 'operator1', 'values': ['$value3', '$value4']}}]}} の構文を使用します。基準で絞り込む場合、{'datasets' : {'dataset1': [{'fields': ['field1'], 'selection': ['$value1', '$value2']}, {'fields': ['field2'], 'filter': { 'operator': 'operator1', 'values': [[$value3]]}}]}} の構文を使用します。選択オプションを使用すると、ダッシュボードにそのすべてのデータが表示され、指定したディメンション値が強調表示されます。検索オプションを使用すると、ダッシュボードには絞り込まれたデータのみが表示されます。詳細は、https://help.salesforce.com/articleView?id=bi_embed_community_builder.htm を参照してください。	
height	Integer	ダッシュボードの高さ (ピクセル単位) を指定します。	
hideOnError	Boolean	エラーのあるダッシュボードをユーザに表示するかどうかを制御します。この属性が true に設定されている場合、ダッシュボードにエラーがあってもページには表示されません。false に設定すると、ダッシュボードは表示されますが、データは表示されません。ユーザにダッシュボードへのアクセス権がないか削除されていると、エラーが発生する可能性があります。	
openLinksInNewWindow	Boolean	false にすると、その他のダッシュボードへのリンクが同じウィンドウで開きます。	
recordId	String	コンポーネントが表示されるコンテキスト内の現在のエンティティの ID。	
showHeader	Boolean	true の場合、ダッシュボードにはダッシュボード情報とコントロールを含むヘッダーバーが表示されます。false の場合、ダッシュボードはヘッダーバーなしで表示されます。showSharing または showTitle が true の場合、ヘッダーバーは自動的に表示されます。	
showSharing	Boolean	trueにすると、ダッシュボードが共有可能な場合、ダッシュボードに [共有] アイコンが表示されます。false にすると、ダッシュボードに [共有] アイコンは表示されません。ダッシュボードに [共有] アイコンを表示するためのサポートされる最小のフレームサイズは 800 × 612 ピクセルです。	

属性名	属性型	説明	必須
showTitle	Boolean	true にすると、ダッシュボードのタイトルがダッシュボードの上に表示されます。false の場合、タイトルなしでダッシュボードが表示されます。	

lightning:accordion

複数のコンテンツ領域を含むセクションを縦に積み上げたコレクション。同時に展開されるのは 1 つのみです。セクションを選択すると、展開されるか折りたたまれます。各セクションには、1 つ以上の Lightning コンポーネントを含めることができます。このコンポーネントはバージョン 41.0 以降で使用できます。

lightning:accordion コンポーネントは、複数のコンテンツ領域を含むセクションを縦に積み上げたコレクションを表示します。同時に展開されるのは1つのみです。セクションを選択すると、展開されるか折りたたまれます。各セクションには、1 つ以上の Lightning コンポーネントを含めることができます。

このコンポーネントは、Lightning Design System のアコーディオンからスタイル設定を継承します。

さらにこのコンポーネントのスタイルを設定するには、Lightning Design System ヘルパークラスを使用します。

次の例では、3 つのセクション (セクション B はデフォルトで展開される) がある基本的なアコーディオンを作成します。

```
<aura:component>
   <lightning:accordion activeSectionName="B">
     <lightning:accordionSection name="A" label="Accordion Title A">This is the content
area for section A</lightning:accordionSection>
     <lightning:accordionSection name="B" label="Accordion Title B">This is the content
area for section B</lightning:accordionSection>
     <lightning:accordionSection name="C" label="Accordion Title C">This is the content
area for section C</lightning:accordionSection>
   </lightning:accordion>
</aura:component>
```

使用上の考慮事項

lightning:accordion の最初のセクションはデフォルトで展開されます。デフォルトを変更するには、activeSectionName 属性を使用します。この属性は大文字と小文字を区別します。

複数のセクションで同じ名前が使用されていて、その名前が activeSectionName として指定されている場合、最初のセクションがデフォルトで展開されます。

属性

属性名	属性型	説明	必須項目
activeSectionName	String	activeSectionName は、デフォルトで展開されるセクションを変更します。アコーディオンの最初のセクションはデフォルトで展開されます。	

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	

lightning:accordionSection

lightning:accordion コンポーネント内にネストされた 1 つのセクション。lightning:accordionSection は、関連コンテンツを1つのコンテナに保持します。セクションを選択すると、展開されるか折りたたまれます。各セクションには、1つ以上のLightningコンポーネントを含めることができます。このコンポーネントは、lightning:accordion と併用します。このコンポーネントはバージョン 41.0 以降で使用できます。

lightning:accordion コンポーネント内にネストされた 1 つのセクション。

lightning:accordionSection は、関連コンテンツを1つのコンテナに保持します。セクションを選択すると、展開されるか折りたたまれます。各セクションには、1 つ以上の Lightning コンポーネントを含めることができます。このコンポーネントは、lightning:accordion と併用します。

このコンポーネントは、Lightning Design System のアコーディオンからスタイル設定を継承します。

さらにこのコンポーネントのスタイルを設定するには、Lightning Design System ヘルパークラスを使用します。

次の例では、3 つのセクション (セクション B はデフォルトで展開される) がある基本的なアコーディオンを作成します。

```
<aura:component>
   <lightning:accordion activeSectionName="B">
      <lightning:accordionSection label="Accordion Title A" name="A">This is the content
area for section A</lightning:accordionSection>
      <lightning:accordionSection label="Accordion Title B" name="B">This is the content
area for section B</lightning:accordionSection>
      <lightning:accordionSection label="Accordion Title C" name="C">This is the content
area for section C</lightning:accordionSection>
   </lightning:accordion>
</aura:component>
```

次の例では、同じ基本的なアコーディオンを作成しますが、最初のセクションに buttonMenu が追加されています。

```
<aura:component>
   <lightning:accordion>
      <lightning:accordionSection label="Accordion Title A" name="A">This is the content
area for section A
         <aura:set attribute="actions">
```

```
            <lightning:buttonMenu aura:id="menu" alternativeText="Show menu">
               <lightning:menuItem value="New" label="Menu Item One" />
            </lightning:buttonMenu>
         </aura:set>
      </lightning:accordionSection>
      <lightning:accordionSection label="Accordion Title B" name="B">This is the content
area for section B</lightning:accordionSection>
      <lightning:accordionSection label="Accordion Title C" name="C">This is the content
area for section C</lightning:accordionSection>
   </lightning:accordion>
</aura:component>
```

使用上の考慮事項

lightning:accordion の最初のセクションはデフォルトで展開されます。デフォルトを変更するには、activeSectionName 属性を使用します。

複数のセクションで同じ名前が使用されていて、その名前が activeSectionName として指定されている場合、最初のセクションがデフォルトで展開されます。

属性

属性名	属性型	説明	必須項目
actions	Component[]	カスタムメニュー実装を有効にします。アクションは、セクションタイトルの右側に表示されます。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
label	String	セクションのタイトルとして表示されるテキスト。	
name	String	lightning:accordion コンポーネントの activeSectionName 属性と共に使用する一意のセクション名。	
title	String	マウスが要素に重ねられたときにツールチップテキストを表示します。	

lightning:avatar

オブジェクトのビジュアル表現。

lightning:avatar コンポーネントは、取引先やユーザなどのオブジェクトを表す画像です。デフォルトで、この画像は中サイズの角丸長方形 (square バリエーションともいう) で表示されます。

このコンポーネントは、Lightning Design System のアバターからスタイル設定を継承します。

class 属性を使用して追加のスタイル設定を適用します。

次に例を示します。

```
<aura:component>
    <lightning:avatar src="/images/codey.jpg" alternativeText="Codey Bear"/>
</aura:component>
```

無効な画像パスの処理

src 属性は、画像への相対パスを解決しますが、アプリケーションがオフラインであったり画像が削除されたりしたために、画像パスが正しく解決できない場合があります。無効な画像パスを処理するには、initials 属性を使用して代替イニシャルを指定できます。この例では、画像パスが無効な場合にはイニシャルの「Sa」が表示されます。

```
<lightning:avatar src="/bad/image/url.jpg" initials="Sa"
      fallbackIconName="standard:account" alternativeText="Salesforce"/>
```

上の例では、イニシャルが指定されなかった場合には、代替アイコン「standard:account」が表示されます。

アクセシビリティ

alternativeText 属性を使用して、ユーザのイニシャルや名前など、アバターを説明します。この説明によって、img HTML タグの alt 属性の値が指定されます。

属性

属性名	属性型	説明	必須項目?
alternativeText	String	アバターの説明に使用する代替テキストで、画像のフロート表示テキストとして表示されます。	はい
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
fallbackIconName	String	画像の読み込みに失敗した場合に代替として使用するアイコンの Lightning Design System 名。代替イニシャルの背景色は、これに依存します。名前は、形式「standard:account」で記述します。「standard」はカテゴリ、「account」は表示するアイコンです。標準およびカスタムカテゴリからのアイコンのみが許可されます。	
initials	String	レコード名に 2 つの単語 (名と姓など) が含まれる場合、各単語の最初の大文字を使用します。レコードに 1 つの単語の名前のみが含まれる場合、1 つの大文字と 1 つの小文字を使用してその単語の最初の 2 文字を使用します。	
size	String	アバターのサイズ。有効な値は、x-small、small、medium、large です。この値のデフォルトは medium です。	

属性名	属性型	説明	必須項目?
src	String	画像の URL。	はい
variant	String	バリエーションによってアバターの形状が変化します。有効な値は、empty、circle、square です。この値のデフォルトは square です。	

lightning:badge

未読の通知数など、少量の情報を保持する表示ラベルを表します。

`lightning:badge` は、少量の情報を保持する表示ラベルです。バッジは、未読の通知を表示するため、またはテキストブロックに表示ラベルを表示するために使用されます。バッジにはハイパーリンクを挿入できないため、ナビゲーションでは動作しません。

このコンポーネントは、Lightning Design System のバッジからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:badge label="Label" />
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
label	String	バッジ内に表示するテキスト。	はい

lightning:breadcrumb

ユーザが表示しているページの階層パス内の項目。

`lightning:breadcrumb` コンポーネントは、ページのパスを親ページに対する相対パスとして表示します。ブレッドクラムは、`lightning:breadcrumbs` コンポーネントにネストされています。各ブレッドクラムは実行可能で、大なり記号で区切られています。ブレッドクラムが表示される順序は、マークアップでリストされている順序によって決まります。

このコンポーネントは、Lightning Design System のブレッドクラムからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:breadcrumbs>
        <lightning:breadcrumb label="Parent Account" href="path/to/place/1"/>
        <lightning:breadcrumb label="Case" href="path/to/place/2"/>
    </lightning:breadcrumbs>
</aura:component>
```

ブレッドクラムの動作はリンクと類似しています。href 属性によってリンクが指定されていない場合、値はデフォルトの javascript:void(0); に設定されます。カスタムナビゲーションを指定するには、onclick ハンドラを使用します。onclick の使用は、たとえば force:navigateToSObject などのイベントを使用して移動する場合に役立ちます。href 属性でリンクを指定する場合は、event.preventDefault() をコールすることで、リンクをスキップして、代わりにカスタムナビゲーションを使用することができます。

```
<aura:component>
    <lightning:breadcrumbs>
        <lightning:breadcrumb label="Parent Account" href="path/to/place/1" onclick="{!
c.navigateToCustomPage1 }"/>
        <lightning:breadcrumb label="Case" href="path/to/place/2" onclick="{!
c.navigateToCustomPage2 }"/>
    </lightning:breadcrumbs>
</aura:component>

/** Client-Side Controller **/
({
    navigateToCustomPage1: function (cmp, event) {
        event.preventDefault();
        //your custom navigation here
    },
    navigateToCustomPage2: function (cmp, event) {
        event.preventDefault();
        //your custom navigation here
    }
})
```

aura:iteration によるブレッドクラムの生成

aura:iteration を使用して項目のリストを反復処理し、ブレッドクラムを生成します。たとえば、表示ラベルと名前の値を使用してブレッドクラムの配列を作成できます。これらの値は init ハンドラ内で設定します。

```
<aura:component>
    <aura:attribute name="myBreadcrumbs" type="Object"/>
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>
    <lightning:breadcrumbs>
        <aura:iteration items="{! v.myBreadcrumbs }" var="crumbs">
            <lightning:breadcrumb label="{! crumbs.label }" onclick="{! c.navigateTo }"
name="{! crumbs.name }"/>
        </aura:iteration>
    </lightning:breadcrumbs>
</aura:component>

/* Client-Side Controller */
```

```
({
    init: function (cmp, event, helper) {
        var myBreadcrumbs = [
            {label: 'Account', name: 'objectName' },
            {label: 'Record Name', name: 'record' }
        ];
        cmp.set('v.myBreadcrumbs', myBreadcrumbs);
    },
    navigateTo: function (cmp, event, helper) {
        //get the name of the breadcrumb that's clicked
        var name = event.getSource().get('v.name');

        //your custom navigation here
    }
})
```

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
href	String	ブレッドクラムの移動先のページの URL。	
label	String	ブレッドクラムのテキスト表示ラベル。	はい
name	String	ブレッドクラムコンポーネントの名前。この値は省略可能であり、コールバック内でブレッドクラムを識別するために使用できます。	
onclick	Action	ブレッドクラムがクリックされたときにトリガされるアクション。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	

lightning:breadcrumbs

Web サイトまたはアプリケーション内で現在アクセスしているページの階層パス。

lightning:breadcrumbs コンポーネントは、親に戻るためのページのパスを表示する順序付きリストです。各ブレッドクラム項目は、lightning:breadcrumb コンポーネントによって表されます。ブレッドクラムは実行可能で、大なり記号で区切られています。

このコンポーネントは、Lightning Design System のブレッドクラムからスタイル設定を継承します。

詳細は、「lightning:breadcrumb」を参照してください。

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	

lightning:button

ボタン要素を表します。

lightning:button コンポーネントは、コントローラでアクションを実行するボタン要素を表します。ボタンをクリックすると、onclick に対して設定されたクライアント側コントローラのメソッドがトリガされます。ボタンは表示ラベルのみ、表示ラベルとアイコン、ボディのみ、ボディとアイコンのいずれかにすることができます。アイコンのみのボタンが必要な場合は、lightning:buttonIcon を使用します。

追加のスタイルを適用するには、variant および class 属性を使用します。

Lightning Design System ユーティリティアイコンカテゴリでは、lightning:button で表示ラベルテキストと一緒に使用できる約 200 個のユーティリティアイコンを提供しています。SLDS では複数のアイコンカテゴリが提供されていますが、このコンポーネントで使用できるのはユーティリティカテゴリのみです。

ユーティリティアイコンを表示するには、https://lightningdesignsystem.com/icons/#utilityにアクセスしてください。

このコンポーネントは、Lightning Design System のボタンからスタイル設定を継承します。

次に、2 つの例を示します。

```
<aura:component>
    <lightning:button variant="brand" label="Submit" onclick="{! c.handleClick }" />
</aura:component>
```

```
<aura:component>
    <lightning:button variant="brand" label="Download" iconName="utility:download"
iconPosition="left" onclick="{! c.handleClick }" />
</aura:component>
```

アクセシビリティ

スクリーンリーダーにボタンが無効化されていることを伝えるには、disabled 属性を true に設定します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
disabled	Boolean	このボタンを無効な状態で表示するかどうかを指定します。無効なボタンをクリックすることはできません。この値のデフォルトは false です。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。	
iconPosition	String	ボディに対するアイコンの位置を記述します。left または right を選択できます。この値のデフォルトは left です。	
label	String	ボタン内に表示するテキスト。	
name	String	ボタン要素の名前。この値は省略可能であり、コールバック内でボタンを識別するために使用できます。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onclick	Action	ボタンがクリックされたときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
title	String	マウスが要素に重ねられたときにツールチップテキストを表示します。	
type	String	ボタンの種類を指定します。有効な値は、button、reset、submit です。この値のデフォルトは button です。	
value	String	ボタン要素の値。この値は省略可能であり、フォームの送信時に使用できます。	
variant	String	バリエーションによってボタンの外観が変更されます。有効なバリエーションとして、base、neutral、brand、	

属性名	属性型	説明	必須項目
		destructive、inverseがあります。この値のデフォルトはneutralです。	

lightning:buttonGroup

ボタンのグループを表します。

lightning:buttonGroup コンポーネントは、ナビゲーションバーを作成するために一緒に表示できるボタンのセットを表します。コンポーネントのボディには lightning:button または lightning:buttonMenu を含めることができます。ナビゲーションタブが必要な場合は、lightning:buttonGroup ではなく lightning:tabset を使用します。

このコンポーネントは、Lightning Design System のボタングループからスタイル設定を継承します。

ボタンを作成するには、次の例のように lightning:button コンポーネントを使用します。

```
<aura:component>
    <lightning:buttonGroup>
        <lightning:button label="Refresh" onclick="{!c.handleClick}"/>
        <lightning:button label="Edit" onclick="{!c.handleClick}"/>
        <lightning:button label="Save" onclick="{!c.handleClick}"/>
    </lightning:buttonGroup>
</aura:component>
```

lightning:button の onclick ハンドラは handleClick クライアント側コントローラをコールし、クライアント側コントローラはクリックされたボタンの表示ラベルを返します。

```
({
    handleClick: function (cmp, event, helper) {
        var selectedButtonLabel = event.getSource().get("v.label");
        alert("Button label: " + selectedButtonLabel);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	

lightning:buttonIcon

アイコンのみの HTML ボタン。

lightning:buttonIcon コンポーネントは、コントローラでアクションを実行するアイコンのみのボタン要素を表します。ボタンをクリックすると、onclick に対して設定されたクライアント側コントローラのメソッドがトリガされます。

variant、size、class、iconClass の各属性を組み合わせて、ボタンやアイコンのスタイルをカスタマイズできます。ボタンコンテナのスタイル設定をカスタマイズするには、class 属性を使用します。ベアバリエーションでは、size クラスがアイコン自体に適用されます。ノンベアバリエーションでは、size クラスがボタンに適用されます。アイコン要素のスタイル設定をカスタマイズするには、iconClass 属性を使用します。次の例では、ベアバリエーションおよびカスタムアイコンというスタイル設定のアイコンのみのボタンを作成します。

```
<!-- Bare variant with custom "dark" CSS class added to icon svg element -->
<lightning:buttonIcon iconName="utility:settings" variant="bare" alternativeText="Settings"
 iconClass="dark"/>
```

Lightning Design System ユーティリティアイコンカテゴリでは、lightning:buttonIcon で使用できる約 200 個のユーティリティアイコンを提供しています。Lightning Design System では複数のアイコンカテゴリが提供されていますが、lightning:buttonIcon で使用できるのはユーティリティカテゴリのみです。

ユーティリティアイコンを表示するには、https://lightningdesignsystem.com/icons/#utilityにアクセスしてください。

このコンポーネントは、Lightning Design System のボタンアイコンからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:buttonIcon iconName="utility:close" variant="bare" onclick="{! c.handleClick
 }" alternativeText="Close window." />
</aura:component>
```

使用上の考慮事項

スタンドアロンアプリケーションで lightning:buttonIcon を使用する場合、アイコンリソースを正しく解決するために force:slds を拡張します。

```
<aura:application extends="force:slds">
     <lightning:buttonIcon iconName="utility:close" alternativeText="Close"/>
</aura:application>
```

アクセシビリティ

アイコンを説明するには、alternativeText 属性を使用します。説明では、アイコンの外観 (「ペーパークリップ」) ではなく、ボタンをクリックしたときに何が起こるか (「ファイルのアップロード」など) を示す必要があります。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目?
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
alternativeText	String	アイコンを説明するために使用する代替テキスト。このテキストでは、アイコンの外観 (「ペーパークリップ」) ではなく、ボタンをクリックしたときに何が起こるか (「ファイルのアップロード」など) を説明する必要があります。	はい
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
disabled	Boolean	このボタンを無効な状態で表示するかどうかを指定します。無効なボタンをクリックすることはできません。この値のデフォルトは false です。	
iconClass	String	含まれるアイコン要素に適用されるクラス。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。注意: このコンポーネントで使用できるのはユーティリティアイコンのみです。	はい
name	String	ボタン要素の名前。この値は省略可能であり、コールバック内でボタンを識別するために使用できます。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onclick	Action	ボタンがクリックされたときに実行されるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
size	String	buttonIconのサイズ。ベアバリエーションのオプションは、x-small、small、medium、large です。ノンベアバリエーションのオプションは、xx-small、x-small、small、medium です。この値のデフォルトは medium です。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	

属性名	属性型	説明	必須項目?
title	String	マウスが要素に重ねられたときにツールチップテキストを表示します。	
type	String	ボタンの種類を指定します。有効な値は、button、reset、submit です。この値のデフォルトは button です。	
value	String	ボタン要素の値。この値は省略可能であり、フォームの送信時に使用できます。	
variant	String	バリエーションによって buttonIcon の外観が変更されます。有効なバリエーションとして、bare、container、border、border-filled、bare-inverse、border-inverse があります。この値のデフォルトは border です。	

lightning:buttonIconStateful

状態を保持するアイコンのみのボタンです。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:buttonIconStateful コンポーネントは、2 つの状態を切り替えるアイコンのみのボタン要素を表します。たとえば、このコンポーネントを使用して、ブログ投稿に関する顧客のフィードバック (賛成または反対) を取得できます。ボタンをクリックすると、onclick に設定されたクライアント側コントローラメソッドがトリガされ、selected 属性を使用してアイコンの状態が変更されます。

Lightning Design System ユーティリティアイコンカテゴリでは、lightning:buttonIconStateful で使用できる約 200 個のユーティリティアイコンを提供しています。Lightning Design System では複数のアイコンカテゴリが提供されていますが、このコンポーネントで使用できるのはユーティリティカテゴリのみです。

ユーティリティアイコンを表示するには、https://lightningdesignsystem.com/icons/#utilityにアクセスしてください。

このコンポーネントは、Lightning Design System のボタンアイコンからスタイル設定を継承します。

variant、size、class の各属性を組み合わせて、ボタンやアイコンのスタイルをカスタマイズできます。ボタンコンテナのスタイル設定をカスタマイズするには、class 属性を使用します。

次の例では、2 つの状態を切り替える [like (賛成)] ボタンを作成します。デフォルトでは、[like (賛成)] ボタンが選択されます。ボタンの状態は selected 属性に保存されます。

```
<aura:component>
    <aura:attribute name="liked" type="Boolean" default="true"/>
    <lightning:buttonIconStateful iconName="utility:like" selected="{!v.liked}"
alternativeText="Like" onclick="{! c.handleToggle }"/>
<aura:component>
```

[dislike (反対)] ボタンを選択すると、[like (賛成)] ボタンの状態も切り替わり、選択解除されます。

```
({
    handleToggle : function (cmp, event) {
        var liked = cmp.get("v.liked");
        cmp.set("v.liked", !liked);
```

```
    }
})
```

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
onfocus	Aura.Action	要素にフォーカスが設定されたときにトリガされるアクション。	
onblur	Aura.Action	要素がフォーカスを解放したときにトリガされるアクション。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスが要素に重ねられたときにツールチップテキストを表示します。	
name	String	ボタン要素の名前。この値は省略可能であり、コールバック内でボタンを識別するために使用できます。	
value	String	ボタン要素の値。この値は省略可能であり、フォームの送信時に使用できます。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。注意: このコンポーネントで使用できるのはユーティリティアイコンのみです。	はい
variant	String	バリエーションによってbuttonIconの外観が変更されます。使用できるバリエーションは border と border-inverse です。この値のデフォルトは border です。	

属性名	属性型	説明	必須項目
size	String	buttonIcon のサイズ。オプションは、xx-small、x-small、small、medium です。この値のデフォルトは medium です。	
disabled	Boolean	このボタンを無効な状態で表示するかどうかを指定します。無効なボタンをクリックすることはできません。この値のデフォルトは false です。	
alternativeText	String	アイコンを説明するために使用する代替テキスト。このテキストでは、アイコンの外観(「ペーパークリップ」)ではなく、ボタンをクリックしたときに何が起こるか(「ファイルのアップロード」など)を説明する必要があります。	はい
onclick	Aura.Action	ボタンがクリックされたときに実行されるアクション。	
selected	Boolean	ボタンが選択済みの状態かどうかを指定します。	

lightning:buttonMenu (ベータ)

アクションまたは機能のリストのドロップダウンメニューを表します。

lightning:buttonMenu は、クリックされるとユーザがアクセス可能なアクションまたは機能のドロップダウンメニューを表示するボタンを表します。

スタイルをカスタマイズするには、variant、size、または class 属性を使用します。

このコンポーネントは、Lightning Design System のメニューからスタイル設定を継承します。

3 つの項目が含まれるドロップダウンメニューの例を次に示します。

```
<lightning:buttonMenu iconName="utility:settings" alternativeText="Settings" onselect="{!
 c.handleMenuSelect }">
    <lightning:menuItem label="Font" value="font" />
    <lightning:menuItem label="Size" value="size"/>
    <lightning:menuItem label="Format" value="format" />
</lightning:buttonMenu>
```

onselect がトリガされると、そのイベントは value パラメータを持ちます。これは、選択されたメニュー項目の値です。値を参照する方法の例を次に示します。

```
handleMenuSelect: function(cmp, event, helper) {
    var selectedMenuItemValue = event.getParam("value");
}
```

lightning:menuItem コンポーネントで checked 属性を使用して、チェックまたはチェック解除可能なメニュー項目を作成し、必要に応じて切り替えることができます。メニュー項目の切り替えを有効にするには、checked 属性の初期値を true または false に設定する必要があります。

他の場所をクリックするとメニューが閉じます。メニュー項目が選択された場合もメニューが閉じて、ボタンにフォーカスが戻されます。

aura:iteration によるメニュー項目の生成

次の例では、初期化中に複数の項目が含まれるボタンメニューを作成します。

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.createItems}" />
    <lightning:buttonMenu alternativeText="Action" onselect="{! c.handleMenuSelect }">

        <aura:iteration var="action" items="{! v.actions }">
            <lightning:menuItem aura:id="actionMenuItems" label="{! action.label }"
value="{! action.value }"/>
        </aura:iteration>
    </lightning:buttonMenu>
</aura:component>
```

クライアント側のコントローラは、メニュー項目の配列を作成し、`actions` 属性でその値を設定します。

```
({
    createItems: function (cmp, event) {
        var items = [
            { label: "New", value: "new" },
            { label: "Edit", value: "edit" },
            { label: "Delete", value: "delete" }
        ];
        cmp.set("v.actions", items);
    }
})
```

使用上の考慮事項

このコンポーネントには、ボタンがトリガされた場合に限って作成されるメニュー項目が含まれます。初期化中、またはボタンがトリガされていない場合は、メニュー項目を参照できません。

アクセシビリティ

スクリーンリーダーにボタンが無効化されていることを伝えるには、`disabled` 属性を true に設定します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
alternativeText	String	ボタンの補助テキスト。	
body	ComponentDefRef[]	コンポーネントのボディ。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	

属性名	属性型	説明	必須項目
disabled	Boolean	true の場合、メニューは無効化されています。メニューを無効化すると、ユーザはメニューを開けなくなります。この値のデフォルトは false です。	
iconName	String	\'utility:down\' 形式で使用するアイコンの名前。この値のデフォルトは utility:down です。utility:down または utility:chevrondown以外のアイコンが使用されている場合は、そのアイコンの右側に utility:down アイコンが追加されます。	
iconSize	String	アイコンのサイズ。オプションは、xx-small、x-small、medium、large のいずれかです。この値のデフォルトは medium です。	
menuAlignment	String	ボタンを基準とするメニューの位置を決定します。選択可能なオプションは、left、center、right です。この値のデフォルトは left です。	
name	String	ボタン要素の名前。この値は省略可能であり、コールバック内でボタンを識別するために使用できます。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
onselect	Action	メニュー項目が選択されたときに起動されるアクション。渡されたイベントの「detail.menuItem」プロパティは、選択されたメニュー項目です。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
title	String	ボタンのツールチップテキスト。	
value	String	ボタン要素の値。この値は省略可能であり、フォームの送信時に使用できます。	
variant	String	バリエーションによってボタンの外観が変更されます。有効なバリエーションとして、bare、container、border、border-filled、bare-inverse、border-inverse があります。この値のデフォルトは border です。	
visible	Boolean	trueの場合、メニュー項目が表示されます。この値のデフォルトは false です。	

lightning:buttonStateful

状態を切り替えるボタン。

lightning:buttonStateful コンポーネントは、ソーシャルメディアのいいね! ボタンのように、状態を切り替えるボタンを表します。ステートフルボタンには、状態に基づいて異なる表示ラベルやアイコンを表示できます。

追加のスタイルを適用するには、variant および class 属性を使用します。

Lightning Design System ユーティリティアイコンカテゴリでは、lightning:button でテキスト表示ラベルと一緒に使用できる約 200 個のユーティリティアイコンを提供しています。Lightning Design System では複数のアイコンカテゴリが提供されていますが、このコンポーネントで使用できるのはユーティリティカテゴリのみです。

ユーティリティアイコンを表示するには、https://lightningdesignsystem.com/icons/#utilityにアクセスしてください。

このコンポーネントは、Lightning Design System のステートフルボタンからスタイル設定を継承します。

ボタンがクリックされたときの状態の変更を処理するには、onclick イベントハンドラを使用します。次の例では、ボタンの状態を切り替えることができます。デフォルトでは「Follow」(フォローする) 表示ラベルが表示され、ボタンが選択されると「Following」(フォロー中) に変わります。ボタンを選択すると状態が true に切り替わり、選択解除すると状態が false に切り替わります。状態が true のときにボタンにマウスポインタまたはフォーカスが置かれると、「Unfollow」(フォローを解除) がボタンに表示されます。

```
<aura:component>
    <aura:attribute name="buttonstate" type="Boolean" default="false"/>
    <lightning:buttonStateful
        labelWhenOff="Follow"
        labelWhenOn="Following"
        labelWhenHover="Unfollow"
        iconNameWhenOff="utility:add"
        iconNameWhenOn="utility:check"
        iconNameWhenHover="utility:close"
        state="{! v.buttonstate }"
        onclick="{! c.handleClick }"
    />
</aura:component>
```

クライアント側のコントローラは、buttonstate 属性によって状態を切り替えます。

```
({
    handleClick : function (cmp, event, helper) {
        var buttonstate = cmp.get('v.buttonstate');
        cmp.set('v.buttonstate', !buttonstate);
    }
})
```

アクセシビリティ

アクセシビリティに対応するには、ボタンに aria-live="assertive" 属性を含めます。aria-live="assertive" 属性を設定すると、ボタン内の <span> の値が変更されるたびに、その値が読み上げられます。

スクリーンリーダーにボタンが無効化されていることを伝えるには、disabled 属性を true に設定します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

認証

属性名	属性型	説明	必須
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
iconNameWhenHover	String	状態が true でボタンにフォーカスが置かれたときに \'utility:close\' 形式で使用するアイコンの名前。	
iconNameWhenOff	String	状態が false のときに \'utility:add\' 形式で使用するアイコンの名前。	
iconNameWhenOn	String	状態が true のときに \'utility:check\' 形式で使用するアイコンの名前。	
labelWhenHover	String	状態が true でボタンにフォーカスが置かれたときにボタン内に表示するテキスト。	
labelWhenOff	String	状態が false のときにボタン内に表示するテキスト。	はい
labelWhenOn	String	状態が true のときにボタン内に表示するテキスト。	はい
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onclick	Action	ボタンがクリックされたときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
state	Boolean	ボタンが選択されたかどうかを示す状態。デフォルトの状態は false です。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
variant	String	バリエーションによってボタンの外観が変更されます。使用できるバリエーションは、brand、inverse、neutral、text です。この値のデフォルトは neutral です。	

lightning:card

カードは、関連する情報のグルーピングにコンテナを適用するために使用されます。

lightning:card は、スタイル設定されたコンテナを情報のグルーピングに適用するために使用されます。情報は、1 つの項目の場合もあれば、関連リストなど項目のグループの場合もあります。

スタイル設定をカスタマイズするには variant 属性または class 属性を使用します。

lightning:card にはタイトル、ボディ、フッターが含まれています。カードのボディのスタイルを設定するには、Lightning Design System ヘルパークラスを使用します。

このコンポーネントは、Lightning Design System のカードからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:card>
        <aura:set attribute="title">
            Hello!
        </aura:set>
        <aura:set attribute="footer">
            <lightning:badge label="footer"/>
        </aura:set>
        <aura:set attribute="actions">
            <lightning:button label="New"/>
        </aura:set>
        <p class="slds-p-horizontal--small">
            Card Body (custom component)
        </p>
    </lightning:card>
</aura:component>
```

使用上の考慮事項

title 属性と footer 属性は Object 型です。つまり、String 型や Component[] 型などの値を渡すことができます。上の例では、title 属性と footer 属性を Component[] 型 (facet とも呼ばれる) として渡しています。Component[] 型は、この例に示すように、タイトルやフッターにマークアップを渡す必要がある場合に役立ちます。

```
<aura:component>
    <aura:attribute name="name" type="String" default="Your Name"/>
    <aura:attribute name="myTitleName" type="Aura.Component[]">
        <h1>Hello {! v.name }</h1>
    </aura:attribute>
    <lightning:card footer="Card Footer">
        <aura:set attribute="title">
            {!v.myTitleName}
        </aura:set>
        <!-- actions and body markup here -->
    </lightning:card>
</aura:component>
```

String 型の値を渡すには、それを <lightning:card> タグ内に含めることができます。

```
<aura:component>
    <aura:attribute name="myTitle" type="String" default="My Card Title"/>
```

```
<lightning:card title="{!v.myTitle}" footer="Card Footer">
    <aura:set attribute="actions">
        <lightning:button label="New"/>
    </aura:set>
    <p class="slds-p-horizontal--small">
        Card Body (custom component)
    </p>
</lightning:card>
```

属性

属性名	属性型	説明	必須項目
actions	Component[]	アクションは、button や buttonIcon などのコンポーネントです。アクションはヘッダーに表示されます。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
footer	Object	フッターにはテキストまたは別のコンポーネントを含めることができます。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。アイコンはヘッダー内のタイトルの左側に表示されます。	
title	Object	タイトルにはテキストまたは別のコンポーネントを含めることができ、ヘッダーに表示されます。	はい
variant	String	バリエーションは、カードの外観を変更します。使用できるバリエーションは、base または narrow です。この値のデフォルトは base です。	

lightning:checkboxGroup

単一または複数のオプションの選択を有効にするチェックボックスグループです。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:checkboxGroup コンポーネントは、単一または複数のオプションの選択を有効にするチェックボックスグループを表します。

required 属性が true に設定されている場合、1 つ以上のチェックボックスを選択する必要があります。ユーザがチェックボックスグループを操作して選択しないと、エラーメッセージが表示されます。messageWhenValueMissing 属性を使用して、カスタムエラーメッセージを提供できます。

disabled 属性が true に設定されている場合、チェックボックスの選択は変更できません。

このコンポーネントは、Lightning Design System のチェックボックスからスタイル設定を継承します。

次の例では、2つのオプションがあり、option1 がデフォルトで選択されているチェックボックスグループを作成します。required 属性が true になっているので 1 つ以上のチェックボックスを選択する必要があります。

```
<aura:component>
    <aura:attribute name="options" type="List" default="[
    {'label': 'Ross', 'value': 'option1'},
    {'label': 'Rachel', 'value': 'option2'},
    ]"/>
    <aura:attribute name="value" type="List" default="option1"/>
    <lightning:checkboxGroup
        aura:id="mygroup"
        name="checkboxGroup"
        label="Checkbox Group"
        options="{! v.options }"
        value="{! v.value }"
        onchange="{! c.handleChange }"
        required="true" />
</aura:component>
```

cmp.find("mygroup").get("v.value") を使用して、どの値が選択されているのかを確認できます。チェックボックスがオンまたはオフになったときに値を取得するには、onchange イベントハンドラを使用して event.getParam("value") をコールします。

```
({
    handleChange: function (cmp, event) {
        var changeValue = event.getParam("value");
        alert(changeValue);
    }
});
```

使用上の考慮事項

lightning:checkboxGroup は、一連のチェックボックスをグループ化するのに便利です。チェックボックスが 1 つの場合、代わりに lightning:input type="checkbox" を使用します。

アクセシビリティ

チェックボックスグループは、legend 要素が含まれる fieldset 要素内でネストされます。legend には label 値が含まれます。fieldset 要素では、関連するチェックボックスをグループ化して、タブおよび音声のナビゲーションを促進し、アクセシビリティに対応できます。同様に、legend 要素では、キャプションを fieldset に割り当てられるようにしてアクセシビリティを改善します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

checkValidity():チェックボックスグループに有効性エラーがあるかどうかを示す ValidityState オブジェクトの有効なプロパティ値 (Boolean) を返します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
name	String	チェックボックスグループの名前。	はい
label	String	チェックボックスグループのテキスト表示ラベル。	はい
options	List	各チェックボックスの表示ラベル - 値のペアの配列。	はい
value	String []	選択されたチェックボックスのリスト。各配列のエントリには、選択されたチェックボックスの値が含まれます。各チェックボックスの値は、options 属性で設定されます。	はい
messageWhenValueMissing	String	チェックボックスが選択されておらず、required 属性が true に設定されている場合に表示される省略可能なメッセージ。	
required	boolean	1 つ以上のチェックボックスを選択する必要がある場合に true に設定します。この値のデフォルトは false です。	
disabled	boolean	チェックボックスグループが無効になっている場合、true に設定されます。無効になっているチェックボックスグループのチェックボックスの選択は変更できません。この値のデフォルトは false です。	
onblur	Aura.Action	チェックボックスグループがフォーカスを解放したときにトリガされるアクション。	
onchange	Aura.Action	チェックボックスの値が変更されたときにトリガされるアクション。	
onfocus	Aura.Action	チェックボックスグループにフォーカスが設定されたときにトリガされるアクション。	

lightning:clickToDial

Open CTI および Voice 用に有効化または無効化されたクリック-to-ダイヤルとして書式設定された電話番号を表示します。このコンポーネントでは、Salesforce とのコンピュータテレフォニーインテグレーション (CTI) の既存のクリック-to-ダイヤルコマンドが優先されます。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:clickToDial` コンポーネントは、Open CTI および Voice 用に有効化または無効化されたクリック-to-ダイヤルとして書式設定された電話番号を表示します。このコンポーネントでは、Salesforce とのコンピュータテレフォニーインテグレーション (CTI) の既存のクリック-to-ダイヤルコマンドが優先されます。

コンポーネントで電話番号にダイヤルするには、まず電話システムを有効にする必要があります。電話システムが有効になった後でユーザが電話番号をクリックすると、コンポーネントは電話番号がクリックされたことを電話システムに通知します。次に、コンポーネントは電話システムでアウトバウンドコールを行うために必要な情報を渡します。

次に、`lightning:clickToDial` コンポーネントを使用する方法の例を示します。最初の電話番号には、recordId やパラメータがありません。2 番目の電話番号には recordId があります。3 番目の電話番号には recordId とパラメータがあります。

```
    <aura:component>
        <lightning:clickToDial value="14155555551"/>
        <lightning:clickToDial value="14155555552" recordId="5003000000D9duF"/>
        <lightning:clickToDial value="14155555553" recordId="5003000000D8cuI"
params="accountSid=xxx, sourceId=xxx, apiVersion=123"/>
    </aura:component>
```

Open CTI の使用に関する考慮事項

`lightning:clickToDial` コンポーネントは、Open CTI for Lightning Experience API メソッド `enableClickToDial`、`disableClickToDial`、`onClickToDial` と連動して動作します。詳細は、『Open CTI 開発者ガイド』を参照してください。

`lightning:clickToDial` を使用して電話番号にダイヤルするには、まず Open CTI メソッド `enableClickToDial` で電話システムを有効にします。電話システムを無効にするには、Open CTI メソッド `disableClickToDial` を使用します。

電話番号をクリックすると、Open CTI メソッド `onClickToDial` に登録された `onClickToDial` リスナーが呼び出されます。

`lightning:clickToDial` には、recordId 属性を含めることができます。この属性を渡す場合、Open CTI メソッド `onClickToDial` に渡されるペイロードにこのレコード ID に関連付けられたレコード情報が含まれます。たとえば、レコード名やオブジェクト種別です。recordId が渡されない場合、レコード情報は `onClickToDial` ハンドラに提供されません。

書式設定された電話番号は、北米形式 (123 456 7890) に従います。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	String	電話番号。	はい
recordId	String	電話番号に関連付けられた Salesforce レコード ID。	
params	String	サードパーティの電話システムに渡すパラメータのカンマ区切りリスト。	

lightning:combobox (ベータ)

選択可能なオプションのドロップダウンリストと共に参照のみの入力項目を提供するウィジェット。

lightning:combobox は、オプションのリストの単一選択を有効にする入力要素です。選択結果は入力値として表示されます。

このコンポーネントは、Lightning Design System のコンボボックスからスタイル設定を継承します。

次の例では、初期化中のオプションのリスト (デフォルト選択あり) を作成します。

```
<aura:component>
    <aura:attribute name="statusOptions" type="List" default="[]"/>
    <aura:handler name="init" value="{! this }" action="{! c.loadOptions }"/>
    <lightning:combobox aura:id="selectItem" name="status" label="Status"
                      placeholder="Choose Status"
                      value="new"
                      onchange="{!c.handleOptionSelected}"
                      options="{!v.statusOptions}"/>
</aura:component>
```

クライアント側コントローラで、オプションの配列を定義し、この配列を statusOptions 属性に割り当てます。各オプションは、ドロップダウンリストのリスト項目に対応します。

```
({
    loadOptions: function (component, event, helper) {
        var options = [
            { value: "new", label: "New" },
            { value: "in-progress", label: "In Progress" },
            { value: "finished", label: "Finished" }
        ];
        component.set("v.statusOptions", options);
```

