



Lightning コンポーネント開発者ガイド

バージョン 35.0, Winter '16



©Copyright 2000–2015 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

使用開始	1
第 1 章: はじめに	1
Lightning コンポーネントフレームワークとは?	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
ブラウザサポート	4
開発者コンソールの使用	5
オープンソースの Aura フレームワーク	6
このガイドのオンラインバージョン	6
第 2 章: クイックスタート	7
ご利用になる前に	8
スタンドアロン Lightning アプリケーションを作成する	9
省略可能: 経費追跡アプリケーションのインストール	11
経費オブジェクトを作成する	12
ステップ 1: 静的モックアップを作成する	15
ステップ 2: ユーザ入力用のコンポーネントを作成する	18
ステップ 3: 経費データを読み込む	23
ステップ 4: ネストされたコンポーネントを作成する	25
ステップ 5: 新しい経費の入力を有効化する	27
ステップ 6: アプリケーションとイベントをインタラクティブにする	30
まとめ	34
Salesforce のコンポーネントの作成	36
省略可能: 取引先責任者リストアプリケーションのインストール	37
取引先責任者の読み込み	38
イベントの起動	46
コンポーネントの作成	50
第 3 章: コンポーネント	50
コンポーネントのマークアップ	52
コンポーネントの名前空間	53
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	53
組織の名前空間の使用	54
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	54
組織の名前空間の作成	54

名前空間の使用例および参照	55
コンポーネントのバンドル	58
コンポーネントの ID	60
コンポーネント内の HTML	60
コンポーネント内の CSS	61
コンポーネントの属性	63
コンポーネントのコンポジション	64
コンポーネントのボディ	68
コンポーネントのファセット	70
条件付きマークアップのベストプラクティス	71
表示ラベルの使用	72
入力コンポーネントの表示ラベル	73
親属性による表示ラベル値の設定	73
ローカライズ	74
Lightning コンポーネントの有効化	75
Salesforce への Lightning コンポーネントの追加	76
Lightning ページと Lightning App Builder のコンポーネントの設定	77
アプリケーションへのコンポーネントの追加	79
コンポーネントのドキュメントの提供	80
第 4 章: 式	83
式の動的出力	84
条件式	84
値プロバイダ	85
グローバル値プロバイダ	86
式の評価	89
式の演算子のリファレンス	90
式の関数のリファレンス	93
第 5 章: UI コンポーネント	97
UI イベント	99
UI コンポーネントの使用	100
日時項目	101
数値項目	102
テキスト項目	105
リッチテキスト項目	107
チェックボックス	108
ラジオボタン	110
ボタン	112
ドロップダウンリスト	114
項目レベルのエラー	117
メニュー	117
第 6 章: アクセシビリティのサポート	119

アクセシビリティの考慮事項	120
ボタンの表示ラベル	120
ヘルプとエラーメッセージ	121
音声メッセージ	121
フォーム、項目、および表示ラベル	122
イベント	122
メニュー	122
 イベントとの通信	124
第 7 章: イベント	124
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	125
コンポーネントイベント	127
コンポーネントイベントの例	129
アプリケーションイベント	133
アプリケーションイベントの例	135
イベント処理のライフサイクル	138
高度なイベントの例	140
非 Lightning コードからの Lightning イベントの起動	147
イベントのベストプラクティス	147
イベントのアンチパターン	148
表示ライフサイクル中に起動されたイベント	149
第 8 章: Salesforce イベント	153
第 9 章: システムイベント	155
 アプリケーションの作成	156
第 10 章: アプリケーションの基本	156
アプリケーションの概要	157
アプリケーションの UI の設計	157
コンテンツセキュリティポリシーの概要	157
第 11 章: アプリケーションのスタイル設定	159
外部 CSS の使用	159
ベンダープレフィックス	160
第 12 章: JavaScript の使用	161
DOM へのアクセス	162
外部 JavaScript ライブラリの使用	162
JavaScript での属性値の操作	163
JavaScript でのコンポーネントのボディの操作	165
コンポーネントのバンドル内の JavaScript コードの共有	166
DOM へのクライアント側表示	169

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	173
項目の検証	174
エラーの発生および処理	178
API コールの実行	180
第 13 章: JavaScript Cookbook	181
コンポーネントの初期化時のアクションの呼び出し	182
データ変更の検出	183
ID によるコンポーネントの検索	184
コンポーネントの動的な作成	184
イベントハンドラの動的な追加	187
マークアップの動的な表示または非表示	188
スタイルの追加と削除	188
第 14 章: Apex の使用	191
コントローラのサーバ側ロジックの作成	192
Apex サーバ側コントローラの概要	192
Apex サーバ側コントローラの作成	193
サーバ側のアクションのコール	194
サーバ側のアクションのキュー配置	197
中止可能なアクション	197
コンポーネントの操作	199
Salesforce レコードの操作	200
Apex コードのテスト	205
Apex からの API コールの実行	206
第 15 章: オブジェクト指向開発の使用	207
継承とは?	208
継承されるコンポーネントの属性	209
抽象コンポーネント	211
インターフェース	211
マーカーインターフェース	212
継承ルール	212
第 16 章: AppCache の使用	214
AppCache の有効化	215
AppCache を使用したリソースの読み込み	215
第 17 章: アクセスの制御	216
アプリケーションのアクセス制御	217
インターフェースのアクセス制御	217
コンポーネントのアクセス制御	217
属性のアクセス制御	218
イベントのアクセス制御	218

第 18 章: アプリケーションとコンポーネントの配布	219
デバッグ	220
第 19 章: デバッグ	220
JavaScript コードのデバッグ	221
ログメッセージ	221
警告メッセージ	223
リファレンス	224
第 20 章: リファレンスの概要	224
リファレンスドキュメントアプリケーション	225
aura:application	225
aura:dependency	226
aura:event	227
aura:interface	227
aura:set	228
スーパー・コンポーネントから継承される属性の設定	228
コンポーネント参照での属性の設定	229
インターフェースから継承される属性の設定	230
コンポーネントの参照	231
aura:component	231
aura:expression	232
aura:html	232
aura:if	233
aura:iteration	234
aura:renderIf	234
aura:text	235
aura:unesapedHtml	235
force:inputField	236
force:outputField	236
force:recordEdit	237
force:recordView	238
forceChatter:feed	238
ltng:require	240
ui:actionMenuItem	240
ui:button	242
ui:checkboxMenuItem	244
ui:inputCheckbox	246
ui:inputCurrency	249
ui:inputDate	252
ui:inputDateTime	256
ui:inputDefaultError	260

目次

ui:inputEmail	262
ui:inputNumber	265
ui:inputPhone	268
ui:inputRadio	271
ui:inputRichText	275
ui:inputSecret	278
ui:inputSelect	280
ui:inputSelectOption	287
ui:inputText	288
ui:inputTextArea	291
ui:inputURL	294
ui:menu	297
ui:menuItem	304
ui:menuItemSeparator	306
ui:menuList	307
ui:menuTrigger	308
ui:menuTriggerLink	309
ui:message	310
ui:outputCheckbox	312
ui:outputCurrency	314
ui:outputDate	316
ui:outputDateTime	318
ui:outputEmail	321
ui:outputNumber	323
ui:outputPhone	324
ui:outputRichText	326
ui:outputText	328
ui:outputTextArea	330
ui:outputURL	332
ui:radioMenuItem	334
ui:spinner	335
イベントの参照	337
force:createRecord	337
force:editRecord	338
force:navigateToList	338
force:navigateToObjectHome	340
force:navigateToRelatedList	341
force:navigateToSObject	341
force:navigateToURL	342
force:recordSave	343
force:recordSaveSuccess	344
force:refreshView	345
force:showToast	345
ui:clearErrors	346

目次

ui:collapse	347
ui:expand	348
ui:menuSelect	349
ui:menuTriggerPress	350
ui:validationError	351
システムイベントの参照	352
aura:doneRendering	352
aura:doneWaiting	353
aura:locationChange	354
aura:systemError	355
aura:valueChange	356
aura:valueDestroy	357
aura:valueInit	358
aura:waiting	359
サポートされる HTML タグ	360
サポートされる aura:attribute の型	360
基本の型	361
オブジェクト型	364
標準オブジェクト型とカスタムオブジェクト型	364
コレクション型	364
カスタム Apex クラス型	366
フレームワーク固有の型	366

使用開始

第1章 はじめに

トピック:

- Lightning コンポーネントフレームワークとは?
- 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
- ブラウザサポート
- 開発者コンソールの使用
- オープンソースのAura フレームワーク
- このガイドのオンラインバージョン

Salesforce1 Lightning の概要

昨今のマルチデバイスの増加を背景に、どのような画面にも対応するアプリケーションを簡単に構築できる Lightning を作成しました。Lightning には、次のような、開発者向けの魅力的なツールが多数用意されています。

1. 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での使用に最適です。
2. Lightning App Builder。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。

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自体が 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 コンポーネントはもちろん、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とは?

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。



メモ: このリリースには、Lightning コンポーネントフレームワークのベータバージョンが含まれています。本番品質ではありますが、いくつかの制限があります。

Lightning コンポーネントフレームワークと 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 (<http://github.com/forcedotcom/aura> で入手可能) 上に構築されています。Aura フレームワークを使用して、独自のサーバ上でホストされる汎用的な Web アプリケーションを作成できます。このフレームワークは、Lightning コンポーネントフレームワークでは使用できない可能性のある機能やコンポーネントも提供します。ここに示すサンプルコードの多くは、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が使用しています。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。

Visualforce と Lightning

Lightning コンポーネントフレームワークは、コンポーネントベースのフレームワークです。アプリケーションのビルディングブロックとなるコンポーネントは、HTML、JavaScript、CSS をカプセル化する一方で、イベントを介してやりとりします。Lightning コンポーネントはクライアント側主体で、動的性能を高めてモバイルで使いやすくします。対照的に、Visualforce コンポーネントはページ主体で、サーバコールに大きく依存します。Visualforce は、テンプレート主導の Web ページやメールメッセージの配信を促進し、要求のライフサイクルにわたって綿密な管理を希望する開発者に適しています。Lightning コンポーネントはサーバ側で Apex を使用するため、Apex コードを使用不能な組織は、Lightning コンポーネントを作成できません(ただし、Visualforce は使用できます)。

Lightning コンポーネントフレームワークを使用する理由

Lightning コンポーネントフレームワークを使用したアプリケーションの構築にはさまざまな利点があります。

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はありません。

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ー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ーフェースに集中できます。

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。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。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](#)

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、基本的には、従来のパブリッシュ/サブスクライブモデルです。イベントが起動すると、そのイベントのハンドラがあるすべてのコンポーネントに通知されます。

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](#)

[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](#)