```
    },
    handleChange: function (cmp, event) {
        // Get the string of the "value" attribute on the selected option
        var selectedOptionValue = event.getParam("value");
        alert("Option selected with value: '" + selectedOptionValue + "'");
    }
})
```

オプションを選択すると、onchange イベントがトリガされ、handleChange クライアント側コントローラがコールされます。クリックされたオプションをチェックするには、event.getParam("value") を使用します。cmp.find("mycombobox").get("v.value"); をコールすると、現在選択されているオプションが返されます。

入力規則

このコンポーネントでは、クライアント側で入力規則が使用できます。選択を必須にするには、required="true" を設定します。required="true" で項目が選択されていない場合、エラーメッセージが自動的に表示されます。

入力の有効性状態を確認するには、ValidityState オブジェクトに基づく validity 属性を使用します。有効性状態にはクライアント側コントローラでアクセスできます。この validity 属性は、boolean プロパティが設定されたオブジェクトを返します。詳細は、lightning:inputを参照してください。

messageWhenValueMissing に独自の値を指定し、デフォルトのメッセージを上書きできます。

アクセシビリティ

アクセシビリティのためのテキスト表示ラベルを指定して、情報が支援技術で使用できるようにする必要があります。label 属性は、入力コンポーネントの HTML label 要素を作成します。表示ラベルをビューに表示させず、支援技術には使用できるようにするには、label-hidden バリエーションを使用します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

checkValidity(): コンボボックスに有効性エラーがあるかどうかを示す ValidityState オブジェクトの有効なプロパティ値 (Boolean) を返します。

setCustomValidity(message): コンボボックスの値が送信されたときに表示されるカスタムエラーメッセージを設定します。

- message (String): エラーを説明する文字列。メッセージが空の文字列の場合、エラーメッセージはリセットされます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
name	String	入力要素の名前を指定します。	はい

属性名	属性型	説明	必須項目
value	Object	入力要素の値を指定します。	
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトはfalseです。	
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
onchange	Aura.Action	値属性が変更された場合にトリガされるアクション。	
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
onfocus	Aura.Action	要素にフォーカスが設定されたときにトリガされるアクション。	
onblur	Aura.Action	要素がフォーカスを解放したときにトリガされるアクション。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
options	Object[]	選択できるオプションのリスト。各オプションには、class、selected、label、value 属性があります。	はい
label	String	コンボボックスのテキスト表示ラベル。	はい
placeholder	String	オプションを選択する前に表示されるテキストで、ユーザにオプションを選択するように求めます。デフォルトは、[オプションを選択] です。	
dropdownAlignment	String	入力を基準とするドロップダウンの位置を決定します。使用可能な値は、left、center、right、bottom-left、bottom-center、bottom-right です。デフォルトは、left です。	

属性名	属性型	説明	必須項目
messageWhenValueMissing	String	必須の入力の値がない場合に表示されるエラーメッセージ。	

lightning:container

AngularJS や React など、サードパーティの JavaScript フレームワークを使用するコンテンツを含めるために使用します。

lightning:container コンポーネントでは、Lightning コンポーネント内でサードパーティのフレームワークを使用して開発されたコンテンツをホストできます。このコンテンツは静的リソースとしてアップロードされ、iFrame でホストされます。lightning:container コンポーネントは、単一ページアプリケーションにのみ使用できます。

以下は、lightning:container の簡単な例です。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes">
    <lightning:container src="{!$Resource.myReactApp + '/index.html'}"/>
</aura:component>
```

また、フレーム化されたアプリケーションとの通信を実装でき、Salesforce とのやりとりが可能になります。JavaScript コントローラで message() 関数を使用して、アプリケーションにメッセージを送信し、コンポーネントの onmessage 属性でメッセージを処理するメソッドを指定します。

次の JavaScript コントローラの例は、message() 関数を使用して、簡単な JSON ペイロードをサードパーティのコンテンツ (この場合は AngularJS アプリケーション) に送信します。

```
({
    sendMessage : function(component, event, helper) {
        var msg = {
            name: "General",
            value: component.get("v.messageToSend")
        };
        component.find("AngularApp").message(msg);
    },
    handleMessage: function(component, message, helper) {
        var payload = message.payload;
        var name = payload.name;
        if (name === "General") {
            var value = payload.value;
            component.set("v.messageReceived", value);
        }
        else if (name === "Foo") {
            // A different response
        }
    },
})
```

付随するコンポーネント定義は、コンテナからLightningコンポーネントに送信するメッセージの属性および受信したメッセージの属性を定義します。lightning:container の onmessage 属性は、JavasSript メソッド handleMessage を参照します。

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >
    <aura:attribute name="messageToSend" type="String" default=""/>
    <aura:attribute name="messageReceived" type="String" default=""/>
    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to Angular app: "/>
        <lightning:button label="Send" onclick="{!c.sendMessage}"/>
        <lightning:textarea name="messageReceived" value="{!v.messageReceived}"
label="Message received from Angular app: "/>
        <lightning:container aura:id="AngularApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"/>
    </div>
</aura:component>
```

コントローラ側のメッセージ処理は自身で定義するため、この処理を使用してあらゆる種類のメッセージペイロードを処理できます。たとえば、単にテキスト文字列を送信することや、構造化された JSON 応答を返すことができます。

使用上の考慮事項

コンテナの src を指定するときに、ホスト名を指定しないでください。代わりに、$Resource をドット表記で使用して、静的リソースとしてアップロードされたアプリケーションを参照します。

アクセシビリティ

alternativeText 属性を使用して、Lightning コンテナの補助テキストを指定します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

message(): ユーザ定義のメッセージをコンポーネントから iFrame コンテンツに送信します。

属性

属性名	属性型	説明	必須項目?
alternativeText	String	アクセシビリティのシナリオで代替テキストに使用します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	iframe 要素の CSS クラス。	
onerror	Action	含まれているアプリケーションへのメッセージ送信時にエラーが発生した場合に実行するクライアント側コントローラのアクション。	

属性名	属性型	説明	必須項目?
onmessage	Action	含まれているアプリケーションからメッセージを受信した場合に実行するクライアント側コントローラのアクション。	
src	String	URL 形式のリソース名、ランディングページ、およびクエリパラメータ。ナビゲーションは、特定された単一のページでのみサポートされます。	はい

lightning:datatable

型に応じて書式設定されたデータの列を表示するテーブルです。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:datatable コンポーネントには、表形式データが表示され、そこにデータ型に応じて各列が表示されます。たとえば、email 型を指定すると、mailto: URL スキームが付いているハイパーリンクとしてメールアドレスを表示します。デフォルトの型は text です。

このコンポーネントは、Lightning Design System のデータテーブルからスタイル設定を継承します。

インライン編集は現在サポートされていません。次の機能がサポートされています。

- 適用するデータ型による列の表示と書式設定
- 列のサイズ変更
- 行の選択
- 昇順または降順による列の並び替え

テーブルは、data、columns、keyField 属性を使用して、初期化中に生成されます。この例では、6 つの列を含むテーブルが作成され、1 列目に行選択のチェックボックスが表示されます。テーブルデータは、init ハンドラを使用して読み込まれます。チェックボックスをオンにすると、データの行全体が選択されて、onrowselection イベントハンドラをトリガできます。

```
<aura:component>
    <aura:attribute name="mydata" type="Object"/>
    <aura:attribute name="mycolumns" type="List"/>
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>
   <lightning:datatable data="{! v.mydata }" columns="{! v.mycolumns }" onrowselection="{!
 c.getSelectedName }"/>
</aura:component>
```

選択可能な行と columns オブジェクトを、対応する列データに作成するクライアント側コントローラを次に示します。

```
({
    init: function (cmp, event, helper) {
    cmp.set('v.mycolumns', [
                {label: 'Opportunity name', fieldName: 'opportunityName', type: 'text'},
                {label: 'Confidence', fieldName: 'confidence', type: 'percent'},
                {label: 'Amount', fieldName: 'amount', type: 'currency', cellAttributes:
```

```
{ currencyCode: 'EUR'}},
                {label: 'Contact Email', fieldName: 'contact', type: 'email'},
                {label: 'Contact Phone', fieldName: 'phone', type: 'phone'}
            ]);
        cmp.set('v.mydata', [{
                id: 'a',
                opportunityName: 'Cloudhub',
                confidence: 0.2,
                amount: 25000,
                contact: 'jrogers@cloudhub.com',
                phone: '2352235235'
            },
            {
                id: 'b',
                opportunityName: 'Quip',
                confidence: 0.78,
                amount: 740000,
                contact: 'quipy@quip.com',
                phone: '2352235235'
            }]);
    },
    getSelectedName: function (cmp, event) {
        var selectedRows = event.getParam('selectedRows');
        // Display that fieldName of the selected rows
        for (var i = 0; i < selectedRows.length; i++){
            alert("You selected: " + selectedRows[i].opportunityName);
        }
    }
})
```

前述の例では、データの 1 行目では、1 列目にチェックボックスが表示されて、Cloudhub、20%、$25,000.00、jrogers@cloudhub.com、(235) 223-5235 というデータを含む列が表示されます。最後の2列は、メールアドレスと電話番号のハイパーリンクとして表示されます。

列データの操作

列データを提供するほか、次の列プロパティを定義する必要があります。

- `label`: 必須。列ヘッダーに表示されるテキスト表示ラベル。
- `fieldName`:必須。列プロパティを関連データに結びつけるための名前。`columns` プロパティはそれぞれ、データ配列の項目に対応している必要があります。
- `type`: 必須。データ書式設定に使用するデータ型。
- `initialWidth`: 初期化されたときの列幅。`minColumnWidth` および `maxColumnWidth` 値の範囲内、またはそれらが指定されていない場合は、50px および 1000px の範囲内である必要があります。
- `cellAttributes`:データ型のコンポーネント属性にカスタム書式設定を提供します。たとえば、`currency` 型の場合、`currencyCode` です。
- `sortable`: 列による並べ替えを有効にするかどうかを指定します。デフォルトは false です。

データ型による書式設定

データテーブルは、選択された型に基づいて形式を判断します。各データ型は、Lightning基本コンポーネントに関連付けられています。たとえば、`text` 型を指定すると、関連データは `lightning:formattedText` コンポーネントを使用して表示されます。有効なデータ型は次のとおりです。

- text: `lightning:formattedText` を使用してデータを表示
- number: `lightning:formattedNumber` を使用してデータを表示
- currency: `lightning:formattedNumber` を使用してデータを表示
- percent: `lightning:formattedNumber` を使用してデータを表示
- date: `lightning:formattedDateTime` を使用してデータを表示
- email: `lightning:formattedEmail` を使用してデータを表示 (`label` 属性はサポートされていません)
- phone: `lightning:formattedPhone` を使用してデータを表示
- url: `lightning:formattedUrl` を使用してデータを表示 (`label` 属性はサポートされていません)
- location: `lightning:formattedLocation` を使用してデータを表示(データの形式は `{latitude : number, longitude : number}` である必要があります)

データ型に基づいた書式設定をカスタマイズするには、対応するLightning基本コンポーネントの属性を渡します。たとえば、カスタム `currencyCode` 値を渡すと、デフォルトの通貨コードが上書きされます。

```
columns: [
    {label: 'Amount', fieldName: 'amount', type: 'currency', cellAttributes: { currencyCode:
 'EUR' }}
]
```

通貨または日時データ型を使用する場合、ロケールの書式設定が指定されていなければ、デフォルトのユーザロケールが使用されます。

これらの属性の詳細は、対応するコンポーネントのドキュメントを参照してください。

テーブルと列のサイズ変更

データテーブルの幅と高さは、コンテナ要素によって決まります。表示対象の行がさらにある場合は、テーブルボディにスクローラが追加されます。たとえば、コンテナ要素にCSSスタイル設定を適用して、高さを300pxに制限できます。

```
<div style="height: 300px;">
    <!-- lightning:datatable goes here -->
</div>
```

デフォルトでは、列はサイズ変更できます。ユーザは、デフォルト値が変更されていない限り、最小50pxおよび最大1000pxの幅にクリックおよびドラッグできます。列はデフォルトでサイズ変更できます。列のサイズ変更を無効にする場合は、`resizeColumnDisabled` を `true` に設定します。最小および最大列幅を変更するには、`minColumnWidth` および `maxColumnWidth` 属性を使用します。

列によるデータの並び替え

列表示ラベルを使用して行データを並べ替えられるようにするには、並べ替えを有効にする列の `sortable` を `true` に設定します。`sortedBy` を列の `fieldName` プロパティに一致するように設定します。`defaultSortDirection` を変更しない限り、列ヘッダーをクリックすると列は昇順で並べ替えられます。続けてクリックすると、降順になります。`onsort` イベントハンドラを処理して、新しい列インデックスと並び替えの方向でテーブルを更新します。

`onsort` イベントハンドラによってコールされるクライアント側コントローラの例を次に示します。

```
({
    // Client-side controller called by the onsort event handler
    updateColumnSorting: function (cmp, event, helper) {
```

```
        var fieldName = event.getParam('fieldName');
        var sortDirection = event.getParam('sortDirection');
       // assign the latest attribute with the sorted column fieldName and sorted direction

        cmp.set("v.sortedBy", fieldName);
        cmp.set("v.sortedDirection", sortDirection);
        helper.sortData(cmp, fieldName, sortDirection);
    }
    })
```

ヘルパー関数は次のとおりです。

```
({
    sortData: function (cmp, fieldName, sortDirection) {
        var data = cmp.get("v.data");
        var reverse = sortDirection !== 'asc';
        //sorts the rows based on the column header that's clicked
        data.sort(this.sortBy(fieldName, reverse))
        cmp.set("v.data", data);
    },
    sortBy: function (field, reverse, primer) {
        var key = primer ?
            function(x) {return primer(x[field])} :
            function(x) {return x[field]};
        //checks if the two rows should switch places
        reverse = !reverse ? 1 : -1;
        return function (a, b) {
            return a = key(a), b = key(b), reverse * ((a > b) - (b > a));
        }
    }
})
```

アクセシビリティ

データテーブルは、キーボードを使用して、ナビゲーションモードとアクションモードで使用できます。ナビゲーションモードを開始するには、Tab キーを押してデータテーブルに切り替えます。テーブルボディの最初のデータセルがフォーカスされます。矢印キーを使用して、テーブルを移動します。

アクションモードでは、列をサイズ変更できます。列のサイズを変更するには、上矢印キーを押してヘッダーに移動します。Enter キーまたはスペースキーを押してアクションモードを開始します。次に、Tab キーを押して列の仕切りを有効化して、左右矢印を使用して列をサイズ変更します。列のサイズ変更を終了してナビゲーションモードに戻るには、Tab キーを押します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`getSelectedRows()`: 各選択行のデータの配列を返します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
columns	List	データ型を定義するために使用される列オブジェクトの配列。必須プロパティには、「label」、「dataKey」、「type」があります。デフォルトの型は「text」です。	
data	Object	表示するデータの配列。	
keyField	String	パフォーマンス向上のために必須。各行を一意の ID に関連付けます。	
hideCheckboxColumn	Boolean	行選択のチェックボックス列の非表示または表示。チェックボックス列を非表示にするには、hideCheckboxColumn を true に設定します。デフォルトは false です。	
resizeColumnDisabled	Boolean	列のサイズ変更を無効にするかどうかを指定します。デフォルトは false です。	
minColumnWidth	Integer	すべての列の最小幅。デフォルト値は 50px です。	
maxColumnWidth	Integer	すべての列の最大幅。デフォルト値は 1000px です。	
resizeStep	Integer	ユーザが左または右矢印を押したときの列サイズ変更幅。デフォルト値は 10px です。	
sortedBy	String	並び順を制御する列 fieldName。onsort イベントハンドラを使用してデータを並び替えます。	
sortedDirection	String	並び替え順を指定します。onsort イベントハンドラを使用してデータを並び替えます。有効なオプションには、「asc」および「desc」が含まれます。	
defaultSortDirection	String	並び替えなしの列のデフォルトの並び替えの方向を指定します。有効なオプションには、「asc」および「desc」が含まれます。デフォルトでは、昇順の「asc」です。	
onrowselection	Aura.Action	行が選択されたときにトリガされるアクション。	
onsort	Aura.Action	列が並び替えられたときにトリガされるアクション。	

lightning:dualListbox

入力リストボックスを提供するウィジェットであり、選択可能なオプションのリストボックスが含まれます。選択されたオプションの順序は保存されます。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:dualListbox` コンポーネントは、横並びの 2 つのリストボックスを表します。左側のリストで 1 つ以上のオプションを選択します。選択したオプションを右側のリストに移動します。選択されたオプションの順序は維持され、オプションは並び替えられます。

このコンポーネントは、Lightning Design System のデュエル選択リストからスタイル設定を継承します。

8 つのオプションを含むシンプルなデュアルリストボックスを作成する例を次に示します。オプションの 7、2、3 は、リストボックスの [2 番目のカテゴリ] で選択されます。オプションの 2 と 7 は必須オプションです。

```
<aura:component>
    <aura:attribute name="listOptions" type="List" default="[]"/>
    <aura:attribute name="defaultOptions" type="List" default="[]"/>
    <aura:attribute name="requiredOptions" type="List" default="[]"/>
    <aura:handler name="init" value="{! this }" action="{! c.initialize }"/>
    <lightning:dualListbox aura:id="selectOptions" name="Select Options"  label= "Select
Options"
                           sourceLabel="Available Options"
                           selectedLabel="Selected Options"
                           options="{! v.listOptions }"
                           value="{! v.defaultOptions }"
                           requiredOptions="{! v.requiredOptions }"
                           onchange="{! c.handleChange }"/>
</aura:component>
```

オプションを読み込み、値の変更を処理するクライアント側コントローラを次に示します。

```
/** Client-Side Controller **/
({
    initialize: function (component, event, helper) {
        var options = [
            { value: "1", label: "Option 1" },
            { value: "2", label: "Option 2" },
            { value: "3", label: "Option 3" },
            { value: "4", label: "Option 4" },
            { value: "5", label: "Option 5" },
            { value: "6", label: "Option 6" },
            { value: "7", label: "Option 7" },
            { value: "8", label: "Option 8" },
        ];
        var values = ["7", "2", "3"];
        var required = ["2", "7"];
        component.set("v.listOptions", options);
        component.set("v.defaultOptions", values);
        component.set("v.requiredOptions", required);
    },
    handleChange: function (cmp, event) {
        // Get the list of the "value" attribute on all the selected options
        var selectedOptionsList = event.getParam("value");
```

```
        alert("Options selected: '" + selectedOptionsList + "'");
    }
})
```

ユーザが選択できるオプションの数を指定するには、`min` および `max` 属性を使用します。たとえば、`min` を 3 に、`max` を 8 に設定した場合、ユーザは 3 つのオプションを選択する必要があり、選択できるオプションは 8 つとなります。

使用上の考慮事項

選択された値を取得するには、onchange ハンドラを使用します。

```
({
    onChange: function (cmp, event) {
    // Retrieve an array of the selected options
    var selectedOptionValue = event.getParam("value");
    }
})
```

リスト間でオプションを移動するために左および右ボタンをクリックしたとき、または「選択されたオプション」リストでオプションの順序を変更したときに、onchange ハンドラがトリガされます。

アクセシビリティ

デュアルリストボックスの作業では、次のキーボードショートカットを使用します。

- クリック - 1 つのオプションを選択します。
- Cmd + クリック - 複数のオプションを選択、または選択したオプションを選択解除します。
- Shift+ クリック - クリックしたオプションと、1 つ前にクリックしていたオプションの間のすべてのオプションを選択します。

オプションがフォーカスされている場合:

- 上矢印 - 選択を前のオプションに移動します。
- 下矢印 - 選択を次のオプションに移動します。
- Cmd / Ctrl + 上矢印 - フォーカスを前のオプションに移動します。
- Cmd / Ctrl + 下矢印 - フォーカスを次のオプションに移動します。
- Ctrl + スペースキー - フォーカスされているオプションの選択を切り替えます。
- Cmd / Ctrl + 右矢印 - 選択したオプションを右のリストボックスに移動します。
- Cmd / Ctrl + 左矢印 - 選択したオプションを左のリストボックスに移動します。
- Tab - フォーカスを次の要素に移動します。

ボタンがフォーカスされている場合:

- Enter - ボタンに関連した動作を実行します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

`checkValidity()`: デュアルリストボックスに有効性エラーがあるかどうかを示す、ValidityState オブジェクトの有効なプロパティ値 (Boolean) を返します。

`setCustomValidity(message)`: デュアルリストボックス値の送信時に表示するカスタムエラーメッセージを設定します。

- `message` (String): エラーを説明する文字列。メッセージが空の文字列の場合、エラーメッセージはリセットされます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
name	String	入力要素の名前を指定します。	はい
value	Object	入力要素の値を指定します。	
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトはfalseです。	
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
onchange	Aura.Action	値属性が変更された場合にトリガされるアクション。	
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
onfocus	Aura.Action	要素にフォーカスが設定されたときにトリガされるアクション。	
onblur	Aura.Action	要素がフォーカスを解放したときにトリガされるアクション。	
label	String	デュアルリストボックスの表示ラベル。	はい
sourceLabel	String	ソースオプションリストボックスの表示ラベル。	はい
selectedLabel	String	「選択されたオプション」リストボックスの表示ラベル。	はい

属性名	属性型	説明	必須項目
options	Object[]	選択できるオプションのリスト。各オプションには、label と value 属性が含まれます。	はい
requiredOptions	List	「選択されたオプション」リストボックスから削除できない必須オプションのリスト。このリストは、options 属性の値から作成されます。	
values	List	「選択されたオプション」リストボックスに含めるデフォルトオプションのリスト。このリストは、options 属性の値から作成されます。	
min	Integer	「選択されたオプション」リストボックスに必要な最小オプション数。	
max	Integer	「選択されたオプション」リストボックスの最大オプション数。	

lightning:dynamicIcon

異なる状態のさまざまなアニメーションアイコンを表します。たとえば、このコンポーネントを使用して、読み込み中のグラフなど、処理中のイベントを視覚的に表示できます。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:dynamicIcon コンポーネントは、異なる状態のさまざまな動的アイコンを表します。たとえば、このコンポーネントを使用して、読み込み中のグラフなど、処理中のイベントを視覚的に表示できます。

このコンポーネントは、Lightning Design System の動的アイコンからスタイル設定を継承します。

Ellie アイコンの代替テキストを例を次に示します。

```
<aura:component>
    <lightning:dynamicIcon type="ellie" alternativeText="Your calculation is running."/>
</aura:component>
```

使用上の考慮事項

次のオプションを使用できます。

- type="ellie" 属性を使用して、鼓動し、1 つのアニメーションサイクルの後に停止する青い円を表示します。このアイコンは、項目計算またはバッググラウンドで実行中のプロセスに適しています。
- type="eq" 属性を使用して、ランダムに上下する 3 本のバーを含むアニメーショングラフを表示します。このアイコンは、更新中のグラフに適しています。
- type="score" 属性を使用して、緑で塗りつぶされた円または赤の塗りつぶされていない円を表示します。このアイコンは、陽性および陰性のインジケータの表示に適しています。
- type="strength" 属性を使用して、横に並んだ 3 つの緑または赤のアニメーションの円を表示します。このアイコンは、Einstein 計算またはパスワード強度の表示に適しています。

- `type="trend"` 属性を使用して、上、下、または横を指すアニメーションの矢印を表示します。このアイコンは、売上予測スコアのように、1つの範囲から別の範囲への値の移動を表示する場合に適しています。
- `type="waffle"` 属性を使用して、3 x 3 の円のグリッドで構成される正方形を表示します。このアイコンは、マウスポインタを置くとアニメーションになります。このアイコンは、Lightning Experienceのアプリケーションランチャーのようなアプリケーション関連項目に適しています。

アクセシビリティ

必要に応じて、`alternativeText` 属性を使用して `dynamicIcon` を説明できます。

`dynamicIcon` が装飾的で説明が不要な場合もあります。しかし、小さい画面やウィンドウでは、`dynamicIcon` が情報となることもあります。この場合、`alternativeText` を含めます。`alternativeText` を含めない場合、小さな画面やウィンドウで、`dynamicIcon` がすべての形式で常に装飾的であることを確認します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
type	String	dynamicIcon の Lightning Design System 名。有効な値は、ellie、eq、score、strength、trend、waffle です。	はい
option	String	option 属性は、dynamicIcon の外観を変更します。使用可能なオプションは、type 属性によって異なります。eq の場合、play (デフォルト) または stop。score の場合、positive (デフォルト) または negative。strength の場合、-3、-2、-1、0 (デフォルト)、1、2、3。trend の場合、neutral (デフォルト)、up、down。	
alternativeText	String	dynamicIconを説明するために使用する代替テキスト。このテキストでは、進行中の事柄を説明します。たとえば「グラフ」といったアイコンの外観ではなく、「グラフの更新中」のように説明します。	
onclick	Aura.Action	アイコンがクリックされたときにトリガされるアクション。	

lightning:fileUpload (ベータ)

レコードにファイルをアップロードし添付するファイルアップローダです。

lightning:fileUpload コンポーネントは、複数のファイルをアップロードするための統合された簡単な方法をユーザに提供します。ファイルアップローダには、ドラッグアンドドロップ機能とファイル種別による絞り込みが含まれています。

このコンポーネントは、Lightning Design System のファイルセレクタからスタイル設定を継承します。

ファイルのアップロードは常にレコードに関連付けられるため、recordId 属性は必須です。アップロードされたファイルは、[自分が所有者] 検索条件の [ファイル] ホーム、およびレコード詳細ページのレコードの [添付ファイル] 関連リストで使用できます。Salesforce でサポートされているすべてのファイル形式が許可されていますが、accept 属性を使用してファイル形式を制限できます。

次の例では、複数の PDF および PNG ファイルのアップロードを許可するファイルアップローダを作成します。recordId 値を適切な値に変更します。

```
<aura:component>
    <aura:attribute name="myRecordId" type="String" description="Record to which the files
 should be attached" />
    <lightning:fileUpload label="Attach receipt"
        multiple="true"
        accept=".pdf, .png"
        recordId="{!v.myRecordId}"
        onuploadfinished="{!c.handleUploadFinished}" />
</aura:component
```

onuploadfinished イベントを処理する必要があります。イベントは、アップロードが完了すると起動します。

```
({
    handleUploadFinished: function (cmp, event) {
        // Get the list of uploaded files
        var uploadedFiles = event.getParam("files");
        alert("Files uploaded : " + uploadedFiles.length);
    }
})
```

event.getParam("files") は、プロパティ name と documentId を使用して、アップロードされたファイルのリストを返します。

- name: filename.extension 形式のファイル名 (たとえば、account.jpg)。
- documentId: 069XXXXXXXXXXXX 形式の ContentDocument ID。

ファイルアップロードの制限

Salesforce システム管理者が制限を変更していない限り、デフォルトでは、最大 10 個のファイルを同時にアップロードできます。同時にアップロードするファイル数の組織の制限は、最大25個および最小3個です。アップロードできるファイルの最大サイズは2GBです。コミュニティの場合、ファイルのサイズ制限と許可される種別は、コミュニティファイルモデレーションが定める設定のとおりです。

使用上の考慮事項

このコンポーネントは、Lightning Out またはスタンドアロンアプリケーションによってサポートされておらず、無効の入力として表示されます。また、組織で [HTML で添付ファイルまたはドキュメントレコードとしてアップロードすることを許可しない] セキュリティ設定が有効になっている場合、ファイルアップローダを使用して、.htm、.html、.htt、.htx、.mhtm、.mhtml、.shtm、.shtml、.acgi、.svg の拡張子を持つファイルをアップロードす

ることはできません。詳細は、Salesforce ヘルプの「ファイルのアップロードおよび共有」を参照してください。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
label	String	ファイルアップローダのテキスト表示ラベル。	はい
recordId	String	アップロードされたファイルが関連付けられるレコードのレコード ID。	はい
multiple	Boolean	ユーザが複数のファイルを同時にアップロードできるかどうかを指定します。この値のデフォルトは false です。	
disabled	Boolean	このコンポーネントを無効な状態で表示するかどうかを指定します。無効なコンポーネントをクリックすることはできません。この値のデフォルトは false です。	
accept	List	.pdf、.jpg、.png のように、.ext 形式でアップロードできるファイル拡張子のカンマ区切りリスト。	
onuploadfinished	Aura.Action	ファイルのアップロードが完了したときにトリガされるアクション。	

lightning:flexipageRegionInfo

上位のコンポーネントに Lightning ページ範囲の情報を提供します。

lightning:flexipageRegionInfo コンポーネントは、上位のコンポーネントに Lightning ページ範囲の情報を提供します。Lightning アプリケーションビルダーでコンポーネントがドロップされた範囲の幅を渡します。詳細は、「lightning:flexipageRegionInfo による Lightning ページコンポーネントでの幅の認識」を参照してください。

```
<aura:component implements="flexipage:availableForAllPageTypes">
<aura:attribute name="width" type="String" description=" width of parent region"/>
  <lightning:flexipageRegionInfo width="{!v.width}"/>
     <div id="MyCustomComponent" class="{! v.width}">
         <!-- Your custom component here -->
```

```
    </div>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
width	String	コンポーネントが存在する範囲の幅。	

lightning:flow

Lightning ランタイムでのフローインタビューを表します。

lightning:flow コンポーネントは、Lightning ランタイムでのフローインタビューを表します。

name 属性で、表示するフローを指定します。該当するフローがある場合、inputVariables 属性を使用してそのフローの入力変数を初期化します。

次の例では「Calculate Discounts (商談の割引計算)」フロー (「割引計算機能の構築」Trailhead プロジェクトのもの) を表示します。inputVariables 属性は Opportunity.Id と Opportunity.AccountId をフローの入力変数に渡します。

```
<aura:component>
   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <lightning:flow aura:id="flow" onstatuschange="{!c.statusChange}"/>
</aura:component>
```

クライアント側コントローラでインタビューの入力変数を初期化した後、インタビューを開始して、インタビューの完了時に何を行うのかを決定します。

```
({
  init : function (cmp) {
    var flow = cmp.find("flow");
    var inputVariables = [
      {
        name : 'OpportunityID',
        type : 'String',
        value : 'Opportunity.Id'
      },
      {
        name : 'AccountID',
        type : 'String',
        value : 'Opportunity.AccountId'
      }
    ];
    flow.startFlow("Calculate_discounts", inputVariables);
  },
```

```
  statusChange : function (cmp, event) {
    if(event.getParam('status') === "FINISHED") {
      //Do something
    }
  }
})
```

使用上の考慮事項

参照されるフローが有効化されている必要があります。

フローインタビューの有効な状況は、次のとおりです。

- STARTED: インタビューは正常に開始しました。
- PAUSED: インタビューは正常に一時停止しました。
- FINISHED: 画面を伴うフローのインタビューが完了しました。
- FINISHED_SCREEN: 画面を伴わないフローのインタビューが完了し、コンポーネントは「フローが終了しました」というメッセージのデフォルト画面を表示しています。
- ERROR: 問題が発生して、インタビューに失敗しました。

各フローコンポーネントには、ナビゲーションボタン(戻る、次へ、一時停止、完了)が含まれており、フロー内を移動できます。デフォルトでは、フローが完了すると、コンポーネントは新しいインタビューの最初の画面を再読み込みします。フローが完了したときの動作をカスタマイズするには、状況に FINISHED が含まれる場合の onstatuschange アクションのイベントハンドラを追加します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

startFlow(flowName, inputVariables): フローインタビューを開始します。

- flowName (String): 表示するフローの一意の名前。
- inputVariables (Object[]): フロー入力変数の初期値を設定します。

resumeFlow(interviewId): 一時停止中のフローインタビューを再開します。

- interviewId (String): 再開するインタビューの ID。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
onstatuschange	Action	フローインタビューの現在の状況。	

lightning:formattedDateTime (ベータ)

書式設定された日付と時刻を表示します。

`lightning:formattedDateTime` コンポーネントは、書式設定された日付と時刻を表示します。このコンポーネントは、Intl.DateTimeFormat JavaScript オブジェクトを使用して、日付値の書式を設定します。アプリケーションのユーザ設定に設定されているロケールによって書式が決まります。次の入力値がサポートされています。

- 日付オブジェクト
- ISO8601 形式の文字列
- タイムスタンプ

ISO8601 形式の文字列は、次のいずれかのパターンに一致します。

- YYYY
- YYYY-MM
- YYYY-MM-DD
- YYYY-MM-DDThh:mmTZD
- YYYY-MM-DDThh:mm:ssTZD
- YYYY-MM-DDThh:mm:ss.sTZD

en-US のロケールに基づく例をいくつか示します。

表示: 8/2/2016

```
<aura:component>
    <lightning:formattedDateTime value="1470174029742" />
</aura:component>
```

表示: Tuesday, Aug 02, 16

```
<aura:component>
    <lightning:formattedDateTime value="1470174029742" year="2-digit" month="short"
day="2-digit" weekday="long"/>
</aura:component>
```

表示: 8/2/2016, 3:15 PM PDT

```
<aura:component>
    <lightning:formattedDateTime value="1470174029742" year="numeric" month="numeric"
day="numeric"  hour="2-digit" minute="2-digit" timeZoneName="short" />
</aura:component>
```

使用上の考慮事項

このコンポーネントは、Apple Safari 10 以下でフォールバック動作を提供します。次のフォーマッティングオプションには、古いブラウザでフォールバック動作を使用するときに例外があります。

- `era` はサポートされていません。
- `timeZoneName` は、短縮形では `GMT`、長い形式では `GMT-h:mm` または `GMT+h:mm` を追加します。
- `timeZone` は `UTC` をサポートします。別のタイムゾーン値を使用する場合、`lightning:formattedDateTime` はブラウザのタイムゾーンを使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
day	String	使用できる値は、numeric または 2-digit です。	
era	String	使用できる値は、narrow、short、long です。	
hour	String	使用できる値は、numeric または 2-digit です。	
hour12	Boolean	時刻が 12 時間として表示されるかどうかを決定します。falseの場合、時刻は24時間として表示されます。デフォルト設定は、ユーザのロケールによって決まります。	
minute	String	使用できる値は、numeric または 2-digit です。	
month	String	使用できる値は、2-digit、narrow、short、long です。	
second	String	使用できる値は、numeric または 2-digit です。	
timeZone	String	使用するタイムゾーン。実装には、IANA タイムゾーンデータベースにリストされている任意のタイムゾーンを含めることができます。デフォルトは、ランタイムのデフォルトのタイムゾーンです。この属性は、デフォルトのタイムゾーンを上書きする場合のみ使用します。	
timeZoneName	String	使用できる値は、shortまたはlongです。たとえば、太平洋タイムゾーンは、「short」を選択すると「PST」、「long」を選択すると「Pacific Standard Time」と表示されます。	
value	Object	書式設定された値であり、日付オブジェクト、タイムスタンプ、または ISO8601 形式の文字列です。	はい
weekday	String	使用できる値は、narrow、short、long です。	
year	String	使用できる値は、numeric または 2-digit です。	

lightning:formattedEmail

mailto: URL スキームが付いているハイパーリンクとしてメールを表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:formattedEmail` コンポーネントは、`mailto:` URLスキームを使用して、参照のみのハイパーリンクとしてメールアドレスを表示します。メールアドレスをクリックすると、デスクトップまたはモバイルデバイスのデフォルトメールアプリケーションが開きます。

次の例は、メールのアイコン付きメールアドレスを表示します。メールアドレスはデフォルトの表示ラベルとして表示されます。

```
<aura:component>
    <lightning:formattedEmail value="hello@myemail.com" />
</aura:component>
```

複数のメールアドレスがサポートされています。次の例では、「Send a group email (グループメールを送信)」という表示ラベルがハイパーリンクとして表示されます。

```
<aura:component>
   <lightning:formattedEmail value="hello@email1.com,hello@email2.com" label="Send a group
 email" />
</aura:component>
```

次の例は、cc、件名、メール本文の値を含むメールアドレスを作成します。この表示ラベルは、ハイパーリンクとして表示されます。

```
<aura:component>
   <lightning:formattedEmail value="hello@myemail.com?cc=cc@myemail.com&subject=My%20subject
 &body=The%20email%20body"
                              label="Send us your feedback" />
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	String	表示ラベルが指定されていない場合に表示されるメールアドレス。	はい
label	String	メールのテキスト表示ラベル。	
onclick	Aura.Action	メールがクリックされたときにトリガされるアクション。	

lightning:formattedLocation

[緯度, 経度] の形式を使用して 10 進数の度数 (DD) で地理位置情報を表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:formattedLocation コンポーネントは、緯度と経度の値を参照のみで表示します。緯度と経度は、10進数の度数で指定された地理座標です。いずれかの値が無効か許可された範囲外にある場合、このコンポーネントには何も表示されません。

緯度の例として、-30、45、37.12345678、-10.0 などがあります。90.5 や -90.5 などの値は有効な緯度ではありません。経度の例として、-100、-120.9762、115.84 などがあります。180.5 や -180.5 などの値は有効な経度ではありません。

次の例では、緯度が 37.7938460、経度が -122.3948370 の地理位置情報を表示します。

```
<aura:component>
    <lightning:formattedLocation latitude="37.7938460" longitude="-122.3948370"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
latitude	数値	地理位置情報の緯度値。緯度値は -90 ～ 90 である必要があります。	はい
longitude	数値	地理位置情報の経度値。経度値は -180 ～ 180 である必要があります。	はい

lightning:formattedNumber (ベータ)

小数、通貨、およびパーセントの書式設定された数値を表示します。

lightning:formattedNumber コンポーネントは、小数、通貨、およびパーセントの書式設定された数値を表示します。このコンポーネントは、Intl.NumberFormat JavaScript オブジェクトを使用して、数値の書式を設定します。アプリケーションのユーザ設定に設定されているロケールによって数値の書式が決まります。

このコンポーネントには、アプリケーションでの数値の書式設定処理方法を指定する属性がいくつかあります。これらの属性の中には minimumSignificantDigits と maximumSignificantDigits があります。有効数字とは数値の精度のことです。たとえば、1000 の有効数字は 1 ですが、1000.0 の有効数字は 5 です。さらに、maximumFractionDigits を使用して小数点以下の桁数をカスタマイズできます。

デフォルトの小数点以下の桁数は 3 桁です。次の例は 1234.568 を返します。

```
<aura:component>
    <lightning:formattedNumber value="1234.5678" />
</aura:component>
```

通貨のデフォルトの小数点以下の桁数は 2 桁です。この例では、書式設定された数値は $5,000.00 と表示されます。

```
<aura:component>
    <lightning:formattedNumber value="5000" style="currency" currency="USD" />
</aura:component>
```

パーセントのデフォルトの小数点以下の桁数は 0 桁です。この例では、書式設定された数値は 50% と表示されます。

```
<aura:component>
    <lightning:formattedNumber value="0.5" style="percent" />
</aura:component>
```

使用上の考慮事項

このコンポーネントは、Apple Safari 10 以下で次のフォールバック動作を提供します。

- style を currency に設定した場合、ロケールとは異なる currencyCode 値を指定すると、記号の代わりに通貨コードが表示されます。次の例では、フォールバックモードの場合に EUR12.34 が表示され、その他の場合に €12.34 が表示されます。

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="EUR"/>
```

- currencyDisplayAs は、記号のみをサポートします。次の例では、currencyCode がユーザのロケール通貨と一致する場合に限ってフォールバックモードで $12.34 が表示され、その他の場合は USD12.34 が表示されます。

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="USD" currencyDisplayAs="symbol"/>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
currencyCode	String	style='currency' の場合のみ使用します。この属性は、どの通貨が表示されるかを決定します。使用できる値は、ISO4217 通貨コード (米ドルを示す「USD」など) です。	
currencyDisplayAs	String	通貨を表示する方法を決定します。使用できる値は、symbol、code、name です。この値のデフォルトは symbol です。	

属性名	属性型	説明	必須項目
maximumFractionDigits	Integer	使用できる小数点以下の最大桁数。	
maximumSignificantDigits	Integer	使用できる有効数字の最大値。使用できる値は 1 ～ 21 です。	
minimumFractionDigits	Integer	使用しなければならない小数点以下の最小桁数。	
minimumIntegerDigits	Integer	使用しなければならない整数の最小桁数。使用できる値は 1 ～ 21 です。	
minimumSignificantDigits	Integer	使用しなければならない有効数字の最小値。使用できる値は 1 ～ 21 です。	
style	String	使用する数値書式スタイル。使用できる値は、decimal、currency、percent です。この値のデフォルトは decimal です。	
value	BigDecimal	書式設定する値。	はい

lightning:formattedPhone

tel: URL スキームが付いているハイパーリンクとして、電話番号を表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:formattedPhone コンポーネントは、tel: URL スキームが付いているハイパーリンクとして、電話番号を参照のみで表示します。デスクトップで電話番号をクリックすると、デフォルトの VOIP コールアプリケーションが開きます。モバイルデバイスでは、電話番号をクリックするとその番号に発信されます。

1 で始まる 10 または 11 桁の電話番号を指定すると、その番号が (999) 999-9999 形式で表示されます。番号の前に「+」記号を付けると、その番号が +19999999999 形式で表示されます。

ハイパーリンクで (425) 333-4444 を表示する 2 つの方法を次に示します。

```
<aura:component>
    <p><lightning:formattedPhone value="4253334444"></lightning:formattedPhone></p>
    <p><lightning:formattedPhone value="14253334444"></lightning:formattedPhone></p>
</aura:component>
```

前の例によって次の HTML が表示されます。

```
<a href="tel:4253334444">(425) 333-4444</a>
<a href="tel:14253334444">(425) 333-4444</a>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	Integer	表示する電話番号を設定します。	
onclick	Aura.Action	電話番号がクリックされたときにトリガされるアクション。	

lightning:formattedRichText

ホワイトリストに登録されているタグと属性を使用して書式設定されたリッチテキストを表示します。その他のタグと属性は削除され、そのテキストコンテンツのみが表示されます。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:formattedRichText` コンポーネントは、リッチテキストを参照のみで表示します。リッチテキストとは、太字テキストの場合は `<b>`、下線付きテキストの場合は `<u>` などの HTML タグで書式設定されたテキストのことです。`lightning:inputRichText` コンポーネントを使用するか、クライアント側コントローラで値をプログラムで設定し、このコンポーネントにリッチテキストを渡すことができます。

次の例では、`lightning:formattedRichText` コンポーネントに結び付けられたリッチテキストエディタを作成します。初期化中にリッチテキストコンテンツが設定されます。

```
<aura:component>
    <aura:handler name="init" value="{! this }" action="{! c.init }" />
    <aura:attribute name="richtext" type="String"/>
    <!-- Rich text editor and formatted output -->
    <lightning:inputRichText value="{!v.richtext}"/>
    <lightning:formattedRichText value="{!v.richtext}" />
</aura:component>
```

クライアント側コントローラでリッチテキストコンテンツを初期化します。

```
({
    init: function(cmp) {
        var content = "<h1>Hello!</h1>";
        cmp.set("v.richtext", content);
    }
})
```

値の定義で二重引用符を使用するには、\ 文字を使用してエスケープします。

```
var rte = "<h1 style=\"color:blue;\">This is a blue heading</h1>";
cmp.set("v.richtext", rte);
```

コンポーネントのマークアップで HTML タグを渡すには、タグを次のようにエスケープします。

```
<lightning:formattedRichText value="&lt;h1>TEST&lt;/h1>" />
```

サポートされる HTML タグおよび属性

コンポーネントは、XSS の脆弱性を防止するために、value 属性に渡される HTML タグをサニタイズします。また、書式設定された出力が有効な HTML になるようにもします。たとえば、<div>My Title</h1> のようにタグが一致しない場合、コンポーネントは <div>My Title</div> を返します。