ブラウザサポート

フレームワークは、主要なプラットフォームの次の Web ブラウザの最新の安定バージョンをサポートしています (例外は注記のとおりです)。

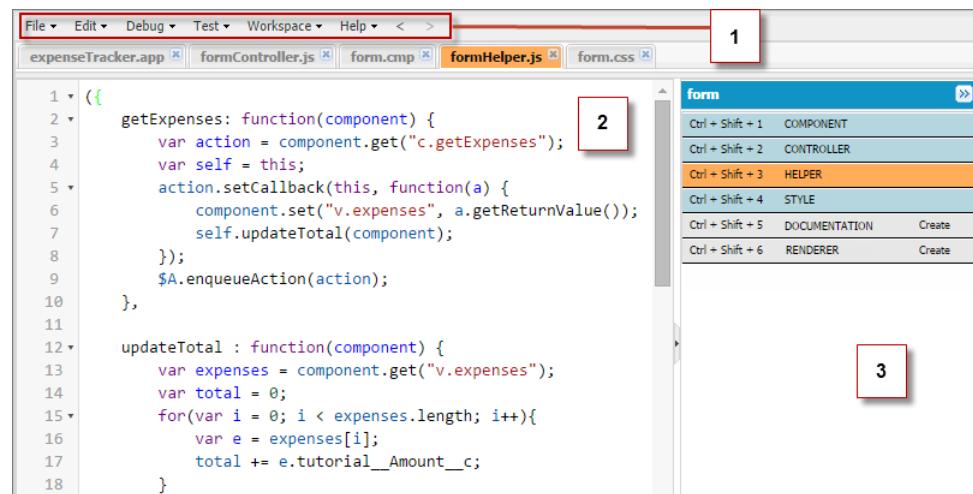
ブラウザ	メモ
Google Chrome™	
Apple® Safari® 5+	Mac OS X および iOS 用
Mozilla® Firefox®	
Microsoft® Internet Explorer®	Internet Explorer 9、10、11 の使用をお勧めします。 Internet Explorer 7 および 8 を使用すると、パフォーマンスが低下することがあります。

 **メモ:** どのブラウザでも、JavaScript および Cookie を有効にする必要があります。

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。

開発者コンソールを開くには、[あなたの名前] > [開発者コンソール] をクリックします。



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
- 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

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

関連トピック:

[コンポーネントのバンドル](#)

オープンソースの Aura フレームワーク

この開発者ガイドでは、全体を通して *Aura* コンポーネントへの参照があります。たとえば、コードサンプルにはコンポーネントの `aura:component` タグがあります。これまで Lightning について述べてきましたが Aura とは何でしょうか? どのような違いがあるのでしょうか? Lightning コンポーネントは、

<https://github.com/forcedotcom/aura> で入手できるオープンソースの Aura フレームワークに基づいています。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。

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。

このガイドのオンラインバージョン

このガイドはオンラインで利用できます。最新バージョンを参照するには、次の場所にアクセスしてください。

<https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/>

第2章 クイックスタート

トピック:

- ご利用になる前に
- スタンドアロン Lightning アプリケーションを作成する
- Salesforce1 のコンポーネントの作成

クイックスタートでは、開発者コンソールから経費を追跡するための、スタンドアロンの簡単な Lightning アプリケーションの作成と実行についてステップごとに説明します。

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. [名前空間プレフィックスを登録する](#)
3. [Lightning コンポーネントを有効化する](#)

 **メモ:** 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 [\[Lightning コンポーネントを有効化\]](#) に移動できます。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要です。このクイックスタートチュートリアルでは名前空間を使用しません。

Developer Edition 組織を作成する

名前空間プレフィックスを登録するには、Developer Edition 組織(略称「DE 組織」)が必要です。DE 組織がない場合は、次の手順に従って作成します。

1. ブラウザで <http://bit.ly/lightningguide> 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@example.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. [Master Subscription Agreement (主登録契約)] を読み、チェックボックスをオンにしてから [Submit Registration (登録を実行)] 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。

名前空間プレフィックスを登録する

次に、名前空間プレフィックスを登録します。名前空間プレフィックスは、すべての Salesforce 組織にわたって必ずグローバルに一意なものを指定します。名前空間プレフィックスでは大文字と小文字が区別され、15 文字以内の英数字で指定できます。

 **メモ:** Lightning コンポーネントを作成するために名前空間は必要ありませんが、管理パッケージを提供する予定の場合は必須です。組織に名前空間がない場合は、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できます。

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] で、[作成] > [パッケージ] をクリックします。
2. [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、[次へ]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。

名前空間は、作成しているコンポーネントおよび Apex クラスのプレフィックスとして使用されます。また、次にアクセスして、作成するアプリケーションをアドレス指定するために名前空間を使用することもできます。

`https://<mySalesforceInstance>.lightning.force.com/<namespace>/<appName>.app`
(<mySalesforceInstance> は、na1 など、組織をホストするインスタンスの名前です)。

Lightning コンポーネントを有効化する

組織で Lightning コンポーネントを有効化するように選択する必要があります。

1. [設定] で、[開発] > [Lightning コンポーネント] をクリックします。
 2. [Lightning コンポーネントを有効化] チェックボックスをオンにします。
-  **警告:** このベータバージョンの Lightning コンポーネントでは、Salesforce1 の Force.com Canvas アプリケーションはサポートされていません。Lightning コンポーネントを有効化すると、組織のすべての Force.com Canvas アプリケーションが Salesforce1 で動作しなくなります。
3. [保存] をクリックします。

スタンドアロン Lightning アプリケーションを作成する

このチュートリアルでは、開発者コンソールを使用して簡単な経費追跡アプリケーションを作成します。

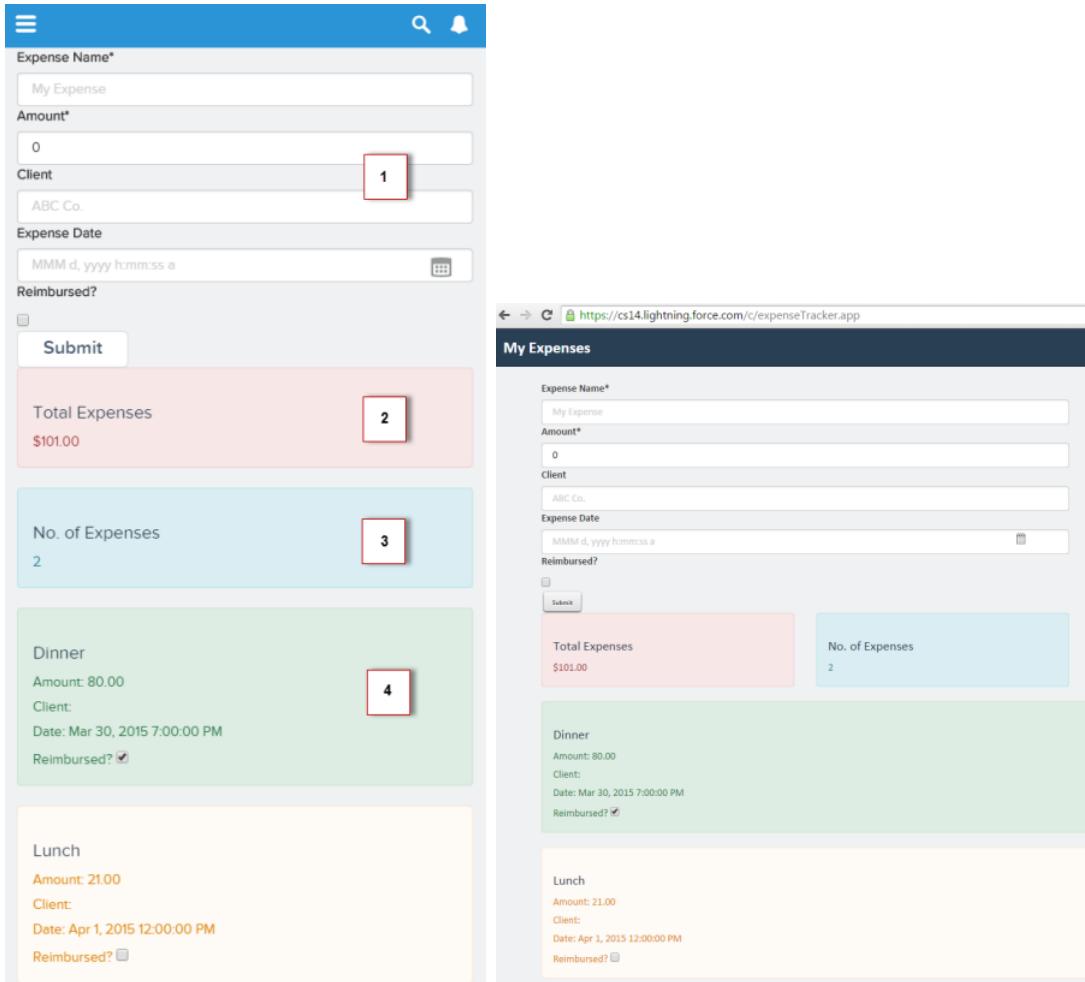
このアプリケーションの目的は、多くの標準 Lightning コンポーネントを利用することと、JavaScript と Apex を使用したクライアントとサーバのやりとりを示すことです。式を使用して動的にデータを操作したり、イベントを使用してコンポーネント間でデータをやりとりしたりする方法をアプリケーションを構築しながら学習していきます。

「[経費オブジェクトを作成する](#)」(ページ 12)で説明されている経費カスタムオブジェクトを作成してあることを確認します。経費データを保存するカスタムオブジェクトを使用して、アプリケーションでレコードを操作する方法、クライアント側のコントローラアクションでユーザ操作を処理する方法、および Apex コントローラでデータ更新を保持する方法を学習します。

コンポーネントを作成したら、[「Salesforce1への Lightning コンポーネントの追加」](#)(ページ 76)の手順に従って、Salesforce1 にそれを追加できます。コンポーネントとアプリケーションのパッケージ化および AppExchange での配布については、[「アプリケーションとコンポーネントの配布」](#)(ページ 219)を参照してください。

 **メモ:** このアプリケーションは、スタンドアロンアプリケーションとしての Salesforce1 から独立して Salesforce1 で使用できます。コンポーネントから既存の Salesforce1 を利用するコンポーネントを作成し、Salesforce1 でのみ使用するには、「[Salesforce1 のコンポーネントの作成](#)」(ページ 36)を参照してください。

次の画像に、Salesforce1 およびスタンドアロンアプリケーションでの経費追跡を示します。



The screenshot displays two views of an expense tracking application. On the left is the Salesforce1 interface, and on the right is a stand-alone Lightning application.

- Form View (Left):** Shows a form with fields: Expense Name (My Expense), Amount (0), Client (ABC Co.), and Expense Date (MM d, yyyy h:mm:ss a). A red box labeled (1) highlights the "Submit" button. Below the form is a summary section with "Total Expenses" (\$101.00) and "No. of Expenses" (2).
- List View (Right):** Shows a list of expenses. The first expense is "Dinner" with an amount of \$80.00, client ABC Co., date Mar 30, 2015 7:00:00 PM, and reimbursed status checked. The second expense is "Lunch" with an amount of \$21.00, client blank, date Apr 1, 2015 12:00:00 PM, and reimbursed status unchecked.