クライアント側コントローラでサポートされていないタグを設定した場合、それらのタグが削除され、テキストコンテンツは保持されます。サポートされている HTML タグは、a、abbr、acronym、address、b、br、big、blockquote、caption、cite、code、col、colgroup、del、div、dl、dd、dt、em、font、h1、h2、h3、h4、h5、h6、hr、i、img、ins、kbd、li、ol、p、q、s、samp、small、span、strong、sub、sup、table、tbody、td、tfoot、th、thead、tr、tt、u、ul、var、strike です。

サポートされているHTML属性は、accept、action、align、alt、autocomplete、background、bgcolor、border、cellpadding、cellspacing、checked、cite、class、clear、color、cols、colspan、coords、data-fileid、datetime、default、dir、disabled、download、enctype、face、for、headers、height、hidden、high、href、hreflang、id、ismap、label、lang、list、loop、low、max、maxlength、media、method、min、multiple、name、noshade、novalidate、nowrap、open、optimum、pattern、placeholder、poster、preload、pubdate、radiogroup、readonly、rel、required、rev、reversed、rows、rowspan、spellcheck、scope、selected、shape、size、span、srclang、start、src、step、style、summary、tabindex、target、title、type、usemap、valign、value、width、xmlns です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	String	表示するリッチテキストを設定します。	

lightning:formattedText

テキストを表示し、改行を禁則処理で置き換え、要求された場合は linkify で表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:formattedText` コンポーネントは、アンカータグ (「linkify」とも呼ばれる) でテキスト、ラッピング URL、およびメールアドレスを参照のみで表示します。また、\r または \n 文字を `<br />` タグに変換します。

アンカータグでテキストブロックに URL とメールアドレスを表示するには、`linkify="true"` を設定します。設定されていない場合、URL とメールアドレスはプレーンテキストで表示されます。`linkify="true"` を設定すると、`format="html" scope="external" type="new-window:HTML"` のアンカータグで URL とメールアドレスがラップされます。URL とメールアドレスには、それぞれ `http://` と `mailto://` が追加されます。

```
<aura:component>
    <lightning:formattedText linkify="true" value="I like salesforce.com and
trailhead.salesforce.com." />
</aura:component>
```

前の例は次のように表示されます。

```
I like <a format="html" scope="external" type="new-window:HTML"
href="http://salesforce.com">salesforce.com</a>
and <a format="html" scope="external" type="new-window:HTML"
href="http://trailhead.salesforce.com">trailhead.salesforce.com</a>.
```

使用上の考慮事項

`lightning:formattedText` では、`http`、`https`、`ftp`、`mailto` プロトコルがサポートされています。

ハイパーリンクを操作していて `target` 値を指定する必要がある場合は、代わりに `lightning:formattedURL` を使用します。メールアドレスのみを操作する場合は、`lightning:formattedEmail` を使用します。

アンカータグ以外のタグを使用するリッチテキストの場合は、代わりに `lightning:formattedRichText` を使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	String	出力するテキスト。	

属性名	属性型	説明	必須項目
linkify	Boolean	テキストをリンクに変換するかどうかを指定します。true に設定した場合、URL とメールアドレスがアンカータグで表示されます。	

lightning:formattedUrl

URL をハイパーリンクとして表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:formattedUrl コンポーネントは、href 属性を使用したハイパーリンクとして、URL を参照のみで表示します。リンクは相対 URL または絶対 URL で指定できます。絶対 URL は、http://、https://、ftp:// などのプロトコルを使用します。このコンポーネントは、href 値として絶対URL、表示テキストとして label を使用してアンカーリンクを表示します。表示ラベルが指定されていない場合、絶対 URL が表示テキストとして使用されます。URL をクリックすると、クリックしたウィンドウと同じウィンドウでその URL にリダイレクトされます。

絶対 URL は、デフォルトで http:// プロトコルを使用して表示されます。

```
<aura:component>
    <lightning:formattedUrl value="www.salesforce.com" />
</aura:component>
```

前の例によって次の HTML が表示されます。

```
<a href="http://www.salesforce.com">http://www.salesforce.com</a>
```

相対 URL は、現在のサイト内のパスに移動します。

```
<aura:component>
    <!-- Resolves to http://current-domain/my/path -->
    <lightning:formattedUrl value="/my/path" />
</aura:component>
```

使用上の考慮事項

リンクを開く場所を変更するには、target 属性を使用します。target を指定しない場合、ハイパーリンクは href 属性なしで表示されます。サポートされている target 値は、次のとおりです。

- _blank: 新しいウィンドウまたはタブでリンクが開きます。
- _self: クリックしたフレームと同じフレームでリンクが開きます。これがデフォルトの動作です。
- _parent: 親フレームでリンクが開きます。親がない場合、これは _self と同様です。
- _top: 最上位の参照コンテキストでリンクが開きます。親がない場合、これは _self と同様です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	String	書式設定する URL。	
target	String	リンクを開く場所を指定します。オプションは、_blank、_parent、_self、_top です。	
label	String	リンクに表示するテキスト。	
tooltip	String	リンクにマウスポインタが置かれたときに表示するテキスト。	
onclick	Aura.Action	URL がクリックされたときにトリガされるアクション。	

lightning:helptext (ベータ)

テキストのポップオーバーがあるアイコンです。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:helptext コンポーネントは、画面の要素を説明する少量のテキストを含むポップオーバーがあるアイコンを表示します。ポップオーバーは、関連付けられているアイコンにユーザがマウスポインタまたはフォーカスを置くと表示されます。このコンポーネントはツールチップと似ており、項目レベルのヘルプテキストを表示する場合などに役立ちます。ツールチップコンテンツでは、HTML マークアップはサポートされていません。

このコンポーネントは、Lightning Design System のツールチップからスタイル設定を継承します。

デフォルトでは、ツールチップは utility:info アイコンを使用します。Lightning Design System ユーティリティアイコンカテゴリでは、lightning:helptext で使用できる約 200 個のユーティリティアイコンを提供しています。Lightning Design System では複数のアイコンカテゴリが提供されていますが、lightning:buttonIcon で使用できるのはユーティリティカテゴリのみです。

ユーティリティアイコンを表示するには、https://lightningdesignsystem.com/icons/#utilityにアクセスしてください。

次の例では、ツールチップがあるアイコンを作成します。

```
<aura:component>
    <lightning:helptext
        content="Your email address will be your login name" />
</aura:component>
```

ポップオーバーはアイコンの左下に固定され、スペースがある場合はアイコンの上に表示されます。ビューポートに従ってその位置が自動的に調整されます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
content	String	ポップオーバーに表示するテキスト。	
iconName	String	表示要素として使用するアイコンの Lightning Design System 名。名前は、形式「utility:info」で記述します。「utility」はカテゴリ、「info」は表示する特定のアイコンです。デフォルト値は「utility:info」です。	

lightning:icon

コンテキストを示し、使いやすさを向上させるビジュアル要素を表します。

lightning:icon は、コンテキストを示し、使いやすさを向上させるビジュアル要素です。アイコンは、別のコンポーネントのボディの内部で使用するか単独で使用できます。

使用できるアイコンについては、https://lightningdesignsystem.com/icons を参照してください。

次に例を示します。

```
<aura:component>
    <lightning:icon iconName="action:approval" size="large" alternativeText="Indicates
approval"/>
</aura:component>
```

スタイルをカスタマイズするには、variant、size、または class 属性を使用します。variant 属性は、ユーティリティアイコンの外観を変更します。たとえば、error バリエーションは、エラーユーティリティアイコンを赤で塗りつぶします。

```
<lightning:icon iconName="utility:error" variant="error"/>
```

アイコンの色やスタイルをさらに変更するには、class 属性を使用します。

使用上の考慮事項

スタンドアロンアプリケーションで lightning:icon を使用する場合、アイコンリソースを正しく解決するために force:slds を拡張します。

```
<aura:application extends="force:slds">
    <lightning:icon iconName="utility:error" variant="error"/>
</aura:application>
```

アクセシビリティ

アイコンを説明するには、alternativeText 属性を使用します。説明では、アイコンの外観 (「ペーパークリップ」) ではなく、ボタンをクリックしたときに何が起こるか (「ファイルのアップロード」など) を示す必要があります。

場合によっては、アイコンは装飾的で説明が不要なこともあります。ただし、画面サイズに基づいて装飾的なアイコンと情報的なアイコンを切り替えることができます。alternativeText の説明を含めない場合、小さな画面やウィンドウで、アイコンがすべての形式で装飾的であることを確認します。

属性

属性名	属性型	説明	必須項目
alternativeText	String	アイコンを説明するために使用する代替テキスト。このテキストでは、アイコンの外観 (「ペーパークリップ」) ではなく、ボタンをクリックしたときに何が起こるか (「ファイルのアップロード」など) を説明する必要があります。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。	はい
size	String	アイコンのサイズ。オプションは、xx-small、x-small、medium、large です。この値のデフォルトは medium です。	
title	String	マウスが要素に重ねられたときにツールチップテキストを表示します。	
variant	String	バリエーションによって、ユーティリティアイコンの外観が変化します。使用できるバリエーションは、inverse、warning、error です。濃い色の背景でユーティリティアイコンに白の余白を実装するには、inverse バリエーションを使用します。	

lightning:input (ベータ)

type 属性に応じてユーザ入力を受け入れる対話型コントロールを表します。

lightning:input コンポーネントは HTML input 要素を作成します。このコンポーネントは、checkbox、date および datetime-local、email、file、password、search、tel、url、number、radio、toggle など、HTML5 の入力種別をサポートします。デフォルトは text です。

onblur、onfocus、onchange などの入力イベントに対してクライアント側のコントローラを定義できます。たとえば、コンポーネントの値が変更された場合にコンポーネントの変更イベントを処理するには、onchange 属性を使用します。

このコンポーネントは、Lightning Design System の入力からスタイル設定を継承します。

チェックボックス

チェックボックスでは、1 つ以上のオプションを選択できます。lightning:input type="checkbox" は、単一チェックボックスの作成に役立ちます。チェックボックスのグループを操作する場合は、代わりに lightning:checkboxGroup を使用します。

```
<lightning:input type="checkbox" label="Red" name="red" checked="true"/>
<lightning:input type="checkbox" label="Blue" name="blue" />
```

チェックボックスボタン

チェックボックスボタンでは、代替ビジュアルデザインの 1 つ以上のオプションを選択できます。

```
<lightning:input type="checkbox" label="Add pepperoni" name="addPepperoni" checked="true"
 value="pepperoni" />
<lightning:input type="checkbox-button" label="Add salami" name="addSalami" value="salami"
 />
```

色

カラーピッカーを使用するかテキスト項目に色を入力して、色を指定できます。ネイティブのカラーピッカーが使用されます。

```
<lightning:input type="color" label="Color" name="color" value="#EEEEEE"/>
```

日付

日付を入力するための入力項目。日付ピッカーは現在、Lightning Design System スタイル設定をサポートしていません。日付形式は、onblur イベント時に自動的に検証されます。

```
<lightning:input type="date" label="Birthday" name="date" />
```

日時 (ローカル)

日時を入力するための入力項目。日付ピッカーは現在、Lightning Design System スタイル設定をサポートしていません。日時形式は、onblur イベント時に自動的に検証されます。

```
<lightning:input type="datetime-local" label="Birthday" name="datetime" />
```

メール

メールアドレスを入力するための入力項目。メールパターンは、onblur イベント時に自動的に検証されます。

```
<lightning:input type="email" label="Email" name="email" value="abc@domain.com" />
```

ファイル

[ファイルをアップロード] ボタンまたはドラッグアンドドロップゾーンを使用して、ファイルをアップロードするための入力項目。選択されたファイルのリストを取得するには、event.getSource().get("v.files"); を使用します。

```
<lightning:input type="file" label="Attachment" name="file" multiple="true"
accept="image/png, .zip" onchange="{! c.handleFilesChange }"/>
```

月

月と年を入力するための入力項目。日付ピッカーは現在、Lightning Design System スタイル設定を継承しません。月と年の形式は、onblur イベント時に自動的に検証されます。

```
<lightning:input type="month" label="Birthday" name="month" />
```

数値

数値を入力するための入力項目。数値入力を操作する場合は、max、min、step などの属性を使用できます。

```
<lightning:input type="number" name="number" label="Number" value="12345"/>
```

数値入力をパーセントまたは通貨として書式設定するには、formatter をそれぞれ percent または currency に設定します。

```
<lightning:input type="number" name="ItemPrice"
    label="Price" value="12345" formatter="currency"/>
```

パーセント入力と通貨入力の項目では、ネイティブ実装によって要求されるように、段階的な増分を 0.01 に指定する必要があります。

```
<lightning:input type="number" name="percentVal" label="Enter a percentage value"
formatter="percent" step="0.01" />
<lightning:input type="number" name="currencyVal" label="Enter a dollar amount"
formatter="currency" step="0.01" />
```

パスワード

パスワードを入力するための入力項目。入力した文字はマスクされます。

```
<lightning:input type="password" label="Password" name="password" />
```

ラジオ

ラジオボタンでは、指定された数のオプションの 1 つのみを選択できます。lightning:input type="radio" は、単一ラジオボックスの作成に役立ちます。ラジオボタンのセットを操作している場合は、代わりに lightning:radioGroup を使用します。

```
<lightning:input type="radio" label="Red" name="red" value="red" checked="true" />
<lightning:input type="radio" label="Blue" name="blue" value="blue" />
```

範囲

数値を入力するためのスライダコントロール。数値入力を操作する場合は、max、min、step などの属性を使用できます。

```
<lightning:input type="range" label="Number" name="number" min="0" max="10" />
```

検索

検索文字列を入力するための入力項目。この項目には、Lightning Design System 検索ユーティリティアイコンが表示されます。

```
<lightning:input type="search" label="Search" name="search" />
```

電話

電話番号を入力するための入力項目。pattern 属性を使用して、項目検証のパターンを定義します。

```
<lightning:input type="tel" label="Telephone" name="tel" value="343-343-3434"
pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
```

テキスト

テキストを入力するための入力項目。これはデフォルトの入力種別です。

```
<lightning:input label="Name" name="myname" />
```

時間

時間を入力するための入力項目。時間形式は、onblur イベント時に自動的に検証されます。

```
<lightning:input type="time" label="Time" name="time" />
```

切り替え

指定された 2 つの値の 1 つを選択するためのチェックボックスの切り替え。

```
<lightning:input type="toggle" label="Toggle value" name="togglevalue" checked="true" />
```

URL

URL を入力するための入力項目。この URL パターンは、onblur イベント時に自動的に検証されます。

```
<lightning:input type="url" label="Website" name="website" />
```

週

週と年を入力するための入力項目。日付ピッカーは現在、Lightning Design System スタイル設定を継承しません。週と年の形式は、onblur イベント時に自動的に検証されます。

```
<lightning:input type="week" label="Week" name="week" />
```

入力規則

このコンポーネントでは、クライアント側で入力規則が使用できます。たとえば、url または email の入力種別に対して URL やメールアドレスを入力する必要がある場合にエラーメッセージが表示されます。

その他の項目要件を定義できます。たとえば、最大長を設定するには maxlength 属性を使用します。

```
<lightning:input name="quantity" value="1234567890" label="Quantity" maxlength="10" />
```

入力の有効性状態を確認するには、ValidityState Web API に基づく validity 属性を使用します。項目が有効かどうかを判断するために、クライアント側コントローラの有効性状態にアクセスできます。たとえば、次の入力項目があるとします。

```
<lightning:input name="input" aura:id="myinput" label="Enter some text" onblur="{!
c.handleBlur }" />
```

valid プロパティは、すべての制約検証が満たされた場合に true を返します (この場合はなし)。

```
handleBlur: function (cmp, event) {
    var validity = cmp.find("myinput").get("v.validity");
    console.log(validity.valid); //returns true
    }
```

たとえば、数種類の項目とボタンが 1 つ設定された次のフォームがあるとします。無効な項目にエラーメッセージを表示するには、showHelpMessageIfInvalid() メソッドを使用します。

```
<aura:component>
        <lightning:input aura:id="field" label="First name" placeholder="First name"
required="true" />
        <lightning:input aura:id="field" label="Last name" placeholder="Last name"
required="true" />
       <lightning:button aura:id="submit" type="submit" label="Submit" onclick="{! c.onClick
 }" />
</aura:component>
```

クライアント側コントローラで項目を検証します。

```
({
    onClick: function (cmp, evt, helper) {
        var allValid = cmp.find('field').reduce(function (validSoFar, inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validSoFar && inputCmp.get('v.validity').valid;
         }, true);
         if (allValid) {
             alert('All form entries look valid. Ready to submit!');
         } else {
             alert('Please update the invalid form entries and try again.');
         }
     }
})
```

この validity 属性は、次の boolean プロパティがあるオブジェクトを返します。

- badInput: 値が無効であることを示します
- patternMismatch: 値が指定されたパターンに一致していないことを示します。
- rangeOverflow: 値が指定された max 属性よりも大きいことを示します。
- rangeUnderflow: 値が指定された min 属性よりも小さいことを示します。
- stepMismatch: 値が指定された step 属性に一致していないことを示します。
- tooLong: 値が指定された maxlength 属性を超えていることを示します。
- typeMismatch: 値がメールまたは url 入力種別の所定の構文に一致していないことを示します。
- valid: 値が有効であることを示します。

- valueMissing: required 属性が true に設定されている場合に、空の値が指定されたことを示します。

エラーメッセージ

入力規則の検証に失敗した場合、デフォルトで次のメッセージが表示されます。

- badInput: 有効な値を入力してください。
- patternMismatch: エントリは許可されているパターンと一致しません。
- rangeOverflow: 数値が高すぎます。
- rangeUnderflow: 数値が低すぎます。
- stepMismatch: エントリは有効な増分ではありません。
- tooLong: エントリが長すぎます。
- typeMismatch: 無効な形式を入力しました。
- valueMissing: この項目を入力してください。

次の属性に独自の値を指定することでデフォルトのメッセージを上書きできます: messageWhenBadInput、messageWhenPatternMismatch、messageWhenTypeMismatch、messageWhenValueMissing、messageWhenRangeOverflow、messageWhenRangeUnderflow、messageWhenStepMismatch、messageWhenTooLong。

たとえば、入力が 5 文字未満の場合にカスタムエラーメッセージを表示します。

```
<lightning:input name="firstname" label="First Name" minlength="5"
    messageWhenBadInput="Your entry must be at least 5 characters." />
```

使用上の考慮事項

maxlength は、入力できる文字数を制限します。許可されている文字数を超えて入力できないため、messageWhenTooLong エラーメッセージはトリガされません。ただし、messageWhenPatternMismatch と pattern を使用して同じ動作を実現できます。

```
<lightning:input type="text" messageWhenPatternMismatch="Too many characters!"
pattern=".{0,5}" name="input-name" label="Enter up to 5 characters" />
```

次の入力種別はサポートされていません。

- button
- hidden
- image
- reset
- submit

入力種別が button、reset、および submit の場合、代わりに lightning:button を使用します。

また、チェックボックス、ラジオボタン、トグルスイッチを操作するときは、aura:id を使用し、コンポーネントの配列をグループ化してトラバースします。get("v.checked") を使用して、どの要素がオンまたはオフになっているかを判断することができ、DOM にアクセスする必要がありません。name 属性と value 属性を使用し、反復中に各コンポーネントを識別することもできます。次の例では、aura:id を使用して 3 個のチェックボックスをグループ化しています。

```
<aura:component>
    <fieldset>
```

```
        <legend>Select your favorite color:</legend>
        <lightning:input type="checkbox" label="Red"
            name="color1" value="1" aura:id="colors"/>
        <lightning:input type="checkbox" label="Blue"
            name="color2" value="2" aura:id="colors"/>
        <lightning:input type="checkbox" label="Green"
            name="color3" value="3" aura:id="colors"/>
    </fieldset>
<lightning:button label="Submit" onclick="{!c.submitForm}"/>
</aura:component>
```

アクセシビリティ

アクセシビリティのためのテキスト表示ラベルを指定して、情報が支援技術で使用できるようにする必要があります。`label` 属性は、入力コンポーネントのHTML `label` 要素を作成します。表示ラベルをビューに表示させず、支援技術には使用できるようにするには、`label-hidden` バリエーションを使用します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

`showHelpMessageIfInvalid()`: フォームコントロールが無効な状態の場合、ヘルプメッセージを表示します。

属性

属性名	属性型	説明	必須項目
accept	String	サーバが受け入れるファイルの種類を指定します。この属性は、type='file' の場合のみ使用できます。	
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
checked	Boolean	チェックボックスをオンにするかどうかを指定します。この値のデフォルトは false です。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
files	Object	選択されたファイルが含まれる FileList。この属性は、type='file' の場合のみ使用できます。	
formatter	String	使用するフォーマッタを伴う文字列値。	

属性名	属性型	説明	必須項目
isLoading	Boolean	データが読み込み中であることを示すスピナーを表示するかどうかを指定します。この値のデフォルトはfalseです。	
label	String	入力のテキスト表示ラベル。	はい
max	decimal	浮動小数点数の値の上限。	
maxlength	Integer	項目内で使用できる最大文字数。	
messageToggleActive	String	切り替えの有効状態を示すテキスト。デフォルトは「Active」(有効) です。	
messageToggleInactive	String	切り替えの無効状態を示すテキスト。デフォルトは「Inactive」(無効) です。	
messageWhenBadInput	String	不正な入力が検出された場合に表示されるエラーメッセージ。	
messageWhenPatternMismatch	String	パターンの不一致が検出された場合に表示されるエラーメッセージ。	
messageWhenRangeOverflow	String	範囲を超えたことが検出された場合に表示されるエラーメッセージ。	
messageWhenRangeUnderflow	String	範囲を下回ったことが検出された場合に表示されるエラーメッセージ。	
messageWhenStepMismatch	String	ステップの不一致が検出された場合に表示されるエラーメッセージ。	
messageWhenTooLong	String	値が長すぎる場合に表示されるエラーメッセージ。	
messageWhenTypeMismatch	String	種別の不一致が検出された場合に表示されるエラーメッセージ。	
messageWhenValueMissing	String	値がない場合に表示されるエラーメッセージ。	
min	decimal	浮動小数点数の値の下限。	
minlength	Integer	項目内で使用できる最小文字数。	
multiple	Boolean	ユーザが複数の値を入力できることを指定します。この属性は、type='file' または type='email' の場合のみ使用できます。	
name	String	入力要素の名前を指定します。	はい
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onchange	Action	値属性が変更された場合にトリガされるアクション。	

属性名	属性型	説明	必須項目
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
pattern	String	入力値をチェックする正規表現を指定します。この属性は、text、date、search、url、tel、email、password の種別でサポートされます。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトはfalseです。	
step	Object	正の浮動小数点の値の粒度。粒度が重要でない場合は、「any」を使用します。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
type	String	入力の種別。この値のデフォルトは text です。	
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
value	Object	入力要素の値を指定します。	
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	

lightning:inputLocation

緯度値と経度値を受け入れる地理位置情報の複合項目を表します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:inputLocation コンポーネントは、緯度値と経度値を受け入れる地理位置情報の複合項目を表します。緯度と経度は、10進数の度数で指定された地理座標です。地理位置情報の複合項目では、その緯度と経度によって位置を特定できます。緯度項目は -90 ～ 90 の値を受け入れ、経度項目は -180 ~ 180 の値を受け入れます。有効な範囲外の値を入力すると、エラーメッセージが表示されます。

緯度の例として、-30、45、37.12345678、-10.0 などがあります。90.5 や -90.5 などの値は有効な緯度ではありません。経度の例として、-100、-120.9762、115.84 などがあります。180.5 や -180.5 などの値は有効な経度ではありません。

次の例では、緯度が 37.7938460、経度が -122.3948370 の地理位置情報の複合項目を表示します。

```
<aura:component>
    <lightning:inputLocation label="My Coordinates" latitude="37.7938460"
longitude="-122.3948370"/>
</aura:component>
```

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

`blur()`: 要素からフォーカスを外します。

`checkValidity()`: コンボボックスに有効性エラーがあるかどうかを示す ValidityState オブジェクトの有効なプロパティ値 (Boolean) を返します。

`showHelpMessageIfInvalid()`: 複合項目が無効な状態の場合、ヘルプメッセージを表示します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
latitude	String	緯度値。緯度値は -90 ～ 90 である必要があります。	
longitude	String	経度値。経度値は -180 ～ 180 である必要があります。	
required	Boolean	複合項目に入力する必要があるかどうかを指定します。ユーザが項目に値を入力していないと、エラーメッセージが表示されます。この値のデフォルトは false です。	
disabled	Boolean	複合項目を無効にするかどうかを指定します。無効な項目はグレー表示され、クリックできません。この値のデフォルトは false です。	
readonly	Boolean	複合項目が参照のみかどうかを指定します。この値のデフォルトは false です。	
variant	String	バリエーションによって複合項目の外観が変更されます。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	
label	String	複合項目のテキスト表示ラベル。	

属性名	属性型	説明	必須項目
onblur	Aura.Action	入力がフォーカスを解放したときにトリガされるアクション。	
onchange	Aura.Action	値が変更された場合にトリガされるアクション。	
onfocus	Aura.Action	入力にフォーカスが設定されたときにトリガされるアクション。	

lightning:inputRichText (ベータ)

カスタマイズ可能なツールバーを備えた、リッチテキスト入力用の WYSIWYG エディタ。

lightning:inputRichText コンポーネントは、Quill.js ライブラリに基づいて、ユーザがリッチテキストを追加、編集、書式設定、削除できるリッチテキストエディタを作成します。ツールバー設定が異なる複数のリッチテキストエディタを作成できます。リッチコンテンツのエディタへの貼り付けは、ツールバーでその機能が使用できる場合にサポートされています。たとえば、ツールバーに太字ボタンがあれば、太字テキストを貼り付けることができます。ツールバーボタンが多くてツールバーの幅に収まらない場合は、オーバーフローメニューが提供されます。

このコンポーネントは、Lightning Design System のリッチテキストエディタからスタイル設定を継承します。

次の例では、リッチテキストエディタを作成し、初期化中にそのコンテンツを設定します。

```
<aura:component>
    <aura:attribute name="myVal" type="String" />
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>
    <lightning:inputRichText value="{!v.myVal}" />
</aura:component>
```

クライアント側コントローラでリッチテキストコンテンツを初期化します。

```
({
    init: function(cmp) {
        cmp.set('v.myVal', '<b>Hello!</b>');
    }
})
```

ツールバーのカスタマイズ

デフォルトでは、ツールバーには、フォントファミリおよびサイズメニューと、[太字]、[斜体]、[下線]、[取り消し線] ボタンが含まれるテキスト書式設定ブロックが表示されます。また、[箇条書き]、[段落番号]、[インデント]、[アウトデント] ボタンが含まれる本文書式設定ブロックと、[テキストを左揃え]、[テキストを中央揃え]、[テキストを右揃え] ボタンが含まれるテキスト整列ブロックも表示されます。[書式設定を削除] ボタンも使用できます。このボタンは、常にツールバーの最後尾に単独で表示されます。

disabledCategories 属性を使用して、カテゴリごとにボタンを無効にすることができます。次のカテゴリがあります。

1. FORMAT_FONT: フォントファミリおよびサイズメニュー
2. FORMAT_TEXT: テキスト書式設定ボタン

3. FORMAT_BODY: 本文書式設定ボタン
4. ALIGN_TEXT: テキスト整列ボタン
5. REMOVE_FORMATTING: 書式設定削除ボタン

フォントメニューで使用できるフォントは、Arial、Courier、Garamond、Salesforce Sans、Tahoma、Times New Roman、Verdana です。フォントのデフォルトは、Salesforce Sans のサイズ 12px です。サポートされているフォントサイズは、8、9、10、11、12、14、16、18、20、22、24、26、28、36、48、72 です。エディタでテキストをコピーして貼り付ける場合、フォントメニューで使用できるフォントである場合のみ、フォントが保持されます。

入力規則

lightning:inputRichText には、検証は組み込まれていませんが、独自の検証ロジックに関連付けることができます。リッチテキストエディタの境界線の色を赤に変更するには、valid 属性を false に設定します。次の例は、リッチテキストコンテンツが空または未定義であるかどうかを確認します。

```
<aura:component>
    <aura:attribute name="myVal" type="String" />
   <aura:attribute name="errorMessage" type="String" default="You haven't composed anything
 yet."/>
    <aura:attribute name="validity" type="Boolean" default="true"/>
    <lightning:inputRichText value="{!v.myVal}" placeholder="Type something interesting"
messageWhenBadInput="{!v.errorMessage}" valid="{!v.validity}"/>
    <lightning:button name="validate" label="Validate" onclick="{!c.validate}"/>
</aura:component>
```

クライアント側コントローラで、リッチテキストエディタの有効性を切り替え、無効な場合はエラーメッセージを表示します。

```
({
    validate: function(cmp) {
        if(!cmp.get("v.myVal")){
            cmp.set("v.validity", false);
        }
        else{
            cmp.set("v.validity", true);
        }
    }
})
```

サポートされる HTML タグ

リッチテキストエディタでは、WYSIWYG インターフェースのみが提供されます。エディタを使用して HTML タグを編集することはできませんが、value 属性で HTML タグを設定できます。Web ページまたは別のソースからコンテンツをコピーしてエディタに貼り付けると、サポートされていないタグは削除されます。有効なツールバーボタンまたはメニューに対応する書式設定のみが保持されます。たとえば、FORMAT_TEXT カテゴリを無効にすると、[太字]、[斜体]、[下線]、[取り消し線] ボタンが使用できなくなります。さらに、FORMAT_TEXT カテゴリを無効にすると、エディタで太字、斜体、下線、取り消し線テキストを貼り付けることはできません。サポートされていないタグで囲まれていたテキストは、プレーンテキストとして保持されます。ただし、テーブル、画像、およびリンクは、対応するツールバーボタンやメニューがない場合でも、エディタでの貼り付けと value 属性を介した設定を行うことができます。

コンポーネントは、XSS の脆弱性を防止するために、value 属性に渡される HTML タグをサニタイズします。ツールバーで使用できる機能に対応する HTML タグのみがサポートされています。クライアント側コントロー

ラでサポートされていないタグを設定した場合、それらのタグが削除され、テキストコンテンツは保持されます。サポートされている HTML タグは、a、b、col、colgroup、em (i に変換)、h1、h2、h3、h4、h5、h6、i、img、li、ol、p、q、s、strike (s に変換)、strong、table、tbody、td、tfoot、th、thead、tr、u、ul、strike です。

div タグと span タグで囲まれたテキストを貼り付けると、それらのタグが p タグに変換されます。たとえば、次ようなテキストをエディタで貼り付けまたは設定するとします。

```
The sky is <span style="color:blue;font-weight:bold">blue</span>.
<div style="color:#0000FF;font-weight:bold">This is some text in a div element.</div>
```

エディタは次のマークアップを返します。

```
<p>The sky is <b>blue</b>.</p>
<p><b>This is some text in a div element.</b></p>
```

font-weight:bold 書式設定は [太字] ツールバーボタンに対応するため、保持されて b タグに変換されています。マークアップでは、色の書式設定は削除されます。

使用上の考慮事項

テーブルの作成、画像の挿入、リンクのためのツールバーボタンはありませんが、それらをプログラムで作成するか、それらの要素をコピーして貼り付けることで、エディタ内で書式設定が保持されます。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

認証

属性名	属性型	説明	必須
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
disabled	Boolean	エディタが無効かどうかを示します。この値のデフォルトは false です。	
disabledCategories	List	ツールバーから削除するボタンカテゴリのカンマ区切りのリスト。	
messageWhenBadInput	String	valid が false の場合に表示されるエラーメッセージ。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
placeholder	String	項目が空の場合に表示されるテキスト。	

属性名	属性型	説明	必須
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
valid	Boolean	エディタコンテンツが有効かどうかを示します。無効の場合、slds-has-error クラスが追加されます。この値のデフォルトは true です。	
value	String	リッチテキストエディタ内の HTML コンテンツ。	
variant	String	バリエーションによってツールバーの外観が変更されます。使用できるバリエーションは、bottom-toolbar です。	

lightning:layout

ページ上でコンテナを配置するための反応型グリッドシステムを表します。

lightning:layout は、ページ内または別のコンテナ内でコンテナを配置するための柔軟なグリッドシステムです。デフォルトのレイアウトはモバイルファーストで、異なるデバイスで動作するように簡単に設定できます。

レイアウトは、次の属性値を設定することによってカスタマイズできます。

horizontalAlign

次の値に基づいてレイアウト項目を横方向に配置します。

- center: グリッドに slds-grid--align-center クラスを追加します。この属性は、レイアウト項目を横方向に間隔なしで並べ、そのグループをコンテナの中央に配置します。
- space: グリッドに slds-grid--align-space クラスを追加します。レイアウト項目はコンテナ内で横方向に間隔を空けて配置されます。最初と最後にも間隔を空けます。
- spread: グリッドに slds-grid--align-spread クラスを追加します。レイアウト項目はコンテナ内で横方向に間隔を空けて配置されます。最初と最後はレイアウト項目になります。
- end: グリッドに slds-grid--align-end クラスを追加します。レイアウト項目はグループ化されてコンテナの右側に横方向に配置されます。

verticalAlign

次の値に基づいてレイアウト項目を縦方向に配置します。

- start: グリッドに slds-grid--vertical-align-start クラスを追加します。レイアウト項目はコンテナの上部に配置されます。
- center: グリッドに slds-grid--vertical-align-center クラスを追加します。レイアウト項目はコンテナの中央に配置されます。
- end: グリッドに slds-grid--vertical-align-end クラスを追加します。レイアウト項目はコンテナの下部に配置されます。
- stretch: グリッドに slds-grid--vertical-stretch クラスを追加します。レイアウト項目はコンテナの縦いっぱいに広げて配置されます。

pullToBoundary

次の値に基づいてレイアウト項目をレイアウトの境界線に寄せます。レイアウト項目にパディングが使用されている場合、この属性はコンテナの両側にある要素を境界線まで寄せます。layoutItem の padding に対応するサイズを選択します。たとえば、`lightning:layoutItem="horizontalSmall"` であれば、`pullToBoundary="small"` を選択します。

- small: グリッドに `slds-grid--pull-padded` クラスを追加します。
- medium: グリッドに `slds-grid--pull-padded-medium` クラスを追加します。
- large: グリッドに `slds-grid-pull-padded-large` クラスを追加します。

スタイル設定を別の方法でカスタマイズするには `class` 属性または `multipleRows` 属性を使用します。

レイアウト項目を `lightning:layout` で囲むと、シンプルなレイアウトが完成します。次に例を示します。

```
<aura:component>
    <div class="c-container">
        <lightning:layout horizontalAlign="space">
            <lightning:layoutItem flexibility="auto" padding="around-small">
                1
            </lightning:layoutItem>
            <lightning:layoutItem flexibility="auto" padding="around-small">
                2
            </lightning:layoutItem>
            <lightning:layoutItem flexibility="auto" padding="around-small">
                3
            </lightning:layoutItem>
            <lightning:layoutItem flexibility="auto" padding="around-small">
                4
            </lightning:layoutItem>
        </lightning:layout>
    </div>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	レイアウトコンポーネントのボディ。	
class	String	外部要素に適用される CSS クラス。このスタイルは、コンポーネントで出力される基本クラスに追加されます。	
horizontalAlign	String	レイアウト項目を横方向に広げる方法を決定します。配置オプションは、center、space、spread、end です。	
multipleRows	Boolean	子アイテムがレイアウトの幅を超えた場合に折り返すかどうかを決定します。true の場合は、項目を次の行に折り返します。この値のデフォルトは false です。	

属性名	属性型	説明	必須項目
pullToBoundary	String	レイアウト項目をレイアウトの境界線に寄せ、レイアウト項目のパディングサイズに対応します。使用できる値は、small、medium、large です。	
verticalAlign	String	レイアウト項目を縦方向に広げる方法を決定します。配置オプションは、start、center、end、stretch です。	

lightning:layoutItem

lightning:layout の基本要素。

lightning:layoutItem は、lightning:layout 内の基本要素です。lightning:layout 内に 1 つ以上のレイアウト項目を配置できます。lightning:layoutItem の属性を使用して、レイアウト項目のサイズを設定し、さまざまなデバイスサイズでレイアウトが設定される方法を変更できます。

レイアウトシステムはモバイルファーストです。size 属性と smallDeviceSize 属性の両方が指定されている場合、size 属性は小さい携帯電話に適用され、smallDeviceSize はスマートフォンに適用されます。サイズ設定属性は付加的で、そのサイズ以上の大きさのデバイスに適用されます。たとえば、mediumDeviceSize=10 で largeDeviceSize が設定されていない場合、mediumDeviceSize はタブレットに加えてデスクトップやさらに大きいデバイスにも適用されます。

smallDeviceSize、mediumDeviceSize、largeDeviceSize のいずれかの属性が指定されている場合は、size 属性が必要です。

次に例を示します。

```
<aura:component>
    <div>
        <lightning:layout>
            <lightning:layoutItem padding="around-small">
                <div>1</div>
            </lightning:layoutItem>
            <lightning:layoutItem padding="around-small">
                <div>2</div>
            </lightning:layoutItem>
            <lightning:layoutItem padding="around-small">
                <div>3</div>
            </lightning:layoutItem>
            <lightning:layoutItem padding="around-small">
                <div>4</div>
            </lightning:layoutItem>
        </lightning:layout>
    </div>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	外部要素に適用される CSS クラス。このスタイルは、コンポーネントで出力される基本クラスに追加されます。	
flexibility	Object	コンテナ内の追加スペースに合わせて拡大したり、スペースが小さい場合は縮小したりできるように、項目を流動的にします。有効な値は auto、shrink、no-shrink、grow、no-grow、no-flex です。	
largeDeviceSize	Integer	ビューポートが 12 個の部分に分割されている場合、この属性はデスクトップよりも大きいデバイス種別でコンテナが占める相対スペースを示します。1 ～ 12 の整数で表されます。	
mediumDeviceSize	Integer	ビューポートが 12 個の部分に分割されている場合、この属性はタブレットよりも大きいデバイス種別でコンテナが占める相対スペースを示します。1 ～ 12 の整数で表されます。	
padding	String	コンテナの右側と左側、またはコンテナのすべての辺のパディングを設定します。有効な値は、horizontal-small、horizontal-medium、horizontal-large、around-small、around-medium、around-large です。	
size	Integer	ビューポートが 12 個の部分に分割されている場合、size はコンテナが占める相対スペースを示します。1 ～ 12 の整数で表されます。この属性はすべてのデバイス種別に適用されます。	
smallDeviceSize	Integer	ビューポートが 12 個の部分に分割されている場合、この属性はモバイルよりも大きいデバイス種別でコンテナが占める相対スペースを示します。1 ～ 12 の整数で表されます。	

lightning:menuItem (ベータ)

メニューのリスト項目を表します。

lightning:menuItem は、lightning:buttonMenu ドロップダウンコンポーネント内のメニュー項目です。チェック済みやチェックなしなどの状態を保持し、アイコンを含めることができます。

class 属性を使用してスタイル設定をカスタマイズします。

このコンポーネントは、Lightning Design System のメニューからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:buttonMenu alternativeText="Toggle menu">
       <lightning:menuItem label="Menu Item 1" value="menuitem1" iconName="utility:table"
 />
    </lightning:buttonMenu>
</aura:component>
```

複数選択メニューを実装するには、checked 属性を使用します。次のクライアント側コントローラの例は、lightning:buttonMenu コンポーネントで onselect イベントを使用して選択を処理します。メニュー項目を選択すると、選択された状態がその項目に適用されます。

```
({
    handleSelect : function (cmp, event) {
        var menuItem = event.getSource();
        // Toggle check mark on the menu item
        menuItem.set("v.checked", !menuItem.get("v.checked"));
    }
})
```

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
checked	Boolean	指定されていない場合、メニュー項目にチェックを付けることはできません。trueの場合、チェックマークがメニュー項目の左に表示されます。false の場合、チェックマークは表示されませんが、チェックマーク用のスペースが表示されます。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
disabled	Boolean	true の場合、メニュー項目は実行不可能で、無効と表示されます。	

属性名	属性型	説明	必須項目
iconName	String	指定されている場合、指定された名前のアイコンがメニュー項目の右に表示されます。	
label	String	メニュー項目のテキスト。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
title	String	ツールチップテキスト。	
value	String	メニュー項目に関連付けられている値。	
onactive	Action	非推奨。このメニュー項目が有効になったときにトリガされるアクション。	

lightning:pill

ピルとは、ユーザが生成する自由形式のテキストではない、データベース内の既存の項目を表します。

lightning:pill コンポーネントは、取引先名やケース番号などの項目を表し、テキスト表示ラベルは角が丸い境界線によって囲まれています。デフォルトでは、ピルは削除ボタンと共に表示されます。ピルは、たとえばメールアドレスのリストやキーワードのリストなど、オンデマンドで追加および削除できる参照のみのテキストを表示する場合に役立ちます。

このコンポーネントは、Lightning Design System のピルからスタイル設定を継承します。

class 属性を使用して追加のスタイル設定を適用します。

次の例では、基本的なピルを作成します。

```
<aura:component>
    <lightning:pill label="Pill Label" href="/path/to/some/where" onremove="{! c.handleRemove
 }"/>
</aura:component>
```

ピルには、2つのクリック可能な要素、テキスト表示ラベルと削除ボタンがあります。いずれの要素も onclick ハンドラをトリガします。href 値を指定すると、テキスト表示ラベルをクリックすることで onclick ハンドラがトリガされ、指定されたパスに移動します。ピルトリガの削除ボタンをクリックすると、onremove ハンドラ、onclick ハンドラの順にトリガされます。これらのイベントハンドラは省略可能です。

onclick ハンドラを実行しないようにするには、onremove ハンドラ内で event.preventDefault() をコールします。

```
<aura:component>
    <lightning:pill label="hello pill" onremove="{! c.handleRemoveOnly }" onclick="{!
c.handleClick }"/>
</aura:component>
```

```
({
    handleRemoveOnly: function (cmp, event) {
        event.preventDefault();
        alert('Remove button was clicked!');
    },
    handleClick: function (cmp, event) {
        // this won't run when you click the remove button
        alert('The pill was clicked!');
    }
})
```

画像の挿入

ピルには、アイコンやアバターなど、オブジェクトの種別を表す画像を含めることができます。ピルに画像を挿入するには、media 属性を使用します。

```
<aura:component>
    <lightning:pill label="Pill Label" href="/path/to/some/where">
        <aura:set attribute="media">
            <lightning:icon iconName="standard:account"  alternativeText="Account"/>
        </aura:set>
    </lightning:pill>
</aura:component>
```

使用上の考慮事項

ピルには、コンテナが事前定義した項目のコレクションに一致しない場合(メールアドレスが無効な場合やケース番号が存在しない場合など)にエラー状態を表示できます。エラーを含むピルを示すには hasError 属性を使用します。hasError を true に設定すると、ピルに警告アイコンが挿入され、境界線が赤に変わります。このコンテキストで独自の画像を指定しても、ピルには影響が及びません。

アクセシビリティ

alternativeText 属性を使用して、ユーザのイニシャルや名前など、アバターを説明します。この説明によって、img HTML タグの alt 属性の値が指定されます。

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	

属性名	属性型	説明	必須
hasError	Boolean	ピルにエラーがあるかどうかを示します。デフォルトは false です。	
href	String	リンク先のページの URL。	
label	String	ピルに表示されるテキスト表示ラベル。	はい
media	Component[]	テキスト情報の横に表示されるアイコンまたは図。	
name	String	ピルの名前。この値は省略可能であり、コールバック内でピルを識別するために使用できます。	
onclick	Action	ボタンがクリックされたときにトリガされるアクション。	
onremove	Action	ピルが削除されたときにトリガされるアクション。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	

lightning:progressBar

操作の進行状況を左から右へ示す、横方向の進行状況バーを表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:progressBar コンポーネントは、ファイルのダウンロードやアップロードなど、操作の進行状況を左から右へ示します。

このコンポーネントは、Lightning Design System の進行状況バーからスタイル設定を継承します。

次の例では、コンポーネントの表示と再表示の進行状況バーを読み込みます。

```
<aura:component>
    <aura:handler name="render" value="{!this}" action="{!c.onRender}"/>
    <aura:attribute name="progress" type="Integer" default="0"/>
    <lightning:progressBar value="{!v.progress}"/>
</aura:component>
```

進行状況バーの値を変更するクライアント側コントローラは、次のようになります。progress === 100 ? clearInterval(interval) : progress + 10 を指定すると、進行状況値が 10% 増加し、進行状況が 100% に達するとアニメーションが停止します。進行状況バーは 200 ミリ秒ごとに更新されます。

```
({
    onRender: function (cmp) {
        var interval = setInterval($A.getCallback(function () {
            var progress = cmp.get('v.progress');
            cmp.set('v.progress', progress === 100 ? clearInterval(interval) : progress +
 10);
        }), 200);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
variant	String	進行状況バーのバリエーション。有効な値は、base および circular です。	
value	Integer	進行状況バーのパーセント値。	
size	String	進行状況バーのサイズ。有効な値は、x-small、small、medium、large です。デフォルト値は、medium です。	

lightning:progressIndicator

特定のプロセスの進行状況を視覚的に示します。このコンポーネントはバージョン 41.0 以降で使用できます。

lightning:progressIndicator コンポーネントは、プロセスのステップのリストを横方向で表示し、指定されたプロセスのステップ数、現在のステップ、および前の完了済みステップを示します。たとえば、セールスパスは進行状況インジケータを使用して、営業担当にセールスプロセスのフェーズを示します。

type 属性を指定して、異なるビジュアルスタイル設定の進行状況インジケータを作成できます。Lightning Design System の進行状況インジケータからスタイル設定を継承するコンポーネントを作成するには、type="base" を設定します。Lightning Design System のパスのスタイル設定を継承するコンポーネントを作成するには、type="path" を設定します。

タイプが指定されていない場合、デフォルトのタイプ (base) が使用されます。ステップを作成するには、1 つ以上の lightning:progressStep コンポーネントを label および value 属性と共に使用します。現在のステップを指定するには、currentStep 属性が lightning:progressStep コンポーネントのいずれかの value 属性と一致する必要があります。

```
<aura:component>
    <lightning:progressIndicator currentStep="step2">
        <lightning:progressStep label="Step One" value="step1"/>
        <lightning:progressStep label="Step Two" value="step2"/>
        <lightning:progressStep label="Step Three" value="step3"/>
    </lightning:progressIndicator>
</aura:component>
```

前の例では、ステップにマウスポインタを置いたときにツールチップで label が表示されます。進行状況インジケータのタイプが path の場合、表示ラベルはステップが完了済みの場合はフロート表示され、現在のステップまたは未完了のステップの場合はそのステップ上に表示されます。

次の例では、「Step Two」で現在のステップを表示するパスを作成します。「Step One」は完了済みとマークされ、「Step Three」は未完了です。

```
<aura:component>
    <lightning:progressIndicator type="path" currentStep="step2">
        <lightning:progressStep label="Step One" value="step1"/>
        <lightning:progressStep label="Step Two" value="step2"/>
        <lightning:progressStep label="Step Three" value="step3"/>
    </lightning:progressIndicator>
</aura:component>
```

アクセシビリティ

各進行状況ステップは、補助テキストでもそのステップの表示ラベルが表示されます。base タイプの場合、Tab キーを使用して次のステップに移動できます。Shift+Tab キーを押すと、前のステップに戻ります。path タイプの場合、Tab キーを押して現在のステップを有効にし、上下左右矢印キーを使用してステップ間を移動します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
currentStep	String	現在のステップ。いずれかの progressStep コンポーネントの value 属性と一致する必要があります。ステップが指定されていない場合、最初の progressStep コンポーネントの値が使用されます。	
hasError	Boolean	現在のステップがエラー状態かどうかを示し、ステップインジケータに警告アイコンを表示します。base タイプのみに適用されます。この値のデフォルトは false です。	
type	String	インジケータのビジュアルパターンを変更します。有効な値は、base および path です。この値のデフォルトは base です。	
variant	String	base タイプのみの進行状況インジケータの外観を変更します。有効な値は、base および shaded です。shaded バリエーションでは、ステップインジケータにライトグレーの境界線が追加されます。この値のデフォルトは base です。	

lightning:radioGroup

1 つのオプションを選択できるラジオボタングループ。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:radioGroup コンポーネントは、1 つのオプションを選択できるラジオボタングループを表します。

required 属性が true の場合、少なくとも 1 つのラジオボタンを選択する必要があります。ユーザがラジオグループで選択を行っていない場合、エラーメッセージが表示されます。

disabled 属性が true の場合、ラジオボタンの選択は変更できません。

このコンポーネントは、Lightning Design System のラジオボタンからスタイル設定を継承します。Lightning Design System のラジオボタングループからスタイル設定を継承するコンポーネントを作成するには、`type="button"` を設定します。

次の例では、2 つのオプションがあり、デフォルトで `option1` が選択されているラジオグループを作成します。required 属性が true になっているため、1 つのラジオボタンを選択する必要があります。

```
<aura:component>
    <aura:attribute name="options" type="List" default="[
    {'label': 'apples', 'value': 'option1'},
    {'label': 'oranges', 'value': 'option2'}
    ]"/>
    <aura:attribute name="value" type="String" default="option1"/>
    <lightning:radioGroup
        aura:id="mygroup"
        name="radioButtonGroup"
        label="Radio Button Group"
        options="{! v.options }"
        value="{! v.value }"
        onchange="{! c.handleChange }"
        required="true" />
</aura:component>
```

`cmp.find("mygroup").get("v.value")` を使用して、どの値が選択されているのかを確認できます。選択が変更されたときにその値を取得するには、`onchange` イベントハンドラを使用して `event.getParam("value")` をコールします。

```
({
    handleChange: function (cmp, event) {
        var changeValue = event.getParam("value");
        alert(changeValue);
    }
});
```

アクセシビリティ

ラジオグループは、`legend` 要素を含む `fieldset` 要素内にネストされます。legend には `label` 値が含まれます。`fieldset` 要素では、関連するラジオボタンをグループ化して、タブナビゲーションと音声ナビゲーションを容易にし、アクセシビリティに役立てることができます。同様に、`legend` 要素では、キャプションを `fieldset` に割り当てられるようにしてアクセシビリティを改善します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

`checkValidity()`: ラジオグループに有効性エラーがあるかどうかを示す、ValidityState の有効なプロパティ値 (Boolean) を返します。

`showHelpMessageIfInvalid()`: フォームコントロールが無効な状態の場合、ヘルプメッセージを表示します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
name	String	入力要素の名前を指定します。	はい
value	Object	入力要素の値を指定します。	
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトはfalseです。	
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
onchange	Aura.Action	値属性が変更された場合にトリガされるアクション。	
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
onfocus	Aura.Action	要素にフォーカスが設定されたときにトリガされるアクション。	
onblur	Aura.Action	要素がフォーカスを解放したときにトリガされるアクション。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
label	String	ラジオグループのテキスト表示ラベル。	はい

属性名	属性型	説明	必須項目
options	List	各ラジオボタンの表示ラベル-値ペアの配列。	はい
type	String	ラジオグループのスタイル。オプションは、radio または button です。デフォルトは、radio です。	
messageWhenValueMissing	String	ラジオボタンが選択されておらず、required 属性が true に設定されている場合、メッセージが表示されます(省略可能)。	

lightning:relativeDateTime

ソースの日時と指定された日時の相対的な時差を表示します。

タイムスタンプまたは JavaScript の Date オブジェクトを指定すると、`lightning:relativeDateTime` には、現在の時刻と指定された時刻の相対時間を表す文字列が表示されます。

使用される単位は、指定された時刻からどれだけの時間が過ぎたかに対応します。たとえば、「数秒前」または「5 分前」などです。将来の時刻を指定すると、「7 か月後」または「5 年後」などの相対時間が返されます。

この例では、現在の時刻と過去および将来の指定時刻との相対時間を返します。時差は `init` ハンドラによって設定されます。

```
<aura:component>
    <aura:handler name="init" value="{! this }" action="{! c.init }" />
    <aura:attribute name="past" type="Object"/>
    <aura:attribute name="future" type="Object"/>
    <p><lightning:relativeDateTime value="{! v.past }"/></p>
    <p><lightning:relativeDateTime value="{! v.future }"/></p>
</aura:component>
```

クライアント側コントローラは、コンポーネントの初期化中にコールされます。`past` 属性と `future` 属性は、次のような値を返します。

- 2 時間前
- 2 日後

```
({
    init: function (cmp) {
        cmp.set('v.past', Date.now()-(2\*60\*60\*1000));
        cmp.set('v.future', Date.now()+(2\*24\*60\*60\*1000));
    }
})
```

出力例には他に次のものがあります。

- 相対的な過去: 数秒前、1 分前、2 分前、1 時間前、2 時間前、2 日前、2 か月前、2 年前
- 相対的な将来: 数秒後、1 分後、2 分後、1 時間後、2 時間後、2 日後、2 か月後、2 年後

時間の単位はユーザのロケール (en-US などの言語コードを返す) を使用してローカライズされます。次の時間の単位がサポートされています。

- seconds
- minutes
- hours
- days
- months
- years

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
value	Object	書式設定の対象となるタイムスタンプまたは JavaScript の Date オブジェクト。	はい

lightning:select

選択入力を表します。

lightning:select コンポーネントは HTML select 要素を作成します。このコンポーネントは、HTML option 要素を使用してドロップダウンリストのオプションを作成し、リストから 1 つのオプションを選択できるようにします。複数選択は現在サポートされていません。

このコンポーネントは、Lightning Design System の選択からスタイル設定を継承します。

ドロップダウンリストでのさまざまな入力イベントを処理するクライアント側コントローラアクションを定義できます。たとえば、コンポーネントでの変更イベントを処理するには、onchange 属性を使用します。cmp.find("selectItem").get("v.value") を使用して、選択した値を取得します。

```
<aura:component>
    <lightning:select name="selectItem" label="Select an item" onchange="{!c.doSomething}">

        <option value="">choose one...</option>
        <option value="1">one</option>
        <option value="2">two</option>
     </lightning:select>
</aura:component>
```

aura:iteration によるオプションの生成

オプションを生成するには、aura:iteration を使用して項目のリストを反復処理します。この例では、項目のリストを反復処理します。

```
<aura:component>
    <aura:attribute name="colors" type="String[]" default="Red,Green,Blue"/>
    <lightning:select name="select" label="Select a Color" required="true"
messageWhenValueMissing="Did you forget to select a color?">
        <option value="">-- None --</option>
```

```
        <aura:iteration items="{!v.colors}" var="color">
            <option value="{!color}" text="{!color}"></option>
        </aura:iteration>
    </lightning:select>
</aura:component>
```

初期化時のオプションの生成

属性を使用して、コンポーネントでオプション値の配列を保存および設定します。次のコンポーネントは、コンポーネントの初期化時にクライアント側コントローラをコールして、オプションを作成します。

```
<aura:component>
    <aura:attribute name="options" type="List" />
    <aura:attribute name="selectedValue" type="String" default="Red"/>
    <aura:handler name="init" value="{!this}" action="{!c.loadOptions}" />
    <lightning:select name="mySelect" label="Select a color:" aura:id="mySelect"
value="{!v.selectedValue}">
        <aura:iteration items="{!v.options}" var="item">
          <option text="{!item.label}" value="{!item.value}" selected="{!item.selected}"/>

         </aura:iteration>
    </lightning:select>
    </aura:component>
```

クライアント側コントローラで、オプションの配列を定義し、この配列を `items` 属性に割り当てます。

```
({
    loadOptions: function (component, event, helper) {
        var opts = [
            { value: "Red", label: "Red" },
            { value: "Green", label: "Green" },
            { value: "Blue", label: "Blue" }
         ];
         component.set("v.options", opts);
    }
})
```

コンポーネントで新しいオプションの配列を指定しても、競合状況が発生して、選択した新しい値がコンポーネントの値に反映されないことがあります。たとえば、新しいオプションを選択した後で `component.find("mySelect").get("v.value")` を実行しても、オプションの表示が完了する前の値を取得するため、以前に選択した値がコンポーネントから返されます。この競合状況を回避するには、前の例のように `lightning:select` コンポーネントの `value` 属性と `selected` 属性をバインドします。また、新しいオプション値の `selected` 属性をバインドし、次の例のようにコンポーネントで選択した値を明示的に設定します。これにより、コンポーネントの値が選択した新しいオプションに対応します。

```
updateSelect: function(component, event, helper){
    var opts = [
        { value: "Cyan", label: "Cyan" },
        { value: "Yellow", label: "Yellow" },
        { value: "Magenta", label: "Magenta", selected: true }];
    component.set('v.options', opts);
    //set the new selected value on the component
    component.set('v.selectedValue', 'Magenta');
    //return the selected value
```

```
    component.find("mySelect").get("v.value");
}
```

入力規則

このコンポーネントでは、クライアント側で入力規則が使用できます。ドロップダウンメニューを必須項目にするには、`required="true"` を設定します。`required="true"` で項目が選択されていない場合、エラーメッセージが自動的に表示されます。

入力の有効性状態を確認するには、`ValidityState` オブジェクトに基づく `validity` 属性を使用します。有効性状態にはクライアント側コントローラでアクセスできます。この `validity` 属性は、`boolean` プロパティが設定されたオブジェクトを返します。詳細は、`lightning:input`を参照してください。

`messageWhenValueMissing` に独自の値を指定し、デフォルトのメッセージを上書きできます。

使用上の考慮事項

`onchange` イベントは、ユーザがマウスをクリックしてドロップダウンリストの値を選択した場合にのみトリガされます。HTML `select` 要素の予期される動作は、マウスのクリックです。`value` 属性をプログラムで変更した場合、その変更が選択要素に伝達されたとしても、このイベントはトリガされません。このイベントを処理するには、`value` の変更ハンドラを指定します。

```
<aura:handler name="change" value="{!v.value}" action="{!c.handleChange}"/>
```

次の例では、クリックしたときに選択されたオプションが変更されるドロップダウンリストとボタンを作成します。

```
<aura:component>
    <aura:attribute name="status" type="String" default="open"/>
    <aura:handler name="change" value="{!v.status}" action="{!c.handleChange}"/>
    <lightning:select aura:id="select" name="select" label="Opportunity Status"
value="{!v.status}">
        <option value="">choose one...</option>
        <option value="open">Open</option>
        <option value="closed">Closed</option>
        <option value="closedwon">Closed Won</option>
    </lightning:select>
    <lightning:button name="selectChange" label="Change" onclick="{!c.changeSelect}"/>
</aura:component>
```

クライアント側コントローラは、`v.status` 値を変更して変更ハンドラをトリガし、選択されたオプションを更新します。

```
({
    changeSelect: function (cmp, event, helper) {
        //Press button to change the selected option
        cmp.find("select").set("v.value", "closed");
    },
    handleChange: function (cmp, event, helper) {
        //Do something with the change handler
        alert(event.getParam('value'));
    }
})
```

アクセシビリティ

アクセシビリティのためのテキスト表示ラベルを指定して、情報が支援技術で使用できるようにする必要があります。label 属性は、入力コンポーネントの HTML label 要素を作成します。表示ラベルをビューに表示させず、支援技術には使用できるようにするには、`label-hidden` バリエーションを使用します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

`showHelpMessageIfInvalid()`: フォームコントロールが無効な状態の場合、ヘルプメッセージを表示します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	外部要素に適用される CSS クラス。このスタイルは、コンポーネントに関連付けられている基本クラスに追加されます。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
label	String	目的の選択入力を説明するテキスト。	はい
messageWhenValueMissing	String	値がない場合に表示されるエラーメッセージ。	
name	String	入力要素の名前を指定します。	はい
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onchange	Action	値属性が変更された場合にトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトは false です。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	

属性名	属性型	説明	必須項目
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
value	String	選択の値。init 中に適切なオプションを選択するデフォルト値としても使用されます。値が指定されていない場合、最初のオプションが選択されます。	
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	

lightning:slider

指定した2つの数字間の値を指定するための入力範囲スライダ。このコンポーネントでは、APIバージョン41.0以降が必要です。

lightning:slider コンポーネントは、指定した 2 つの数字間の値を指定する横方向または縦方向のスライダです。たとえば、このスライダを使用して、注文数量についてのユーザ入力を取得できます。また、type="range" の入力項目を使用する場合にこのスライダを使用できます。スライダの向きを垂直方向にするには、type="vertical" を設定します。スライダをサポートしない古いブラウザではこの設定が無効になり、スライダは type="text" として処理されます。

このコンポーネントは、Lightning Design System のスライダからスタイル設定を継承します。

次に、段階的な増分を 10 に設定したスライダの例を示します。

```
<aura:component>
    <aura:attribute name="myval" default="10" type="Integer"/>
    <lightning:slider step="10" value="{!v.myval}" onchange="{! c.handleRangeChange }"/>
</aura:component>
```

クライアント側のコントローラで値の変更を処理し、値を最新の値で更新します。

```
({
    handleRangeChange: function (cmp, event) {
        var detail = cmp.set("v.value", event.getParam("value"));
    }
})
```

入力規則

入力の有効性状態を確認するには、checkValidity() メソッドを使用します。これは、ValidityState オブジェクトの valid プロパティ値が true の場合に true を返します。また、setCustomValidity() を使用してカスタムエラーメッセージを提供することもできます (例: setCustomValidity(this.messageWhenRangeUnderflow))。

基盤となる input 要素が type="range" の場合、以下の条件で入力値がサニタイズされます。このいずれかの条件を満たす場合、スライダは無効になり、エラーメッセージが表示され、value 属性の正しい値の入力が要求されます。

- value を min 値より小さい値に設定した場合、スライダは入力値を min 値に設定します。
- value を max 値より大きい値に設定した場合、スライダは入力値を max 値に設定します。
- value が step 値の倍数でない場合、スライダは入力値を最も近い倍数に設定します。たとえば、value を 18、step を 5、min を 10、max を 50 に設定した場合、スライダは入力値を 20 に設定します。
- 誤って min 値と max 値を逆に設定した場合、スライダは値を修正しませんが、入力値を min 値に設定します。たとえば、value を 18、min を 50、max を 10 に設定した場合、スライダは入力値を 50 に設定します。

使用上の考慮事項

デフォルトでは、min 値と max 値は 0 と 100 ですが、独自の値を指定できます。また、独自の段階的な増分値を指定した場合、段階的な増分のみに基づいてスライダをドラッグできます。min 値より低い値を設定した場合、値は min 値に設定されます。同様に、max 値より高い値を設定すると、値は max 値に設定されます。精密な数値の場合は、代わりに type="number" の lightning:input コンポーネントを使用することをお勧めします。

メソッド

このコンポーネントは、次のメソッドをサポートします。

blur(): 入力要素に設定されたキーボードフォーカスを削除します。

checkValidity(): 入力項目値に有効性エラーがあるかどうかを示す、ValidityState オブジェクトの valid プロパティ値 (Boolean) を返します。

focus(): フォーカスを入力要素に設定します。

setCustomValidity(message): 条件を満たした場合にカスタムエラーメッセージを表示するように設定します。

- message (String): エラーを説明する文字列。メッセージが空の文字列の場合、エラーメッセージはリセットされます。

showHelpMessageIfInvalid(): スライドバーにエラーメッセージを表示します。1 つ以上の制約検証に失敗した場合、スライダ値は無効になり、checkValidity() がコールされると false が返されます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
value	Integer	入力範囲の数値。この値のデフォルトは 0 です。	

属性名	属性型	説明	必須項目
onchange	String	スライダ値が変更された場合にトリガされるアクション。新しく選択した値をスライダコンポーネントに戻して、新しい値をスライダにバインドする必要があります。	
min	Integer	入力範囲の最小値。この値のデフォルトは 0 です。	
max	Integer	入力範囲の最大値。この値のデフォルトは 100 です。	
step	String	入力範囲の段階的な増分値。たとえば、段階的な増分値として0.1、1、10があります。この値のデフォルトは1です。	
size	String	入力範囲のサイズ値。この値のデフォルトは空であり、これがベースです。x-small、small、medium、large をサポートします。	
type	String	入力範囲位置の種別。この値のデフォルトは horizontal です。	
label	String	入力範囲のテキスト表示ラベル。スライダを説明する独自の表示ラベルを指定します。指定しない場合、表示ラベルは表示されません。	
disabled	Boolean	無効化された入力範囲の値。この値のデフォルトはfalseです。	
variant	String	バリエーションによってスライダの外観が変更されます。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	
messageWhenBadInput	String	不正な入力が検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenPatternMismatch	String	パターンの不一致が検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenTypeMismatch	String	種別の不一致が検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenValueMissing	String	値がない場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenRangeOverflow	String	範囲を超えたことが検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenRangeUnderflow	String	範囲を下回ったことが検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
messageWhenStepMismatch	String	ステップの不一致が検出された場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	

属性名	属性型	説明	必須項目
messageWhenTooLong	String	値が長すぎる場合に表示されるエラーメッセージ。setCustomValidity と共に使用します。	
onblur	Aura.Action	スライダがフォーカスを解放したときにトリガされるアクション。	
onfocus	Aura.Action	スライダにフォーカスが設定されたときにトリガされるアクション。	

lightning:spinner

アニメーションスピナーを表示します。

lightning:spinner は、機能が読み込み中であることを示すアニメーションスピナー画像を表示します。このコンポーネントは、データを取得しているとき、または操作がすぐに完了しないときに使用できます。

variant 属性は、スピナーの外観を変更します。variant="brand" に設定した場合、スピナーは Lightning Design System ブランドの色と同じになります。variant="inverse" に設定すると、白いスピナーが表示されます。デフォルトのスピナーの色はダークブルーです。

このコンポーネントは、Lightning Design System のスピナーからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:spinner variant="brand" size="large"/>
</aura:component>
```

lightning:spinner は、条件付きで使用します。aura:if または Lightning Design System ユーティリティクラスを使用して、スピナーを表示または非表示にすることができます。

```
<aura:component>
    <lightning:button label="Toggle" variant="brand" onclick="{!c.toggle}"/>
    <div class="exampleHolder">
        <lightning:spinner aura:id="mySpinner" />
    </div>
</aura:component>
```

次のクライアント側コントローラは、スピナーの slds-hide クラスを切り替えます。

```
({
    toggle: function (cmp, event) {
        var spinner = cmp.find("mySpinner");
        $A.util.toggleClass(spinner, "slds-hide");
    }
})
```

属性

属性名	属性型	説明	必須項目
alternativeText	String	待機する理由とスピナーの必要性の説明に使用する代替テキスト。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
size	String	スピナーのサイズ。有効なサイズは、small、medium、large です。この値のデフォルトは medium です。	
variant	String	variant は、スピナーの外観を変更します。variant の有効な値は、brand および inverse です。	

lightning:tab (ベータ)

lightning:tabset コンポーネントにネストされる 1 つのタブ。

lightning:tab は、関連コンテンツを 1 つのコンテナに保持します。ユーザがタブをクリックすると、タブのコンテンツが表示されます。lightning:tab は、lightning:tabset と併用します。

このコンポーネントは、Lightning Design System のタブからスタイル設定を継承します。

label 属性には、テキストまたは複雑なマークアップを含めることができます。次の例では、lightning:icon を含む label の指定に aura:set が使用されています。

```
<aura:component>
    <lightning:tabset>
        <lightning:tab>
            <aura:set attribute="label">
                Item One
                <lightning:icon iconName="utility:connected_apps" />
            </aura:set>
        </lightning:tab>
    </lightning:tabset>
</aura:component>
```

使用上の考慮事項

このコンポーネントでは、実行中に本文が作成されます。初期化中にはコンポーネントを参照できません。代わりに、コンポーネント属性とバインドする値を使用して、コンテンツを設定できます。詳細は、lightning:tabsetを参照してください。

メソッド

このコンポーネントは、次のメソッドをサポートします。

`focus()`: フォーカスを要素に設定します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	
body	ComponentDefRef[]	tab のボディ。	
id	String	ID (省略可能) は、タブセットの onSelect イベント時に、どのタブがクリックされたかを判断するために使用されます。	
label	Component[]	タブに表示されるテキスト。	
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
title	String	タブにマウスポインタを置くと、タイトルが表示されます。スクリーンリーダーのタブのコンテンツを説明するタイトルを指定します。	
onactive	Action	このタブが有効になったときにトリガされるアクション。	

`lightning:tabset` (ベータ)

タブのリストを表します。

`lightning:tabset` は、複数のコンテンツ領域があり、一度に 1 つのみが表示されるタブ付きコンテナを表示します。タブは横一列で表示され、その下にコンテンツが表示されます。tabset は、そのボディの一部として複数の lightning:tab コンポーネントを保持できます。最初のタブがデフォルトで有効になりますが、対象タブで `selectedTabId` 属性を設定してデフォルトのタブを変更できます。

タブセットの外観を変更するには、`variant` 属性を使用します。`variant` 属性は、default、scoped、または vertical に設定できます。variant が default の場合、有効なタブに下線が表示されます。タブセットのスタイル設定が scoped の場合、コンテナが閉じた状態で表示され、有効なタブの境界線が強調表示されます。垂直方向 (vertical) のタブセットでは、水平方向ではなく垂直方向にタブを表示する、スコープ設定された (scoped) タブセットが表示されます。

このコンポーネントは、Lightning Design System のタブからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:tabset>
        <lightning:tab label="Item One">
            Sample Content One
        </lightning:tab>
        <lightning:tab label="Item Two">
            Sample Content Two
        </lightning:tab>
    </lightning:tabset>
</aura:component>
```

タブにコンテンツを遅延読み込みする場合は、`onactive` 属性を使用して、タブのボディをプログラムで挿入します。次に、有効な場合にコンテンツが読み込まれる 2 つのタブの例を示します。

```
<lightning:tabset variant="scoped">
    <lightning:tab onactive="{! c.handleActive }" label="Accounts" id="accounts" />
    <lightning:tab onactive="{! c.handleActive }" label="Cases" id="cases" />
</lightning:tabset>
```

クライアント側ヘルパーで、`$A.createComponent()` を使用して、コンテンツを追加する前に選択されたタブを渡します。

```
({
    handleActive: function (cmp, event) {
        var tab = event.getSource();
        switch (tab.get('v.id')) {
            case 'accounts' :
                this.injectComponent('c:myAccountComponent', tab);
                break;
            case 'cases' :
                this.injectComponent('c:myCaseComponent', tab);
                break;
        }
    },
    injectComponent: function (name, target) {
        $A.createComponent(name, {
        }, function (contentComponent, status, error) {
            if (status === "SUCCESS") {
                target.set('v.body', contentComponent);
            } else {
                throw new Error(error);
            }
        });
    }
})
```

使用上の考慮事項

読み込んだタブ数が多くてビューポートの幅に収まらない場合は、タブセットがオーバーフローしたタブのナビゲーションボタンを設定します。

このコンポーネントでは、実行中に本文が作成されます。初期化中にはコンポーネントを参照できません。代わりに、コンポーネント属性とバインドする値を使用して、コンテンツを設定できます。

たとえば、init ハンドラを使用して初期化時にオプションのリストを動的に読み込む方法で、タブセットに lightning:select コンポーネントを作成することはできません。他方、コンポーネント属性を値にバインドする方法で、オプションのリストを作成することはできます。デフォルトで、オプションの value 属性には、渡されたオプションと同じ値が指定されます (ただし、明示的に値を割り当てる場合を除く)。

```
<aura:component>
    <aura:attribute name="opts" type="List" default="['red', 'blue', 'green']" />
    <lightning:tabset>
        <lightning:tab label="View Options">
           <lightning:select name="colors" label="Select a color:">
                    <aura:iteration items="{!v.opts}" var="option">
                        <option>{! option }</option>
                    </aura:iteration>
            </lightning:select>
        </lightning:tab>
    </lightning:tabset>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	ComponentDefRef[]	コンポーネントのボディ。1 つ以上の lightning:tab コンポーネントを含めることができます。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
onselect	Action	タブがクリックされたときに実行されるアクション。	
selectedTabId	String	デフォルトで開く特定のタブを設定できます。この属性が使用されていない場合、デフォルトで最初のタブが開きます。	
variant	String	variant は、タブセットの外観を変更します。variant の有効な値は、default および scoped です。	

lightning:textarea

複数のテキスト入力を表します。

lightning:textarea コンポーネントは、複数行のテキストを入力するための HTML textarea 要素を作成します。テキストエリアは、無制限の文字数を保持できます。

このコンポーネントは、Lightning Design System のテキストエリアからスタイル設定を継承します。

rows および cols HTML 属性はサポートされていません。テキストエリアにカスタムの高さと幅を適用するには、class 属性を使用します。テキストエリアの入力を設定するには、value 属性を使用してその値を設定します。この値を設定すると、指定された最初の値は上書きされます。

次の例は、最大文字数が 300 文字のテキストエリアを作成します。

```
<lightning:textarea name="myTextArea" value="initial value"
    label="What are you thinking about?" maxlength="300" />
```

onblur、onfocus、onchange などの入力イベントを処理するクライアント側コントローラアクションを定義できます。たとえば、コンポーネントでの変更イベントを処理するには、onchange 属性を使用します。

```
<lightning:textarea name="myTextArea" value="initial value"
    label="What are you thinking about?" onchange="{!c.countLength}" />
```

入力規則

このコンポーネントでは、クライアント側で入力規則が使用できます。maxlength 属性を使用して最大文字数を設定するか、minlength 属性を使用して最小文字数を設定します。テキストエリアを必須項目にするには、required="true" を選択します。次の場合、エラーメッセージが自動的に表示されます。

- required が true に設定されていて必須項目が空の場合。
- 入力値が minlength 属性で指定された文字数未満の場合。
- 入力値が maxlength 属性で指定された文字数を超えている場合。

入力の有効性状態を確認するには、ValidityState オブジェクトに基づく validity 属性を使用します。有効性状態にはクライアント側コントローラでアクセスできます。この validity 属性は、boolean プロパティが設定されたオブジェクトを返します。詳細は、lightning:inputを参照してください。

messageWhenValueMissing、messageWhenBadInput、または messageWhenTooLong に独自の値を指定し、デフォルトのメッセージを上書きできます。

次に例を示します。

```
<lightning:textarea name="myText" required="true" label="Your Name"
        messageWhenValueMissing="This field is required."/>
```

アクセシビリティ

アクセシビリティのためのテキスト表示ラベルを指定して、情報が支援技術で使用できるようにする必要があります。label 属性は、入力コンポーネントの HTML label 要素を作成します。表示ラベルをビューに表示させず、支援技術には使用できるようにするには、label-hidden バリエーションを使用します。

メソッド

このコンポーネントは、次のメソッドをサポートします。

focus(): フォーカスを要素に設定します。

showHelpMessageIfInvalid(): フォームコントロールが無効な状態の場合、ヘルプメッセージを表示します。

属性

属性名	属性型	説明	必須項目
accesskey	String	要素の有効化またはフォーカス設定のためのショートカットキーを指定します。	

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	外部要素に適用される CSS クラス。このスタイルは、コンポーネントに関連付けられている基本クラスに追加されます。	
disabled	Boolean	入力要素を無効にするかどうかを指定します。この値のデフォルトは false です。	
label	String	目的のテキストエリア入力を説明するテキスト。	はい
maxlength	Integer	テキストエリアに入力できる最大文字数。	
messageWhenBadInput	String	不正な入力が検出された場合に表示されるエラーメッセージ。	
messageWhenTooLong	String	値が長すぎる場合に表示されるエラーメッセージ。	
messageWhenValueMissing	String	値がない場合に表示されるエラーメッセージ。	
minlength	Integer	テキストエリアに入力できる最小文字数。	
name	String	入力要素の名前を指定します。	はい
onblur	Action	要素がフォーカスを解放したときにトリガされるアクション。	
onchange	Action	値属性が変更された場合にトリガされるアクション。	
onfocus	Action	要素にフォーカスが設定されたときにトリガされるアクション。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
readonly	Boolean	入力項目が参照のみであることを指定します。この値のデフォルトは false です。	
required	Boolean	フォームを送信する前に入力項目が入力されている必要があることを指定します。この値のデフォルトはfalseです。	
tabindex	Integer	要素のタブ順序を指定します (タブボタンがナビゲーションに使用される場合)。	
validity	Object	制約検証に対して要素が取ることのできる有効性状態を表します。	
value	String	テキストエリアの値。init 中のデフォルト値としても使用されます。	

属性名	属性型	説明	必須項目
variant	String	バリエーションは入力項目の外観を変更します。使用できるバリエーションは standard と label-hidden です。この値のデフォルトは standard です。	

lightning:tile

レコードに関連付けられた関連情報のグループ。

lightning:tile コンポーネントは、レコードに関連付けられた関連情報をグループ化します。この情報はアクション可能にでき、lightning:icon や lightning:avatar コンポーネントなどの図とペアにできます。

class 属性を使用してスタイル設定をカスタマイズします。たとえば、slds-tile--board クラスを指定して board バリエーションを作成します。タイルのボディのスタイルを設定するには、Lightning Design System ヘルパークラスを使用します。

このコンポーネントは、Lightning Design System のタイルからスタイル設定を継承します。

次に例を示します。

```
<aura:component>
    <lightning:tile label="Lightning component team" href="/path/to/somewhere">
        <p class="slds-truncate" title="7 Members">7 Members</p>
    </lightning:tile>
</aura:component>
```

アイコンまたはアバターを挿入するには、これを media 属性に渡します。定義リストを使用して、アイコンを表示するタイルを作成できます。次の例では、slds-dl--horizontal などのヘルパークラスを使用して、アイコンと一部のテキストを揃えます。

```
<aura:component>
    <lightning:tile label="Salesforce UX" href="/path/to/somewhere">
        <aura:set attribute="media">
            <lightning:icon iconName="standard:groups"/>
        </aura:set>
        <dl class="slds-dl--horizontal">
            <dt class="slds-dl--horizontal__label">
                <p class="slds-truncate" title="Company">Company:</p>
            </dt>
            <dd class="slds-dl--horizontal__detail slds-tile__meta">
                <p class="slds-truncate" title="Salesforce">Salesforce</p>
            </dd>
            <dt class="slds-dl--horizontal__label">
                <p class="slds-truncate" title="Email">Email:</p>
            </dt>
            <dd class="slds-dl--horizontal__detail slds-tile__meta">
                <p class="slds-truncate"
title="salesforce-ux@salesforce.com">salesforce-ux@salesforce.com</p>
            </dd>
        </dl>
```

```
    </lightning:tile>
</aura:component>
```

また、次の例のように、順序なしリストを使用して、アバターを表示するタイルのリストを作成することもできます。

```
<aura:component>
    <ul class="slds-has-dividers--bottom-space">
        <li class="slds-item">
            <lightning:tile  label="Astro" href="/path/to/somewhere">
                <aura:set attribute="media">
                    <lightning:avatar src="/path/to/img" alternativeText="Astro"/>
                </aura:set>
                <ul class="slds-list--horizontal slds-has-dividers--right">
                    <li class="slds-item">Trailblazer, Salesforce</li>
                    <li class="slds-item">Trailhead Explorer</li>
                </ul>
            </lightning:tile>
        </li>
        <!-- More list items here -->
    </ul>
</aura:component>
```

認証

属性名	属性型	説明	必須
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	外部要素に適用される CSS クラス。このスタイルは、コンポーネントに関連付けられている基本クラスに追加されます。	
href	String	リンク先のページの URL。	
label	String	タイルやフロート表示テキストに表示されるテキスト表示ラベル。	はい
media	Component[]	テキスト情報の横に表示されるアイコンまたは図。	

lightning:tree

ネストされたツリーを表示します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:tree コンポーネントは、Web サイトのサイトマップや組織のロール階層など、構造階層を視覚的に表示します。項目はハイパーリンクとして表示され、階層内の項目はネストすることができます。ネストした項目を含む項目は、ブランチとも呼ばれます。

このコンポーネントは、Lightning Design System のツリーからスタイル設定を継承します。

ツリーを作成するには、キー - 値のペアの配列を items 属性に渡します。キーは次のとおりです。

- disabled (Boolean): ブランチを無効化するかどうかを指定します。無効化されたブランチを展開することはできません。デフォルトは false です。
- expanded (Boolean): ブランチを展開するかどうかを指定します。展開されたブランチには、ネストされた項目が視覚的に表示されます。デフォルトは false です。
- href (String): リンクの URL。
- name (String): クリックされたツリー項目を返す、`onselect` イベントハンドラの項目の一意の名前。
- items (Object): キー - 値のペアの配列としてネストされた項目。
- label (String): 必須。ハイパーリンクのタイトルと表示ラベル。

次に、複数レベルのネストを持つツリーの例を示します。

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />
    <aura:attribute name="items" type="Object"/>
    <lightning:tree items="{! v.items }" header="Roles"/>
</aura:component>
```

ツリーはコンポーネントの初期化中に作成されます。

```
({
    doInit: function (cmp, event, helper) {
    var items = [{
            "label": "Western Sales Director",
            "name": "1",
            "expanded": true,
            "items": [{
                "label": "Western Sales Manager",
                "name": "2",
                "expanded": true,
                "items" :[{
                    "label": "CA Sales Rep",
                    "name": "3",
                    "expanded": true,
                    "items" : []
                },{
                    "label": "OR Sales Rep",
                    "name": "4",
                    "expanded": true,
                    "items" : []
                }]
            }]
        }, {
            "label": "Eastern Sales Director",
            "name": "5",
            "expanded": false,
            "items": [{
                "label": "Easter Sales Manager",
                "name": "6",
                "expanded": true,
                "items" :[{
                    "label": "NY Sales Rep",
                    "name": "7",
                    "expanded": true,
```

```
                    "items" : []
                }, {
                    "label": "MA Sales Rep",
                    "name": "8",
                    "expanded": true,
                    "items" : []
                }]
            }]
        }];
        cmp.set('v.items', items);
    }
})
```

選択された項目の ID を取得するには、`onselect` 属性を使用してイベントハンドラにバインドします (次の例の `handleSelect()` を参照)。`href` 値を持つ項目を選択した場合、`select` イベントも起動されます。

```
({
    handleSelect: function (cmp, event, helper) {
        //return name of selected tree item
        var myName = event.getParam('name');
        alert("You selected: " + myName);
    }
})
```

ツリーの項目を追加または削除できます。次のようなツリーがあるとします。

```
var items = [{
            label: "Go to Record 1",
            href: "#record1",
            items: []
        },{
            label: "Go to Record 2",
            href: "#record2",
            items: []
        }, {
            label: "Go to Record 3",
            href: "#record3",
            items: []
        }];
```

この例では、ネストされた項目がツリーの最後に追加されます。

```
({
    addItem: function (cmp, event, helper) {
        var items = cmp.get('v.items');
        var branch = items.length - 1;
        var label = 'New item added at ' + branch;
        var newItem = {
            label: label,
            expanded: true,
            disabled: false,
            items: []
        };
        items[branch].items.push(newItem);
        cmp.set('v.items', items);
```

```
        alert("Added new item at branch: " + branch);
    }
})
```

項目に href 値を提供した場合、ハイパーリンクに移動する前に onselect イベントハンドラがトリガされます。

アクセシビリティ

キーボードを使用してツリーを操作できます。Tab キーを使用してツリーに移動し、上矢印または下矢印キーを使用してツリー項目にフォーカスを設定します。展開されたブランチを折りたたむには、左矢印キーを押します。ブランチを展開するには、右矢印キーを押します。Enterキーまたはスペースバーを押すと、onclickイベントと同様に、項目のデフォルトアクションが実行されます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
items	Object	ツリーを説明する、キー-値のペアの配列。キーには label、name、disabled、expanded、および items が含まれます。	
header	String	ツリーのヘッダーとして表示されるテキスト。	
onselect	Aura.Action	ツリー項目が選択されたときにトリガされるアクション。	

lightning:verticalNavigation

縦方向のリンクのリストを表します。このリンクにより、ユーザは別のページまたは現在のページの別の部分に移動できます。このコンポーネントでは、API バージョン 41.0 以降が必要です。

lightning:verticalNavigation コンポーネントは、1 レベルのみの深さを持つリンクのリストを表し、折りたたまれるおよび展開されるオーバーフローセクションをサポートします。オーバーフローセクションは、lightning:verticalNavigationOverflow を使用して作成される必要があり、ビューポートに基づいて自動的に調整されることはありません。

このコンポーネントは、Lightning Design System の垂直ナビゲーションからスタイル設定を継承します。

lightning:verticalNavigation は、次のサブコンポーネントと共に使用されます。

- lightning:verticalNavigationSection
- lightning:verticalNavigationItem

- `lightning:verticalNavigationOverflow`
- `lightning:verticalNavigationItemBadge`
- `lightning:verticalNavigationItemIcon`

この例では、基本的な垂直ナビゲーションメニューを作成します。

```
<aura:component>
    <lightning:verticalNavigation>
            <lightning:verticalNavigationSection label="Reports">
                <lightning:verticalNavigationItem label="Recent" name="recent" />
                <lightning:verticalNavigationItem label="Created by Me" name="created" />

                <lightning:verticalNavigationItem label="Private Reports" name="private"
/>
                <lightning:verticalNavigationItem label="Public Reports" name="public" />

                <lightning:verticalNavigationItem label="All Reports" name="all" />
            </lightning:verticalNavigationSection>
        </lightning:verticalNavigation>
</aura:component>
```

有効なナビゲーション項目を定義するには、`lightning:verticalNavigation` の `selectedItem="itemName"` を使用します。ここで、`itemName` は、強調表示する `lightning:verticalNavigationItem` コンポーネントの `name` と一致します。