1. フォームには、[実行]ボタンが押されたときにビューおよび経費レコードを更新する Lightning 入力コンポーネント (1) が含まれます。
 2. 経費の総額と数でカウンタが初期化され (2)、レコードの作成または削除で更新されます。合計が \$100 を超えると、カウンタが赤になります。
 3. 経費リスト (3) の表示には Lightning 出力コンポーネントが使用され、経費が追加されるたびに更新されます。
 4. 経費リストに対するユーザ操作 (4) により、レコードの変更を保存する更新イベントがトリガれます。
- 経費追跡アプリケーションで作成するリソースは次のとおりです。

Resources

説明

expenseTracker バンドル

Resources	説明
expenseTracker.app	他のすべてのコンポーネントが含まれる最上位コンポーネント
Form バンドル	
form.cmp	ユーザ入力を収集する Lightning 入力コンポーネントのコレクション
formController.js	フォームに対するユーザ操作を処理するアクションが含まれるクライアント側のコントローラ
formHelper.js	コントローラアクションによってコールされるクライアント側のヘルパー関数
form.css	フォームコンポーネントのスタイル
expenseList バンドル	
expenseList.cmp	経費レコードのデータを表示する Lightning 出力コンポーネントのコレクション
expenseListController.js	経費リストの表示に対するユーザ操作を処理するアクションが含まれるクライアント側のコントローラ
Apex クラス	
ExpenseController.apxc	データの読み込み、経費レコードの挿入または更新を行う Apex コントローラ
イベント	
updateExpenseItem.evt	経費リストの表示から経費項目が更新されるときに起動されるイベント

省略可能: 経費追跡アプリケーションのインストール

クイックスタートチュートリアルをスキップする場合は、経費追跡アプリケーションをパッケージとしてインストールできます。

パッケージとは、組織にインストール可能なコンポーネントの集合体です。こうしたパッケージアプリケーションは、クイックスタートチュートリアルを使用せずに Lightning アプリケーションについて学習する場合に役立ちます。Lightning コンポーネントを初めて利用する場合は、クイックスタートチュートリアルの説明に従うことをお勧めします。

 **メモ:** クイックスタートのオブジェクトと API 名が同じオブジェクトのない組織にパッケージをインストールします。

経費追跡アプリケーションをインストールする手順は、次のとおりです。

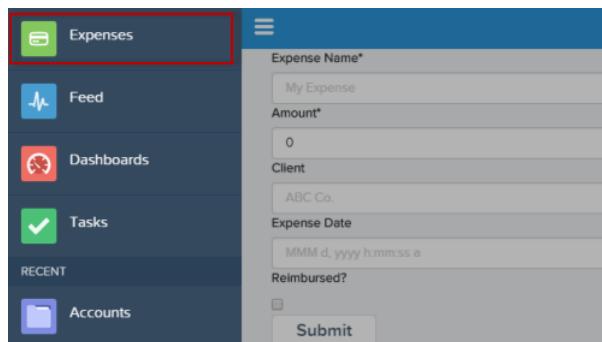
- インストール用 URL リンク <https://login.salesforce.com/packaging/installPackage.apexp?p0=04to0000003onL> をクリックします。
- ユーザ名とパスワードを入力して組織にログインします。

3. [パッケージインストールの詳細] ページで、[続行] をクリックします。
4. [次へ] をクリックし、[セキュリティレベル] ページで [次へ] をクリックします。
5. [インストール] をクリックします。
6. [今すぐリリース] をクリックし、[リリース] をクリックします。

インストールが完了したら、ユーザインターフェースの [Expenses (経費)] タブを選択して、新しい経費レコードを追加できます。



Salesforce1 ナビゲーションメニューにも [Expenses (経費)] メニュー項目が表示されます。Salesforce1 にメニュー項目が表示されない場合は、[モバイル管理] > [モバイルナビゲーション] に移動して追加します。



続いて、開発者コンソールでコードを変更したり、

<https://<mySalesforceInstance>.lightning.force.com/c/expenseTracker.app> でスタンダードアロン アプリケーションを探索したりすることができます。この <mySalesforceInstance> は、組織をホストしているインスタンスの名前 (na1 など) です。

メモ: パッケージを削除するには、[設定] > [インストール済みパッケージ] をクリックします。

経費オブジェクトを作成する

アプリケーションの経費レコードとデータを保存するため、経費オブジェクトを作成します。

チュートリアルの 「[スタンダードアロン Lightning アプリケーションを作成する](#)」 (ページ 9) に従っている場合は、このオブジェクトを作成する必要があります。

1. [設定] で、[作成] > [オブジェクト] をクリックします。
2. [新規カスタムオブジェクト] をクリックします。
3. カスタムオブジェクト定義を入力します。

- ・ [表示ラベル] に 「Expense」 (経費) と入力します。
- ・ [表示ラベル (複数形)] に 「Expenses」 (経費) と入力します。

4. [保存] をクリックして、新規オブジェクトの作成を終了します。経費の詳細ページが表示されます。

 **メモ:** 名前空間プレフィックスを使用している場合は、Expense__c の代わりに namespace__Expense__c が表示されることがあります。

5. 経費の詳細ページで、次のカスタム項目を追加します。

データ型	項目表示ラベル
数値 (16, 2)	Amount (金額)
テキスト (20)	Client (クライアント)
日付/時間	Date (日付)
チェックボックス	Reimbursed? (払い戻し済み?)

カスタムオブジェクトの作成が完了すると、経費定義の詳細ページは次のようにになります。

Custom Object Expense

[Help for this Page](#)

[Standard Fields \[4\]](#) | [Custom Fields & Relationships \[4\]](#) | [Validation Rules \[0\]](#) | [Page Layouts \[1\]](#) | [Field Sets \[0\]](#) |
[Compact Layouts \[1\]](#) | [Search Layouts \[6\]](#) | [Buttons, Links, and Actions \[8\]](#) | [Record Types \[0\]](#) |
[Apex Sharing Reasons \[0\]](#) | [Apex Sharing Recalculation \[0\]](#) | [Object Limits \[10\]](#)

Custom Object Definition Detail

[Edit](#) [Delete](#)

Singular Label	Expense	Description
Plural Label	Expenses	Enable Reports <input type="checkbox"/>
Object Name	Expense	Track Activities <input type="checkbox"/>
API Name	Expense__c	Allow Sharing <input checked="" type="checkbox"/> Allow Bulk API Access <input checked="" type="checkbox"/> Allow Streaming API Access <input checked="" type="checkbox"/> Track Field History <input type="checkbox"/>
		Deployment Status Deployed
		Help Settings Standard salesforce.com Help Window
Created By	Jane Smith, 3/26/2015 11:04 AM	Modified By Jane Smith, 3/26/2015 11:04 AM

Standard Fields

[Standard Fields Help](#)

Action	Field Label	Field Name	Data Type	Controlling Field	Indexed
	Created By	CreatedBy	Lookup(User)		
Edit	Expense Name	Name	Text(80)		<input checked="" type="checkbox"/>
	Last Modified By	LastModifiedBy	Lookup(User)		
Edit	Owner	Owner	Lookup(User,Queue)		<input checked="" type="checkbox"/>

Custom Fields & Relationships

[New](#) [Field Dependencies](#)
[Custom Fields & Relationships Help](#)

Action	Field Label	API Name	Data Type	Indexed	Controlling Field	Modified By
Edit Del	Amount	Amount__c	Number(16,2)			Jane Smith, 3/26/2015 11:04 AM
Edit Del	Client	Client__c	Text(20)			Jane Smith, 3/26/2015 11:04 AM
Edit Del	Date	Date__c	Date/Time			Jane Smith, 3/26/2015 11:04 AM
Edit Del	Reimbursed?	Reimbursed__c	Checkbox			Jane Smith, 3/26/2015 11:04 AM

6. 経費レコードを表示するため、カスタムオブジェクトタブを作成します。
 - a. [設定] で、[作成]>[タブ] をクリックします。
 - b. [カスタムオブジェクトタブ] 関連リストで、[新規] をクリックして新規カスタムタブウィザードを起動します。
 - [オブジェクト] で、[Expense (経費)] を選択します。
 - [タブスタイル] でルックアップアイコンをクリックし、クレジットカードアイコンを選択します。
 - c. 残りのデフォルトを受け入れ、[次へ] をクリックします。
 - d. [次へ] および [保存] をクリックして、タブの作成を終了します。

これで、経費用のタブが画面上部に表示されます。

7. 経費レコードをいくつか作成します。

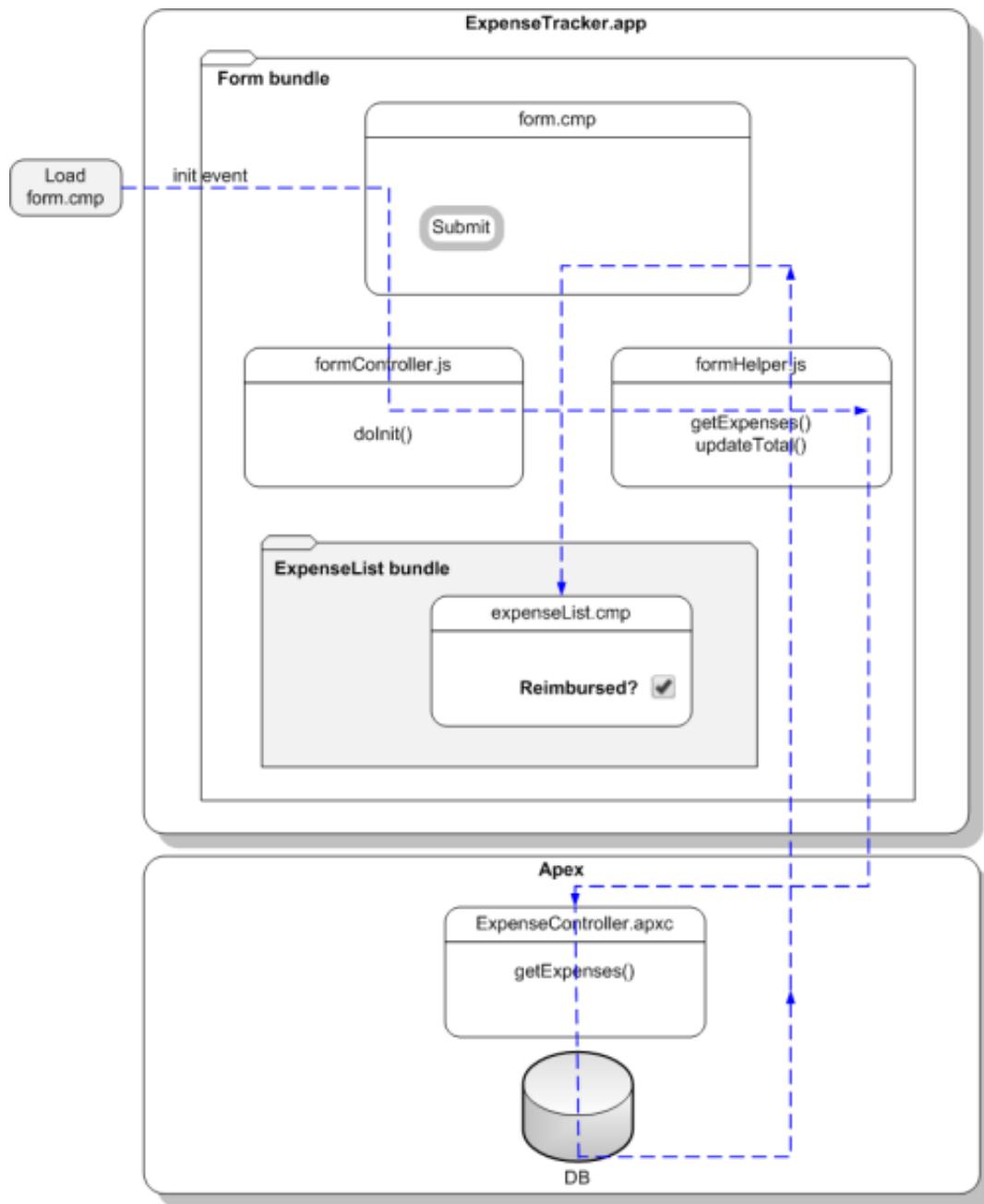
- a. [Expenses (経費)] タブをクリックし、[新規] をクリックします。
- b. 次の項目に値を入力し、2つ目のレコードについても同じ操作を繰り返します。

Expense Name (経費名)	Amount (金額)	Client	Date (日付)	Reimbursed? (払い戻し済み?)
昼食	21		4/1/2015 12:00 PM	オフ
夕食	70	ABC Co.	3/30/2015 7:00 PM	オン

ステップ 1: 静的モックアップを作成する

アプリケーションのエントリポイントとなる .app ファイルに静的モックアップを作成します。他のコンポーネントおよび HTML マークアップを含めることができます。

次のフローチャートに、アプリケーションでのデータフローをまとめます。アプリケーションでは、クライアント側のコントローラ、ヘルパー関数、Apex コントローラを組み合わせてレコードからデータが取得されます。これは、このクイックスタートで後ほど作成します。



1. [あなたの名前] > [開発者コンソール] をクリックして、開発者コンソールを開きます。
2. 開発者コンソールで、[File(ファイル)] > [New(新規)] > [Lightning Application (Lightning アプリケーション)] をクリックします。
3. [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)] 項目に入力します。これにより、`expenseTracker.app` という新しいアプリケーションが作成されます。

4. ソースコードエディタで、次のコードを入力します。

```
<aura:application>

<ltng:require styles="/resource/bootstrap"/>

<div class="bootstrap-sf1">

    <div class="navbar navbar-inverse">

        <div class="navbar-header">

            <a href="#" class="navbar-brand">My Expenses</a>

        </div>

    </div>

</div>

</aura:application>
```

アプリケーションは最上位レベルのコンポーネントであり、コンポーネントへのメインエントリポイントとなります。これには、コンポーネントや、`<div>` タグおよび `<header>` タグなどの HTML マークアップを含めることができます。

 **メモ:** ファイルパス名 `bootstrap` は、次にアップロードする静的リソースの名前に対応します。リソースが見つからないというエラーがブラウザコンソールに表示されても、このクイックスタートでは使用しないため心配しないでください。

5. Salesforce1のスタイルを使用したテーマを <http://developer.salesforcefoundation.org/bootstrap-sf1/> からダウンロードし、`bootstrap-namespaced.min.css` ファイルを静的リソースとしてアップロードします。このファイルは、`/dist/css` ディレクトリにあります。この CSS リソースには、他の名前空間とのスタイルの競合を回避する `.bootstrap-sf1` セレクタが含まれます。
 - a. [設定] から、[開発者]>[静的リソース] をクリックします。[新規] をクリックします。
 - b. [名前] 項目に「`bootstrap`」と入力します。[ファイルを選択] ボタンをクリックし、`bootstrap-namespaced.min.css` ファイルを選択します。このチュートリアルを簡略化するため、他のファイルはアップロードしません。
 - c. [保存] をクリックします。
6. 変更を保存し、サイドバーの[プレビュー]をクリックしてアプリケーションをプレビューします。または、`https://<mySalesforceInstance>.lightning.force.com/<namespace>/expenseTracker.app` (`<mySalesforceInstance>` は、na1 など、組織をホストするインスタンスの名前です)に移動します。名前空間を使用しない場合は、`/c/expenseTracker.app` でアプリケーションを使用できます。

関連トピック:

[aura:application](#)

ステップ 2: ユーザ入力用のコンポーネントを作成する

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。コンポーネントを Apex コントローラクラスに接続して、データを読み込むことができます。このステップで作成するコンポーネントにより、経費の金額や日付など、経費に関するユーザ入力を受け入れるフォームが提供されます。

1. [ファイル] > [新規] > [Lightning コンポーネント] をクリックします。
2. [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)] 項目には、「*form*」(フォーム) と入力します。これにより、*form.cmp* という新しいコンポーネントが作成されます。
3. ソースコードエディタで、次のコードを入力します。

 **メモ:** 次のコードは、経費を作成するためにユーザ入力を受け入れる入力フォームを作成します。これは、スタンダードアロンアプリケーションと Salesforce1 の両方で動作します。Salesforce1 に固有の Lightning アプリケーションでは、`force:createRecord` を使用してレコードの作成ページを開くことができます。

```
<aura:component>

<ltng:require styles="/resource/bootstrap"/>

<aura:attribute name="expenses" type="Expense__c[]"/>

<aura:attribute name="newExpense" type="Expense__c"
    default="{ 'sobjectType': 'Expense__c',
        'Name': '',
        'Amount__c': 0,
        'Client__c': '',
        'Date__c': '',
        'Reimbursed__c': false
    }"/>

<!-- Attributes for Expense Counters -->
<aura:attribute name="total" type="Double" default="0.00" />
<aura:attribute name="exp" type="Double" default="0" />

<!-- Input Form using components -->
<div class="bootstrap-sf1">
    <div class="container">
```

```
<form>

    <fieldset>

        <ui:inputText aura:id="expname" label="Expense Name"
            class="form-control"
            value="{!v.newExpense.Name}"
            placeholder="My Expense" required="true"/>

        <ui:inputNumber aura:id="amount" label="Amount"
            class="form-control"
            value="{!v.newExpense.Amount__c}"
            placeholder="20.80" required="true"/>

        <ui:inputText aura:id="client" label="Client"
            class="form-control"
            value="{!v.newExpense.Client__c}"
            placeholder="ABC Co."/>

        <ui:inputDateTime aura:id="exptime" label="Expense Date"
            class="form-control"
            value="{!v.newExpense.Date__c}"
            displayDatePicker="true"/>

        <ui:inputCheckbox aura:id="reimbursed" label="Reimbursed?" 
            class="checkbox"
            value="{!v.newExpense.Reimbursed__c}"/>

        <ui:button label="Submit" press=" {!c.createExpense} "/>

    </fieldset>

</form>

</div><!-- ./container-->

<!-- Expense Counters -->
```

```
<div class="container">

    <div class="row">

        <div class="col-sm-6">

            <!-- Make the counter red if total amount is more than 100 -->

            <div class="{'!v.total >= 100 ? 'alert alert-danger' : 'alert alert-info'}">

                <h3>Total Expenses</h3>$<ui:outputNumber value="{'!v.total}" format=".00"/>

            </div>

        </div>

        <div class="col-sm-6">

            <div class="alert alert-info">

                <h3>No. of Expenses</h3><ui:outputNumber value="{'!v.exp}"/>

            </div>

        </div>

    </div>

    <!-- Display expense records -->

    <div class="container">

        <div id="list" class="row">

            <aura:iteration items="{'!v.expenses}" var="expense">

                <!-- If you're using a namespace, use the format
                {'!expense.myNamespace__myField__c} instead. -->

                <p>{'!expense.Name}, {'!expense.Client__c}, {'!expense.Amount__c},
                {'!expense.Date__c}, {'!expense.Reimbursed__c}</p>

            </aura:iteration>

        </div>

    </div>

</div><!-- ./bootstrap-sf1-->
```

```
</aura:component>
```

コンポーネントでは、豊富な属性セットが提供され、ブラウザイベントがサポートされます。属性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、式の構文を使用して参照できます。すべての `aura:attribute` タグには、名前とデータ型の値があります。詳細は、「[サポートされる `aura:attribute` の型](#)」(ページ 360) を参照してください。

この属性と式は、アプリケーションを作成していく過程で明確になります。`{!v.exp}` は経費レコード数を評価し、`{!v.total}` は合計金額を評価します。`{!c.createExpense}` は、[実行] ボタンをクリックしたときに実行されるクライアント側のコントローラアクションを表します。このアクションにより、新しい経費が作成されます。`ui:button` の `press` イベントでは、ボタンをクリックしたときのアクションを結び付けられます。

式 `{!v.expenses}` は、コンポーネントを経費オブジェクトに接続します。`var="expense"` は、反復内の各項目に使用する変数の名前を示します。`{!expense.Client__c}` は、経費オブジェクトのクライアント項目にバインドするデータを表します。

 **メモ:** `Expense__c` 型の `newExpense` のデフォルト値は、`sObjectType` を含め正しい項目で初期化する必要があります。デフォルト値を初期化することで、経費が正しい形式で保存されます。

4. サイドバーで [STYLE] をクリックして、`form.css` という名前の新しいリソースを作成します。次の CSS ルールセットを入力します。

```
.THIS .uiInputDateTime+.datePicker-openIcon {  
    position: absolute;  
    left: 90%;  
    top: 30px;  
}  
  
.THIS .uiInputDefaultError li {  
    list-style: none;  
}
```

 **メモ:** `THIS` は、CSS に名前空間を追加して別のコンポーネントとのスタイルの競合を回避するためのキーワードです。`.uiInputDefaultError` セレクタは、「[ステップ 5: 新しい経費の入力を有効化する](#)」(ページ 27)で項目検証を追加するときのデフォルトのエラーコンポーネントのスタイルを設定します。

5. アプリケーションにコンポーネントを追加します。`expenseTracker.app` で、マークアップに新しいコンポーネントを追加します。

次の手順は、`<c:form />` をマークアップに追加します。名前空間を使用している場合は、`<myNamespace:form />` を代わりに使用できます。

```
<aura:application>

<ltng:require styles="/resource/bootstrap"/>

<aura:handler name="init" value="{!this}" action=" {!c.doInit} "/>

<div class="bootstrap-sf1">
    <div class="navbar navbar-inverse">
        <div class="navbar-header">
            <a href="#" class="navbar-brand">My Expenses</a>
        </div>
    </div>
    <div class="container">
        <c:form />
    </div>
</div>

</aura:application>
```

6. 変更を保存し、サイドバーの[プレビューを更新]をクリックしてアプリケーションをプレビューします。または、ブラウザを再読み込みします。

 **メモ:** このステップでは、Apex コントローラークラスをまだ作成していないため、作成したコンポーネントにデータは表示されません。

お疲れ様です。入力フォームを提供し、経費を表示するコンポーネントが作成されました。

Beyond the Basics

Lightning コンポーネントフレームワークには、aura や ui などの異なる名前空間に整理され、すぐに使用できる一連のコンポーネントが含まれています。ui 名前空間には、UI フレームワークの一般的なコンポーネントがあります。たとえば、ui:inputText はテキスト項目に対応します。aura 名前空間に

は、このステップで使用する `aura:iteration` など、中核となるフレームワーク機能のための多くのコンポーネントが含まれています。

関連トピック:

- [コンポーネントのマークアップ](#)
- [コンポーネントのボディ](#)

ステップ 3: 経費データを読み込む

Apexコントローラクラスを使用して、経費データを読み込みます。コンポーネントの属性を介してこのデータを表示し、カウンタを動的に更新します。

経費コントローラクラスを作成します。

1. [File(ファイル)] > [New(新規)] > [Apex クラス] をクリックし、[New Class(新規クラス)] ウィンドウで `ExpenseController` と入力します。これにより、`ExpenseController.apxc` という新しい Apex クラスが作成されます。
2. 次のコードを入力します。

```
public class ExpenseController {
    @AuraEnabled
    public static List<Expense__c> getExpenses() {
        return [SELECT id, name, amount__c, client__c, date__c,
                reimbursed__c, createdDate FROM Expense__c];
    }
}
```

`getExpenses()` メソッドには、すべての経費レコードを返すための SOQL クエリが含まれています。

`getExpenses()` メソッドの結果をコンポーネントのマークアップに表示する、`form.cmp` の構文 `{!v.expenses}` を思い出してください。