この例では、強調表示された項目とオーバーフローセクションを持つナビゲーションメニューを作成します。

```
<aura:component>
    <lightning:verticalNavigation selectedItem="recent">
        <lightning:verticalNavigationSection label="Reports">
            <lightning:verticalNavigationItem label="Recent" name="recent" />
            <lightning:verticalNavigationItem label="All Reports" name="all" />
        </lightning:verticalNavigationSection>
        <lightning:verticalNavigationOverflow>
            <lightning:verticalNavigationItem label="Regional Sales East" name="east" />
            <lightning:verticalNavigationItem label="Regional Sales West" name="west" />
        </lightning:verticalNavigationOverflow>
    </lightning:verticalNavigation>
</aura:component>
```

ナビゲーションメニューの動的な作成

JavaScript を使用してナビゲーションメニューを作成するには、サブコンポーネントを定義する、キー - 値のペアの対応付けを渡します。次に、コンポーネントの初期化時にナビゲーションメニューを作成する例を示します。

```
<aura:component>
    <aura:attribute name="navigationData" type="Object" description="The list of sections
 and their items." />
    <aura:handler name="init" value="{! this }" action="{! c.init }" />
    <lightning:verticalNavigation>
            <aura:iteration items="{! v.navigationData }" var="section">
                <lightning:verticalNavigationSection label="{! section.label }">
                    <aura:iteration items="{! section.items }" var="item">
                        <aura:if isTrue="{! !empty(item.icon) }">
```

```
                            <lightning:verticalNavigationItemIcon
                                label="{! item.label }"
                                name="{! item.name }"
                                iconName="{! item.icon }" />
                            <aura:set attribute="else">
                                <lightning:verticalNavigationItem
                                    label="{! item.label }"
                                    name="{! item.name }" />
                            </aura:set>
                        </aura:if>
                    </aura:iteration>
                </lightning:verticalNavigationSection>
            </aura:iteration>
        </lightning:verticalNavigation>
</aura:component>
```

クライアント側のコントローラで、2 つのセクションを作成し、各セクションに 2 つのナビゲーション項目を含めます。

```
({
    init: function (component) {
        var sections = [
          {
            label: "Reports",
            items: [
              {
                label: "Created by Me",
                name: "default_created"
              },
              {
                label: "Public Reports",
                name: "default_public"
              }
            ]
          },
          {
            label: "Dashboards",
            items: [
              {
                label: "Favorites",
                name: "default_favorites",
                icon: "utility:favorite"
              },
              {
                label: "Most Popular",
                name: "custom_mostpopular"
              }
            ]
          }
        ];
        component.set('v.navigationData', sections);
    }
})
```

使用上の考慮事項

2 レベル以上の深さを持つナビゲーションメニューを作成する場合、代わりに `lightning:tree` を使用することを考慮してください。

ナビゲーションメニューは画面の幅全体を使用します。CSS を使用して幅を指定できます。

```
.THIS {
    width: 320px;
}
```

アクセシビリティ

Tab キーと Shift+Tab キーを使用して、メニューを上下に移動します。オーバーフローセクションを展開するまたは折りたたむには、Enter キーまたはスペースバーを押します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
compact	Boolean	ナビゲーション項目間の間隔を小さくするには、true を指定します。この値のデフォルトは false です。	
onselect	Action	項目が選択されたときに起動されるアクション。イベントパラメータには、選択された項目の「name」が含まれます。	
selectedItem	String	有効にするナビゲーション項目の名前。	
shaded	Boolean	網掛け表示の背景上に垂直ナビゲーションを配置する場合は true を指定します。この値のデフォルトは false です。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	

lightning:verticalNavigationItem

lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のテキストのみのリンク。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:verticalNavigationItem` コンポーネントは `lightning:verticalNavigation` 内のナビゲーション項目です。

詳細は、「`lightning:verticalNavigation`」を参照してください。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
label	String	ナビゲーション項目に表示するテキスト。	はい
name	String	ナビゲーション項目の一意の識別子。	はい
href	String	ナビゲーション項目の移動先のページの URL。	

lightning:verticalNavigationItemBadge

lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のリンクとバッジ。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:verticalNavigationItemBadge` コンポーネントは、項目の表示ラベルの右に数値バッジを表示するナビゲーション項目です。

次に、バッジが含まれる項目を持つナビゲーションメニューを作成する例を示します。

```
<aura:component>
    <lightning:verticalNavigation selectedItem="recent">
        <lightning:verticalNavigationSection label="Reports">
            <lightning:verticalNavigationItemBadge label="Recent" name="recent"
badgeCount="3" />
            <lightning:verticalNavigationItem label="Created by Me" name="usercreated" />

            <lightning:verticalNavigationItem label="Private Reports" name="private" />
            <lightning:verticalNavigationItem label="Public Reports" name="public" />
            <lightning:verticalNavigationItem label="All Reports" name="all" />
        </lightning:verticalNavigationSection>
    </lightning:verticalNavigation>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
label	String	このナビゲーション項目に表示するテキスト。	はい
name	String	このナビゲーション項目の一意の識別子。	はい
href	String	ナビゲーション項目の移動先のページの URL。	
badgeCount	数値	バッジ内に表示するテキスト。この値がゼロの場合、バッジは非表示になります。	
assistiveText	String	バッジ内の番号を説明する補助テキスト。これはアクセシビリティを向上するために使用され、ユーザには表示されません。	

lightning:verticalNavigationItemIcon

lightning:verticalNavigationSection または lightning:verticalNavigationOverflow 内のリンクとアイコン。このコンポーネントでは、API バージョン 41.0 以降が必要です。

`lightning:verticalNavigationItemIcon` コンポーネントは、項目の表示ラベルの左にアイコンを表示するナビゲーション項目です。

次に、ナビゲーション項目にアイコンを表示するナビゲーションメニューを作成する例を示します。

```
<aura:component>
    <lightning:verticalNavigation>
        <lightning:verticalNavigationSection label="Reports">
            <lightning:verticalNavigationItemIcon
                label="Recent"
                name="recent"
                iconName="utility:clock" />
            <lightning:verticalNavigationItemIcon
                label="Created by Me"
                name="created"
                iconName="utility:user" />
            <lightning:verticalNavigationItem
                label="All Reports"
                name="all"
                iconName="utility:open_folder" />
        </lightning:verticalNavigationSection>
    </lightning:verticalNavigation>
</aura:component>
```

Lightning Design System のアイコンがサポートされます。使用できるアイコンについては、https://lightningdesignsystem.com/icons を参照してください。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントの基本クラスに加え、外部要素の CSS クラス。	
title	String	マウスポインタが要素に重ねられたときにツールチップテキストを表示します。	
label	String	このナビゲーション項目に表示するテキスト。	はい
name	String	このナビゲーション項目の一意の識別子。	はい
href	String	ナビゲーション項目の移動先のページの URL。	
iconName	String	アイコンの Lightning Design System 名。名前は、形式「\utility:down\」で記述します。「utility」はカテゴリ、「down」は表示する特定のアイコンです。	

lightning:verticalNavigationOverflow

前述の lightning:verticalNavigationSection からの項目のオーバーフローを表し、表示を切り替える機能があります。このコンポーネントでは、API 41.0 以降が必要です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

lightning:verticalNavigationSection

lightning:verticalNavigation 内のセクションを表します。このコンポーネントでは、API バージョン 41.0 以降が必要です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
label	String	このセクションのヘッダーテキスト。	はい

ltng:require

連動関係の順序を維持しながら、スクリプトとスタイルシートを読み込みます。スタイルはリストの順序で読み込まれます。同じコンポーネントまたは異なるコンポーネントの複数の<ltng:require>タグでスタイルが指定されていても、スタイルが読み込まれるのは 1 回のみです。

ltng:require では、外部 CSS と JavaScript ライブラリを静的リソースとしてアップロードした後で読み込むことができます。

```
<aura:component>

    <ltng:require

        styles="{!$Resource.SLDSv1 + '/assets/styles/lightning-design-system-ltng.css'}"

        scripts="{!$Resource.jsLibraries + '/jsLibOne.js'}"

        afterScriptsLoaded="{!c.scriptsLoaded}" />

</aura:component>
```

式で $Resource が解析される方法に予測できない動作があるため、複数の $Resource 参照を 1 つの属性に含めるには join 演算子を使用します。たとえば、コンポーネントに追加する JavaScript ライブラリが複数ある場合、scripts 属性は次のようにする必要があります。

```
scripts="{!join(',',

    $Resource.jsLibraries + '/jsLibOne.js',

    $Resource.jsLibraries + '/jsLibTwo.js')}"
```

リソースのカンマ区切りリストは、scripts および styles 属性に入力された順序で読み込まれます。スクリプトが読み込まれると、クライアント側コントローラの afterScriptsLoaded アクションがコールされます。カプセル化および再利用を確実に行うには、CSS または JavaScript ライブラリを使用する .cmp または .app リソースのそれぞれに <ltng:require> タグを追加します。

同じコンポーネントまたは異なるコンポーネントの複数の <ltng:require> タグでリソースが指定されていても、リソースが読み込まれるのは 1 回のみです。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
scripts	String []	読み込まれる連動関係の順序で表示したスクリプトのセット。	
styles	String []	読み込まれる連動関係の順序で示したスタイルシートのセット。	

イベント

イベント名	イベントタイプ	説明
afterScriptsLoaded	COMPONENT	ltng:require.scripts にリストされたすべてのスクリプトが ltng:require で読み込まれると起動します。
beforeLoadingResources	COMPONENT	ltng:require でリソースの読み込みが開始される前に起動します。

ui:actionMenuItem

アクションをトリガするメニュー項目。このコンポーネントは、ui:menu コンポーネントでネストされます。

ui:actionMenuItem コンポーネントは、クリックしたときにアクションをトリガするメニューリスト項目を表します。リストの値を反復処理してメニュー項目を表示するには、aura:iteration を使用します。ui:menuTriggerLink コンポーネントは、メニュー項目を表示したり、非表示にしたりします。

```
<aura:attribute name="status" type="String[]" default="Open, Closed, Closed Won, Any"/>
    <ui:menu>
        <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
            <aura:iteration items="{!v.status}" var="s">
                <ui:actionMenuItem label="{!s}" click="{!c.doSomething}"/>
            </aura:iteration>
        </ui:menuList>
    </ui:menu>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
hideMenuAfterSelected	Boolean	メニュー項目の選択後にメニューを非表示にするには、true に設定します。	
label	String	コンポーネントに表示されるテキスト。	
selected	Boolean	メニュー項目の状況。True はこのメニュー項目が選択されていることを示し、False は選択されていないことを示します。	
type	String	メニュー項目の具体的な種別。有効な値は、「action」、「checkbox」、「radio」、「separator」、または ns:xxxxmenuItem などの名前空間を含むコンポーネント記述子です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。

イベント名	イベントタイプ	説明
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。

ui:button

ボタン要素を表します。

ui:button コンポーネントは、コントローラで定義されたアクションを実行するボタン要素を表します。ボタンをクリックすると、press イベントに対して設定されたクライアント側コントローラのメソッドがトリガされます。ボタンは、さまざまな方法で作成できます。

テキストのみのボタンに設定されているのは label 属性のみです。

```
<ui:button label="Find"/>
```

画像のみのボタンでは、CSS で label と labelClass の両方の属性を使用します。

```
<!-- Component markup -->
<ui:button label="Find" labelClass="assistiveText" class="img" />

/** CSS **/
THIS.uiButton.img {
background: url(/path/to/img) no-repeat;
width:50px;
height:25px;
}
```

assistiveText クラスは、ビューに表示ラベルは表示されませんが、支援技術には使用できます。画像とテキストの両方を含むボタンを作成するには、label 属性を使用し、ボタンのスタイルを追加します。

```
<!-- Component markup -->
<ui:button label="Find" />

/** CSS **/
THIS.uiButton {
background: url(/path/to/img) no-repeat;
}
```

テキストと画像を含むボタンの前述のマークアップの結果、次の HTML になります。

```
<button class="button uiButton--default uiButton" accesskey type="button">
<span class="label bBody truncate" dir="ltr">Find</span>
</button>
```

次の例に、入力値を表示するボタンを示します。

```
<aura:component access="global">
 <ui:inputText aura:id="name" label="Enter Name:" placeholder="Your Name" />
 <ui:button aura:id="button" buttonTitle="Click to see what you put into the field"
class="button" label="Click me" press="{!c.getInput}"/>
 <ui:outputText aura:id="outName" value="" class="text"/>
</aura:component>
```

```
({
    getInput : function(cmp, evt) {
        var myName = cmp.find("name").get("v.value");
        var myText = cmp.find("outName");
        var greet = "Hi, " + myName;
        myText.set("v.value", greet);
    }
})
```

属性

属性名	属性型	説明	必須項目
accesskey	String	ボタンにフォーカスを置くキーボードのアクセスキー。ボタンにフォーカスがあるときに Enter キーを押すと、ボタンがクリックされます。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
buttonTitle	String	ボタンにマウスポインタを置いたときにツールチップとして表示されるテキスト。	
buttonType	String	ボタンの種類を指定します。指定可能な値は、reset、submit、または button です。この値のデフォルトは button です。	
class	String	ボタンに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	このボタンを無効な状態で表示するかどうかを指定します。無効なボタンをクリックすることはできません。デフォルト値は「false」です。	
label	String	ボタンに表示されるテキスト。表示される HTML 入力要素の value 属性に対応します。	

属性名	属性型	説明	必須項目
labelClass	String	表示ラベルに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	

イベント

イベント名	イベントタイプ	説明
press	COMPONENT	ボタンがクリックされたときに起動されるイベント。

ui:checkboxMenuItem

複数選択をサポートしてアクションを呼び出すことができる、チェックボックスを含むメニュー項目。このコンポーネントは、ui:menu コンポーネントでネストされます。

ui:checkboxMenuItem コンポーネントは、複数選択を有効にするメニューリスト項目を表します。リストの値を反復処理してメニュー項目を表示するには、aura:iteration を使用します。ui:menuTriggerLink コンポーネントは、メニュー項目を表示したり、非表示にしたりします。

```
<aura:attribute name="status" type="String[]" default="Open, Closed, Closed Won, Any"/>
    <ui:menu>
    <ui:menuTriggerLink aura:id="checkboxMenuLabel" label="Multiple selection"/>
       <ui:menuList aura:id="checkboxMenu" class="checkboxMenu">
        <aura:iteration items="{!v.status}" var="s">
            <ui:checkboxMenuItem label="{!s}"/>
            </aura:iteration>
        </ui:menuList>
    </ui:menu>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	

属性名	属性型	説明	必須項目
hideMenuAfterSelected	Boolean	メニュー項目の選択後にメニューを非表示にするには、true に設定します。	
label	String	コンポーネントに表示されるテキスト。	
selected	Boolean	メニュー項目の状況。True はこのメニュー項目が選択されていることを示し、False は選択されていないことを示します。	
type	String	メニュー項目の具体的な種別。有効な値は、「action」、「checkbox」、「radio」、「separator」、または ns:xxxxmenuItem などの名前空間を含むコンポーネント記述子です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。

イベント名	イベントタイプ	説明
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。

ui:inputCheckbox

チェックボックスを表します。クリックや変更などのイベントを使用して、動作を設定できます。

ui:inputCheckbox コンポーネントは、value および disabled 属性によって状態が制御されるチェックボックスを表します。checkbox 型の HTML input タグとして表示されます。ui:inputCheckbox コンポーネントからの出力を表示するには、ui:outputCheckbox コンポーネントを使用します。

次に、チェックボックスの基本設定を示します。

```
<ui:inputCheckbox label="Reimbursed?"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputCheckbox uiInput--default uiInput--checkbox">
  <label class="uiLabel-left form-element__label uiLabel">
    <span>Reimbursed?</span>
  </label>
  <input type="checkbox">
</div>
```

value 属性はチェックボックスの状態を制御し、click や change などのイベントはその動作を決定します。次の例は、クリックイベント時のチェックボックスの CSS クラスを更新します。

```
<!-- Component Markup -->
<ui:inputCheckbox label="Color me" click="{!c.update}"/>

/** Client-Side Controller **/
update : function (cmp, event) {
  $A.util.toggleClass(event.getSource(), "red");
}
```

次の例は、ui:inputCheckbox コンポーネントの値を取得します。

```
<aura:component>
 <aura:attribute name="myBool" type="Boolean" default="true"/>
 <ui:inputCheckbox aura:id="checkbox" label="Select?" change="{!c.onCheck}"/>
 <p>Selected:</p>
 <p><ui:outputText class="result" aura:id="checkResult" value="false" /></p>
 <p>The following checkbox uses a component attribute to bind its value.</p>
 <ui:outputCheckbox aura:id="output" value="{!v.myBool}"/>
</aura:component>
```

```
({
```

```
  onCheck: function(cmp, evt) {
   var checkCmp = cmp.find("checkbox");
   resultCmp = cmp.find("checkResult");
   resultCmp.set("v.value", ""+checkCmp.get("v.value"));

  }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	コンポーネントに表示されるテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
name	String	コンポーネントの名前。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
text	String	入力値の属性。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change,click」です。	
value	Boolean	オプションの状況が選択されているかどうかを示します。デフォルト値は「false」です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。

イベント名	イベントタイプ	説明
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputCurrency

通貨を入力するための入力項目。

ui:inputCurrency コンポーネントは、text 型の HTML input 要素として表示される、通貨である数値の入力項目を表します。これは、JavaScriptの数値型を使用して、サポートされる桁数を決定します。デフォルトでは、ブラウザのロケールが使用されます。ui:inputCurrency コンポーネントからの出力を表示するには、ui:outputCurrency コンポーネントを使用します。

次に、ブラウザの通貨ロケールが $ の場合に、値 $50.00 を含む入力項目を表示する ui:inputCurrency コンポーネントの基本設定を示します。

```
<ui:inputCurrency aura:id="amount" label="Amount" class="field" value="50"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInput--default uiInput--input">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Amount</span>
    </label>
    <input class="field input" max="99999999999999" step="1" type="text"
min="-99999999999999">
</div>
```

ブラウザのロケールを上書きするには、ui:inputCurrency コンポーネントの v.format 属性で新しい形式を設定します。次の例は、値 £50.00 を含む入力項目を表示します。

```
var curr = component.find("amount");
curr.set("v.format", '£#,###.00');
```

次の例は、ui:inputCurrency コンポーネントの値を ui:outputCurrency にバインドします。

```

<aura:component>
    <aura:attribute name="myCurrency" type="integer" default="50"/>
   <ui:inputCurrency aura:id="amount" label="Amount" class="field" value="{!v.myCurrency}"
 updateOn="keyup"/>
 You entered: <ui:outputCurrency value="{!v.myCurrency}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
format	String	数値の形式。たとえば、format=".00" は、小数点以下 2 桁の数値を表示します。指定されていない場合は、ロケールのデフォルト形式が使用されます。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	Decimal	数値の入力値。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputDate

日付を入力するための入力項目。

ui:inputDate コンポーネントは、デスクトップ上に text 型の HTML input タグとして表示される、日付の入力項目を表します。携帯電話およびタブレットで実行される Web アプリケーションは、Internet Explorer を除くすべてのブラウザで date 型の入力項目を使用します。値は、$Locale.dateFormat で返されるブラウザのロケールに基づいて表示されます (たとえば、MMM d, yyyy など)。

ロケール形式に基づいて項目値 Jan 30, 2014 を表示する、日付ピッカーアイコンがある日付項目の基本設定を次に示します。デスクトップ上で、input タグは form タグでラップされます。

```
<ui:inputDate aura:id="dateField" label="Birthday" value="2014-01-30"
displayDatePicker="true"/>
```

モバイルデバイスでの日付の選択

モバイルまたはタブレットに表示する場合、ui:inputDate コンポーネントはネイティブの日付ピッカーを使用します。この場合、format 属性はサポートされません。入力項目の日付値の変更は、値変更ハンドラを使用して取得することをお勧めします。iOS デバイスでは、日付ピッカーで日付を選択すると、コンポーネントで変更ハンドラがトリガされますが、値は blur イベントでのみバインドされます。この例では、日付値を値変更ハンドラにバインドします。

```
<aura:component>
    <aura:attribute name="myDate" type="Date" />
    <!-- Value change handler -->
    <aura:handler name="change" value="{!v.myDate}" action="{!c.handleValueChange}"/>

    <ui:inputDate aura:id="mySelectedDate"
                  label="Select a date" displayDatePicker="true"
                  value="{!v.myDate}"/>
</aura:component>
```

次の例は、ui:inputDate コンポーネントで今日の日付を設定し、その値を取得し、ui:outputDate を使用して値を表示します。init ハンドラは、コンポーネントで日付を初期化し、設定します。

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDate aura:id="expdate" label="Today's Date" class="field" value="{!v.today}"
 displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputDate aura:id="oDate" value="" />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
    },
```

```
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');
        var expdate = component.find("expdate").get("v.value");

        var oDate = component.find("oDate");
        oDate.set("v.value", expdate);

    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
displayDatePicker	Boolean	日付ピッカーをトリガするアイコンを項目に表示するかどうかを示します。デフォルトは false です。	
errors	ArrayList	表示するエラーのリスト。	
format	String	java.text.SimpleDateFormat スタイル形式の文字列。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
langLocale	String	非推奨。日時の書式設定に使用される言語ロケール。ロケール値プロバイダによって提供される値のみを使用できます。これ以外の場合は、$Locale.langLocale の値に戻ります。この属性は今後のリリースで廃止されます。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	

属性名	属性型	説明	必須項目
value	String	ISO 文字列としての日付/時間の入力値。	

イベント

イベント名	イベントタイプ	説明
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。

イベント名	イベントタイプ	説明
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputDateTime

日時を入力するための入力項目。

ui:inputDateTime コンポーネントは、デスクトップ上に text 型の HTML input タグとして表示される、日時の入力項目を表します。携帯電話およびタブレットで実行される Web アプリケーションは、Internet Explorer を除くすべてのブラウザで datetime-local 型の入力項目を使用します。値は、$Locale.dateFormat および $Locale.timeFormat で返されるブラウザのロケールに基づいて表示されます (たとえば、MMM d, yyyy と h:mm:ss a など)。

これは、日付ピッカーアイコンを使用した、日付と時間のペアの項目の基本的な設定です。クライアント側のコントローラで現在の日付と時間を項目に設定します。デスクトップでは、input タグが form タグでラップされ、日付と時間の項目が 2 つの別個の項目として表示されます。時間ピッカーには、30 分単位で増分する時間のリストが表示されます。

```
<!-- Component markup -->
<aura:attribute name="today" type="DateTime" />
<ui:inputDateTime aura:id="expdate" label="Expense Date" class="form-control"
   value="{!v.today}" displayDatePicker="true" />

/** Client-Side Controller **/
 var today = new Date();
// Convert the date to an ISO string
component.set("v.today", today.toISOString());
```

モバイルデバイスでの日付と時間の選択

モバイルまたはタブレットに表示する場合、ui:inputDateTime コンポーネントはネイティブの日時ピッカーを使用します。この場合、format 属性はサポートされません。入力項目の日時値の変更は、値変更ハンドラを使用して取得することをお勧めします。iOS デバイスでは、日付ピッカーで日付と時間を選択すると、コンポーネントで変更ハンドラがトリガされますが、値は blur イベントでのみバインドされます。この例では、日付値を値変更ハンドラにバインドします。

```
<aura:component>
    <aura:attribute name="myDateTime" type="DateTime" />
    <!-- Value change handler -->
    <aura:handler name="change" value="{!v.myDateTime}" action="{!c.handleValueChange}"/>


    <ui:inputDateTime aura:id="mySelectedDateTime"
                  label="Select a date and time"
```

```
                  value="{!v.myDateTime}"/>
</aura:component>
```

次の例は、ui:inputDateTime コンポーネントの値を取得し、ui:outputDateTime を使用して値を表示します。

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDateTime aura:id="today" label="Time" class="field" value=""
displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

    <div aura:id="msg" class="hide">
  You entered: <ui:outputDateTime aura:id="oDateTime" value=""  />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
    },

    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var todayVal = component.find("today").get("v.value");
        var oDateTime = component.find("oDateTime");
        oDateTime.set("v.value", todayVal);

    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	

属性名	属性型	説明	必須項目
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
displayDatePicker	Boolean	日付ピッカーをトリガするアイコンを項目に表示するかどうかを示します。デフォルトは false です。	
errors	ArrayList	表示するエラーのリスト。	
format	String	java.text.SimpleDateFormat スタイル形式の文字列。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
langLocale	String	非推奨。日時の書式設定に使用される言語ロケール。ロケール値プロバイダによって提供される値のみを使用できます。これ以外の場合は、$Locale.langLocale の値に戻ります。この属性は今後のリリースで廃止されます。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	ISO 文字列としての日付/時間の入力値。	

イベント

イベント名	イベントタイプ	説明
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputDefaultError

値を反復処理してメッセージを表示する、項目レベルのエラーのデフォルト実装。

ui:inputDefaultError は、入力コンポーネントのデフォルトのエラー処理です。このコンポーネントは、エラーのリストとして項目の下に表示されます。項目レベルのエラーメッセージは、set("v.errors") を使用して追加できます。エラー属性を使用して、エラーメッセージを表示できます。たとえば、次のコンポーネントは、入力が数値であるかどうかを検証します。

```
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp" label="number"/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

次のクライアント側のコントローラは、入力が数値でない場合にエラーを表示します。

```
doAction : function(component, event) {
    var inputCmp = cmp.find("inputCmp");
    var value = inputCmp.get("v.value");
    if (isNaN(value)) {
        inputCmp.set("v.errors", [{message:"Input not a number: " + value}]);
    } else {
        //clear error
        inputCmp.set("v.errors", null);
    }
}
```

または、独自の ui:inputDefaultError コンポーネントを指定することもできます。次の例は、warnings 属性にメッセージが含まれている場合にエラーメッセージを返します。

```
<aura:component>
      <aura:attribute name="warnings" type="String[]" description="Warnings for input
text"/>
    Enter a number: <ui:inputNumber aura:id="inputCmp" label="number"/>
    <ui:button label="Submit" press="{!c.doAction}"/>
    <ui:inputDefaultError aura:id="number" value="{!v.warnings}" />
</aura:component>
```

次のクライアント側のコントローラは、warnings 属性に文字列を追加することによってエラーを表示します。

```
doAction : function(component, event) {
    var inputCmp = component.find("inputCmp");
    var value = inputCmp.get("v.value");

    // is input numeric?
    if (isNaN(value)) {
       component.set("v.warnings", "Input is not a number");
    } else {
       // clear error
       component.set("v.warnings", null);
    }
}
```

次の例に、デフォルトのエラー処理をする ui:inputText コンポーネントと、テキストを表示するための対応する ui:outputText コンポーネントを示します。

```
<aura:component>
 <ui:inputText aura:id="color" label="Enter some text: " placeholder="Blue" />
 <ui:button label="Validate" press="{!c.checkInput}" />
 <ui:outputText aura:id="outColor" value="" class="text"/>
</aura:component>
```

```
({
    checkInput : function(cmp) {
     var colorCmp = cmp.find("color");
```

```
        var myColor = colorCmp.get("v.value");

        var myOutput = cmp.find("outColor");
        var greet = "You entered: " + myColor;
        myOutput.set("v.value", greet);

        if (!myColor) {
            colorCmp.set("v.errors", [{message:"Enter some text"}]);
        }
        else {
            colorCmp.set("v.errors", null);
        }
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
value	String []	表示するエラー文字列のリスト。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:inputEmail

メールアドレスを入力するための入力項目を表します。

`ui:inputEmail` コンポーネントは、`email` 型の HTML `input` タグとして表示される、メールの入力項目を表します。`ui:inputEmail` コンポーネントからの出力を表示するには、`ui:outputEmail` コンポーネントを使用します。

次に、メール項目の基本設定を示します。

```
<ui:inputEmail aura:id="email" label="Email" placeholder="abc@email.com"/>
```

この例の結果、次の HTML になります。

```
<ui:inputEmail aura:id="email" label="Email" placeholder="abc@email.com"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputEmail uiInput--default uiInput--input">
  <label class="uiLabel-left form-element__label uiLabel">
    <span>Email</span>
  </label>
  <input placeholder="abc@email.com" type="email" class="field input">
</div>
```

次の例は、`ui:inputEmail` コンポーネントの値を取得し、`ui:outputEmail` を使用して値を表示します。

```

<aura:component>
    <ui:inputEmail aura:id="email" label="Email" class="field" value="manager@email.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
     You entered: <ui:outputEmail aura:id="oEmail" value="Email" />
 </div>

</aura:component>
```

```

({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var email = component.find("email").get("v.value");
        var oEmail = component.find("oEmail");
        oEmail.set("v.value", email);

```

```
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputNumber

使用可能な場合にクライアントの入力支援と検証を利用する、数値を入力するための入力項目。

ui:inputNumber コンポーネントは、text 型の HTML input 要素として表示される、数値の入力項目を表します。これは、JavaScript の数値型を使用して、サポートされる桁数を決定します。

次の例に、10 の値を表示する数値項目を示します。

```
<aura:attribute name="num" type="integer" default="10"/>
<ui:inputNumber aura:id="num" label="Age" value="{!v.num}"/>
```

前の例の結果、次の HTML になります。

```
<div class="uiInput uiInputNumber uiInput--default uiInput--input">
<label class="uiLabel-left form-element__label uiLabel">
    <span>Age</span>
</label>
<input max="99999999999999" step="1" type="text"
        min="-99999999999999" class="input">
</div>
```

ui:inputNumber コンポーネントからの出力を表示するには、ui:outputNumber コンポーネントを使用します。カンマを含む数値を指定する場合は、type="integer" を使用します。次の例は 100,000 を返します。

```
<aura:attribute name="number" type="integer" default="100,000"/>
<ui:inputNumber label="Number" value="{!v.number}"/>
```

type="string" の場合は、正しい形式で出力するためカンマを含まない数値を指定します。次の例も 100,000 を返します。

```
<aura:attribute name="number" type="string" default="100000"/>
<ui:inputNumber label="Number" value="{!v.number}"/>
```

format="#,##0,000.00#" を指定すると、10,000.00 のように書式設定された数値が返されます。

```
<ui:inputNumber label="Cost" aura:id="costField" format="#,##0,000.00#" value="10000"/>
```

次の例は、ui:inputNumber コンポーネントの値を ui:outputNumber にバインドします。

```

<aura:component>
    <aura:attribute name="myNumber" type="integer" default="10"/>
 <ui:inputNumber label="Enter a number: " value="{!v.myNumber}" updateOn="keyup"/> <br/>
    <ui:outputNumber value="{!v.myNumber}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
format	String	数値の形式。たとえば、format=".00" は、小数点以下 2 桁の数値を表示します。指定されていない場合は、ロケールのデフォルト形式が使用されます。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	Decimal	数値の入力値。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputPhone

電話番号を入力するための入力項目を表します。

ui:inputPhone コンポーネントは、tel 型の HTML input タグとして表示される、電話番号を入力するための入力項目を表します。ui:inputPhone コンポーネントからの出力を表示するには、ui:outputPhone コンポーネントを使用します。

次の例は、指定された電話番号を表示する電話項目を示します。

```
<ui:inputPhone label="Phone" value="415-123-4567" />
```

前の例の結果、次の HTML になります。

```
<ui:inputPhone label="Phone" value="415-123-4567" />
```

前の例の結果、次の HTML になります。

```
<div class="uiInput uiInputPhone uiInput--default uiInput--input">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Phone</span>
    </label>
    <input class="input" type="tel">
</div>
```

次の例は、ui:inputPhone コンポーネントの値を取得し、ui:outputPhone を使用して値を表示します。

```

<aura:component>
   <ui:inputPhone aura:id="phone" label="Phone Number" class="field" value="415-123-4567"
 />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputPhone aura:id="oPhone" value="" />
 </div>
</aura:component>
```

```

({

    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var phone = component.find("phone").get("v.value");
        var oPhone = component.find("oPhone");
        oPhone.set("v.value", phone);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputRadio

入力で使用されるラジオボタン。

ui:inputRadio コンポーネントは、value および disabled 属性によって状態が制御されるラジオボタンを表します。radio 型の HTML input タグとして表示されます。ラジオボタンをまとめてグループ化するには、name 属性を一意の名前で指定します。

次の例は、ラジオボタンの基本設定です。

```
<ui:inputRadio label="Yes"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputRadio uiInput--default uiInput--radio">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Yes</span>
    </label>
    <input type="radio">
</div>
```

次の例では、選択された ui:inputRadio コンポーネントの値を取得します。

```

<aura:component>
    <aura:attribute name="stages" type="String[]" default="Any,Open,Closed,Closed Won"/>
    <aura:iteration items="{!v.stages}" var="stage">
     <ui:inputRadio label="{!stage}" change="{!c.onRadio}" />
    </aura:iteration>

   <b>Selected Item:</b>
   <p><ui:outputText class="result" aura:id="radioResult" value="" /></p>

   <b>Radio Buttons - Group</b>
   <ui:inputRadio aura:id="r0" name="others" label="Prospecting" change="{!c.onGroup}"/>
   <ui:inputRadio aura:id="r1" name="others" label="Qualification" change="{!c.onGroup}"
value="true"/>
   <ui:inputRadio aura:id="r2" name="others" label="Needs Analysis" change="{!c.onGroup}"/>

   <ui:inputRadio aura:id="r3" name="others" label="Closed Lost" change="{!c.onGroup}"/>
   <b>Selected Items:</b>
   <p><ui:outputText class="result" aura:id="radioGroupResult" value="" /></p>

</aura:component>
```

```

({
 onRadio: function(cmp, evt) {
   var selected = evt.getSource().get("v.label");
   resultCmp = cmp.find("radioResult");
   resultCmp.set("v.value", selected);
  },

  onGroup: function(cmp, evt) {
   var selected = evt.getSource().get("v.label");
   resultCmp = cmp.find("radioGroupResult");
   resultCmp.set("v.value", selected);
  }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	このラジオボタンを無効な状態で表示するかどうかを指定します。無効なラジオボタンはクリックできません。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	コンポーネントに表示されるテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
name	String	コンポーネントの名前。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
text	String	入力値の属性。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	Boolean	オプションの状況が選択されているかどうかを示します。デフォルト値は「false」です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputRichText

リッチテキストを入力するための入力項目。このコンポーネントは、LockerService ではサポートされていません。

メモ: `ui:inputRichText` の代わりに `lightning:inputRichText` を使用することをお勧めします。`ui:inputRichText` は、LockerService が有効化されている場合にサポートされなくなりました。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
cols	Integer	テキストエリアの幅。1 行に一度に表示する文字数で定義します。デフォルト値は「20」です。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
height	String	編集エリアの高さ (エディタのコンテンツを含む)。整数 (ピクセルサイズ) または CSS で定義されている長さの単位で指定できます。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML textarea 要素の maxlength 属性に対応します。	
placeholder	String	デフォルトで表示されるテキスト。	
readonly	Boolean	このテキストエリアを参照のみとして表示するかどうかを指定します。デフォルト値は「false」です。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
resizable	Boolean	テキストエリアのサイズを変更できるかどうかを指定します。デフォルトは true です。	
rows	Integer	テキストエリアの高さ。一度に表示する行数で定義されます。デフォルト値は「2」です。	

属性名	属性型	説明	必須項目
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	
width	String	エディタ UI の外側の余白の幅。整数 (ピクセルサイズ) または CSS で定義されている単位で指定できます。isRichText が false に設定されている場合は、代わりに cols 属性を使用します。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。

イベント名	イベントタイプ	説明
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputSecret

password 型の秘密のテキストを入力するための入力項目。

ui:inputSecret コンポーネントは、password 型の HTML input タグとして表示されるパスワード項目を表します。

次の例は、パスワード項目の基本設定です。

```
<ui:inputSecret aura:id="secret" label="Pin" class="field" value="123456"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputSecret uiInput--default uiInput--input">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Pin</span>
    </label>
    <input class="field input" type="password">
</div>
```

次の例では、デフォルト値で ui:inputSecret コンポーネントを表示します。

```

<aura:component>
    <ui:inputSecret aura:id="secret" label="Pin" class="field" value="123456"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputSelect

オプションを含むドロップダウンリストを表します。

ui:inputSelect コンポーネントは、HTML select 要素として表示されます。ui:inputSelectOption コンポーネントで表されるオプションが含まれます。複数選択を有効にするには、multiple="true" を設定します。入力値選択時のクライアント側のロジックを関連付けるには、change イベントを使用します。

```
<ui:inputSelect multiple="true">
    <ui:inputSelectOption text="All Primary" label="All Contacts" value="true"/>
    <ui:inputSelectOption text="All Primary" label="All Primary"/>
    <ui:inputSelectOption text="All Secondary" label="All Secondary"/>
</ui:inputSelect>
```

v.value はオプションのHTML selected 属性を表し、v.text はオプションのHTML value 属性を表します。

aura:iteration によるオプションの生成

オプションを生成するには、aura:iteration を使用して項目のリストを反復処理します。次の例では、項目のリストを反復処理し、変更イベントを処理します。

```
<aura:attribute name="contactLevel" type="String[]" default="Primary Contact, Secondary
Contact, Other"/>
    <ui:inputSelect aura:id="levels" label="Contact Levels" change="{!c.onSelectChange}">

        <aura:iteration items="{!v.contactLevel}" var="level">
             <ui:inputSelectOption text="{!level}" label="{!level}"/>
        </aura:iteration>
    </ui:inputSelect>
```

選択したオプションが変更されると、次のクライアント側のコントローラで新しいテキスト値が取得されます。

```
onSelectChange : function(component, event, helper) {
    var selected = component.find("levels").get("v.value");
    //do something else
}
```

動的なオプションの生成

コントローラ側のアクションを使用して、コンポーネントの初期化時に動的にオプションを生成します。

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <ui:inputSelect label="Select me:" class="dynamic" aura:id="InputSelectDynamic"/>
</aura:component>
```

次のクライアント側のコントローラは、ui:inputSelect コンポーネントで options 属性を使用してオプションを生成します。v.options はオブジェクトのリストを取得し、リストオプションに変換します。opts オブジェクトは、ui:inputSelect 内に ui:inputSelectOptions コンポーネントを作成する InputOption オブジェクトを構築します。このサンプルコードは初期化中にオプションを生成しますが、オプションのリストは v.options でリストを操作するときにいつでも変更できます。コンポーネントは自動的に更新され、新しいオプションが表示されます。

```
({
    doInit : function(cmp) {
        var opts = [
```

```
            { class: "optionClass", label: "Option1", value: "opt1", selected: "true" },
            { class: "optionClass", label: "Option2", value: "opt2" },
            { class: "optionClass", label: "Option3", value: "opt3" }

        ];
        cmp.find("InputSelectDynamic").set("v.options", opts);
    }
})
```

`class` は、古いバージョンの Internet Explorer では動作しない可能性がある予約キーワードです。二重引用符で囲んだ `"class"` を使用することをお勧めします。複数のドロップダウンリストで同じオプションセットを再利用する場合は、各オプションセットで異なる属性を使用します。この操作を行わないと、1 つのリストで異なるオプションを選択したときに、他のリストのオプションも同じ属性に更新されます。

```
<aura:attribute name="options1" type="String" />
<aura:attribute name="options2" type="String" />
<ui:inputSelect aura:id="Select1" label="Select1" options="{!v.options1}" />
<ui:inputSelect aura:id="Select2" label="Select2" options="{!v.options2}" />
```

次の例では、単一および複数選択が有効なドロップダウンリストと、動的に生成されたリストオプションを使用する別のドロップダウンリストを表示します。`ui:inputSelect` コンポーネントの選択値を取得します。

```

<aura:component>
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

<div class="row">
<p class="title">Single Selection</p>
<ui:inputSelect class="single" aura:id="InputSelectSingle"
change="{!c.onSingleSelectChange}">

            <ui:inputSelectOption text="Any"/>
            <ui:inputSelectOption text="Open" value="true"/>
            <ui:inputSelectOption text="Closed"/>
            <ui:inputSelectOption text="Closed Won"/>
         <ui:inputSelectOption text="Prospecting"/>
            <ui:inputSelectOption text="Qualification"/>
            <ui:inputSelectOption text="Needs Analysis"/>
            <ui:inputSelectOption text="Closed Lost"/>
    </ui:inputSelect>
    <p>Selected Item:</p>
      <p><ui:outputText class="result" aura:id="singleResult" value="" /></p>
</div>

<div class="row">
    <p class="title">Multiple Selection</p>
    <ui:inputSelect multiple="true" class="multiple" aura:id="InputSelectMultiple"
change="{!c.onMultiSelectChange}">

            <ui:inputSelectOption text="Any"/>
            <ui:inputSelectOption text="Open"/>
            <ui:inputSelectOption text="Closed"/>
            <ui:inputSelectOption text="Closed Won"/>
```

```
            <ui:inputSelectOption text="Prospecting"/>
            <ui:inputSelectOption text="Qualification"/>
            <ui:inputSelectOption text="Needs Analysis"/>
            <ui:inputSelectOption text="Closed Lost"/>

    </ui:inputSelect>
    <p>Selected Items:</p>
     <p><ui:outputText class="result" aura:id="multiResult" value="" /></p>
</div>

<div class="row">
   <p class="title">Dynamic Option Generation</p>
   <ui:inputSelect label="Select me: " class="dynamic" aura:id="InputSelectDynamic"
change="{!c.onChange}" />
   <p>Selected Items:</p>
   <p><ui:outputText class="result" aura:id="dynamicResult" value="" /></p>
</div>

</aura:component>
```

```
({
    doInit : function(cmp) {
     // Initialize input select options
        var opts = [
            { "class": "optionClass", label: "Option1", value: "opt1", selected: "true"
},
            { "class": "optionClass", label: "Option2", value: "opt2" },
            { "class": "optionClass", label: "Option3", value: "opt3" }

        ];
        cmp.find("InputSelectDynamic").set("v.options", opts);

    },

 onSingleSelectChange: function(cmp) {
         var selectCmp = cmp.find("InputSelectSingle");
         var resultCmp = cmp.find("singleResult");
         resultCmp.set("v.value", selectCmp.get("v.value"));
  },

  onMultiSelectChange: function(cmp) {
         var selectCmp = cmp.find("InputSelectMultiple");
         var resultCmp = cmp.find("multiResult");
         resultCmp.set("v.value", selectCmp.get("v.value"));
  },

  onChange: function(cmp) {
   var dynamicCmp = cmp.find("InputSelectDynamic");
   var resultCmp = cmp.find("dynamicResult");
   resultCmp.set("v.value", dynamicCmp.get("v.value"));
  }

})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
multiple	Boolean	入力が複数選択かどうかを指定します。デフォルト値は「false」です。	
options	List	選択で使用するオプションのリスト。メモ: この属性を設定すると、コンポーネントは v.body を無視します。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputSelectOption

ui:inputSelect コンポーネント内にネストされた HTML オプション要素。リストで選択可能なオプションを示します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
label	String	コンポーネントに表示されるテキスト。	
name	String	コンポーネントの名前。	
text	String	入力値の属性。	
value	Boolean	オプションの状況が選択されているかどうかを示します。デフォルト値は「false」です。	