 **メモ:** SOQL の使用についての詳細は、『[Force.com SOQL および SOSL リファレンス](#)』を参照してください。

`@AuraEnabled` により、クライアント側およびサーバ側からコントローラメソッドへのアクセスが可能になります。サーバ側のコントローラは静的である必要があります。特定のコンポーネントのすべてのインスタンスは 1 つの静的コントローラを共有します。これらのコントローラは、リストや対応付けなど、任意の型で受け渡しできます。

 **メモ:** サーバ側のコントローラについての詳細は、『[Apex サーバ側コントローラの概要](#)』(ページ 192) を参照してください。

3. `form.cmp` で、`controller` 属性を含めるように `aura:component` タグを更新します。コンポーネントの初期化時にデータを読み込むには、`init` ハンドラを追加します。

```
<aura:component controller="ExpenseController">
<aura:handler name="init" value="{!this}" action=" {!c.doInit} " />
    <!-- Other aura:attribute tags here -->
    <!-- Other code here -->
</aura:component>
```

初期化時に、次に作成する `doInit` アクションが、このイベントハンドラで実行されます。この `init` イベントは、コンポーネントが表示される前に起動します。

4. `init` ハンドラに、クライアント側のコントローラアクションを追加します。サイドバーで [CONTROLLER] をクリックして、新しいリソース `formController.js` を作成します。次のコードを入力します。

```
({
  doInit : function(component, event, helper) {
    //Update expense counters
    helper.getExpenses(component);
  },//Delimiter for future code
})
```

コンポーネントの初期化時に、経費の最新の合計金額と総数が経費カウンタに反映されます。経費カウンタについては、次のステップでヘルパー関数 `getExpenses(component)` を使用して追加します。

- メモ: クライアント側のコントローラでは、コンポーネント内のイベントを処理し、コントローラが属するコンポーネント、アクションで処理するイベント、ヘルパー(使用する場合)という3つのパラメータを指定できます。ヘルパーは、コンポーネントのバンドルで再利用するコードを保存するためのリソースであり、コードの再利用性および特殊化が向上します。クライアント側のコントローラとヘルパーの使用についての詳細は、「[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125)および「[コンポーネントのバンドル内のJavaScriptコードの共有](#)」(ページ166)を参照してください。

5. 経費レコードを表示し、カウンタを動的に更新するヘルパー関数を作成します。[HELPER] をクリックして新しいリソース `formHelper.js` を作成し、次のコードを入力します。

```
({
  getExpenses: function(component) {
    var action = component.get("c.getExpenses");
    var self = this;
    action.setCallback(this, function(response) {
      var state = response.getState();
      if (component.isValid() && state === "SUCCESS") {
        component.set("v.expenses", response.getReturnValue());
        self.updateTotal(component);
      }
    });
    $A.enqueueAction(action);
  },
  updateTotal : function(component) {
    var expenses = component.get("v.expenses");
    var total = 0;
    for(var i=0; i<expenses.length; i++) {
      var e = expenses[i];
      //If you're using a namespace, use e.myNamespace__Amount__c instead
      total += e.Amount__c;
    }
    //Update counters
    component.set("v.total", total);
    component.set("v.exp", expenses.length);
  },//Delimiter for future code
})
```

```
})
```

component.get("c.getExpenses") は、サーバ側アクションのインスタンスを返します。

action.setCallback() は、サーバ応答の後でコールされる関数を渡します。updateTotal では、経費を取得し、金額値と経費期間の合計を求め、total および exp 属性でそれらの値を設定します。

 **メモ:** \$A.enqueueAction(action) は、アクションをキューに追加します。すべてのアクションコードは非同期であり、バッチで実行されます。サーバ側のアクションについての詳細は、「[サーバ側のアクションのコール](#)」(ページ 194)を参照してください。

6. 変更を保存し、ブラウザを再読み込みします。

「[経費オブジェクトを作成する](#)」(ページ 12)で作成した経費レコードが表示されます。カウンタについては、プログラムロジックを後から追加するため、この時点では動作しません。

これで、アプリケーションで経費オブジェクトが取得され、レコードがリストとして表示され、aura:iteration によって反復処理されるようになりました。カウンタには、経費の合計金額と総数が反映されます。

このステップでは、経費データを読み込む Apex コントローラークラスを作成しました。getExpenses() から、経費レコードのリストが返されます。デフォルトでは、フレームワークで getter はコールされません。メソッドにアクセスするには、そのメソッドでデータを公開する @AuraEnabled アノテーションをメソッドに付加します。コンポーネントにアクセスできるのは、コントローラークラスで @AuraEnabled アノテーションが付加されたメソッドのみです。

ExpenseController クラスを使用するコンポーネントのマークアップでは、「[ステップ 2: ユーザ入力用のコンポーネントを作成する](#)」(ページ 18)に示すように、{!expense.name} または {!expense.id} という式で経費名または経費 ID を表示できます。

Beyond the Basics

クライアント側のコントローラ定義は、角括弧および中括弧で囲まれます。中括弧は JSON オブジェクトを表し、オブジェクト内のすべてのものは名前-値のペアの対応付けです。たとえば、updateTotal は、クライアント側のアクションに対応する名前で、値は関数です。関数は、他のオブジェクトと同様に JavaScript で順に渡されます。

ステップ 4: ネストされたコンポーネントを作成する

コンポーネントの拡大に伴い、粒度とカプセル化を保持するために分割する場合があります。このステップでは、繰り返しのデータを含みその属性が親コンポーネントに渡されるコンポーネントの作成について説明します。また、コンポーネントの初期化時にデータを読み込むため、クライアント側のコントローラアクションも追加します。

- [ファイル] > [新規] > [Lightning コンポーネント] をクリックします。
- [New Lightning Bundle(新規 Lightning バンドル)] ポップアップウィンドウで、「expenseList」と入力します。これにより、expenseList.cmp という新しいコンポーネントが作成されます。
- expenseList.cmp で、次のコードを入力します。

 **メモ:** 項目値をバインドするには、項目の API 名を使用します。たとえば、名前空間を使用している場合は、`{!v.expense.Amount__c}` の代わりに `{!v.expense.myNamespace__Amount__c}` を使用する必要があります。

```
<aura:component>

    <aura:attribute name="expense" type="Expense__c"/>

    <!-- Color the item blue if the expense is reimbursed -->
    <div class="{!v.expense.Reimbursed__c == true
        ? 'alert alert-success' : 'alert alert-warning'}">

        <a aura:id="expense" href="{!! '/' + v.expense.Id}">
            <h3>{!v.expense.Name}</h3>
        </a>

        <p>Amount:</p>
        <ui:outputNumber value="{!v.expense.Amount__c}" format=".00"/>
    </p>

        <p>Client:</p>
        <ui:outputText value="{!v.expense.Client__c}"/>
    </p>

        <p>Date:</p>
        <ui:outputDateTime value="{!v.expense.Date__c}" />
    </p>

        <p>Reimbursed?</p>
        <ui:inputCheckbox value="{!v.expense.Reimbursed__c}" click=" {!c.update} "/>
    </p>
</div>

</aura:component>
```

ここでは、`{!expense.Amount__c}` ではなく、`{!v.expense.Amount__c}` を使用しています。この式は、`expense` オブジェクトおよびその `amount` 値にアクセスします。

また、`href="{!! '/' + v.expense.Id}"` は、経費IDを使用して、各経費レコードの詳細ページへのリンクを設定します。

4. `form.cmp` で、ネストされた新しいコンポーネント `expenseList` を使用するように `aura:iteration` タグを更新します。既存の `aura:iteration` タグを見つけます。

```
<aura:iteration items="{!!v.expenses}" var="expense">  
  
    <p>{!!expense.Name}, {!!expense.Client__c}, {!!expense.Amount__c}, {!!expense.Date__c},  
    {!!expense.Reimbursed__c}</p>  
  
</aura:iteration>
```

そのタグを、`expenseList` コンポーネントを使用する `aura:iteration` タグに置き換えます。

```
<aura:iteration items="{!!v.expenses}" var="expense">  
  
    <!--If you're using a namespace, use myNamespace:expenseList instead-->  
  
    <c:expenseList expense="{!!expense}" />  
  
</aura:iteration>
```

経費の詳細の表示を処理する `expenseList` コンポーネントに各 `expense` レコードを単に渡す場合と、マークアップが似ていることがわかります。

5. 変更を保存し、ブラウザを再読み込みします。

ネストされたコンポーネントを作成したので、その属性を親コンポーネントに渡します。次に、ユーザ入力の処理方法と、経費オブジェクトの更新方法について説明します。

Beyond the Basics

コンポーネントを作成すると、そのコンポーネントの定義を提供することになります。コンポーネントを別のコンポーネントに挿入する場合は、そのコンポーネントへの参照を作成します。つまり、同じコンポーネントの異なる属性を持つ複数のインスタンスを追加できるということです。コンポーネントの属性についての詳細は、「[コンポーネントのコンポジション](#)」(ページ 64)を参照してください。

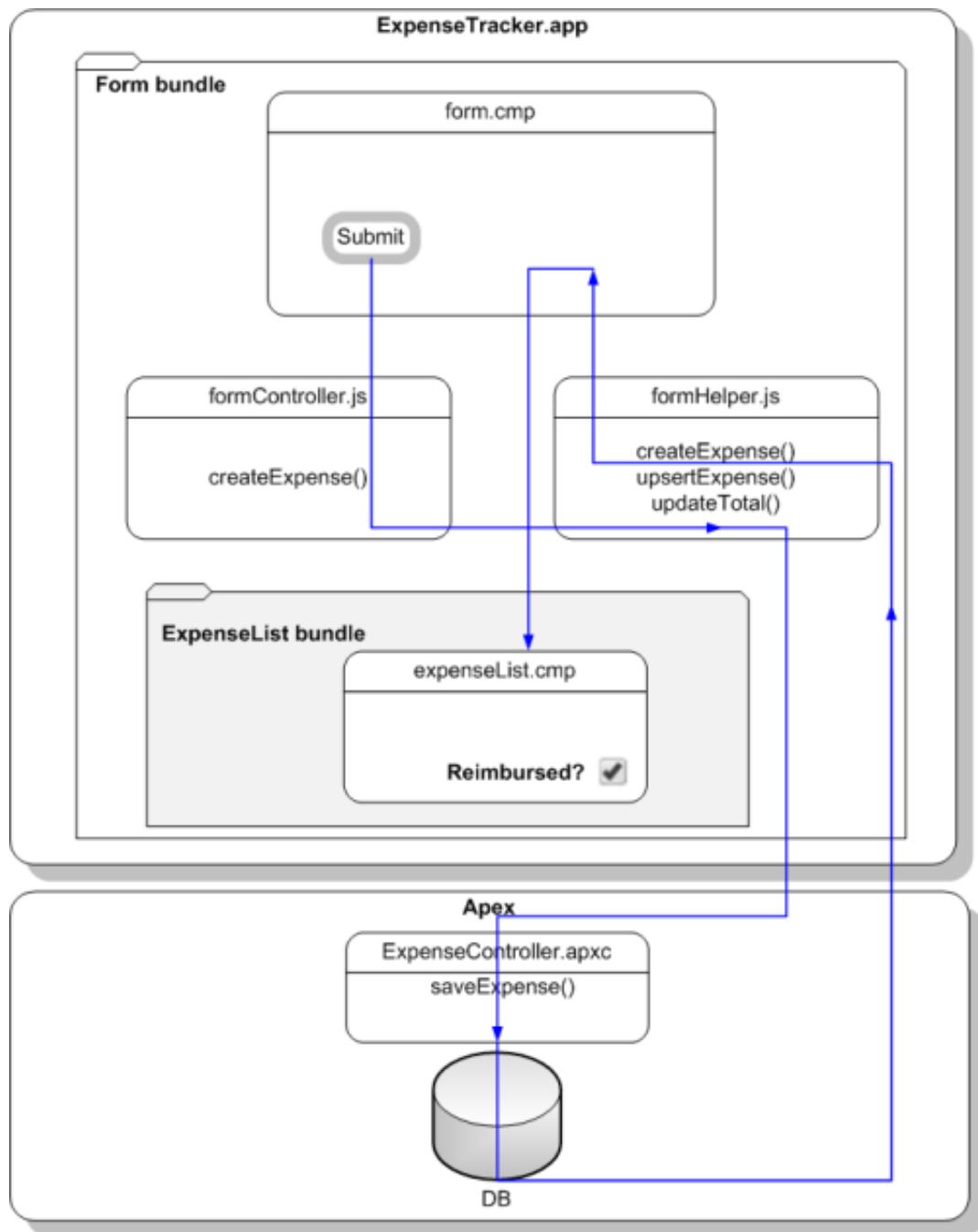
関連トピック:

[コンポーネントの属性](#)

ステップ 5: 新しい経費の入力を有効化する

テキストをフォームに入力して[送信]を押すときに、新しい経費レコードを挿入します。このアクションは、`press` 属性でボタンコンポーネントに結び付けられます。

次のフローチャートに、新しい経費を作成するときの、アプリケーションでのデータフローを示します。コンポーネント `form.cmp` で[送信]ボタンをクリックするとデータが取得され、JavaScriptコードで処理され、サーバ側のコントローラに送信してレコードとして保存されます。レコードのデータは、前のステップで作成したネストされたコンポーネントに表示されます。



まず、レコードを挿入または更新する新しいメソッドで Apex コントローラを更新します。

1. ExpenseController クラスで、次のコードを `getExpenses()` メソッドの下に入力します。

```

@AuraEnabled
public static Expense__c saveExpense(Expense__c expense) {
    upsert expense;
    return expense;
}

```

`saveExpenses()` メソッドでは、`upsert` 操作を使用して経費レコードを挿入または更新できます。

 **メモ:** `upsert` 操作についての詳細は、『[Apex コード開発者ガイド](#)』を参照してください。

2. [送信]ボタンが押されたときに新しい経費レコードを作成するコントローラ側のアクションを作成します。
`formController.js` で、`doInit` アクションの後に次のコードを追加します。

```
createExpense : function(component, event, helper) {
    var amtField = component.find("amount");
    var amt = amtField.get("v.value");
    if (isNaN(amt) || amt==''){
        amtField.setValid("v.value", false);
        amtField.addErrors("v.value", [{message:"Enter an expense amount."}]);
    }
    else {
        amtField.setValid("v.value", true);
        var newExpense = component.get("v.newExpense");
        helper.createExpense(component, newExpense);
    }
},//Delimiter for future code
```

`createExpense` では、`ui:inputDefaultError` で表されるエラーメッセージを追加するデフォルトのエラー処理を使用して、金額項目を検証します。コントローラでは、`setValid(false)` を使用して入力値を無効化し、`setValid(true)` を使用してエラーをクリアします。項目検証についての詳細は、「[項目の検証](#)」(ページ 174)を参照してください。

引数を `helper.createExpense()` というヘルパー関数に渡すと、`saveExpense` という Apex クラスがトリガされます。

 **メモ:** 「[ステップ 2: ユーザー入力用のコンポーネントを作成する](#)」(ページ 18)では、`aura:id` 属性を指定しました。`aura:id` により、コンポーネントとそのコントローラの範囲内で、構文 `component.find("amount")` を使用してコンポーネントを名前で検索できます。

3. レコードの作成を処理するヘルパー関数を作成します。`updateTotal` 関数の後に次のヘルパー関数を追加します。

```
createExpense: function(component, expense) {
    this.upsertExpense(component, expense, function(a) {
        var expenses = component.get("v.expenses");
        expenses.push(a.getReturnValue());
        component.set("v.expenses", expenses);
        this.updateTotal(component);
    });
},
upsertExpense : function(component, expense, callback) {
    var action = component.get("c.saveExpense");
    action.setParams({
        "expense": expense
    });
    if (callback) {
        action.setCallback(this, callback);
    }
    $A.enqueueAction(action);
}
```

`createExpense` から `upsertExpense` がコールされ、`saveExpense` というサーバ側のアクションのインスタンスが定義されて、`expense` オブジェクトがパラメータとして設定されます。サーバ側のアクションが返された後にコールバックが実行され、レコード、ビュー、およびカウンタが更新されます。
`$A.enqueueAction(action)` により、サーバ側のアクションがアクション実行キューに追加されます。

 **メモ:** さまざまなアクションの状態を使用でき、コールバックでそれらの動作をカスタマイズできます。アクションのコールバックについての詳細は、「[サーバ側のアクションのコール](#)」を参照してください。

4. 変更を保存し、ブラウザを再読み込みします。Breakfast, 10, ABC Co., Apr 30, 2014 9:00:00 AMと入力してアプリケーションをテストします。日付項目では、日付ピッカーを使用して、日時値を設定することもできます。[送信] ボタンをクリックします。レコードがコンポーネントのビューとレコードの両方に追加され、カウンタが更新されます。

 **メモ:** Apex コードをデバッグするには、開発者コンソールの [Logs (ログ)] タブを使用します。たとえば、日時項目の入力規則がない場合に無効な日時形式を入力すると、`INVALID_TYPE_ON_FIELD_IN_RECORD` 例外が発生する可能性があります。これは、開発者コンソールの [Logs (タブ)] とブラウザの応答ヘッダーの両方にリストされます。入力規則がある場合は、Apexエラーがブラウザに表示されます。JavaScript コードのデバッグについての詳細は、「[JavaScriptコードのデバッグ](#)」(ページ 221)を参照してください。

以上でこのステップは完了です。いくつかのコンポーネント、クライアント側およびサーバ側のコントローラ、ヘルパー関数が含まれる簡単な経費追跡アプリケーションを正常に作成できました。これで、アプリケーションは、ビューおよびデータベースを更新するユーザ入力値を受け入れるようになります。また、カウンタは新しいユーザ入力値を入力するたびに動的に更新されます。次のステップでは、インタラクティブにイベントを使用するレイヤを追加する方法について説明します。

関連トピック:

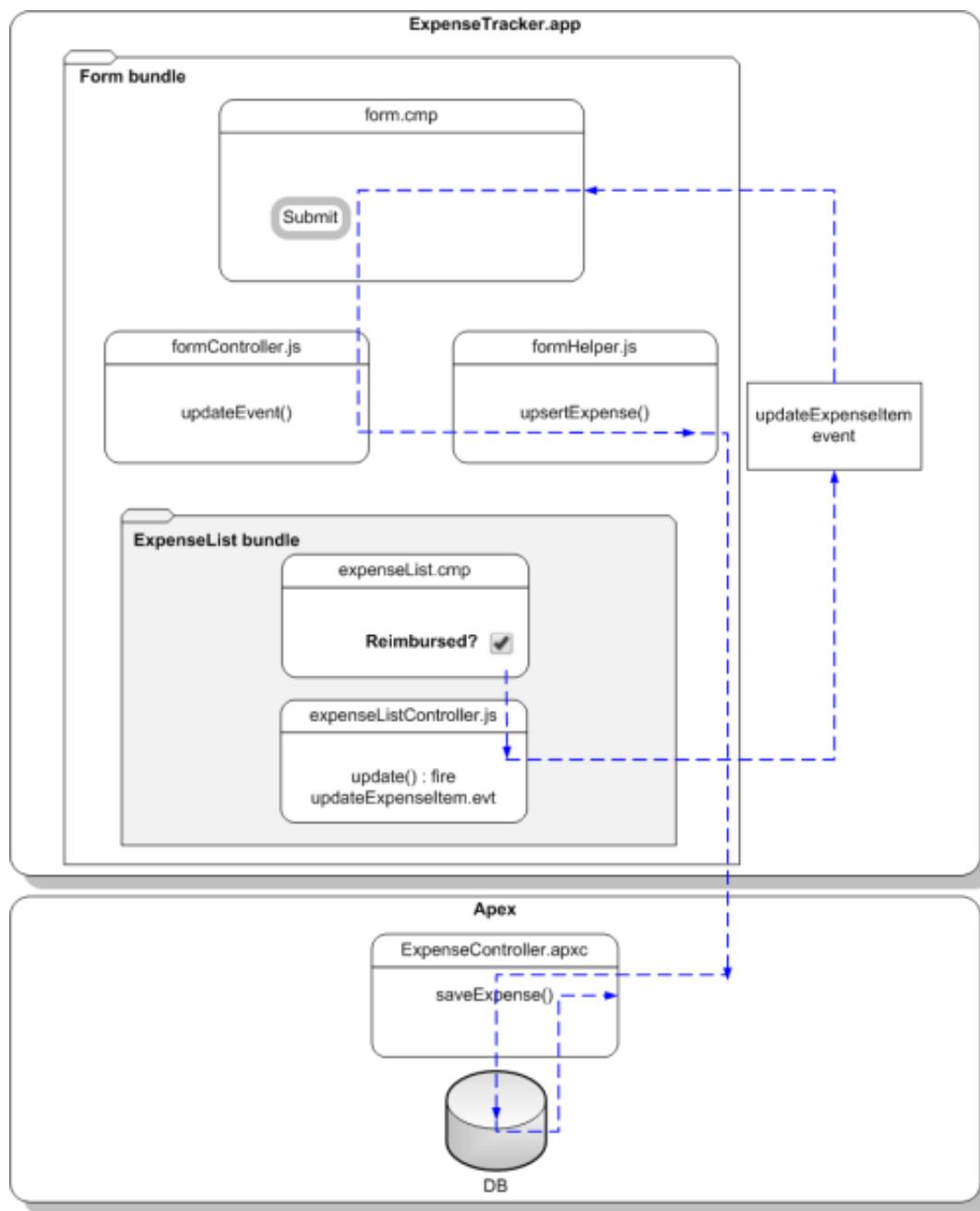
[クライアント側コントローラを使用したイベントの処理](#)

[サーバ側のアクションのコール](#)

ステップ 6: アプリケーションとイベントをインタラクティブにする

イベントは、コンポーネント間でデータの共有を可能にすることによって、インタラクティブルレイヤをアプリケーションに追加します。経費リストビューのチェックボックスをオン/オフにしたときに、関連するコンポーネントデータに基づいてビューとレコードの両方を更新するイベントを起動する必要がある場合があります。

次のフローチャートに、`expenseList` コンポーネントでチェックボックスをオン/オフにすることによってデータの変更が取得されるときの、アプリケーションでのデータフローを示します。[Reimbursed? (払い戻し済み?)] チェックボックスをオン/オフにすると、このブラウザのクリックイベントにより、ここで作成するアプリケーションイベントが起動されます。このイベントは、経費オブジェクトをハンドラコンポーネントに渡し、そのコントローラは Apex コントローラメソッドをコールして関連する経費レコードを更新します。ここでは、このサーバ応答を処理しないため、その後の応答はクライアント側で無視されます。



イベントを起動して親コンポーネントのイベントを処理するには、まずイベントとそのハンドラを作成します。

1. [ファイル] > [新規] > [Lightning イベント] をクリックします。
2. [New Event (新規イベント)] ウィンドウで、`updateExpenseItem` と入力します。これにより、`updateExpenseItem.evt` という新しいイベントが作成されます。
3. `updateExpenseItem.evt` で、次のコードを入力します。

イベントで定義している属性が、起動元のコンポーネントからハンドラに渡されます。

```
<aura:event type="APPLICATION">

    <!-- If you're using a namespace, use myNamespace.Expense__c instead. -->

    <aura:attribute name="expense" type="Expense__c"/>

</aura:event>
```

フレームワークには、コンポーネントイベントとアプリケーションイベントという 2 種類のイベントがあります。ここでは、アプリケーションイベントを使用します。アプリケーションイベントが起動されると、ハンドラに通知されます。この場合は、form.cmp に通知されて、イベントが処理されます。

expenseList.cmp には、change=" {!c.update}" で示されるクライアント側のコントローラアクションに接続されるチェックボックスが含まれていることを思い出してください。次に、update アクションを設定します。

4. expenseList サイドバーで、[CONTROLLER] をクリックします。これにより、expenseListController.js という新しいリソースが作成されます。次のコードを入力します。

```
({

    update: function(component, evt, helper) {

        var expense = component.get("v.expense");

        //If you're using a namespace, use e.myNamespace:updateExpenseItem instead

        var updateEvent = $A.get("e.c:updateExpenseItem");

        updateEvent.setParams({ "expense": expense }).fire();

    }

})
```

チェックボックスをオンまたはオフにすると、update アクションが実行され、reimbursed パラメータの値が true または false に設定されます。updateExpenseItem.evt イベントは、更新済みの expense オブジェクトを使用して起動されます。

5. ハンドラコンポーネント form.cmp で、次のハンドラコードを <aura:attribute> タグの前に追加します。

```
<!-- If you're using a namespace, use myNamespace:updateExpenseItem instead -->

<aura:handler event="c:updateExpenseItem" action=" {!c.updateEvent}" />
```

このイベントハンドラは、作成したアプリケーションイベントが起動されると updateEvent アクションを実行します。

6. イベントを処理するための `updateEvent` アクションを結び付けます。 `formController.js` で、次のコードを `createExpense` コントローラアクションの後に入力します。

```
updateEvent : function(component, event, helper) {
    helper.upsertExpense(component, event.getParam("expense"));
}
```

このアクションは、ヘルパー関数をコールし、`{ Name : "Lunch" , Client__c : "ABC Co." , Reimbursed__c : true , CreatedDate : "2014-08-12T20:53:09.000Z" , Amount__c : 20 }` 形式のパラメータと値が含まれた経費オブジェクトを含む `event.getParam("expense")` を渡します。

完成です! アプリケーションイベントを使用して、経費追跡アプリケーションでインタラクションのレイヤが正常に追加されました。ビューで払い戻し済み状況を変更すると、`update` イベントが起動され、親コンポーネントで処理されてから、サーバ側のコントローラアクション `saveExpense` が実行され、経費レコードが更新されます。

Beyond the Basics

ライフサイクルの表示中には、このチュートリアルで使用した `init` イベントなどのいくつかのイベントがフレームワークによって起動されます。たとえば、サーバ応答をクライアントが待機しているときや、応答を受信したことを示すために `doneWaiting` イベントが起動されたときに、`waiting` イベント実行中のアプリケーションの動作をカスタマイズすることもできます。次の例に、`waiting` イベント実行中にアプリケーションでテキストを追加する方法と、`doneWaiting` イベントの起動時にテキストを削除する方法を示します。

```
<!-- form.cmp markup -->

<aura:handler event="aura:waiting" action="{!c.waiting}" />
<aura:handler event="aura:doneWaiting" action="{!c.doneWaiting}" />
<aura:attribute name="wait" type="String"/>

<div class="wait">
    {!v.wait}
</div>

/** formController.js **/

waiting : function(component, event, helper) {
    component.set("v.wait", "updating...");
},
doneWaiting : function(component, event, helper) {
```

```
        component.set("v.wait", "");  
    }  
}
```

新しいレコードを作成するために[送信]ボタンをクリックするか、経費項目のチェックボックスをオンにすると、このテキストがアプリケーションによって表示されます。詳細は、「[表示ライフサイクル中に起動されたイベント](#)」(ページ 149)を参照してください。

[<https://<mySalesforceInstance>.lightning.force.com/<namespace>/expenseTracker.app](https://<mySalesforceInstance>.lightning.force.com/<namespace>/expenseTracker.app)

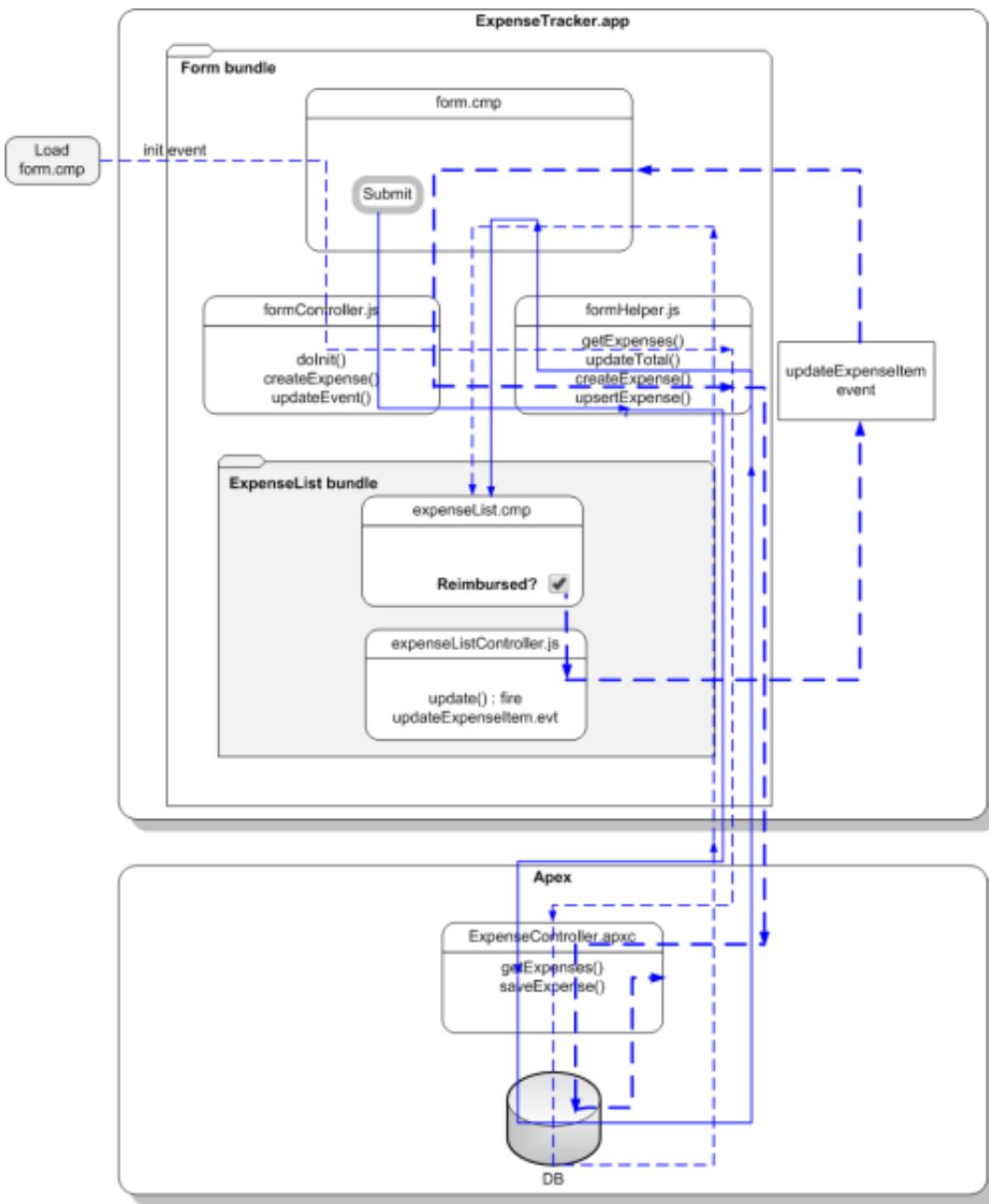
(<mySalesforceInstance> は、na1 など、組織をホストしているインスタンスの名前です)にアクセスすると、スタンドアロンアプリケーションとして先ほど作成したアプリケーションに現時点でアクセスできます。Salesforce1 でアクセスできるようにするには、「[Salesforce1 への Lightning コンポーネントの追加](#)」(ページ 76)を参照してください。アプリケーションをパッケージ化してAppExchange で配布するには、「[アプリケーションとコンポーネントの配布](#)」(ページ 219)を参照してください。

関連トピック:

[アプリケーションイベント](#)

まとめ

ここでは、経費レコードを操作するコントローラやイベントを含むいくつかのコンポーネントを作成しました。経費追跡アプリケーションでは、アプリケーションの初期化時に経費データとカウンタを読み込む、ユーザ入力を受け入れて新規レコードを作成しビューを更新する、イベントを介して関連コンポーネントデータとやりとりすることでユーザ操作を処理する、という 3 つの異なるタスクが実行されます。



`form.cmp` が初期化されると、`init` ハンドラがクライアント側の `doInit` コントローラをトリガし、そのコントローラが `getExpenses` ヘルパー関数をコールします。`getExpenses` は、経費を読み込むためにサーバ側の `getExpenses` コントローラをコールします。コールバックが、`v.expenses` 属性の経費データを設定し、カウンタを更新するために `updateTotal` をコールします。

[送信]ボタンをクリックすると、クライアント側の `createExpense` コントローラがトリガれます。項目の検証後、`createExpense` ヘルパー関数が実行され、そこでは `upsertExpense` ヘルパー関数によってサーバ側の `saveExpense` コントローラがコールされ、レコードが保存されます。コールバックは、新しい経費を経費リストに転送し、`form.cmp` の `v.expenses` 属性を更新します。これにより、`expenseList.cmp` の経費が

更新されます。最後に、ヘルパーが `updateTotal` をコールし、`v.total` および `v.exp` 属性で示されるカウントを更新します。

`expenseList cmp` が、経費のリストを表示します。`[Reimbursed? (払い戻し済み?)]` チェックボックスをオン/オフにすると、`click` イベントがクライアント側の `update` コントローラをトリガします。パラメータとして渡された関連経費を使用して、`updateExpenseItem` イベントが起動されます。`form cmp` がイベントを処理し、クライアント側の `updateEvent` コントローラをトリガします。このコントローラアクションが `upsertExpense` ヘルパー関数をコールし、このヘルパー関数がサーバ側の `saveExpense` コントローラをコールして関連レコードを保存します。

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み、Salesforce1 と連動するコンポーネントを作成します。