イベント

イベント名	イベントタイプ	説明
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:inputText

1 行の自由形式テキストを入力するのに適した入力項目を表します。

ui:inputText コンポーネントは、text 型の HTML input タグとして表示される、テキスト入力項目を表します。ui:inputText コンポーネントからの出力を表示するには、ui:outputText コンポーネントを使用します。

次の例は、テキスト項目の基本設定です。

```
<ui:inputText label="Expense Name" value="My Expense" required="true"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputTextuiInput--default uiInput--input">
  <label class="uiLabel-left form-element__label uiLabel">
    <span>Expense Name</span>
    <span class="required">*</span>
  </label>
  <input required="required" class="input" type="text">
</div>
```

次の例は、ui:inputText コンポーネントの値を ui:outputText にバインドします。

```

<aura:component>
    <aura:attribute name="myText" type="string" default="Hello there!"/>
 <ui:inputText label="Enter some text" class="field" value="{!v.myText}" updateOn="click"/>

 You entered: <ui:outputText value="{!v.myText}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputTextArea

編集可能または参照のみに設定できる HTML textarea 要素。Android デバイスの Chrome ブラウザではスクロールバーが表示されない場合がありますが、テキストエリアにフォーカスを置くとスクロールを有効にできます。

ui:inputTextArea コンポーネントは、HTML textarea タグとして表示される、複数行テキスト入力項目を表します。ui:inputTextArea コンポーネントからの出力を表示するには、ui:outputTextArea コンポーネントを使用します。

次の例は、ui:inputTextArea コンポーネントの基本設定です。

```
<ui:inputTextArea aura:id="comments" label="Comments" value="My comments" rows="5"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputTextArea uiInput--default uiInput--textarea">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Comments</span>
    </label>
    <textarea class="textarea" cols="20" rows="5">
    </textarea>
</div>
```

次の例は、ui:inputTextArea コンポーネントの値を取得し、ui:outputTextArea を使用して値を表示します。

```

<aura:component>
     <ui:inputTextArea aura:id="comments" label="Comments"  value="My comments" rows="5"/>

      <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

      <div aura:id="msg" class="hide">
  You entered: <ui:outputTextArea aura:id="oTextarea" value=""/>
 </div>
</aura:component>
```

```

({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var comments = component.find("comments").get("v.value");
        var oTextarea = component.find("oTextarea");
        oTextarea.set("v.value", comments);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
cols	Integer	テキストエリアの幅。1 行に一度に表示する文字数で定義します。デフォルト値は「20」です。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML textarea 要素の maxlength 属性に対応します。	
placeholder	String	デフォルトで表示されるテキスト。	
readonly	Boolean	このテキストエリアを参照のみとして表示するかどうかを指定します。デフォルト値は「false」です。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
resizable	Boolean	テキストエリアのサイズを変更できるかどうかを指定します。デフォルトは true です。	
rows	Integer	テキストエリアの高さ。一度に表示する行数で定義されます。デフォルト値は「2」です。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。

イベント名	イベントタイプ	説明
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:inputURL

URL を入力するための入力項目。

ui:inputURL コンポーネントは、url 型の HTML input タグとして表示される、URL の入力項目を表します。ui:inputURL コンポーネントからの出力を表示するには、ui:outputURL コンポーネントを使用します。

次の例は、ui:inputURL コンポーネントの基本設定です。

```
<ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputText uiInputURL uiInput--default uiInput--input">
    <label class="uiLabel-left form-element__label uiLabel">
        <span>Venue URL</span>
    </label>
    <input class="field input" type="url">
</div>
```

次の例は、ui:inputURL コンポーネントの値を取得し、ui:outputURL を使用して値を表示します。

```

<aura:component>
   <ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>
 <div aura:id="msg" class="hide">
  You entered: <ui:outputURL aura:id="oURL" value=""/>
 </div>
</aura:component>
```

```

({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var url = component.find("url").get("v.value");
        var oURL = component.find("oURL");
        oURL.set("v.value", url);
        oURL.set("v.label", url);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
errors	List	表示するエラーのリスト。	
label	String	表示ラベルコンポーネントのテキスト。	
labelClass	String	表示ラベルコンポーネントの CSS クラス。	
maxlength	Integer	入力項目に入力できる最大文字数。表示される HTML 入力要素の maxlength 属性に対応します。	
placeholder	String	項目が空の場合に、ユーザに有効な入力を求めるためのテキスト。	
required	Boolean	入力が必須かどうかを指定します。デフォルト値は「false」です。	
requiredIndicatorClass	String	必須のインジケータコンポーネントの CSS クラス。	
size	Integer	入力項目の長さを示す文字数。表示される HTML 入力要素の size 属性に対応します。	
updateOn	String	処理されたイベントに updateOn 属性が設定されている場合、コンポーネントの値のバインドを更新します。デフォルト値は「change」です。	
value	String	現在、値は入力項目にあります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
cut	COMPONENT	ユーザがコンテンツをクリップボードに切り取ると起動されるイベント。
onError	COMPONENT	コンポーネントに検証エラーがあると起動されるイベント。
onClearErrors	COMPONENT	いずれかの検証エラーをクリアする必要がある場合に起動されるイベント。
change	COMPONENT	ユーザが入力のコンテンツを変更したときに起動されるイベント。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーすると起動されるイベント。
paste	COMPONENT	ユーザがクリップボードからコンテンツを貼り付けると起動されるイベント。

ui:menu

表示を制御するトリガを含むドロップダウンメニューリスト。クリック可能なリンクやメニュー項目を作成するには、ui:menuTriggerLink および ui:menuList コンポーネントを使用します。

ui:menu コンポーネントにはトリガとリスト項目が含まれます。クライアント側コントローラでリスト項目をアクションに関連付けて、項目が選択されるとアクションがトリガされるようにすることができます。次の例では、リスト項目が含まれるメニューを表示し、リスト項目が押されるとトリガの表示ラベルを更新します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
              <ui:actionMenuItem aura:id="item1" label="Any"
click="{!c.updateTriggerLabel}"/>
            <ui:actionMenuItem aura:id="item2" label="Open" click="{!c.updateTriggerLabel}"
 disabled="true"/>
              <ui:actionMenuItem aura:id="item3" label="Closed"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item4" label="Closed Won"
click="{!c.updateTriggerLabel}"/>
          </ui:menuList>
</ui:menu>
```

このクライアント側コントローラは、メニュー項目がクリックされるとトリガの表示ラベルを更新します。

```
({
    updateTriggerLabel: function(cmp, event) {
        var triggerCmp = cmp.find("trigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
        }
    }
})
```

ドロップダウンメニューとそのメニュー項目は、デフォルトでは非表示になっています。この設定を変更するには、ui:menuList コンポーネントの visible 属性を true に設定します。メニュー項目は、ui:menuTriggerLink コンポーネントをクリックしたときにのみ表示されます。

メニューを開くトリガを使用するには、ui:menuTriggerLink コンポーネントを ui:menu 内にネストします。リスト項目には、ui:menuList コンポーネントを使用し、クライアント側コントローラアクションをトリガできる次のいずれかのリスト項目コンポーネントを含めます。

- ui:actionMenuItem - メニュー項目
- ui:checkboxMenuItem - 複数選択をサポートするチェックボックス
- ui:radioMenuItem - 単一選択をサポートするラジオ項目

これらのメニュー項目に区切り文字を追加するには、ui:menuItemSeparator を使用します。

次の例では、メニューを作成する複数の方法を示します。

```
<aura:component access="global">
    <aura:attribute name="status" type="String[]" default="Open, Closed, Closed Won, Any"/>

        <ui:menu>
            <ui:menuTriggerLink aura:id="trigger" label="Single selection with actionable
 menu item"/>
            <ui:menuList class="actionMenu" aura:id="actionMenu">
                <aura:iteration items="{!v.status}" var="s">
                    <ui:actionMenuItem label="{!s}" click="{!c.updateTriggerLabel}"/>
                </aura:iteration>
            </ui:menuList>
        </ui:menu>
        <hr/>
        <ui:menu>
        <ui:menuTriggerLink class="checkboxMenuLabel" aura:id="checkboxMenuLabel"
label="Multiple selection"/>
           <ui:menuList aura:id="checkboxMenu" class="checkboxMenu">
            <aura:iteration aura:id="checkbox" items="{!v.status}" var="s">
                <ui:checkboxMenuItem label="{!s}"/>
                </aura:iteration>
            </ui:menuList>
        </ui:menu>
         <p><ui:button class="checkboxButton" aura:id="checkboxButton"
press="{!c.getMenuSelected}" label="Check the selected menu items"/></p>
          <p><ui:outputText class="result" aura:id="result" value="Which items get
selected"/></p>
 <hr/>
         <ui:menu>
             <ui:menuTriggerLink class="radioMenuLabel" aura:id="radioMenuLabel"
label="Select a status"/>
             <ui:menuList class="radioMenu" aura:id="radioMenu">
                    <aura:iteration aura:id="radio" items="{!v.status}" var="s">
                     <ui:radioMenuItem label="{!s}"/>
                    </aura:iteration>
             </ui:menuList>
         </ui:menu>
        <p><ui:button class="radioButton" aura:id="radioButton"
press="{!c.getRadioMenuSelected}" label="Check the selected menu items"/></p>
         <p><ui:outputText class="radioResult" aura:id="radioResult" value="Which items
get selected"/> </p>
 <hr/>
 <div style="margin:20px;">
     <div style="display:inline-block;width:50%;vertical-align:top;">
         Combination menu items
         <ui:menu>
             <ui:menuTriggerLink aura:id="mytrigger" label="Select Menu Items"/>
             <ui:menuList>
                <ui:actionMenuItem label="Red" click="{!c.updateLabel}" disabled="true"/>

                 <ui:actionMenuItem label="Green" click="{!c.updateLabel}"/>
                 <ui:actionMenuItem label="Blue" click="{!c.updateLabel}"/>
                 <ui:actionMenuItem label="Yellow United" click="{!c.updateLabel}"/>
```

```
                <ui:menuItemSeparator/>
                <ui:checkboxMenuItem label="A"/>
                <ui:checkboxMenuItem label="B"/>
                <ui:checkboxMenuItem label="C"/>
                <ui:checkboxMenuItem label="All"/>
                <ui:menuItemSeparator/>
                <ui:radioMenuItem label="A only"/>
                <ui:radioMenuItem label="B only"/>
                <ui:radioMenuItem label="C only"/>
                <ui:radioMenuItem label="None"/>
            </ui:menuList>
        </ui:menu>
      </div>
 </div>
</aura:component>
```

```
({
    updateTriggerLabel: function(cmp, event) {
        var triggerCmp = cmp.find("trigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
        }
    },
    updateLabel: function(cmp, event) {
        var triggerCmp = cmp.find("mytrigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
        }
    },
    getMenuSelected: function(cmp) {
        var menuItems = cmp.find("checkbox");
        var values = [];
        for (var i = 0; i < menuItems.length; i++) {
            var c = menuItems[i];
            if (c.get("v.selected") === true) {
                values.push(c.get("v.label"));
            }
        }
        var resultCmp = cmp.find("result");
        resultCmp.set("v.value", values.join(","));
    },
    getRadioMenuSelected: function(cmp) {
        var menuItems = cmp.find("radio");
        var values = [];
        for (var i = 0; i < menuItems.length; i++) {
            var c = menuItems[i];
            if (c.get("v.selected") === true) {
                values.push(c.get("v.label"));
            }
```

```
        }
        var resultCmp = cmp.find("radioResult");
        resultCmp.set("v.value", values.join(","));
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:menuItem

ui:menuList コンポーネント内の UI メニュー項目。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
hideMenuAfterSelected	Boolean	メニュー項目の選択後にメニューを非表示にするには、true に設定します。	
label	String	コンポーネントに表示されるテキスト。	
selected	Boolean	メニュー項目の状況。True はこのメニュー項目が選択されていることを示し、False は選択されていないことを示します。	
type	String	メニュー項目の具体的な種別。有効な値は、「action」、「checkbox」、「radio」、「separator」、または ns:xxxxmenuItem などの名前空間を含むコンポーネント記述子です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。

ui:menuItemSeparator

メニュー項目を分けるメニュー区切り文字 (ui:radioMenuItem など)。ui:menuList コンポーネントで使用されます。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:menuList

メニュー項目が含まれるメニューコンポーネント。

このコンポーネントは ui:menu コンポーネント内にネストされ、ui:menuTriggerLink コンポーネントと一緒に使用できます。メニュートリガをクリックすると、コンテナとメニュー項目が表示されます。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Click me to display menu items"/>
    <ui:menuList class="actionMenu" aura:id="actionMenu">
        <ui:actionMenuItem aura:id="item1" label="Item 1" click="{!c.doSomething}"/>
        <ui:actionMenuItem aura:id="item2" label="Item 2" click="{!c.doSomething}"/>
        <ui:actionMenuItem aura:id="item3" label="Item 3" click="{!c.doSomething}"/>
        <ui:actionMenuItem aura:id="item4" label="Item 4" click="{!c.doSomething}"/>
   </ui:menuList>
</ui:menu>
```

ui:menuList には、これらのコンポーネントを格納でき、クリックするとクライアント側のコントローラが実行されます。

- ui:actionMenuItem
- ui:checkboxMenuItem
- ui:radioMenuItem
- ui:menuItemSeparator

詳細は、ui:menuを参照してください。

属性

属性名	属性型	説明	必須項目
autoPosition	Boolean	下に十分な表示スペースがない場合は、ポップアップターゲットを上に移動します。注意:autoPosition が false に設定さ	

属性名	属性型	説明	必須項目
		れていても、ポップアップはメニューをトリガと相対的に位置付けます。デフォルトの位置指定を上書きするには、manualPosition 属性を使用します。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
closeOnClickOutside	Boolean	ユーザがターゲットの外側をクリックまたはタップしたら、ターゲットを閉じます。	
closeOnTabKey	Boolean	Tab キーによってターゲットリストを閉じるかどうかを示します。	
curtain	Boolean	ターゲットの下にフロート表示を適用するかどうか。	
menuItems	List	Java クラスのインスタンス aura.components.ui.MenuItem を使用して明示的に設定されたメニュー項目のリスト。	
visible	Boolean	メニューの表示設定を制御します。デフォルトはfalseで、メニューは非表示になります。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
menuExpand	COMPONENT	メニューリストが表示されると起動されるイベント。
menuSelect	COMPONENT	ユーザがメニュー項目を選択すると起動されるイベント。
menuCollapse	COMPONENT	メニューリストが折りたたまれると起動されるイベント。
menuFocusChange	COMPONENT	メニューリストのフォーカスが 1 つのメニュー項目から別のメニュー項目に変更されると起動されるイベント。

ui:menuTrigger

メニューを展開したり折りたたんだりするクリック可能なリンク。ui:menu のリンクを作成するには、代わりに ui:menuTriggerLink を使用します。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
label	String	コンポーネントに表示されるテキスト。	
title	String	このコンポーネントにマウスポインタが置かれたときにツールチップとして表示されるテキスト。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
menuTriggerPress	COMPONENT	トリガがクリックされると起動されるイベント。

ui:menuTriggerLink

ui:menu で使用されるドロップダウンメニューをトリガするリンク。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
label	String	コンポーネントに表示されるテキスト。	

属性名	属性型	説明	必須項目
title	String	このコンポーネントにマウスポインタが置かれたときにツールチップとして表示されるテキスト。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがトリガから離れると起動されるイベント。
focus	COMPONENT	ユーザがトリガにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。
menuTriggerPress	COMPONENT	トリガがクリックされると起動されるイベント。

ui:message

さまざまな重要度のメッセージを表します。

severity 属性は、メッセージの重要度を示し、メッセージを表示するときに使用するスタイルを決定します。closable 属性が true に設定されている場合、x 記号を押すとメッセージを消去できます。

次の例では、消去できる確認メッセージを表示します。

```
<ui:message title="Confirmation" severity="confirm" closable="true">
      This is a confirmation message.
  </ui:message>
```

次の例では、さまざまな重要度のメッセージを表示します。

```

<aura:component access="global">
 <ui:message title="Confirmation" severity="confirm" closable="true">
   This is a confirmation message.
 </ui:message>
 <ui:message title="Information" severity="info" closable="true">
   This is a message.
 </ui:message>
 <ui:message title="Warning" severity="warning" closable="true">
   This is a warning.
 </ui:message>
 <ui:message title="Error" severity="error" closable="true">
   This is an error message.
 </ui:message>

</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
closable	Boolean	クリックされるとアラートを閉じる [x] を表示するかどうかを指定します。デフォルト値は「false」です。	
severity	String	メッセージの重要度。有効な値は、message (デフォルト)、confirm、info、warning、error です。	
title	String	メッセージのタイトルテキスト。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputCheckbox

チェックボックスをオンまたはオフの状態で表示します。

ui:outputCheckbox コンポーネントは、HTML img タグとして表示されるチェックボックスを表します。このコンポーネントを ui:inputCheckbox と共に使用すると、ユーザがチェックボックスをオンまたはオフにできます。チェックボックスをオンまたはオフにするには、value 属性を true または false に設定します。チェックボックスを表示する場合、属性値を使用して ui:outputCheckbox コンポーネントにバインドできます。

```
<aura:attribute name="myBool" type="Boolean" default="true"/>
<ui:outputCheckbox value="{!v.myBool}"/>
```

前の例によって次の HTML が表示されます。

```
<img class="checked uiImage uiOutputCheckbox" alt="checkbox checked" src="path/to/checkbox">
```

次の例は、ui:inputCheckbox コンポーネントを使用する方法を示します。

```

<aura:component>
 <aura:attribute name="myBool" type="Boolean" default="true"/>
 <ui:inputCheckbox aura:id="checkbox" label="Select?" change="{!c.onCheck}"/>
 <p>Selected:</p>
 <p><ui:outputText class="result" aura:id="checkResult" value="false" /></p>
 <p>The following checkbox uses a component attribute to bind its value.</p>
```

```
 <ui:outputCheckbox aura:id="output" value="{!v.myBool}"/>
</aura:component>
```

```
({
  onCheck: function(cmp, evt) {
   var checkCmp = cmp.find("checkbox");
   resultCmp = cmp.find("checkResult");
   resultCmp.set("v.value", ""+checkCmp.get("v.value"));

  }
})
```

属性

属性名	属性型	説明	必須項目
altChecked	String	チェックボックスがオンの場合の代替テキストによる説明。デフォルト値は「True」です。	
altUnchecked	String	チェックボックスがオフの場合の代替テキストによる説明。デフォルト値は「False」です。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
value	Boolean	チェックボックスをオンにするかどうかを指定します。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。

イベント名	イベントタイプ	説明
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputCurrency

通貨をデフォルトまたは指定形式 (特定の通貨コードまたは小数点の使用など) で表示します。

ui:outputCurrency コンポーネントは、数値を HTML span タグでラップされた通貨として表します。このコンポーネントは、数値を通貨として取り込む ui:inputCurrency と共に使用できます。通貨を表示する場合、属性値を使用して ui:outputCurrency コンポーネントにバインドできます。

```
<aura:attribute name="myCurr" type="Decimal" default="50000"/>
<ui:outputCurrency aura:id="curr" value="{!v.myCurr}"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputCurrency">$50,000.00</span>
```

ブラウザのロケールを上書きするには、currencySymbol 属性を使用します。

```
<aura:attribute name="myCurr" type="Decimal" default="50" currencySymbol="£"/>
```

形式を指定して上書きすることもできます。

```
var curr = cmp.find("curr");
curr.set("v.format", '£#,###.00');
```

次の例は、ui:inputCurrency コンポーネントのデータをバインドする方法を示します。

```

<aura:component>
    <aura:attribute name="myCurrency" type="integer" default="50"/>
   <ui:inputCurrency aura:id="amount" label="Amount" class="field" value="{!v.myCurrency}"
 updateOn="keyup"/>
 You entered: <ui:outputCurrency value="{!v.myCurrency}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付されるCSSスタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
currencyCode	String	文字列として指定されたISO4217通貨コード(「USD」など)。	
currencySymbol	String	文字列として指定された通貨記号。	
format	String	数値の形式。たとえば、format=".00" は、小数点以下2桁の数値を表示します。指定されない場合は、ブラウザのロケールに基づくデフォルトの形式が使用されます。	
value	Decimal	Decimal 型で定義された通貨の出力値。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputDate

ユーザのロケールに基づいたデフォルトまたは指定形式で日付を表示します。

ui:outputDate コンポーネントは、YYYY-MM-DD形式の日付出力を表し、HTML span タグでラップされます。このコンポーネントは、日付入力を取り込む ui:inputDate と共に使用できます。ui:outputDate は、ブ

ラウザのロケール情報を取得し、それに従って日付を表示します。日付を表示する場合、属性値を使用して ui:outputDate コンポーネントにバインドできます。

```
<aura:attribute name="myDate" type="Date" default="2014-09-29"/>
<ui:outputDate value="{!v.myDate}"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputDate">Sep 29, 2014</span>
```

次の例は、ui:inputDate コンポーネントのデータをバインドする方法を示します。

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDate aura:id="expdate" label="Today's Date" class="field" value="{!v.today}"
 displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputDate aura:id="oDate" value="" />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
    },

    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');
        var expdate = component.find("expdate").get("v.value");

        var oDate = component.find("oDate");
        oDate.set("v.value", expdate);

    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
format	String	value 属性の日時の形式設定に使用される文字列 (パターン文字は java.text.SimpleDateFormat で定義されます)。	
langLocale	String	非推奨。日付値の形式設定に使用される言語ロケール。ロケール値プロバイダによって提供される値のみを使用できます。これ以外の場合は、$Locale.langLocale の値に戻ります。この属性は今後のリリースで廃止されます。	
value	String	日付の出力値。ISO-8601 形式 (YYYY-MM-DD) の日付文字列である必要があります。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputDateTime

ユーザのロケールに基づいた指定またはデフォルト形式で日時を表示します。

ui:outputDateTime コンポーネントは、HTML span タグでラップされた日時出力を表します。このコンポーネントは、日付入力を取り込む ui:inputDateTime と共に使用できます。ui:outputDateTime は、ブラウ

ザのロケール情報を取得し、それに従って日付を表示します。日時を表示する場合、属性値を使用して `ui:outputDateTime` コンポーネントにバインドできます。

```
<aura:attribute name="myDateTime" type="Date" default="2014-09-29T00:17:08z"/>
<ui:outputDateTime value="{!v.myDateTime}"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputDateTime">Sep 29, 2014 12:17:08 AM</span>
```

次の例は、`ui:inputDateTime` コンポーネントのデータをバインドする方法を示します。

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDateTime aura:id="today" label="Time" class="field" value=""
displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

    <div aura:id="msg" class="hide">
  You entered: <ui:outputDateTime aura:id="oDateTime" value=""  />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
    },

    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var todayVal = component.find("today").get("v.value");
        var oDateTime = component.find("oDateTime");
        oDateTime.set("v.value", todayVal);

    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	

属性名	属性型	説明	必須項目
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
format	String	value 属性の日時の形式設定に使用される文字列 (パターン文字は java.text.SimpleDateFormat で定義されます)。	
langLocale	String	非推奨。日付値の形式設定に使用される言語ロケール。ロケール値プロバイダによって提供される値のみを使用できます。これ以外の場合は、$Locale.langLocale の値に戻ります。この属性は今後のリリースで廃止されます。	
timezone	String	タイムゾーン ID (America/Los_Angeles など)。	
value	String	日時を表す ISO8601 形式の文字列。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputEmail

HTML アンカー (<a>) 要素内にメールアドレスを表示します。先頭および末尾の空白は削除されます。

ui:outputEmail コンポーネントは、HTML span タグでラップされたメール出力を表します。このコンポーネントは、メール入力を取り込む ui:inputEmail と共に使用できます。メール出力は HTML アンカー要素で

ラップされ、mailto が自動的に付加されます。次の例は、ui:outputEmail コンポーネントの簡単な設定です。

```
<ui:outputEmail value="abc@email.com"/>
```

前の例によって次の HTML が表示されます。

```
<span><a href="mailto:abc@email.com" class="uiOutputEmail">abc@email.com</a></span>
```

次の例は、ui:inputEmail コンポーネントのデータをバインドする方法を示します。

```
<aura:component>
    <ui:inputEmail aura:id="email" label="Email" class="field" value="manager@email.com"/>

     <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
     You entered: <ui:outputEmail aura:id="oEmail" value="Email" />
 </div>

</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var email = component.find("email").get("v.value");
        var oEmail = component.find("oEmail");
        oEmail.set("v.value", email);

    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
value	String	メールの出力値。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputNumber

デフォルトまたは指定形式で数値を表示します。最大 18 桁の整数部をサポートします。

ui:outputNumber コンポーネントは、HTML span タグとして表示される数値出力を表します。このコンポーネントは、数値入力を取り込む ui:inputNumber と共に使用できます。ui:outputNumber は、ロケール情報を取得し、それに基づいた 10 進数形式で表示します。数値を表示する場合、属性値を使用して ui:outputNumber コンポーネントにバインドできます。

```
<aura:attribute name="myNum" type="Decimal" default="10.10"/>
<ui:outputNumber value="{!v.myNum}" format=".00"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputNumber">10.10</span>
```

次の例は、ui:intputNumber コンポーネントの値を取得し、入力を検証し、ui:outputNumber を使用して値を表示します。

```

<aura:component>
    <aura:attribute name="myNumber" type="integer" default="10"/>
 <ui:inputNumber label="Enter a number: " value="{!v.myNumber}" updateOn="keyup"/> <br/>
    <ui:outputNumber value="{!v.myNumber}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
format	String	数値の形式。たとえば、format=".00" は、小数点以下 2 桁の数値を表示します。指定されていない場合は、ロケールのデフォルト形式が使用されます。	
value	decimal	このコンポーネントと共に表示される数値。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputPhone

電話番号を URL リンク形式で表示します。

ui:outputPhone コンポーネントは、HTML span タグでラップされた電話番号出力を表します。このコンポーネントは、電話番号入力を取り込む ui:inputPhone と共に使用できます。次の例は、ui:outputPhone コンポーネントの簡単な設定です。

```
<ui:outputPhone value="415-123-4567"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputPhone">415-123-4567</span>
```

モバイルデバイスで表示すると、この例はアクション可能なリンクとして表示されます。

```
<span class="uiOutputPhone">
    <a href="tel:415-123-4567">415-123-4567</a>
</span>
```

次の例は、ui:inputPhone コンポーネントのデータをバインドする方法を示します。

```

<aura:component>
   <ui:inputPhone aura:id="phone" label="Phone Number" class="field" value="415-123-4567"
 />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputPhone aura:id="oPhone" value="" />
 </div>
</aura:component>
```

```

({

    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var phone = component.find("phone").get("v.value");
        var oPhone = component.find("oPhone");
        oPhone.set("v.value", phone);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	

属性名	属性型	説明	必須項目
value	String	このコンポーネントと共に表示される電話番号。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputRichText

value 属性の指定に従って、パラグラフ、画像、ハイパーリンクなど、タグを含む書式設定済みテキストを表示します。

ui:outputRichText コンポーネントは、書式設定済みテキストを表し、lightning:inputRichText または ui:inputRichText コンポーネントからの入力の表示に使用できます。ui:inputRichText は、LockerService が有効化されている場合にサポートされなくなったため、lightning:inputRichText を使用することをお勧めします。ui:outputRichText は、書式設定済みテキストを表示します。たとえば、アンカータグで囲まれた URL やメールアドレスはハイパーリンクとして表示されます。

次の例では、太字テキストを設定して、値を lightning:inputRichText および ui:outputRichText コンポーネントにバインドします。

```
<aura:component>
    <aura:attribute name="myVal" type="String" />
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>

    <lightning:inputRichText value="{!v.myVal}"/>
    <ui:outputRichText value="{!v.myVal}"/>
</aura:component>
```

初期化時に、lightning:inputRichText と ui:outputRichText の両方のコンポーネントに値が設定されます。

```
({
    init: function(cmp) {
        cmp.set('v.myVal', '<b>Hello!</b>');
    }
})
```

ui:outputRichText でサポートされている HTML タグは、a、b、br、big、blockquote、caption、cite、code、col、colgroup、del、div、em、h1、h2、h3、hr、i、img、ins、kbd、li、ol、p、param、pre、q、s、samp、small、span、strong、sub、sup、table、tbody、td、tfoot、th、thead、tr、tt、u、ul、var、strike です。

サポートされている HTML 属性は、accept、action、align、alt、autocomplete、background、bgcolor、border、cellpadding、cellspacing、checked、cite、class、clear、color、cols、colspan、coords、datetime、default、dir、disabled、download、enctype、face、for、headers、height、hidden、high、href、hreflang、id、ismap、label、lang、list、loop、low、max、maxlength、media、method、min、multiple、name、noshade、novalidate、nowrap、open、optimum、pattern、placeholder、poster、preload、pubdate、radiogroup、readonly、rel、required、rev、reversed、rows、rowspan、spellcheck、scope、selected、shape、size、span、srclang、start、src、step、style、summary、tabindex、target、title、type、usemap、valign、value、width、xmlns です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
linkify	Boolean	テキスト内の URL をハイパーリンクとして表示するように設定しているかどうかを示します。	
value	String	出力に使用される書式設定済みテキスト。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputText

value 属性の指定に従ってテキストを表示します。

ui:outputText コンポーネントは、HTML span タグでラップされたテキスト出力を表します。このコンポーネントは、テキスト入力を取り込む ui:inputText と共に使用できます。テキストを表示する場合、属性値を使用して ui:outputText コンポーネントにバインドできます。

```
<aura:attribute name="myText" type="String" default="some string"/>
<ui:outputText value="{!v.myText}" />
```

前の例によって次の HTML が表示されます。

```
<span dir="ltr" class="uiOutputText">
    some string
</span>
```

次の例は、ui:inputText コンポーネントのデータをバインドする方法を示します。

```

<aura:component>
    <aura:attribute name="myText" type="string" default="Hello there!"/>
 <ui:inputText label="Enter some text" class="field" value="{!v.myText}" updateOn="click"/>

 You entered: <ui:outputText value="{!v.myText}"/>
</aura:component>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
title	String	追加情報をフロート表示テキストとして表示します。	
value	String	このコンポーネントと共に表示されるテキスト。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputTextArea

value 属性の指定に従ってテキストエリアを表示します。

ui:outputTextArea コンポーネントは、HTML span タグでラップされたテキスト出力を表します。このコンポーネントは、複数のテキスト入力を取り込む ui:inputTextArea と共に使用できます。テキストを表示す

る場合、属性値を使用して `ui:outputTextArea` コンポーネントにバインドできます。`ui:outputTextArea` コンポーネントは、URL およびメールアドレスをハイパーリンクとして表示します。

```
<aura:attribute name="myTextArea" type="String" default="some string"/>
<ui:outputTextArea value="{!v.myTextArea}"/>
```

前の例によって次の HTML が表示されます。

```
<span class="uiOutputTextArea">some string</span>
```

次の例は、`ui:inputTextArea` コンポーネントのデータをバインドする方法を示します。

```
<aura:component>
     <ui:inputTextArea aura:id="comments" label="Comments"  value="My comments" rows="5"/>

      <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

      <div aura:id="msg" class="hide">
  You entered: <ui:outputTextArea aura:id="oTextarea" value=""/>
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var comments = component.find("comments").get("v.value");
        var oTextarea = component.find("oTextarea");
        oTextarea.set("v.value", comments);
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
linkify	Boolean	テキスト内の URL をハイパーリンクとして表示するように設定しているかどうかを示します。	
value	String	表示するテキスト。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:outputURL

value 属性の指定に従って URL へのリンクを表示します。テキスト (label 属性) および画像が指定されていれば一緒に表示します。

ui:outputURL コンポーネントは、HTML a タグでラップされた URL を表します。このコンポーネントは、URL 入力を取り込む ui:inputURL と共に使用できます。URL を表示する場合、属性値を使用し、ui:outputURL コンポーネントにバインドできます。

```
<aura:attribute name="myURL" type="String" default="http://www.google.com"/>
<ui:outputURL value="{!v.myURL}" label="Search"/>
```

前の例によって次の HTML が表示されます。

```
<a href="http://www.google.com" class="uiOutputURL">Search</a>
```

次の例は、ui:inputURL コンポーネントのデータをバインドする方法を示します。

```

<aura:component>
    <ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>

     <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>
  <div aura:id="msg" class="hide">
    You entered: <ui:outputURL aura:id="oURL" value=""/>
```

```
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var cmpMsg = component.find("msg");
     $A.util.removeClass(cmpMsg, 'hide');

        var url = component.find("url").get("v.value");
        var oURL = component.find("oURL");
        oURL.set("v.value", url);
        oURL.set("v.label", url);
    }
})
```

属性

属性名	属性型	説明	必須項目
alt	String	代替テキストによる画像の説明 (表示ラベルがない場合に使用)	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
iconClass	String	アイコンまたは画像の表示に使用される CSS スタイル。	
label	String	コンポーネントに表示されるテキスト。	
target	String	このコンポーネントが表示される場所。有効な値は、_blank、_parent、_self、_top です。	
title	String	このコンポーネントにマウスポインタが置かれたときにツールチップとして表示されるテキスト。	
value	String	リンク先のページの URL。	はい

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。

ui:radioMenuItem

いずれか1つのみを選択する必要があり、アクションの呼び出しに使用可能であることを示すラジオボタンを含むメニュー項目。このコンポーネントは、ui:menu コンポーネントでネストされます。

ui:radioMenuItem コンポーネントは、単一選択のメニューリスト項目を表します。リストの値を反復処理してメニュー項目を表示するには、aura:iteration を使用します。ui:menuTriggerLink コンポーネントは、メニュー項目を表示したり、非表示にしたりします。

```
<aura:attribute name="status" type="String[]" default="Open, Closed, Closed Won, Any"/>
    <ui:menu>
        <ui:menuTriggerLink class="radioMenuLabel" aura:id="radioMenuLabel" label="Select
 a status"/>
        <ui:menuList class="radioMenu" aura:id="radioMenu">
            <aura:iteration items="{!v.status}" var="s">
                <ui:radioMenuItem label="{!s}"/>
            </aura:iteration>
        </ui:menuList>
    </ui:menu>
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
disabled	Boolean	コンポーネントを無効な状態で表示するかどうかを指定します。デフォルト値は「false」です。	
hideMenuAfterSelected	Boolean	メニュー項目の選択後にメニューを非表示にするには、true に設定します。	
label	String	コンポーネントに表示されるテキスト。	
selected	Boolean	メニュー項目の状況。True はこのメニュー項目が選択されていることを示し、False は選択されていないことを示します。	
type	String	メニュー項目の具体的な種別。有効な値は、「action」、「checkbox」、「radio」、「separator」、または ns:xxxxmenuItem などの名前空間を含むコンポーネント記述子です。	

イベント

イベント名	イベントタイプ	説明
dblclick	COMPONENT	ユーザがコンポーネントをダブルクリックすると起動されるイベント。
mouseover	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
mouseout	COMPONENT	ユーザがコンポーネントからマウスポインタを移動すると起動されるイベント。
mouseup	COMPONENT	ユーザがコンポーネント上でマウスボタンを放すと起動されるイベント。
mousemove	COMPONENT	ユーザがコンポーネントにマウスポインタを重ねると起動されるイベント。
click	COMPONENT	ユーザがコンポーネントをクリックすると起動されるイベント。

イベント名	イベントタイプ	説明
mousedown	COMPONENT	ユーザがコンポーネント上でマウスボタンをクリックすると起動されるイベント。
select	COMPONENT	ユーザが何らかのテキストを選択すると起動されるイベント。
blur	COMPONENT	ユーザがコンポーネントから離れると起動されるイベント。
focus	COMPONENT	ユーザがコンポーネントにフォーカスを移動すると起動されるイベント。
keypress	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すまたは押したままにすると起動されるイベント。
keyup	COMPONENT	ユーザがコンポーネント上でキーボードのキーを放すと起動されるイベント。
keydown	COMPONENT	ユーザがコンポーネント上でキーボードのキーを押すと起動されるイベント。

ui:scrollerWrapper

Salesforce1 でのネイティブスクロールを有効にするコンテナを作成します。

ui:scrollerWrapper は、Salesforce1 でのネイティブスクロールを有効にするコンテナを作成します。このコンポーネントを使用すると、コンテナ内の複数のスクローラをネストできます。class 属性を使用して、コンテナの高さと幅を定義します。スクロールを有効にするには、コンテンツよりも低い高さを指定します。

次の例では、300 ピクセルの高さのスクロール可能な領域を作成します。

```
<aura:component>
    <ui:scrollerWrapper class="scrollerSize">
        <!--Scrollable content here -->
    </ui:scrollerWrapper>
</aura:component>

/** CSS **/
.THIS.scrollerSize {
    height: 300px;
}
```

Lightning Design System の scrollable クラスは、モバイルデバイスのネイティブスクロールと互換性がありません。Salesforce1 でスクロールを有効にする必要がある場合は ui:scrollerWrapper を使用します。

使用上の考慮事項

モバイルデバイスの Google Chrome では、border-radius CSS プロパティがゼロ以外の値に設定されていると、ネストされた ui:scrollerWrapper コンポーネントはスクロールできません。この場合、スクロールを有効にするには、外側の ui:scrollerWrapper コンポーネントの border-radius をゼロ以外の値に設定します。

次に例を示します。

```
<aura:component>
    <ui:scrollerWrapper class="outerScroller">
        <!-- Scrollable content here -->
        <ui:scrollerWrapper class="innerScroller">
            <!-- Scrollable content here -->
        </ui:scrollerWrapper>
        <!-- Scrollable content here -->
    </ui:scrollerWrapper>
</aura:component>

/** CSS **/
.THIS.outerScroller {
    /* fix innerScroller not scrollable */
    border-radius: 1px;
}
.THIS.innerScroller {
    /* make innerScroller rounded */
    border-radius: 10px;
}
```

メソッド

このコンポーネントは、次のメソッドをサポートします。

`scrollTo(destination, xcoord, ycoord)`: 指定された場所までコンテンツをスクロールします。

- `destination` (String): ターゲットの場所。有効な値: custom、top、bottom、left、right。カスタムの場所の場合、xcoord と ycoord を使用してターゲットの場所を決定します。
- `xcoord` (Integer): カスタムの場所の X 座標。デフォルト値は 0 です。
- `ycoord` (Integer): カスタムの場所の Y 座標。デフォルト値は 0 です。

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	外側の要素に適用される CSS クラス。このスタイルは、コンポーネントで出力される基本クラスに追加されます。	

ui:spinner

実際のコンポーネントのボディを読み込み中に使用する読み込みスピナー。

スピナーを切り替えるには、`get("e.toggle")` を使用し、`isVisible` パラメータを `true` または `false` に設定して、イベントを起動します。

次の例では、切り替え可能なスピナーを表示します。

```
<aura:component access="global">
 <ui:spinner aura:id="spinner"/>
 <ui:button press="{!c.toggleSpinner}" label="Toggle Spinner" />
</aura:component>
```

```
({
    toggleSpinner: function(cmp) {
        var spinner = cmp.find('spinner');
        var evt = spinner.get("e.toggle");

    if(!$A.util.hasClass(spinner, 'hideEl')){
        evt.setParams({ isVisible : false });
     }
    else {
        evt.setParams({ isVisible : true });
 }
    evt.fire();
    }
})
```

属性

属性名	属性型	説明	必須項目
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
class	String	コンポーネントに添付される CSS スタイル。このスタイルは、コンポーネントで出力される基本スタイルに追加されます。	
isVisible	Boolean	このスピナーを表示するかどうかを指定します。デフォルトは true です。	

イベント

イベント名	イベントタイプ	説明
toggle	COMPONENT	スピナーが切り替えられると起動されるイベント。

wave:waveDashboard

このコンポーネントを使用して、Salesforce Analytics ダッシュボードを Lightning Experience ページに追加します。

属性

属性名	属性型	説明	必須項目
accessToken	String	Salesforce にログインすることによって取得される有効なアクセストークン。コンポーネントが非 Salesforce ドメインで Lightning Out によって使用される場合に有用です。	
body	Component[]	コンポーネントのボディ。マークアップでは、これはタグのボディに含まれるすべてを指します。	
dashboardId	String	ダッシュボードの一意の ID。ダッシュボードの URL からダッシュボードの ID (0FK から始まる 18 文字のコード) を取得できます。また、API を使用してこの ID を要求することもできます。開発者名の代わりにこの属性を使用できますが、名前が設定されている場合にこれを含めることはできません。2 つのいずれかは必要です。	
developerName	String	ダッシュボードの一意の開発者名。開発者名は API で要求できます。ダッシュボード IDの代わりにこの属性を使用できますが、IDが設定されている場合にこれを含めることはできません。2 つのいずれかは必要です。	
filter	String	実行時に埋め込みダッシュボードに選択または条件を追加します。filter属性はJSONを使用して設定されます。ディメンションで絞り込む場合、{'datasets' : {'dataset1': [{'fields': ['field1'], 'selection': ['$value1', '$value2']}, {'fields': ['field2'], 'filter': { 'operator': 'operator1', 'values': ['$value3', '$value4']}}]}} の構文を使用します。基準で絞り込む場合、{'datasets' : {'dataset1': [{'fields': ['field1'], 'selection': ['$value1', '$value2']}, {'fields': ['field2'], 'filter': { 'operator': 'operator1', 'values': [[$value3]]}}]}} の構文を使用します。選択オプションを使用すると、ダッシュボードにそのすべてのデータが表示され、指定したディメンション値が強調表示されます。検索オプションを使用すると、ダッシュボードには絞り込まれたデータのみが表示されます。詳細は、https://help.salesforce.com/articleView?id=bi_embed_lightning.htm を参照してください。	
height	Integer	ダッシュボードの高さ (ピクセル単位) を指定します。	
hideOnError	Boolean	エラーのあるダッシュボードをユーザに表示するかどうかを制御します。この属性が true に設定されている場合、ダッシュボードにエラーがあってもページには表示されません。false に設定すると、ダッシュボードは表示されますが、データは表示されません。ユーザにダッシュボードへのアクセス権がないか削除されていると、エラーが発生する可能性があります。	

属性名	属性型	説明	必須項目
openLinksInNewWindow	Boolean	false にすると、その他のダッシュボードへのリンクが同じウィンドウで開きます。	
recordId	String	コンポーネントが表示されるコンテキスト内の現在のエンティティの ID。	
showHeader	Boolean	true の場合、ダッシュボードにはダッシュボード情報とコントロールを含むヘッダーバーが表示されます。false の場合、ダッシュボードはヘッダーバーなしで表示されます。showSharing または showTitle が true の場合、ヘッダーバーは自動的に表示されます。	
showSharing	Boolean	trueにすると、ダッシュボードが共有可能な場合、ダッシュボードに [共有] アイコンが表示されます。false にすると、ダッシュボードに [共有] アイコンは表示されません。ダッシュボードに [共有] アイコンを表示するためのサポートされる最小のフレームサイズは 800 × 612 ピクセルです。	
showTitle	Boolean	true にすると、ダッシュボードのタイトルがダッシュボードの上に表示されます。false の場合、タイトルなしでダッシュボードが表示されます。	

メッセージングコンポーネントの参照

メッセージングコンポーネントには、関連情報をユーザに伝える通知とフロート表示が含まれます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

lightning:notificationsLibrary

`lightning:notificationsLibrary` は、アプリケーションでメッセージを簡単に表示できるようにします。このコンポーネントでは、API バージョン 41.0 以降が必要です。このコンポーネントは、Lightning Experience、Salesforce1、および Lightning コミュニティでのみサポートされています。

メッセージを通知およびトーストで表示できます。通知は、ユーザにシステム関連の問題と更新をアラートで伝えます。トーストでは、ユーザがアクションを実行した後に、フィードバックを提供でき、確認メカニズムとして使用できます。通知をトリガするコンポーネントに `<lightning:notificationsLibrary aura:id="notifLib"/>` タグを 1 つ含めます。`aura:id` は、一意のローカル ID です。1 つのタグのみで複数の通知に対応します。

通知

通知はユーザのワークフローを中断し、ページのその他の要素をすべてブロックします。ユーザがアプリケーションを操作できる状態に戻るには、通知を確認する必要があります。その点を踏まえて、通知は慎重に使用

してください。レコードを削除する前段階といった、ユーザのアクションを確認する用途には適しません。現在、通知を閉じるには [OK] ボタンしかサポートされていません。

Something has gone wrong!

Unfortunately, there was a problem updating the record. Make sure you're connected and try again.

OK

ボタンを含む例を次に示します。クリックされると、ボタンは `error` バリエーションを含む通知を表示します。

```
<aura:component>
    <lightning:notificationsLibrary aura:id="notifLib"/>
    <lightning:button name="notice" label="Show Notice" onclick="{!c.handleShowNotice}"/>
</aura:component>
```

クライアント側コントローラは、通知を表示します。

```
({
    handleShowNotice : function(component, event, helper) {
        component.find('notifLib').showNotice({
            "variant": "error",
            "header": "Something has gone wrong!",
            "message": "Unfortunately, there was a problem updating the record.",
            closeCallback: function() {
                alert('You closed the alert!');
            }
        });
    }
})
```

通知を作成および表示するには、`component.find('notifLib').showNotice()` を使用して通知属性を渡します。`notifLib` は、`lightning:notificationsLibrary` インスタンスの `aura:id` に整合します。

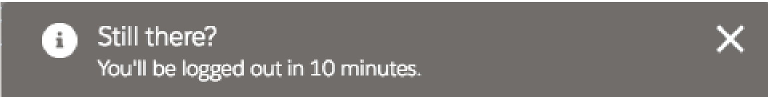
属性

属性名	属性型	説明	必須項目
header	String	通知の最上部に表示されるヘッダー。	
title	String	太字で表示される通知のタイトル。	
message	String	通知本文のメッセージ。 は改行、アンカーはテキストリンクに置換されます。	
variant	String	通知の外観を変更します。使用できるバリエーションは、info、warning、error です。この値のデフォルトは info です。	

属性名	属性型	説明	必須項目
closeCallback	Function	通知が閉じられたときにコールされるコールバック。	

トースト

トーストは、通知よりは操作に影響せず、レコードを作成した後のように、アクション後にユーザにフィードバックを提供する場合に適しています。トーストは、閉じることも、定義した期間が経過するまで表示したままにすることもできます。



ボタンを含む例を次に示します。ボタンをクリックすると、info バリエーションを含むトーストが表示されます。これは、終了ボタン (右上の X) を押すまで表示されたままになります。

```
<aura:component>
    <lightning:notificationsLibrary aura:id="notifLib"/>
    <lightning:button name="toast" label="Show Toast" onclick="{!c.handleShowToast}"/>
</aura:component>
```

クライアント側コントローラは、トーストを表示します。

```
({
    handleShowToast : function(component, event, helper) {
        component.find('notifLib').showToast({
            "title": "Notif library Success!",
            "message": "The record has been updated successfully."
        });
    }
})
```

トーストを作成および表示するには、component.find('notifLib').showToast() を使用してトースト属性を渡します。notifLib は、lightning:notificationsLibrary インスタンスの aura:id に整合します。

属性

属性名	属性型	説明	必須項目
title	String	ヘッダーとして表示されるトーストのタイトル。	
message	String	メッセージを表す文字列。{0} ... {N} の形式でプレースホルダを含めることができます。このプレースホルダは、メッセージデータのアクションリンクに置換されます。	
messageData	Object	トーストメッセージテンプレートを置換するインラインアクションリンクの配列。	

属性名	属性型	説明	必須項目
variant	String	トーストの外観を変更します。使用できるバリエーションは、info、success、warning、error です。この値のデフォルトは info です。	
mode	String	トーストの表示期間を指定します。デフォルトは dismissable です。有効なモードは次のとおりです。 • dismissable: 終了ボタンが押されるか、3 秒間経過するまで、表示され続けます。 • pester:終了ボタンがクリックされるまで表示され続けます。 • sticky: 3 秒間表示されます。	

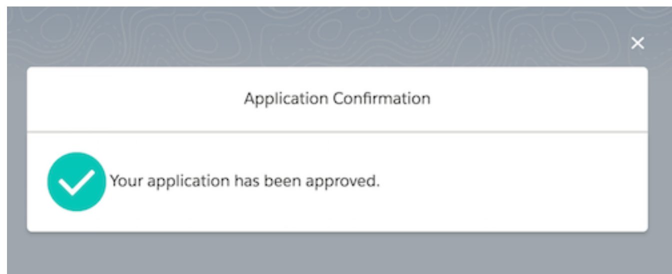
lightning:overlayLibrary

lightning:overlayLibrary は、関連情報とフィードバックを簡単に表示できるようにします。このコンポーネントでは、API バージョン 41.0 以降が必要です。このコンポーネントは、Lightning Experience、Salesforce1、および Lightning コミュニティでのみサポートされています。

メッセージをモーダルおよびポップオーバーで表示できます。モーダルでは、アプリケーションの前面にダイアログを表示し、ユーザのワークフローを中断してメッセージに注意を引き付けます。ポップオーバーでは、参照要素にマウスポインタが置かれると、関連情報を表示します。メッセージをトリガするコンポーネントに <lightning:overlayLibrary aura:id="overlayLib"/> タグを 1 つ含めます。aura:id は、一意のローカル ID です。1 つのタグのみで複数のメッセージに対応します。

モーダル

モーダルは、終了しない限り、ページのその他の要素をすべてブロックします。ユーザがアプリケーションを操作できる状態に戻るには、モーダルを確認する必要があります。モーダルは、ボタンまたはリンクのクリックを含む、ユーザ操作によってトリガされます。モーダルのヘッダー、本文、フッターをカスタマイズできます。Esc キーを押すか、終了ボタンをクリックすると、モーダルは閉じます。



ボタンを含む例を次に示します。クリックされると、ボタンはカスタム本文を含むモーダルを表示します。

```
<aura:component>
    <lightning:overlayLibrary aura:id="overlayLib"/>
```

```
    <lightning:button name="modal" label="Show Modal" onclick="{!c.handleShowModal}"/>
</aura:component>
```

クライアント側コントローラは、モーダルを表示します。モーダルを作成および表示するには、component.find('overlayLib').showCustomModal() を使用してモーダル属性を渡します。overlayLib は、lightning:overlayLibrary インスタンスの aura:id に整合します。

```
({
    handleShowModal: function(component, evt, helper) {
        var modalBody;
        $A.createComponent("c:modalContent", {},
                           function(content, status) {
                               if (status === "SUCCESS") {
                                   modalBody = content;
                                   component.find('overlayLib').showCustomModal({
                                       header: "Application Confirmation",
                                       body: modalBody,
                                       showCloseButton: true,
                                       cssClass: "mymodal",
                                       closeCallback: function() {
                                           alert('You closed the alert!');
                                       }
                                   })

                               }

                           });
    }
})
```

c:modalContent は、アイコンとメッセージを表示するカスタムコンポーネントです。

```
<aura:component>
    <lightning:icon size="medium" iconName="action:approval" alternativeText="Approved"
/>
    Your application has been approved.
</aura:component>
```

footer 属性を使用して、独自のフッターを渡せます。次の例では、$A.createComponents() を使用してカスタム本文とフッタを作成します。

```
handleShowModalFooter : function (component, event, helper) {
    var modalBody;
    var modalFooter;
    $A.createComponents([
        ["c:modalContent",{}],
        ["c:modalFooter",{}]
    ],
    function(components, status){
        if (status === "SUCCESS") {
            modalBody = components[0];
            modalFooter = components[1];
            component.find('overlayLib').showCustomModal({
               header: "Application Confirmation",
               body: modalBody,
```

```
                footer: modalFooter,
                showCloseButton: true,
                cssClass: "mymodal",
                closeCallback: function() {
                    alert('You closed the alert!');
                }
            })
        }
    }
  );
}
```

c:modalFooter は、2 つのボタンを表示するカスタムコンポーネントです。

```
<aura:component>
    <lightning:overlayLibrary aura:id="overlayLib"/>
    <lightning:button name="cancel" label="Cancel" onclick="{!c.handleCancel}"/>
    <lightning:button name="ok" label="OK" variant="brand" onclick="{!c.handleOK}"/>
</aura:component>
```

ボタンのクリック時の動作をクライアント側コントローラで定義します。

```
({
    handleCancel : function(component, event, helper) {
        component.find("overlayLib").notifyClose();
    },
    handleOK : function(component, event, helper) {
        //do something
    }
})
```

showCustomModal() はプロミスを返します。これは、表示時にモーダルへの参照を取得する場合に便利です。

```
component.find('overlayLib').showCustomModal({
   //modal attributes
}).then(function (overlay) {
    //closes the modal immediately
    overlay.close();
});
```

属性

属性名	属性型	説明	必須項目
header	Object	モーダルの最上部に表示されるヘッダー。	
body	Object	モーダルの本文。	
footer	Object	モーダルフッター。	
showCloseButton	Boolean	モーダルに終了ボタンを表示するかどうかを指定します。デフォルトは、true です。	
cssClass	String	モーダルの CSS クラス。	

属性名	属性型	説明	必須項目
closeCallback	Function	モーダルが閉じられたときにコールされるコールバック。	

メソッド

プロミスによって返されるモーダルインスタンスでは、次のメソッドを使用できます。

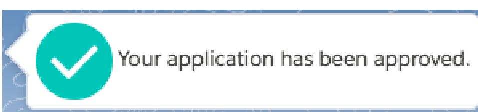
close(): モーダルを閉じて廃棄します。

hide(): モーダルをビューで非表示します。

show(): モーダルを表示します。

ポップオーバー

ポップオーバーは参照要素にコンテキスト情報を表示するもので、モーダルのように他の要素を中断しません。ポップオーバーは、参照要素にマウスポインタが置かれたり、要素がクリックされたりすると表示されます。Esc キーを押すと、ポップオーバーは閉じます。ポップオーバーのデフォルトの位置設定は、参照要素の右です。



ボタンと、参照 div 要素を含む例を次に示します。クリックされると、ボタンはポップオーバーを表示します。div 要素にマウスポインタが置かれた場合も、ポップオーバーは表示されます。

```
<aura:component>
    <lightning:overlayLibrary aura:id="overlayLib"/>
   <lightning:button name="popover" label="Show Popover" onclick="{!c.handleShowPopover}"/>

    <div class="mypopover" onmouseover="{!c.handleShowPopover}">Popover should display if
 you hover over here.</div>
</aura:component>
```

クライアント側コントローラは、ポップオーバーを表示します。この例ではポップオーバー本文に文字列を渡しますが、上のモーダルの例のようにカスタムコンポーネントを渡すこともできます。

```
({
    handleShowPopover : function(component, event, helper) {
        component.find('overlayLib').showCustomPopover({
            body: "Popovers are positioned relative to a reference element",
            referenceSelector: ".mypopover",
            cssClass: "popoverclass"
        }).then(function (overlay) {
            setTimeout(function(){
                //close the popover after 3 seconds
                overlay.close();
            }, 3000);
        });
```

```
    }
})
```

ポップオーバーを作成および表示するには、`component.find('overlayLib').showCustomPopover()` を使用してポップオーバー属性を渡します。`overlayLib` は、`lightning:overlayLibrary` インスタンスの `aura:id` に整合します。

CSS クラスは、ポップオーバーの高さの最小値を設定します。

```
.THIS .popoverclass {
    min-height: 100px;
}
```

属性

属性名	属性型	説明	必須項目
body	Object	モーダルの本文。	
referenceSelector	Object	ポップオーバーを追加する参照要素ポップオーバーは参照要素の右に追加されます。	
cssClass	String	モーダルの CSS クラス。	

メソッド

プロミスによって返されるモーダルインスタンスでは、次のメソッドを使用できます。

`close()`: モーダルを閉じて廃棄します。

`hide()`: モーダルをビューで非表示します。

`show()`: モーダルを表示します。

インターフェースの参照

これらのプラットフォームインターフェースを実装し、コンポーネントをさまざまなコンテキストで使用したり、コンポーネントで追加のコンテキストデータを受信したりできます。コンポーネントは、複数のインターフェースを実装できます。インターフェースのなかには、一緒に実装するものと、相互に排他的なものがあります。また、Lightning Experience と Salesforce1 にのみ影響するインターフェースもあります。

clients:availableForMailAppAppPage

Lightning アプリケーションビルダー、Lightning for Outlook または Lightning for Gmail の Lightning ページに表示するには、コンポーネントに `clients:availableForMailAppAppPage` インターフェースを実装する必要があります。詳細は、「Lightning for Outlook および Lightning for Gmail のコンポーネントの作成」を参照してください。

clients:hasEventContext

Lightning for Outlook および Lightning for Gmail で行動の日付または場所属性にコンポーネントが割り当てられるようにします。詳細は、「Lightning for Outlook および Lightning for Gmail のコンポーネントの作成」を参照してください。

clients:hasItemContext

Lightning for Outlook および Lightning for Gmail でメールまたは行動の項目属性にコンポーネントが割り当てられるようにします。詳細は、「Lightning for Outlook および Lightning for Gmail のコンポーネントの作成」を参照してください。

flexipage:availableForAllPageTypes

Lightning アプリケーションビルダーやあらゆる Lightning ページのタイプに対応するコンポーネントを作成するグローバルインターフェースです。詳細は、「Lightning ページと Lightning アプリケーションビルダーのコンポーネントの設定」を参照してください。

ユーティリティバーに表示するには、コンポーネントに `flexipage:availableForAllPageTypes` インターフェースを実装する必要があります。詳細は、Salesforce ヘルプの「Lightning アプリケーションへのユーティリティバーの追加」を参照してください。

flexipage:availableForRecordHome

コンポーネントがレコードページ専用に設計されている場合は、`flexipage:availableForAllPageTypes` の代わりに `flexipage:availableForRecordHome` インターフェースを実装します。詳細は、「Lightning Experience のレコードホームページのコンポーネントの設定」を参照してください。

forceCommunity:availableForAllPageTypes

コミュニティビルダーに表示するには、コンポーネントに `forceCommunity:availableForAllPageTypes` インターフェースを実装する必要があります。詳細は、「コミュニティのコンポーネントの設定」を参照してください。

force:appHostable

Lightning Experience または Salesforce1 でコンポーネントがカスタムタブとして使用されるようにします。詳細は、「Lightning Experience のカスタムタブとしての Lightning コンポーネントの追加」を参照してください。

force:lightningQuickAction

[キャンセル] ボタンなどの標準アクションコントロールがあるパネルにコンポーネントを表示できるようにします。これらのコンポーネントにも専用のコントロールを表示および実装できますが、標準コントロールからのイベントを処理する必要があります。`force:lightningQuickAction` を実装する場合は、同じコンポーネント内に `force:lightningQuickActionWithoutHeader` を実装できません。詳細は、「カスタムアクション用のコンポーネントの設定」を参照してください。

force:lightningQuickActionWithoutHeader

追加のコントロールなしでパネルにコンポーネントを表示できるようにします。このコンポーネントは、アクションの完全なユーザインターフェースを提供します。`force:lightningQuickActionWithoutHeader` を実装する場合は、同じコンポーネント内に `force:lightningQuickAction` を実装できません。詳細は、「カスタムアクション用のコンポーネントの設定」を参照してください。

ltng:allowGuestAccess

Lightning Out 連動関係アプリケーションに `ltng:allowGuestAccess` インターフェースを追加すると、ユーザは Salesforce の認証を行わずにそのアプリケーションにアクセスできます。このインターフェースでは、Lightning コンポーネントを含むアプリケーションを作成し、あらゆる場所のすべてのユーザにリリースできます。詳細は、「未認証ユーザとの Lightning Out アプリケーションの共有」を参照してください。

このセクションの内容:

force:hasRecordId

`force:hasRecordId` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの ID をコンポーネントに割り当てることができます。現在のレコード ID は、Lightning Experience または Salesforce1 のオブジェクト固有のカスタムアクションやアクション上書きなどとして、コンポーネントをLightning レコードページで使用する場合に便利です。このインターフェースは、Lightning Experience、Salesforce1、およびテンプレートベースのコミュニティ内で使用される場合以外は影響しません。

force:hasSObjectName

`force:hasSObjectName` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの sObject 型の API 参照名をコンポーネントに割り当てることができます。さまざまな sObject 型のレコードでコンポーネントを使用でき、現在のレコードの特定のタイプに適応する必要がある場合は、sObject 名が便利です。このインターフェースは、Lightning Experience、Salesforce1、およびテンプレートベースのコミュニティ内で使用される場合以外は影響しません。

lightning:actionOverride

`lightning:actionOverride` インターフェースを Lightning コンポーネントに追加して、コンポーネントを使用してオブジェクトに対する標準アクションを上書きできるようにします。ほとんどの標準コンポーネントおよびすべてのカスタムコンポーネントで、表示、新規、編集、およびタブ標準アクションを上書きできます。このインターフェースは、Lightning Experience と Salesforce1 内で使用される場合以外は影響しません。

lightning:appHomeTemplate

`lightning:appHomeTemplate` インターフェースを実装して、ページ種別がアプリケーションページのカスタムLightningページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience と Salesforce1 内で使用される場合以外は影響しません。

lightning:availableForChatterExtensionComposer

`lightning:availableForChatterExtensionComposer` インターフェースを使用してカスタムアプリケーションを Chatter パブリッシャーと統合し、カスタムアプリケーションのペイロードをフィードに配置します。このインターフェースは、Lightning コミュニティで使用できます。

lightning:availableForChatterExtensionRenderer

`lightning:availableForChatterExtensionRenderer` インターフェースを使用してカスタムアプリケーションを Chatter パブリッシャーと統合し、カスタムアプリケーションのペイロードをフィードに配置します。このインターフェースは、Lightning コミュニティで使用できます。

lightning:homeTemplate

`lightning:homeTemplate` インターフェースを実装して、Lightning Experience ホームページのカスタムLightningページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience 内で使用される場合以外は影響しません。

lightning:recordHomeTemplate

`lightning:recordHomeTemplate` インターフェースを実装して、オブジェクトレコードページのカスタム Lightning ページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience 内で使用される場合以外は影響しません。

force:hasRecordId

`force:hasRecordId` インターフェースを Lightning コンポーネントに追加すると、現在のレコードの ID をコンポーネントに割り当てることができます。現在のレコード ID は、Lightning Experience または Salesforce1 のオブジェクト固有のカスタムアクションやアクション上書きなどとして、コンポーネントをLightning レコードページで使用する場合に便利です。このインターフェースは、Lightning Experience、Salesforce1、およびテンプレートベースのコミュニティ内で使用される場合以外は影響しません。

このインターフェースはマーカーインターフェースです。マーカーインターフェースは、インターフェースの動作をコンポーネントに追加するよう伝える、コンポーネントのコンテナへの信号です。コンポーネントに特定のメソッドまたは属性を実装する必要はありません。コンポーネントの `implements` 属性にインターフェース名を追加するだけです。

`force:hasRecordId` インターフェースは、このインターフェースを実装するコンポーネントに対して 2 つのことを行います。

- `recordId` という名前の属性をコンポーネントに追加します。この属性は文字列型であり、その値は 18 文字の Salesforce レコード ID (001xx000003DGSWAA4 など) です。これを自分で追加した場合、属性の定義は次のようなマークアップになります。

  ```
  <aura:attribute name="recordId" type="String" />
  ```

  メモ: コンポーネントで `force:hasRecordId` を実装する場合、`recordId` 属性をコンポーネントに自分で追加する必要はありません。追加する場合は、属性のアクセスレベルまたは型を変更しないでください。変更すると、コンポーネントでランタイムエラーが発生します。

- Lightning Experience または Salesforce1 のレコードコンテキストでコンポーネントを呼び出す場合、`recordId` を、表示するレコードの ID に設定します。

`recordId` 属性は、レコードのコンテキストでコンポーネントを配置または呼び出す場合にのみ設定されます。たとえば、レコードページにコンポーネントを配置する場合、またはレコードページやオブジェクトホームからコンポーネントをアクションとして呼び出す場合などがこれに該当します。その他の場合(このコンポーネントをプログラムで別のコンポーネント内に作成する場合など)、`recordId` は設定されないため、コンポーネントでこれを使用しないでください。

例: 次の例に、`force:hasRecordId` インターフェースを Lightning コンポーネントに追加するために必要なマークアップを示します。

```
<aura:component implements="force:lightningQuickAction,force:hasRecordId">

    <!-- ... -->

</aura:component>
```

コンポーネントのコントローラは、`component.get("v.recordId")` を使用して、`recordId` 属性から現在のレコードの ID にアクセスできます。`recordId` 属性は、`force:hasRecordId` インターフェースによって自動的にコンポーネントに追加されます。

force:hasSObjectName

force:hasSObjectName インターフェースを Lightning コンポーネントに追加すると、現在のレコードの sObject 型の API 参照名をコンポーネントに割り当てることができます。さまざまな sObject 型のレコードでコンポーネントを使用でき、現在のレコードの特定のタイプに適応する必要がある場合は、sObject 名が便利です。このインターフェースは、Lightning Experience、Salesforce1、およびテンプレートベースのコミュニティ内で使用される場合以外は影響しません。

このインターフェースはマーカーインターフェースです。マーカーインターフェースは、インターフェースの動作をコンポーネントに追加するよう伝える、コンポーネントのコンテナへの信号です。コンポーネントに特定のメソッドまたは属性を実装する必要はありません。コンポーネントの implements 属性にインターフェース名を追加するだけです。

このインターフェースは、sObjectName という名前の属性をコンポーネントに追加します。この属性は文字列型であり、その値は Account や myNamespace__myObject__c のようなオブジェクトの API 名です。この例を次に示します。

```
<aura:attribute name="sObjectName" type="String" />
```

メモ: コンポーネントで force:hasSObjectName を実装する場合、sObjectName 属性をコンポーネントに自分で追加する必要はありません。追加する場合は、属性のアクセスレベルまたは型を変更しないでください。変更すると、コンポーネントでランタイムエラーが発生します。

sObjectName 属性は、レコードのコンテキストでコンポーネントを配置または呼び出す場合にのみ設定されます。たとえば、レコードページにコンポーネントを配置する場合、またはレコードページやオブジェクトホームからコンポーネントをアクションとして呼び出す場合などがこれに該当します。その他の場合 (このコンポーネントをプログラムで別のコンポーネント内に作成する場合など)、sObjectName は設定されないため、コンポーネントでこれを使用しないでください。

例: 次の例に、force:hasSObjectName インターフェースを Lightning コンポーネントに追加するために必要なマークアップを示します。

```
<aura:component implements="force:lightningQuickAction,force:hasSObjectName">

    <!-- ... -->

</aura:component>
```

コンポーネントのコントローラは、component.get("v.sObjectName") を使用して、recordId 属性から現在のレコードの ID にアクセスできます。recordId 属性は、force:hasSObjectName インターフェースによって自動的にコンポーネントに追加されます。

lightning:actionOverride

lightning:actionOverride インターフェースを Lightning コンポーネントに追加して、コンポーネントを使用してオブジェクトに対する標準アクションを上書きできるようにします。ほとんどの標準コンポーネントおよびすべてのカスタムコンポーネントで、表示、新規、編集、およびタブ標準アクションを上書きできます。このインターフェースは、Lightning Experience と Salesforce1 内で使用される場合以外は影響しません。

このインターフェースはマーカーインターフェースです。マーカーインターフェースは、インターフェースの動作をコンポーネントに追加するよう伝える、コンポーネントのコンテナへの信号です。コンポーネントに特定のメソッドまたは属性を実装する必要はありません。コンポーネントの `implements` 属性にインターフェース名を追加するだけです。

`lightning:actionOverride` は、実装するコンポーネントに属性を必要とせず、追加もしません。このインターフェースを実装するコンポーネントは、自動的にはアクションを上書きしません。[設定] で関連するアクションを手動で上書きする必要があります。

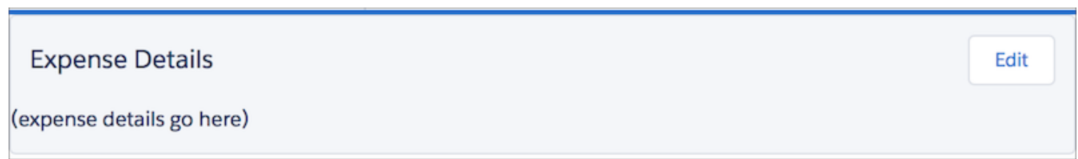
このインターフェースを実装するコンポーネントのみがオブジェクトアクションの[プロパティの上書き]パネルの **[Lightning コンポーネントバンドル]** メニューに表示されます。

例: 次の例に、`lightning:actionOverride` インターフェースを Lightning コンポーネントに追加するために必要なマークアップを示します。

```
<aura:component
    implements="lightning:actionOverride,force:hasRecordId,force:hasSObjectName">

    <article class="slds-card">
      <div class="slds-card__header slds-grid">
        <header class="slds-media slds-media--center slds-has-flexi-truncate">
          <div class="slds-media__body">
            <h2><span class="slds-text-heading--small">Expense Details</span></h2>
          </div>
        </header>
        <div class="slds-no-flex">
            <lightning:button label="Edit" onclick="{!c.handleEdit}"/>
        </div>
      </div>
      <div class="slds-card__body">(expense details go here)</div>
    </article>
</aura:component>
```

Lightning Experience では、標準のタブアクションと表示アクションはページとして表示され、標準の新規アクションと編集アクションは、フロート表示されたパネルに表示されます。アクション上書きとして使用する場合、`lightning:actionOverride` インターフェースを実装する Lightning コンポーネントは標準の動作を完全に置き換えます。ただし、上書きされたアクションはパネルとしてではなく常にページとして表示されます。コンポーネントに、Lightning Experience のメインナビゲーションバー以外のコントロールは表示されません。コンポーネントでは、ナビゲーションバーのほかに、ナビゲーションやアクションを含めアクションの完全なユーザインターフェースが提供されることが予想されます。



lightning:appHomeTemplate

`lightning:appHomeTemplate` インターフェースを実装して、ページ種別がアプリケーションページのカスタム Lightning ページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience と Salesforce1 内で使用される場合以外は影響しません。

このインターフェースを実装するコンポーネントは、Lightning アプリケーションビルダーのアプリケーションページ用新規ページウィザードの [カスタムテンプレート] セクションに表示されます。

重要: 各テンプレートコンポーネントでは、1 つのテンプレートインターフェースのみを実装する必要があります。テンプレートコンポーネントで `flexipage:availableForAllPageTypes` や `force:hasRecordId` などの他のインターフェース型を実装しないでください。テンプレートコンポーネントでは、通常の Lightning コンポーネントのようにマルチタスクを実行できません。これはテンプレートまたはそれ以外になります。

lightning:availableForChatterExtensionComposer

`lightning:availableForChatterExtensionComposer` インターフェースを使用してカスタムアプリケーションを Chatter パブリッシャーと統合し、カスタムアプリケーションのペイロードをフィードに配置します。このインターフェースは、Lightning コミュニティで使用できます。

`lightning:availableForChatterExtensionComposer` インターフェースは、`lightning:availableForChatterExtensionRenderer` インターフェースと `lightning:sendChatterExtensionPayload` イベントと連携して、カスタムアプリケーションを Chatter パブリッシャーに統合し、アプリケーションのペイロードをフィードに表示します。

メモ: 統合についての詳細は、「Chatter パブリッシャーへのカスタムアプリケーションの統合」を参照してください。

lightning:availableForChatterExtensionRenderer

`lightning:availableForChatterExtensionRenderer` インターフェースを使用してカスタムアプリケーションを Chatter パブリッシャーと統合し、カスタムアプリケーションのペイロードをフィードに配置します。このインターフェースは、Lightning コミュニティで使用できます。

`lightning:availableForChatterExtensionRenderer` インターフェースは、`lightning:availableForChatterExtensionComposer` インターフェースと `lightning:sendChatterExtensionPayload` イベントと連携して、カスタムアプリケーションを Chatter パブリッシャーに統合し、アプリケーションのペイロードをフィードに表示します。

メモ: 統合についての詳細は、「Chatter パブリッシャーへのカスタムアプリケーションの統合」を参照してください。

項目

属性名	型	説明	必須項目
payload	Object	フィード項目と一緒に保存されたペイロードデータは、このインターフェースを実装しているコンポーネントに提供されます。	いいえ
variant	String	PREVIEW または RENDER のいずれかの値を取れる enum。選択された値は、このインターフェースを実装しているコンポーネントに提供されます。PREVIEW は、パブリッシャーでプレビューとして添付ファイルを使用するように指定します。RENDER は、添付ファイルをフィード項目と共に表示するように指定します。	いいえ

lightning:homeTemplate

`lightning:homeTemplate` インターフェースを実装して、Lightning Experience ホームページのカスタム Lightning ページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience 内で使用される場合以外は影響しません。

このインターフェースを実装するコンポーネントは、Lightning アプリケーションビルダーのホームページ用新規ページウィザードの [カスタムテンプレート] セクションに表示されます。

重要: 各テンプレートコンポーネントでは、1 つのテンプレートインターフェースのみを実装する必要があります。テンプレートコンポーネントで `flexipage:availableForAllPageTypes` や `force:hasRecordId` などの他のインターフェース型を実装しないでください。テンプレートコンポーネントでは、通常の Lightning コンポーネントのようにマルチタスクを実行できません。これはテンプレートまたはそれ以外になります。

lightning:recordHomeTemplate

`lightning:recordHomeTemplate` インターフェースを実装して、オブジェクトレコードページのカスタム Lightning ページテンプレートとしてコンポーネントを使用できるようにします。このインターフェースは、Lightning Experience 内で使用される場合以外は影響しません。

このインターフェースを実装するコンポーネントは、Lightning アプリケーションビルダーのレコードページ用新規ページウィザードの [カスタムテンプレート] セクションに表示されます。

重要: 各テンプレートコンポーネントでは、1 つのテンプレートインターフェースのみを実装する必要があります。テンプレートコンポーネントで `flexipage:availableForAllPageTypes` や `force:hasRecordId` などの他のインターフェース型を実装しないでください。テンプレートコンポーネントでは、通常の Lightning コンポーネントのようにマルチタスクを実行できません。これはテンプレートまたはそれ以外になります。

イベントの参照

標準搭載のイベントを使用して、Lightning Experience または Salesforce1 内、または Lightning コンポーネント内でコンポーネントがやりとりできるようにします。たとえば、次のイベントは、レコードの作成ページまたは編集ページを開いたり、レコードに移動したりするコンポーネントを有効にします。

イベントは、次をはじめとするさまざまな名前空間に属します。

force
Lightning Experience および Salesforce1 で処理されるイベントを提供します。

forceCommunity
コミュニティで処理されるイベントを提供します。

Lightning
Lightning Experience、Salesforce1、およびコミュニティで処理されるイベントを提供します。

ltng
別のコンポーネントにレコード ID または汎用的なメッセージを送信するイベントを提供します。

ui
`ui` コンポーネントで処理されるイベントを提供します。

wave
Wave Analytics で処理されるイベントを提供します。

これらの `force` または `lightning` イベントのいずれかを Salesforce1 または Lightning Experience 外の Lightning アプリケーション/コンポーネントで起動する場合、次のようになります。

- 処理コンポーネントの `<aura:handler>` タグを使用して、イベントを処理する必要があります。
- 必要に応じて、イベントがクライアントに送信されるように `<aura:registerEvent>` または `<aura:dependency>` タグを使用します。

関連トピック:

aura:dependency
Salesforce1 と Lightning Experience で処理されるイベント
コンポーネントイベントの起動
アプリケーションイベントの起動

force:closeQuickAction

クイックアクションパネルを閉じます。アプリケーションで一度に開くことができるクイックアクションパネルは 1 つのみです。

アクションが完了したか、キャンセルするために、クイックアクションパネルを閉じるには、`$A.get("e.force:closeQuickAction").fire();` を実行します。

次の例では、パネルのユーザインターフェースからの入力を処理し、処理結果の「トースト」メッセージを表示した後、クイックアクションパネルを閉じます。処理とトーストは、クイックアクションの終了とは無関係

ですが、順序が重要です。force:closeQuickAction の起動は、クイックアクションハンドラが最後に実行する処理であることが必要です。

```
/*quickAddController.js*/
({
    clickAdd: function(component, event, helper) {

        // Get the values from the form
        var n1 = component.find("num1").get("v.value");
        var n2 = component.find("num2").get("v.value");

        // Display the total in a "toast" status message
        var resultsToast = $A.get("e.force:showToast");
        resultsToast.setParams({
            "title": "Quick Add: " + n1 + " + " + n2,
            "message": "The total is: " + (n1 + n2) + "."
        });
        resultsToast.fire();

        // Close the action panel
        var dismissActionPanel = $A.get("e.force:closeQuickAction");
        dismissActionPanel.fire();
    }

})
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience および Salesforce1 でのみサポートされています。

force:createRecord

指定した entityApiName (「Account」や「myNamespace__MyObject__c」など) のレコードを作成するページを開きます。

オブジェクトのレコードの作成ページを表示するには、entityApiName 属性でオブジェクト名を設定し、イベントを起動します。recordTypeId は省略可能ですが、使用する場合は、作成されるオブジェクトのレコードタイプを指定します。defaultFieldValues は省略可能ですが、使用する場合、レコード作成フォームの事前入力で使用する値を指定できます。

次の例では、取引先責任者のレコード作成パネルを表示します。

```
createRecord : function (component, event, helper) {
    var createRecordEvent = $A.get("e.force:createRecord");
    createRecordEvent.setParams({
        "entityApiName": "Contact"
    });
    createRecordEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。このイベントは、レコードを作成する標準的なページを表します。つまり、オブジェクトの作成アクションへの上書きを無視します。

項目値の事前入力

defaultFieldValues 属性を使用すると、デフォルトまたは計算による項目値を用いてレコード作成フォームを事前入力できます。値の事前入力により、すばやいデータ入力や、データの一貫性向上、レコード作成プロセスの簡素化を実現します。JavaScript オブジェクトの名前-値ペアの形式でデフォルト項目値を指定します。

次の例では、2 つの項目が自動入力された、取引先責任者のレコード作成パネルを表示します。

```
var createAcountContactEvent = $A.get("e.force:createRecord");
createAcountContactEvent.setParams({
    "entityApiName": "Contact",
    "defaultFieldValues": {
        'Phone' : '415-240-6590',
        'Account' : '001xxxxxxxxxxxxxxx'
    }
});
createAcountContactEvent.fire();
```

レコード作成フォームでは使用できない項目の値も指定できます。

- 項目がページレイアウトに含まれず非表示である場合、defaultFieldValues で指定された値は、新しいレコードで保存されます。
- 項目レベルセキュリティにより現在のユーザが項目の作成アクセス権を持っていない場合、新しいレコードを保存しようとするとエラーが発生します。

重要: エラーメッセージは、現在のユーザがアクセスできない項目を参照できません。このため、ユーザはエラー発生の原因や解決策を把握できません。

force:createRecord イベントが起動すると、標準レコード作成ページを使用するようにアプリケーションに通知されます。システム管理者はそこで発生したエラーを把握できません。また、わかりやすいエラーメッセージを表示するなど、作成ページのインターフェースや動作を変更することもできません。そのため、必ず独自のコードでアクセス権チェックを実行した後に、イベントを起動してください。

Id またはレコード変更タイムスタンプなどのシステムが管理する項目は事前入力できません。これらの項目のデフォルト値に対しては、メッセージを表示せずに無視します。

リッチテキスト項目の事前入力はサポートされていません。簡単な値の場合は動作する可能性がありますが、リッチテキスト項目の内部形式はドキュメント化されていないため、書式設定を含む複雑な値を設定すると問題につながることがあります。各自の責任で使用してください。

日時項目値は ISO 8601 形式を使用する必要があります。次に例を示します。

- 日付: 2017-07-18
- 日時: 2017-07-18T03:00:00Z

メモ: レコード作成パネルではユーザのローカル時間の日時値が表示されますが、項目に事前入力するには、日時を UTC に変換する必要があります。

属性名	型	説明	必須
entityApiName	String	カスタムオブジェクトまたは標準オブジェクトの API 名 (「Account」、「Case」、「Contact」、「Lead」、「Opportunity」、「namespace__objectName__c」など)。	はい

属性名	型	説明	必須
defaultFieldValues	String	パネルに表示されない項目も含め、レコード作成パネルの項目を自動入力します。ID項目とリッチテキスト項目は自動入力できません。ユーザには、値が事前入力される項目に対する作成アクセス権が必要です。項目のアクセス制限によって発生した保存時のエラーは、エラーメッセージには表示されません。	
recordTypeId	String	レコードタイプ ID (オブジェクトにレコードタイプを使用できる場合)。	

force:editRecord

recordId で指定したレコードを編集するページを開きます。

オブジェクトのレコード編集ページを表示するには、recordId 属性でオブジェクト名を設定し、イベントを起動します。次の例では、recordId で指定された取引先責任者のレコード編集ページを表示します。

```
editRecord : function(component, event, helper) {
    var editRecordEvent = $A.get("e.force:editRecord");
    editRecordEvent.setParams({
         "recordId": component.get("v.contact.Id")
   });
    editRecordEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
recordId	String	編集するレコードに関連付けられたレコード ID。	はい

force:navigateToList

listViewId で指定したリストビューに移動します。

リストビューに移動するには、listViewId 属性でリストビュー ID を設定し、イベントを起動します。次の例では、取引先責任者のリストビューを表示します。

```
gotoList : function (component, event, helper) {
    var action = component.get("c.getListViews");
    action.setCallback(this, function(response){
        var state = response.getState();
        if (state === "SUCCESS") {
            var listviews = response.getReturnValue();
            var navEvent = $A.get("e.force:navigateToList");
            navEvent.setParams({
                "listViewId": listviews.Id,
```

```
                "listViewName": null,
                "scope": "Contact"
            });
            navEvent.fire();
        }
    });
    $A.enqueueAction(action);
}
```

次の Apex コントローラからは、取引先責任者オブジェクトのすべてのリストビューが返されます。

```
@AuraEnabled
public static List<ListView> getListViews() {
    List<ListView> listviews =
        [SELECT Id, Name FROM ListView WHERE SobjectType = 'Contact'];

    // Perform isAccessible() check here
    return listviews;
}
```

また、移動先となるリストビューの名前を SOQL クエリで指定して、1 つのリストビュー ID を指定することもできます。

```
SELECT Id, Name FROM ListView WHERE SobjectType = 'Contact' and Name='All Contacts'
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
listViewId	String	表示するリストビューの ID。	はい
listViewName	String	リストビューの名前を指定しますが、実際の名前と一致する必要はありません。リストビューに保存されている実際の名前を使用するには、listViewName を null に設定します。	
scope	String	ビューの sObject の名前 (「Account」や「namespace__MyObject__c」など)。	

関連トピック:

CRUD および項目レベルセキュリティ (FLS)

force:navigateToObjectHome

scope 属性で指定したオブジェクトホームに移動します。

オブジェクトホームに移動するには、scope 属性でオブジェクト名を設定し、イベントを起動します。次の例では、カスタムオブジェクトのホームページを表示します。

```
navHome : function (component, event, helper) {
    var homeEvent = $A.get("e.force:navigateToObjectHome");
    homeEvent.setParams({
```

```
        "scope": "myNamespace__myObject__c"
    });
    homeEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
scope	String	カスタムオブジェクトまたは標準オブジェクトの API 名 (「Contact」や「namespace__objectName__c」など)。	はい
resetHistory	Boolean	true に設定されていると、履歴がリセットされます。デフォルトは false で、Salesforce1 で [戻る] ボタンが表示されます。	

force:navigateToRelatedList

parentRecordId で指定した関連リストに移動します。

関連リストに移動するには、parentRecordId 属性で親レコード ID を設定し、イベントを起動します。たとえば、取引先責任者オブジェクトの関連リストを表示する場合、parentRecordId は Contact.Id です。次の例では、取引先責任者レコードの関連ケースを表示します。

```
gotoRelatedList : function (component, event, helper) {
    var relatedListEvent = $A.get("e.force:navigateToRelatedList");
    relatedListEvent.setParams({
        "relatedListId": "Cases",
        "parentRecordId": component.get("v.contact.Id")
    });
    relatedListEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
parentRecordId	String	親レコードの ID。	はい
relatedListId	String	表示する関連リストの API 名 (「Contacts」や「Opportunities」など)。	はい

force:navigateToSObject

recordId で指定した sObject レコードに移動します。

レコードビューを表示するには、recordId 属性でレコード ID を設定し、イベントを起動します。

レコードビューには、Chatter フィード、レコード詳細、および関連情報を表示するスライドが含まれます。次の例では、指定されたレコード ID のレコードビューの関連情報スライドを表示します。

メモ: Salesforce1 では特定のスライドを設定できますが、Lightning Experience ではできません。

```
createRecord : function (component, event, helper) {
    var navEvt = $A.get("e.force:navigateToSObject");
    navEvt.setParams({
      "recordId": "00QB0000000ybNX",
      "slideDevName": "related"
    });
    navEvt.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
isredirect	Boolean	ナビゲーション履歴の現在の URL を新しい URL に置き換えることを示します。デフォルトは、false です。	
recordId	String	レコード ID。	はい
slideDevName	String	最初に表示するレコードビュー内のスライドを指定します。有効なオプションは、次のとおりです。 • detail: レコード詳細スライド。これはデフォルト値です。 • chatter: Chatter スライド。 • related: 関連情報スライド。 Lightning Experience ではこの属性は無効です。	

force:navigateToURL

指定した URL に移動します。

相対 URL と絶対 URL がサポートされています。相対 URL は、Salesforce1 モバイルブラウザアプリケーションドメインに対して相対的で、ナビゲーション履歴を保持します。外部 URL は、別のブラウザウィンドウで開きます。

アプリケーション内のさまざまな画面に移動するには相対URLを使用します。ユーザに別のサイトまたはアプリケーションへのアクセスを許可するには外部URLを使用します。ユーザは移動先のサイトまたはアプリケーションで、元のアプリケーションに保持する必要のないアクションを実行できます。ユーザが元のアプリケーションに戻るには、別のアプリケーションを終了したときに、外部URLによって開かれた別のウィンドウを閉じる必要があります。この新しいウィンドウは、元のアプリケーションとは別の履歴を持ち、ウィンドウを閉じるとこの履歴は破棄されます。つまり、ユーザは [戻る] ボタンをクリックして元のアプリケーションに戻ることはできません。ユーザは新しいウィンドウを閉じる必要があります。

外部アプリケーションを起動し、ユーザが適切な操作を行えるようにするため、mailto:、tel:、geo: などの URL スキームがサポートされています。ただし、サポートはモバイルプラットフォームとデバイスによって

異なります。mailto: と tel: は信頼できますが、他の URL については、使用が想定されるさまざまなデバイスでテストすることをお勧めします。

mailto: および tel: URL スキームを使用している場合、ui:outputEmail および ui:outputURL コンポーネントの使用も考慮できます。

次の例では、相対 URL を使用してユーザを商談ページ /006/o に移動させます。

```
gotoURL : function (component, event, helper) {
    var urlEvent = $A.get("e.force:navigateToURL");
    urlEvent.setParams({
      "url": "/006/o"
    });
    urlEvent.fire();
}
```

次の例では、リンクがクリックされたときに外部 Web サイトを開きます。

```
navigate : function(component, event, helper) {

    //Find the text value of the component with aura:id set to "address"
    var address = component.find("address").get("v.value");

    var urlEvent = $A.get("e.force:navigateToURL");
    urlEvent.setParams({
      "url": 'https://www.google.com/maps/place/' + address
    });
    urlEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

属性名	型	説明	必須
isredirect	Boolean	ナビゲーション履歴の現在の URL を新しい URL に置き換えることを示します。デフォルトは false です。	
url	String	対象の URL。	はい

force:recordSave

レコードを保存します。

force:recordSave は force:recordEdit コンポーネントで処理されます。次の例に、ユーザ入力を取得して recordId 属性で指定されたレコードを更新する force:recordEdit コンポーネントを示します。ボタンは force:recordSave イベントを起動します。

```
<force:recordEdit aura:id="edit" recordId="a02D0000006V8Ni"/>
<ui:button label="Save" press="{!c.save}"/>
```

このクライアント側のコントローラは、レコードを保存するイベントを起動します。

```
save : function(component, event, helper) {
    component.find("edit").get("e.recordSave").fire();
    // Update the component
    helper.getRecords(component);
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience および Salesforce1 でのみサポートされています。

force:recordSaveSuccess

レコードが正常に保存されたことを示します。

force:recordSaveSuccess は force:recordEdit コンポーネントで使用されます。次の例に、ユーザ入力を取得して recordId 属性で指定されたレコードを更新する force:recordEdit コンポーネントを示します。ボタンは force:recordSave イベントを起動します。

```
<aura:attribute name="recordId" type="String" default="a02D0000006V8Ni"/>
<aura:attribute name="saveState" type="String" default="UNSAVED" />
<aura:handler name="onSaveSuccess" event="force:recordSaveSuccess"
action="{!c.handleSaveSuccess}"/>

<force:recordEdit aura:id="edit" recordId="{!v.recordId}" />
<ui:button label="Save" press="{!c.save}"/>
Record save status: {!v.saveState}
```

このクライアント側コントローラは、レコードを保存するイベントを起動し、適宜処理します。

```
({
    save : function(cmp, event) {
        // Save the record
        cmp.find("edit").get("e.recordSave").fire();
    },

    handleSaveSuccess : function(cmp, event) {
        // Display the save status
        cmp.set("v.saveState", "SAVED");
    }
})
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience および Salesforce1 でのみサポートされています。

force:refreshView

ビューを再読み込みします。

ビューを更新するには、ビューのすべてのデータを再読み込みする `$A.get("e.force:refreshView").fire();` を実行します。

次の例では、アクションが正常に完了した後にビューを更新します。

```
refresh : function(component, event, helper) {
    var action = cmp.get('c.myController');
    action.setCallback(cmp,
        function(response) {
            var state = response.getState();
            if (state === 'SUCCESS'){
                $A.get('e.force:refreshView').fire();
            } else {
                 //do something
            }
        }
    );
    $A.enqueueAction(action);
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

force:showToast

トースト通知にメッセージを表示します。

トーストによって、ビューの上部のヘッダーの下にメッセージが表示されます。メッセージは message 属性で指定されます。

次の例は、「**Success!** The record has been updated successfully. (成功! レコードは正常に更新されました。)」というトーストメッセージを表示します。

```
showToast : function(component, event, helper) {
    var toastEvent = $A.get("e.force:showToast");
    toastEvent.setParams({
        "title": "Success!",
        "message": "The record has been updated successfully."
    });
    toastEvent.fire();
}
```

メモ: このイベントは、one.app コンテナによって処理されます。Lightning Experience、Salesforce1、および Lightning コミュニティでサポートされています。

トーストで使用される背景色とアイコンは、type 属性によって制御されます。たとえば、success に設定すると、トースト通知は緑の背景とチェックマークアイコンで表示されます。mode 属性が dismissible の場合、トーストは 5000 ミリ秒間表示され、右上隅に [閉じる] ボタンが表示されます。

message はテキストのみの文字列をサポートするのに対し、messageTemplate はリンクを含む文字列をサポートします。messageTemplateData で指定した表示ラベルで置き換えられる、プレースホルダの文字列を指定できます。パラメータは、ゼロから順に番号が付けられます。たとえば、{0}、{1}、{2} という 3 つのパラメータがある場合、表示ラベルは指定された順に置換されます。表示ラベルは、アンカータグの title 属性にも使用されます。

次の例では、リンクを含むメッセージのトーストが表示されます。

```
showMyToast : function(component, event, helper) {
    var toastEvent = $A.get("e.force:showToast");
    toastEvent.setParams({
        mode: 'sticky',
        message: 'This is a required message',
        messageTemplate: 'Record {0} created! See it {1}!',
        messageTemplateData: ['Salesforce', {
            url: 'http://www.salesforce.com/',
            label: 'here',
            }
        ]
    });
    toastEvent.fire();
}
```

属性名	型	説明	必須
title	String	太字で表示されるトーストのタイトルを指定します。	
message	String	表示するメッセージを指定します。	はい
messageTemplate	String	指定したメッセージでメッセージ文字列を上書きします。messageTemplateData が必要です。	
messageTemplateData	Object	messageTemplate で使用されるテキストとアクションの配列。	
key	String	トースト種別が other の場合のアイコンを指定します。アイコンキーは「Lightning Design System Resources」(Lightning Design System のリソース) ページに記載されています。	
duration	Integer	トーストの期間 (ミリ秒)。デフォルト値は 5000 ミリ秒です。	
type	String	トースト種別。error、warning、success、info のいずれかを指定できます。デフォルトは other で、info トーストのようなスタイルですが、key 属性で指定しない限りアイコンは表示されません。	
mode	String	トーストモード。ユーザがトーストを消去できる場合の消去方法を制御します。デフォルトは dismissible で、[閉じる] ボタンを表示します。 有効な値は次のとおりです。 • dismissible: [閉じる] ボタンを押すか、duration が経過するかのいずれかが生じるまで表示されたままになります。 • pester: duration が経過するまで表示されたままになります。[閉じる] ボタンは表示されません。 • sticky: [閉じる] ボタンを押すまで表示されたままになります。	

forceCommunity:analyticsInteraction

コミュニティのカスタムコンポーネントによってトリガされたイベントを追跡し、データを Google Analytics に送信します。

たとえば、カスタムボタンを作成し、ボタンのクライアント側コントローラに `forceCommunity:analyticsInteraction` イベントを含めることができます。このボタンをクリックすると、イベントデータが Google Analytics に送信されます。

```
onClick : function(cmp, event, helper) {
    var analyticsInteraction = $A.get("e.forceCommunity:analyticsInteraction");
    analyticsInteraction.setParams({
        hitType : 'event',
        eventCategory : 'Button',
        eventAction : 'click',
        eventLabel : 'Winter Campaign Button',
        eventValue: 200
    });
    analyticsInteraction.fire();
}
```

メモ:

- このイベントは、Lightning コミュニティでのみサポートされます。イベント追跡を有効にするには、コミュニティビルダーの [設定] > [上級] に Google Analytics トラッキング ID を追加して、コミュニティを公開します。
- Google Analytics は Sandbox 環境ではサポートされません。

属性名	型	説明
`hitType`	String	必須。ヒットの種別。許容される唯一の値は `'event'` です。
`eventCategory`	String	必須。ボタンや動画など操作した項目の種別またはカテゴリ。
`eventAction`	String	必須。アクションの種別。たとえば動画プレーヤーの場合、アクションには再生、一時停止、共有などが含まれます。
`eventLabel`	String	イベントに関する追加情報を提供するために使用できます。
`eventValue`	Integer	イベントに関連付けられている正の数値。

forceCommunity:routeChange

ページの URL が変更されると、システムが `forceCommunity:routeChange` イベントを起動します。カスタム Lightning コンポーネントは、このシステムイベントをリスンし、分析や SEO などの目的で必要に応じて処理できます。

メモ: このイベントは、Lightning コミュニティでのみサポートされます。

このサンプルコンポーネントは、システムイベントをリスンします。

```
<aura:component implements="forceCommunity:availableForAllPageTypes">
    <aura:attribute name="routeChangeCounter" default="0" type="Integer" required="false"/>

     <aura:handler event="forceCommunity:routeChange" action="{!c.handleRouteChange}"/>
     <h1>Route was changed: {!v.routeChangeCounter} times</h1>
</aura:component>
```

このクライアント側コントローラの例は、システムイベントを処理します。

```
({handleRouteChange : function(component, event, helper) {
    component.set('v.routeChangeCounter', component.get('v.routeChangeCounter') + 1);
    }
})
```

lightning:openFiles

ContentDocument および ContentHubItem オブジェクトの 1 つ以上のファイルレコードを開きます。

デスクトップでは、このイベントによって SVG ファイルプレビュープレーヤーが開き、画像、ドキュメント、その他のファイルをブラウザでプレビューできます。ファイルプレビュープレーヤーでは、全画面プレゼンテーションモードがサポートされており、アップロード、削除、ダウンロード、共有などのファイルアクションにすばやくアクセスできます。

モバイルデバイスでは、ファイルがダウンロードされます。デバイスでファイルプレビューがサポートされている場合は、デバイスのプレビューアプリケーションが開きます。

この例では 1 つのファイルが開きます。

```
openSingleFile: function(cmp, event, helper) {
    $A.get('e.lightning:openFiles').fire({
        recordIds: [component.get("v.currentContentDocumentId")]
    });
}
```

この例では複数のファイルが開きます。

```
openMultipleFiles: function(cmp, event, helper) {
    $A.get('e.lightning:openFiles').fire({
        recordIds: component.get("v.allContentDocumentIds"),
        selectedRecordId: component.get("v.currentContentDocumentId")
    });
}
```

メモ: このイベントは、Lightning Experience、Salesforce1 モバイルブラウザアプリケーション、Lightning コミュニティでサポートされます。非推奨の Koa および Kokua コミュニティテンプレートではサポートされていません。

属性名	型	説明
recordIds	String []	必須。開くレコードの ID。

属性名	型	説明
selectedRecordId	String	recordIds で指定されたリストから開く最初のレコードの ID。値が指定されていない場合または正しくない場合は、リストの最初の項目が使用されます。

lightning:sendChatterExtensionPayload

拡張の構成中に保存されるペイロードとメタデータを更新します。

このイベントは、lightning:availableForChatterExtensionComposer および lightning:availableForChatterExtensionRenderer インターフェースで使用されます。

項目

属性名	種別	説明	必須
payload	Object	フィード項目と一緒に保存するペイロードデータ	はい
extensionTitle	String	フィード項目と一緒に保存するアプリケーションのタイトル	はい
extensionDescription	String	フィード項目と一緒に保存するアプリケーションの説明	はい
extensionThumbnailUrl	String	フィード項目と一緒に保存するアプリケーションのサムネールの URL	いいえ

関連トピック:

lightning:availableForChatterExtensionComposer

lightning:availableForChatterExtensionRenderer

Chatter パブリッシャーへのカスタムアプリケーションの統合

ltng:selectSObject

UI でオブジェクトが選択されたときに、そのオブジェクトの recordId を送信します。

オブジェクトを選択するには、recordId 属性にレコード ID を設定します。特定のイベントメッセージをリスンする場合、必要に応じてこのイベントのチャネルを指定し、コンポーネントで選択できるようにします。

```
selectedObj: function(component, event) {
 var selectedObjEvent = $A.get("e.ltng:selectSObject");
 selectedObjEvent.setParams({
          "recordId": "0061a000004x8e1",
          "channel": "AccountsChannel"
 });
 selectedObj.fire();
}
```

属性名	型	説明
recordId	String	必須。選択するレコードに関連付けられたレコード ID。
channel	String	特定のコンポーネントで一部のイベントメッセージを処理し、その他のイベントメッセージは無視する場合、この項目を指定します。

ltng:sendMessage

2 つのコンポーネント間のメッセージを渡します。

メッセージを送信するには、コンポーネント間で渡すテキストの文字列を指定します。特定のイベントメッセージをリスンする場合、必要に応じてこのイベントのチャネルを指定し、コンポーネントで選択できるようにします。

```
sendMsg: function(component, event) {
 var sendMsgEvent = $A.get("e.ltng:sendMessage");
 sendMsgEvent.setParams({
          "message": "Hello World",
          "channel": "AccountsChannel"
 });
 sendMsgEvent.fire();
}
```

属性名	型	説明
message	String	必須。コンポーネント間で渡すテキスト。
channel	String	特定のコンポーネントで一部のイベントメッセージを処理し、その他のイベントメッセージは無視する場合、この項目を指定します。

ui:clearErrors

検証エラーをクリアする必要があることを示します。

ui:clearErrors イベントのハンドラを設定するには、ui:inputNumber などの ui:input を拡張するコンポーネントで onClearErrors システム属性を使用します。

次の ui:inputNumber コンポーネントは、ui:button コンポーネントが押されたときにエラーを処理します。これらのイベントは、クライアント側コントローラで起動および処理できます。

```
<aura:component>
    Enter a number:
    <!-- onError calls your client-side controller to handle a validation error -->
    <!-- onClearErrors calls your client-side controller to handle clearing of errors -->

    <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/>

    <!-- press calls your client-side controller to trigger validation errors -->
```

```
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

詳細は、「項目の検証」 (ページ 290) を参照してください。

ui:collapse

メニューコンポーネントが折りたたまれていることを示します。

たとえば、ui:menuList コンポーネントはこのイベントを登録し、その起動時に処理します。

```
<aura:registerEvent name="menuCollapse"  type="ui:collapse"
                    description="The event fired when the menu list collapses." />
```

このイベントは ui:menuList コンポーネントインスタンスで処理できます。次の例に、2 つのリスト項目があるメニューコンポーネントを示します。ここでは、ui:collapse および ui:expand イベントを処理します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu"
                     menuCollapse="{!c.addMyClass}" menuExpand="{!c.removeMyClass}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
                               click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

このクライアント側のコントローラは、メニューが折りたたまれるとトリガに CSS クラスを追加し、メニューが展開されるとそのクラスを削除します。

```
({
    addMyClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.addClass(trigger, "myClass");
    },
    removeMyClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.removeClass(trigger, "myClass");
    }
})
```

ui:expand

メニューコンポーネントが展開されていることを示します。

たとえば、ui:menuList コンポーネントはこのイベントを登録し、その起動時に処理します。

```
<aura:registerEvent name="menuExpand"  type="ui:expand"
                    description="The event fired when the menu list displays." />
```

このイベントは ui:menuList コンポーネントインスタンスで処理できます。次の例に、2 つのリスト項目があるメニューコンポーネントを示します。ここでは、ui:collapse および ui:expand イベントを処理します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu"
                     menuCollapse="{!c.addMyClass}" menuExpand="{!c.removeMyClass}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
                               click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

このクライアント側のコントローラは、メニューが折りたたまれるとトリガに CSS クラスを追加し、メニューが展開されるとそのクラスを削除します。

```
({
    addMyClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.addClass(trigger, "myClass");
    },
    removeMyClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.removeClass(trigger, "myClass");
    }
})
```

ui:menuFocusChange

ユーザがメニューコンポーネント内のメニュー項目フォーカスを変更したことを示します。

たとえば、ユーザがメニューリストを上や下にスクロールすると、メニュー項目のフォーカスが変わり、このイベントが起動されます。ui:menuList コンポーネントはこのイベントを登録し、その起動時に処理します。

```
<aura:registerEvent name="menuFocusChange"  type="ui:menuFocusChange"
                   description="The event fired when the menu list focus changes from one
 menu item to another." />
```

このイベントは ui:menuList コンポーネントインスタンスで処理できます。次の例に、2 つのリスト項目があるメニューコンポーネントを示します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu"
                     menuFocusChange="{!c.handleChange}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts" />
            <ui:actionMenuItem aura:id="item2" label="All Primary" />
        </ui:menuList>
</ui:menu>
```

ui:menuSelect

メニューコンポーネントで 1 つのメニュー項目が選択されたことを示します。

たとえば、ui:menuList コンポーネントはこのイベントを登録し、コンポーネントでイベントを起動できるようにします。

```
<aura:registerEvent name="menuSelect"  type="ui:menuSelect"
                    description="The event fired when a menu item is selected." />
```

このイベントは ui:menuList コンポーネントインスタンスで処理できます。次の例に、2 つのリスト項目があるメニューコンポーネントを示します。ui:menuSelect イベントと click イベントを処理します。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu" menuSelect="{!c.selected}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
                               click="{!c.doSomething}"/>
            <ui:actionMenuItem aura:id="item2" label="All Primary"
                               click="{!c.doSomething}"/>
        </ui:menuList>
</ui:menu>
```

メニュー項目がクリックされると、click イベントが処理されてから、ui:menuSelect イベント (次の例の doSomething および selected クライアント側コントローラに対応) が処理されます。

```
({
    selected : function(component, event, helper) {
        var selected = event.getParam("selectedItem");

        // returns label of selected item
        var selectedLabel = selected.get("v.label");
    },

    doSomething : function(component, event, helper) {
        console.log("do something");
    }
})
```

属性名	型	説明
selectedItem	Component[]	選択されているメニュー項目
hideMenu	Boolean	True に設定した場合はメニューを非表示にします
deselectSiblings	Boolean	現在選択されているメニュー項目の同階層を選択解除します
focusTrigger	Boolean	フォーカスを ui:menuTrigger コンポーネントに設定します

ui:menuTriggerPress

メニュートリガがクリックされたことを示します。

たとえば、ui:menuTrigger コンポーネントはこのイベントを登録し、コンポーネントでイベントを起動できるようにします。

```
<aura:registerEvent name="menuTriggerPress"  type="ui:menuTriggerPress"
                    description="The event fired when the trigger is clicked." />
```

このイベントは、ui:menuTrigger を拡張する ui:menuTriggerLink コンポーネントインスタンスなどのコンポーネントで処理できます。

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"
menuTriggerPress="{!c.triggered}"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

このクライアント側のコントローラは、クリックされたときにトリガの表示ラベルを取得します。

```
({
    triggered : function(component, event, helper) {
        var trigger = component.find("trigger");

        // Get the label on the trigger
        var triggerLabel = trigger.get("v.label");
    }
})
```

ui:validationError

コンポーネントに検証エラーがあることを示します。

ui:validationError イベントのハンドラを設定するには、ui:inputNumber などの ui:input を拡張するコンポーネントで onError システム属性を使用します。

次の ui:inputNumber コンポーネントは、ui:button コンポーネントが押されたときにエラーを処理します。これらのイベントは、クライアント側コントローラで起動および処理できます。

```
<aura:component>
    Enter a number:
    <!-- onError calls your client-side controller to handle a validation error -->
    <!-- onClearErrors calls your client-side controller to handle clearing of errors -->

    <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/>

    <!-- press calls your client-side controller to trigger validation errors -->
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

詳細は、「項目の検証」 (ページ 290) を参照してください。

属性名	型	説明
errors	Object[]	エラーメッセージの配列

wave:discoverDashboard

このイベントは、リスンしている Analytics ダッシュボードアセットに対し、識別した情報を返すようにグローバル要求を送信します。必要に応じて、応答に含める独自のパラメータを追加できます。

wave:discoverDashboard および *wave:discoverResponse* イベントは連携します。また、ダッシュボードが動的にページに追加されたときを確認したり、複数のダッシュボードがページにあるかどうか検出したりする場合に特に便利です。

このイベントには、1 つの属性 (応答データに含まれる任意の識別子) が含まれます。

この例では、Lightning コンポーネントはすでに定義済み、ハンドラは設定済みで、*wave:discoverDashboard* と *wave:discoverResponse* イベントがカスタムコンポーネントマークアップに登録されています。次のコントローラコードは、*wave:discoverDashboard* イベントの起動方法と、*wave:discoverResponse* イベントが起動した場合の結果の使用方法を示しています。ダッシュボードコンポーネントの作成方法も示します。

```
({
    addDashboard: function(component, event, helper) {
        var selectCmp = component.find("idTextBox");
        component.set("v.dashboardId", selectCmp.get("v.value"));
        var config = {
            "dashboardId": selectCmp.get("v.value"),
            "showHeader": false,
            "height": 400
        };
        $A.createComponent("wave:waveDashboard", config,
            function(dashboard, status, err) {
                if (status === "SUCCESS") {
                    dashboard.set("v.rendered", true);
                    dashboard.set("v.showHeader", false);
                    component.set("v.body", dashboard);

                } else if (status === "INCOMPLETE") {
                    console.log("No response from server or client is offline.")
                } else if (status === "ERROR") {
                    console.log("Error: " + err);
                }
            }
        );
    },
    discoverDashboard: function(component, event, helper) {
        $A.get("e.wave:discover").fire();
    },
    handleDiscoverResponse: function(cmp, event, helper) {
        var myText = cmp.find("outName");
        myText.set("v.value", event.getParam("id"));
```

```
    },
})
```

メモ: Analytics Platform ライセンス Insights Builder PSL が必要です。

属性名	型	説明
UID	String	応答データに含まれる任意の識別子。

Lightning での Analytics SDK の使用についての詳細は、『Analytics SDK Developer Guide (Analytics SDK 開発者ガイド)』を参照してください。

wave:discoverResponse

このイベントは、Analytics ダッシュボードに対するアセット識別要求の後に応答を提供します。

このイベントペイロードには、ダッシュボードの一意のID、コンポーネントの種類、ダッシュボードのタイトル、ダッシュボードの読み込み状況、および要求で送信された場合は省略可能なパラメータの5つの属性があります。

wave:discoverDashboard および *wave:discoverResponse* イベントは連携します。また、ダッシュボードが動的にページに追加されたときを確認したり、複数のダッシュボードがページにあるかどうか検出したりする場合に特に便利です。

wave:discoverResponse の使用の詳細と例については、*wave:discoverDashboard* イベントを参照してください。

メモ: Analytics Platform ライセンス Insights Builder PSL が必要です。

属性名	型	説明
id	String	ダッシュボードの一意の識別子 (標準の 18 文字 ID 形式)。
type	String	コンポーネントの種類。通常はダッシュボードです。
title	String	ダッシュボードのタイトル。
isLoaded	Boolean	ダッシュボードが読み込まれたかどうか、またはまだ読み込み中か。
UID	String	要求で送信された省略可能なパラメータ (存在する場合)。

Lightning での Analytics SDK の使用についての詳細は、『Analytics SDK Developer Guide (Analytics SDK 開発者ガイド)』を参照してください。

wave:selectionChanged

Wave ダッシュボードで起動されるイベント。必要なステップの名前、および現在の選択を表すオブジェクトの配列を含む、選択情報が提供されます。

この例では、Lightningコンポーネントはすでに定義済みですべて登録されているため、次のコントローラコードはペイロードの受信と反復方法を示しています。ペイロードは、現在の選択を表すオブジェクトの配列です。

```
({
  handleselectionChanged: function(component, event, helper) {
    var params = event.getParams();
    var payload = params.payload;
    if (payload) {
      var step = payload.step;
      var data = payload.data;
      data.forEach(function(obj) {
        for (var k in obj) {
          if (k === 'Id') {
            component.set("v.recordId", obj[k]);
          }
        }
      });
    }
  }
})
```

メモ: Analytics Platform ライセンス Insights Builder PSL が必要です。

属性名	種別	説明
Id	String	選択変更イベントが発生した Wave アセットの一意の識別子。
noun	String	選択イベントが発生した Wave アセットの一意の識別子。現在、dashboard のみがサポートされています。
payload	String	イベントを起動したアセットからの選択情報を含んでいます。 payload.step (文字列)。選択が発生したステップの名前。 payload.data (オブジェクトの配列)。現在の選択を表すオブジェクトの配列。配列内の各オブジェクトには、選択に基づく1つ以上の属性が含まれます。
verb	String	Wave アセットで発生したアクション。現在、selection のみがサポートされています。

Lightning での Analytics SDK の使用についての詳細は、『Analytics SDK Developer Guide (Analytics SDK 開発者ガイド)』を参照してください。

wave:update

このイベントは、Wave Analytics ダッシュボードの検索条件を設定したり、選択を動的に変更してダッシュボードとやりとりするために使用します。

このイベントには、検索条件を適用する Wave アセットの一意の ID、ペイロード、およびアセットタイプ (現在はダッシュボードのみ) の 3 つの属性があります。ペイロードは、データセットおよびディメンションと項目値を識別する JSON 文字列です。

この例では、Lightning コンポーネントはすでに定義済み、ハンドラは設定済みで、更新イベントがカスタムコンポーネントマークアップに登録されています。次のコントローラコードは、更新イベントのペイロードの作成方法を示しています。この場合、oppty_test ダッシュボードの StageName 設定を「Closed Won」(商談成立) に設定しています。

```
({
    doInit: function(component, event, helper) {
      component.set('v.filter', '{"oppty_test": {"StageName": ["Closed Won"]}}');
    },

    handleSendFilter: function(component, event, helper) {
      var filter = component.get('v.filter');
      var dashboardId = component.get('v.dashboardId');
      var evt = $A.get('e.wave:update);
        evt.setParams({
          id: dashboardId,
          value: filter,
          Type: "dashboard"
        }); evt.fire();
      }
})
```

メモ: Analytics Platform ライセンス Insights Builder PSL が必要です。

属性名	種別	説明
id	String	選択変更イベントが発生した Wave アセットの一意の識別子 (標準の 18 文字 ID 形式)。
value	String	アセットに適用される検索条件または選択を表す JSON。
type	String	Wave アセットの種別。現在、dashboard のみがサポートされています。

Lightning での Analytics SDK の使用についての詳細は、『Analytics SDK Developer Guide (Analytics SDK 開発者ガイド)』を参照してください。

システムイベントの参照

システムイベントは、そのライフサイクルの間にフレームワークによって起動されます。これらのイベントは、Lightning アプリケーション/コンポーネント、および Salesforce1 内で処理できます。たとえば、次のイベントでは、属性値の変更や URL の変更を処理したり、アプリケーションまたはコンポーネントでサーバ応答を待機している場合の処理を行うことができます。

aura:doneRendering

ルートアプリケーションの初期表示が完了したことを示します。

メモ: aura:doneRendering イベントは、最後の手段としてのみ使用することをお勧めします。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience や Salesforce1 などの複雑なアプリケーションに含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、イベントハンドラを複数回トリガすることがあります。

このイベントは、表示する必要があるコンポーネントが他にない場合、またはいずれかの属性値が変更されたため再表示する必要がある場合に自動的に起動されます。aura:doneRendering イベントは、クライアント側のコントローラで処理されます。このイベントを処理するには、コンポーネントに <aura:handler> タグを 1 つだけ指定します。

```
<aura:handler event="aura:doneRendering" action="{!c.doneRendering}"/>
```

たとえば、アプリケーションが初回の表示を完了した後の動作をカスタマイズし、その後の再表示時の動作はカスタマイズしないとします。初回の表示かどうか判定するための属性を作成します。

```
<aura:component>
    <aura:handler event="aura:doneRendering" action="{!c.doneRendering}"/>
    <aura:attribute name="isDoneRendering" type="Boolean" default="false"/>
    <!-- Other component markup here -->
    <p>My component</p>
</aura:component>
```

次のクライアント側のコントローラは、aura:doneRendering イベントが 1 回だけ起動されたことを確認します。

```
({
  doneRendering: function(cmp, event, helper) {
    if(!cmp.get("v.isDoneRendering")){
      cmp.set("v.isDoneRendering", true);
      //do something after component is first rendered
    }
  }
})
```

メモ: aura:doneRendering が起動されると、component.isRendered() から true が返されます。要素が DOM で表示されるかどうかを確認するには、component.getElement()、component.hasClass()、または element.style.display などのユーティリティを使用します。

aura:doneRendering ハンドラには、次の必須属性があります。

属性名	型	説明
event	String	イベント名。aura:doneRendering に設定する必要があります。
action	Object	イベントを処理するクライアント側のコントローラアクション。

aura:doneWaiting

アプリケーションでサーバ要求への応答の待機が終了したことを示します。このイベントの前には aura:waiting イベントがあります。このイベントは、aura:waiting の後で起動されます。

メモ: aura:doneWaiting イベントは、最後の手段としてのみ使用することをお勧めします。aura:doneWaiting アプリケーションイベントは、サーバ応答(アプリケーションの他のコンポーネントからの応答も含む) ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。

このイベントは、サーバから他の応答が予期されない場合に自動的に起動されます。aura:doneWaiting イベントは、クライアント側のコントローラで処理されます。このイベントを処理するには、コンポーネントに <aura:handler> タグを 1 つだけ指定します。

```
<aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>
```

次の例では、aura:doneWaiting が起動されたときにスピナーを非表示にします。

```
<aura:component>
     <aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>
    <!-- Other component markup here -->
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

次のクライアント側のコントローラは、スピナーを非表示にするイベントを起動します。

```
({
    hideSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : false });
        evt.fire();
    }
})
```

aura:doneWaiting ハンドラには、次の必須属性があります。

属性名	型	説明
event	String	イベント名。aura:doneWaiting に設定する必要があります。
action	Object	イベントを処理するクライアント側のコントローラアクション。

aura:locationChange

URL のハッシュ部分が変更されたことを示します。

このイベントは、新しい場所トークンがハッシュに追加されるなど、URL のハッシュ部分が変更された場合に自動的に起動されます。aura:locationChange イベントは、クライアント側のコントローラで処理されま

す。このイベントを処理するには、コンポーネントに `<aura:handler event="aura:locationChange">` タグを 1 つだけ指定します。

```
<aura:handler event="aura:locationChange" action="{!c.update}"/>
```

次のクライアント側のコントローラは、aura:locationChange イベントを処理します。

```
({
    update : function (component, event, helper) {
        // Get the new location token from the event
        var loc = event.getParam("token");
        // Do something else
    }
})
```

aura:locationChange ハンドラには、次の必須属性があります。

属性名	型	説明
event	String	イベント名。aura:locationChange に設定する必要があります。
action	Object	イベントを処理するクライアント側のコントローラアクション。

aura:locationChange イベントには、次の属性があります。

属性名	型	説明
token	String	URL のハッシュ部分。
querystring	Object	ハッシュのクエリ文字列部分。

aura:systemError

エラーが発生したことを示します。

このイベントは、サーバ側のアクションの実行中にエラーが発生した場合に自動的に起動されます。aura:systemError イベントは、クライアント側のコントローラで処理されます。このイベントを処理するには、マークアップでコンポーネントに `<aura:handler event="aura:systemError">` タグを 1 つだけ指定します。

```
<aura:handler event="aura:systemError" action="{!c.handleError}"/>
```

次の例に、エラーをトリガするボタンと、aura:systemError イベントのハンドラを示します。

```
<aura:component controller="namespace.myController">
    <aura:handler event="aura:systemError" action="{!c.showSystemError}"/>
    <aura:attribute name="response" type="Aura.Action"/>
    <!-- Other component markup here -->
    <ui:button aura:id="trigger" label="Trigger error" press="{!c.trigger}"/>
</aura:component>
```

次のクライアント側のコントローラは、エラーの起動をトリガし、そのエラーを処理します。

```
({
    trigger: function(cmp, event) {
        // Call an Apex controller that throws an error
        var action = cmp.get("c.throwError");
        action.setCallback(cmp, function(response){
            cmp.set("v.response", response);
        });
        $A.enqueueAction(action);
    },

    showSystemError: function(cmp, event) {
        // Handle system error
        console.log(cmp);
        console.log(event);
    }
})
```

aura:systemError イベントの aura:handler タグには、次の必須属性があります。

属性名	型	説明
event	String	イベント名。aura:systemError に設定する必要があります。
action	Object	イベントを処理するクライアント側のコントローラアクション。

aura:systemError イベントには、次の属性があります。event.getParam("***attributeName***") を使用して属性値を取得できます。

属性名	型	説明
message	String	エラーメッセージ。
error	String	error オブジェクト。

関連トピック:

エラーの発生および処理

aura:valueChange

属性値が変更されたことを示します。

このイベントは、属性値が変更された場合に自動的に起動されます。aura:valueChange イベントは、クライアント側のコントローラで処理されます。コンポーネントに複数の <aura:handler name="change"> タグを設定して、さまざまな属性の変更を検出できます。

```
<aura:handler name="change" value="{!v.items}" action="{!c.itemsChange}"/>
```

次の例に、aura:valueChange イベントを自動的に起動する Boolean 値の更新を示します。

```
<aura:component>
    <aura:attribute name="myBool" type="Boolean" default="true"/>

    <!-- Handles the aura:valueChange event -->
    <aura:handler name="change" value="{!v.myBool}" action="{!c.handleValueChange}"/>
    <ui:button label="change value" press="{!c.changeValue}"/>
</aura:component>
```

次のクライアント側コントローラのアクションは、値の変更をトリガし、それを処理します。

```
({
    changeValue : function (component, event, helper) {
      component.set("v.myBool", false);
    },

    handleValueChange : function (component, event, helper) {
        // handle value change
        console.log("old value: " + event.getParam("oldValue"));
        console.log("current value: " + event.getParam("value"));
    }
})
```

valueChange イベントは、ハンドラのアクションで以前の値 (oldValue) と現在の値 (value) にアクセスできるようにします。この例では、oldValue が true を返し、value が false を返します。

change ハンドラには、次の必須属性があります。

属性名	型	説明
name	String	ハンドラ名。change に設定する必要があります。
value	Object	変更を検出する属性。
action	Object	値の変更を処理するクライアント側のコントローラアクション。

関連トピック:

変更ハンドラを使用したデータ変更の検出

aura:valueDestroy

コンポーネントが破棄されたことを示します。コンポーネントの破棄時にカスタムクリーンアップを実行する場合、このイベントを処理します。

このイベントは、コンポーネントの破棄処理中に自動的に起動されます。aura:valueDestroy イベントは、クライアント側コントローラで処理されます。

このイベントを処理するには、コンポーネントに <aura:handler name="destroy"> タグを 1 つだけ指定します。

```
<aura:handler name="destroy" value="{!this}" action="{!c.handleDestroy}"/>
```

次のクライアント側のコントローラは、aura:valueDestroy イベントを処理します。

```
({
    handleDestroy : function (component, event, helper) {
      var val = event.getParam("value");
      // Do something else here
    }
})
```

たとえば、Salesforce1 でコンポーネントを表示しているとします。Salesforce1 ナビゲーションメニューで異なるメニュー項目をタップすると、aura:valueDestroy イベントがトリガされ、コンポーネントが破棄されます。この例では、イベントの value パラメータによって、破棄処理中のコンポーネントが返されます。

aura:valueDestroy イベントの <aura:handler> タグには、次の必須属性があります。

属性名	型	説明
name	String	ハンドラ名。destroy に設定する必要があります。
value	Object	イベントを検出する値。破棄処理中の値。常に value="{!this}" を設定します。
action	Object	破棄イベントを処理するクライアント側コントローラのアクション。

aura:valueDestroy イベントには、次の属性があります。

属性名	型	説明
value	String	event.getParam("value") を使用して取得される破棄処理中のコンポーネント。

aura:valueInit

アプリケーションまたはコンポーネントが初期化されたことを示します。

このイベントは、アプリケーションまたはコンポーネントが表示前に初期化された場合に自動的に起動されます。aura:valueInit イベントは、クライアント側のコントローラで処理されます。このイベントを処理するには、コンポーネントに <aura:handler name="init"> タグを 1 つだけ指定します。

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

「コンポーネントの初期化時のアクションの呼び出し」(ページ 271)の例を参照してください。

メモ: value="{!this}" を設定すると、これ自体が値のイベントとしてマークされます。init イベントでは、常にこの設定を使用する必要があります。

init ハンドラには、次の必須属性があります。

属性名	型	説明
name	String	ハンドラ名。init に設定する必要があります。

属性名	型	説明
value	Object	初期化される値。{!this} に設定する必要があります。
action	Object	値の変更を処理するクライアント側のコントローラアクション。

関連トピック:

コンポーネントの初期化時のアクションの呼び出し

aura:valueRender

aura:valueRender

アプリケーションまたはコンポーネントが表示または再表示されたことを示します。

このイベントは、アプリケーションまたはコンポーネントが表示または再表示された場合に自動的に起動されます。aura:valueRender イベントは、クライアント側コントローラで処理されます。このイベントを処理するには、コンポーネントに <aura:handler name="render"> タグを 1 つだけ指定します。

```
<aura:handler name="render" value="{!this}" action="{!c.onRender}"/>
```

この例では、クライアント側コントローラの onRender アクションがコンポーネントの最初の表示と再表示を処理します。action 属性には任意の名前を選択できます。

メモ: value="{!this}" を設定すると、これ自体が値のイベントとしてマークされます。render イベントでは、常にこの設定を使用する必要があります。

render イベントは、コンポーネントを構築してから表示するまでの間に起動される init イベントの後に起動されます。

aura:valueRender イベントには、属性が 1 つだけあります。

属性名	属性型	説明
value	Object	表示または再表示されたコンポーネント。

関連トピック:

aura:valueInit

表示ライフサイクル中に起動されたイベント

aura:waiting

アプリケーションでサーバ要求への応答を待機していることを示します。このイベントは、aura:doneWaiting の前に起動されます。

メモ: 最後の手段として使用する場合を除き、従来の aura:waiting イベントの使用はおすすめしません。アプリケーションの他のコンポーネントからの要求であっても、aura:waiting アプリケーションイベントはサーバー要求ごとに起動されます。コンポーネントがスタンドアロンアプリケーションで完全に独立した状態で実行されていて、Lightning Experience または Salesforce1 に含まれていない場合を除き、

> このアプリケーションイベントを処理することはおそらくないでしょう。コンテナアプリケーションは、サーバ側アクションを起動して、イベントハンドラを複数回トリガすることがあります。

このイベントは、`$A.enqueueAction()` を使用してサーバ側のアクションが追加されその後で実行された場合、または Apex コントローラからの応答を予期している場合に自動的に起動されます。`aura:waiting` イベントは、クライアント側のコントローラで処理されます。このイベントを処理するには、コンポーネントに `<aura:handler>` タグを 1 つだけ指定します。

```
<aura:handler event="aura:waiting" action="{!c.showSpinner}"/>
```

次の例に、`aura:waiting` が起動されたときのスピナーを示します。

```
<aura:component>
     <aura:handler event="aura:waiting" action="{!c.showSpinner}"/>
    <!-- Other component markup here -->
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

次のクライアント側のコントローラは、スピナーを表示するイベントを起動します。

```
({
    showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : true });
        evt.fire();
    }
})
```

`aura:waiting` ハンドラには、次の必須属性があります。

属性名	型	説明
event	String	イベント名。aura:waiting に設定する必要があります。
action	Object	イベントを処理するクライアント側のコントローラアクション。

サポートされる HTML タグ

フレームワークでは、大半の HTML5 タグを含む、ほとんどの HTML タグがサポートされています。

HTML タグは、フレームワークで第一級のコンポーネントとして処理されます。各HTML タグは、`<aura:html>` コンポーネントに変換され、他のコンポーネントと同様の権限を使用できます。

たとえば、フレームワークは、標準の HTML `<div>` タグを次のコンポーネントに自動的に変換します。

```
<aura:html tag="div" />
```

HTML タグよりコンポーネントを優先して使用することをお勧めします。たとえば、`<button>` ではなく `lightning:button` を使用します。

コンポーネントはアクセシビリティを念頭に置いて設計されているため、障害があるユーザや支援技術を使用するユーザもアプリケーションを使用できます。より複雑なコンポーネントを構築する場合、再利用可能な標

準コンポーネントを使用すれば、本来であれば自分で作成しなければならないプラミングの一部を標準コンポーネントが処理してくれるため、作業を簡略化できます。また、これらのコンポーネントは安全であり、パフォーマンスが最適化されています。

厳密な XHTML を使用する必要がある点に注意してください。たとえば、`<br>` ではなく `<br/>` を使用します。

一部の HTML タグは、安全でないか不要です。フレームワークでは、次のタグはサポートされていません。

こちらのオープンソースの Aura ファイルの `HtmlTag` enum は、サポート対象の HTML タグを列挙しています。`(false)` の付いているタグはサポートされていません。たとえば、`applet(false)` は `applet` タグがサポートされていないことを示します。

メモ: リンク先のファイルは、オープンソースの Aura プロジェクトのマスタブランチに含まれています。マスタブランチとは、Salesforce の現在の開発ブランチのことで、Lightning コンポーネントフレームワークの最新リリースより進んだ状態にあります。一方で、このファイルは変更されることがほとんどなく、タグが現在または将来サポートされるかどうかを確認するには最適です。

アンカータグの `href` 属性での `#` の回避

ハッシュ記号 (#) は URL フラグメント識別子であり、ページ内のナビゲーションの Web 開発でよく使用されます。Lightning コンポーネントのアンカータグの `href` 属性で `#` は使用しないでください。特に Salesforce1 モバイルアプリケーションで、予期しないナビゲーション変更が発生する可能性があるためです。たとえば、`href="#"` ではなく `href=""` を使用します。

関連トピック:

アクセシビリティのサポート