このコンポーネントには次の機能があります。

- すべての取引先責任者が正常に読み込まれたらトーストメッセージ (1) を表示する

- ネストしたコンポーネントを使用して、入力選択値⁽²⁾が変更されたら、すべての取引先責任者を表示するか、緑色のすべての主取引先責任者を表示する
- [新規作成]⁽³⁾ アイコンがクリックされたら、新規取引先責任者を作成するためにレコードの作成ページを開く
- [取引先責任者の編集]⁽⁴⁾ アイコンがクリックされたら、選択された取引先責任者を更新するためにレコードの編集ページを開く
- 取引先責任者名⁽⁵⁾ がクリックされたらレコードに移動する
- 郵送先住所⁽⁵⁾ がクリックされたら地図に移動する
- [ケース参照] ボタン⁽⁶⁾ がクリックされたら関連ケースに移動する

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	取引先責任者データを読み込むコンポーネント
contactsController.js	取引先責任者データを読み込み、入力選択変更イベントを処理し、レコードの作成ページを開くクライアント側コントローラアクション
contactsHelper.js	取引先責任者データを取得し、読み込み状況に基づいてトーストメッセージを表示するヘルパー関数
contacts.css	コンポーネントのスタイル
contactList バンドル	
contactList.cmp	取引先責任者リストコンポーネント
contactListController.js	レコードの編集ページを開き、取引先責任者レコード、関連ケース、および取引先責任者の住所の地図に移動するクライアント側コントローラアクション
contactList.css	コンポーネントのスタイル
Apex コントローラ	
ContactController.apxc	取引先責任者レコードをクエリする Apex コントローラ

省略可能: 取引先責任者リストアプリケーションのインストール

クイックスタートチュートリアルをスキップする場合は、取引先責任者リストアプリケーションをパッケージとしてインストールできます。

こうしたパッケージアプリケーションは、クイックスタートチュートリアルを使用せずにLightning アプリケーションについて学習する場合に役立ちます。Lightning コンポーネントを初めて利用する場合は、クイックスタートチュートリアルの説明に従うことをお勧めします。

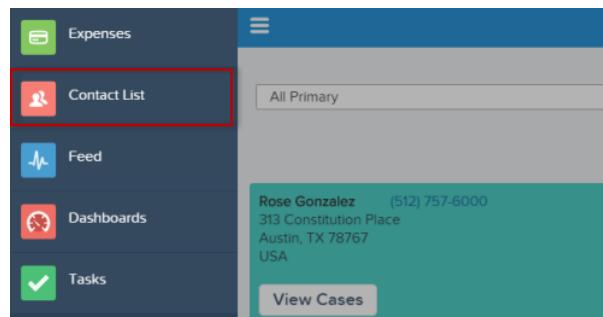
-  **メモ:** クイックスタートのオブジェクトと API 名が同じオブジェクトのない組織にパッケージをインストールします。

取引先責任者リストアプリケーションをインストールする手順は、次のとおりです。

1. インストール用 URL リンク <https://login.salesforce.com/packaging/installPackage.apexp?p0=04to00000003pLQ> をクリックします。
2. ユーザ名とパスワードを入力して組織にログインします。
3. [パッケージインストールの詳細] ページで、[続行] をクリックします。
4. [次へ] をクリックし、[セキュリティレベル] ページで [次へ] をクリックします。
5. [インストール] をクリックします。
6. [今すぐリリース] をクリックし、[リリース] をクリックします。

インストールが完了したら、取引先責任者ページレイアウトにカスタムの [Level (レベル)] 項目を追加し、ユーザインターフェースの [取引先責任者] タブを選択して、新しい取引先責任者レコードを追加できます。

Salesforce1 ナビゲーションメニューに [ContactList(取引先責任者リスト)] メニュー項目が表示されます。Salesforce1 にメニュー項目が表示されない場合は、[モバイル管理] > [モバイルナビゲーション] に移動して追加します。



続いて、開発者コンソールでコードを変更したり、Salesforce1 でアプリケーションを探索したりすることができます。

メモ: パッケージを削除するには、[設定] > [インストール済みパッケージ] をクリックして、パッケージをアンインストールして削除します。

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。

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。このチュートリアルでは、`Level__c` という API 名で表されるカスタム選択リスト項目 `Level` が使用されます。この項目には、`Primary`、`Secondary`、`Tertiary` という 3 つの選択リスト値が含まれます。

1. [ファイル] > [新規] > [Apex クラス] をクリックして、[新規クラス] ウィンドウに `ContactController` と入力します。これにより、`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。

組織で名前空間を使用している場合は、`Level__c` を `myNamespace__Level__c` に置き換えます。

```
public class ContactController {
```

```
    @AuraEnabled
```

```
public static List<Contact> getContacts() {  
  
    List<Contact> contacts =  
  
        [SELECT Id, Name, MailingStreet, Phone, Email, Level__c FROM Contact];  
  
    return contacts;  
  
}  
  
  
@AuraEnabled  
  
// Retrieve all primary contacts  
  
public static List<Contact> getPrimary() {  
  
    List<Contact> primaryContacts =  
  
        [SELECT Id, Name, MailingStreet, Phone, Email, Level__c FROM Contact WHERE  
namespace__Level__c = 'Primary'];  
  
    return primaryContacts;  
  
}  
}
```

getPrimary() によって、Level__c 項目が Primary に設定されているすべての取引先責任者が返されます。

2. [ファイル]>[新規]>[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 [名前] 項目に「contactList」と入力します。これにより、contactList.cmp という新しい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"/>  
  
    <!-- If you're using a namespace, use {!v.contact.myNamespace__Level__c} instead  
-->  
  
    <div class="={!v.contact.Level__c == 'Primary'  
          ? 'row primary' : 'row '}>  
  
        <!-- Display a contact name  
        and navigate to record when the name is clicked -->
```

```
<div class="col-sm-4">  
    <ui:outputText value="{!v.contact.Name}" click="={!c.gotoRecord}"/>  
</div>  
  
<div class="col-sm-4">  
    <ui:outputEmail value="{!v.contact.Email}"/>  
    <ui:outputPhone value="{!v.contact.Phone}"/>  
  
    <!-- Display the edit record page when the icon is clicked -->  
    <div onclick=" {!c.editRecord}">  
          
    </div>  
    </div>  
  
<div class="col-sm-4">  
    <ui:outputTextArea aura:id="address" value="{!v.contact.MailingStreet}"  
    click=" {!c.navigate}"/>  
    </div>  
  
    <!-- Navigate to the related list when the button is clicked -->  
    <ui:button label="View Cases" press=" {!c.relatedList}"/>  
    </div>  
</aura:component>
```

3. contactList サイドバーで、[STYLE] をクリックして、contactList.css という新しい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。

```
.THIS.primary{  
    background: #4ECDC4 !important;  
}  
  
.THIS .uiOutputText {
```

```
width: 25%;  
float: left;  
font-weight: bold;  
}  
  
.THIS .uiOutputPhone {  
color: #2574A9;  
}  
  
.THIS .uiOutputTextArea {  
float: clear;  
}  
  
.THIS .uiOutputEmail {  
float: right;  
}  
  
.THIS .uiButton {  
margin-top: 20px !important;  
}
```

4. [ファイル] > [新規] > [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 [名前] 項目に 「contacts」 と入力します。これにより、contacts.cmp という新しい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。

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

```
<!-- Handle loading events by displaying a spinner -->

<aura:handler event="aura:waiting" action="{!c.showSpinner}"/>

<aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>

<!-- Dynamically load the list of contacts -->

<aura:attribute name="contacts" type="Contact[]"/>

<!-- Create a drop-down list with two options -->

<ui:inputSelect aura:id="selection" change="{!c.select}>

    <ui:inputSelectOption text="All Contacts" label="All Contacts"/>

    <ui:inputSelectOption text="All Primary" label="All Primary"/>

</ui:inputSelect>

<div class="icons">

</div>

<div><center><ui:spinner aura:id="spinner"/></center></div>

<!-- 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:contactList -->

    <c:contactList contact="{!contact}"/>

</aura:iteration>

</aura:component>
```

5. contacts サイドバーで、[CONTROLLER] をクリックして、contactsController.js という新しい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.getContacts(component);  
},  
  
showSpinner : function (component, event, helper) {  
  
    var spinner = component.find('spinner');  
  
    var evt = spinner.get("e.toggle");  
  
    evt.setParams({ isVisible : true });  
  
    evt.fire();  
},  
  
hideSpinner : function (component, event, helper) {  
  
    var spinner = component.find('spinner');  
  
    var evt = spinner.get("e.toggle");  
  
    evt.setParams({ isVisible : false });  
  
    evt.fire();  
}, //Delimiter for future code  
})
```

6. **contacts** サイドバーで、[HELPER]をクリックして、`contactsHelper.js` という新しい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。

```
({  
  
    getContacts : function(cmp) {  
  
        // Load all contact data  
  
        var action = cmp.get("c.getContacts");  
  
        var self = this;  
  
        action.setCallback(this, function(response) {  
  
            var state = response.getState();  
        })  
    }  
})
```

```
if (cmp.isValid() && state === "SUCCESS") {  
  
    cmp.set("v.contacts", response.getReturnValue());  
  
}  
  
  
// 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);  
  
}  
})
```

7. **contacts** サイドバーで、[STYLE] をクリックして、`contacts.css` という新しい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。

```
.THIS.row {  
  
background: #fff;
```

```
max-width:90%;  
  
border-bottom: 2px solid #f0f1f2;  
  
padding: 10px;  
  
margin-left: 2%;  
  
margin-bottom: 10px;  
  
min-height: 70px;  
  
border-radius: 4px;  
  
}  
  
  
.THIS.uiInputSelect {  
  
width: 80%;  
  
padding-left: 10px;  
  
min-height: 28px;  
  
margin-bottom: 20px;  
  
margin-left: 20px;  
  
margin-top: 40px;  
  
}  
  
  
.THIS img {  
  
float: right;  
  
padding-right:5%;  
  
padding-top: 20px;  
  
}  
  
  
.THIS.icons {  
  
height:60px;  
  
}
```

8. 「Salesforce1 への Lightning コンポーネントの追加」(ページ 76)の手順に従って、新しい [Lightning コンポーネント] タブを作成します。このコンポーネントが Salesforce1 ナビゲーションメニューに表示されていることを確認します。

最後に、Salesforce1 モバイルブラウザアプリケーションに移動して、出力を確認できます。コンポーネントが読み込まれると、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。

次に、他のイベントを結び付けて、入力選択にすべての取引先責任者か、緑色の主取引先責任者のみが表示されるようにします。また、レコードの作成ページおよびレコードの編集ページを開くイベントと、レコードおよび URL 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、Salesforce1 で処理されます。

このデモは、「取引先責任者の読み込み」(ページ 38)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. contactList サイドバーで、[CONTROLLER] をクリックして、contactListController.js という新しい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"),
            "slideDevName": 'related'
        })
        sObjectEvent.fire();
    },

    editRecord : function(component, event, helper) {

        // Fire the event to navigate to the edit contact page

        var editRecordEvent = $A.get("e.force:editRecord");
        editRecordEvent.setParams({
            "recordId": component.get("v.contact.Id")
        })
    }
});
```

```
        }) ;

    editRecordEvent.fire();

    },
}

navigate : function(component, event, helper) {

    // Navigate to an external URL

    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();
},
}

relatedList : function (component, event, helper) {

    // Navigate to the related cases

    var relatedListEvent = $A.get("e.force:navigateToRelatedList");

    relatedListEvent.setParams({
        "relatedListId": "Cases",
        "parentRecordId": component.get("v.contact.Id")
    });

    relatedListEvent.fire();
}
})
```

2. Salesforce1 モバイルブラウザアプリケーションを更新し、次の要素をクリックしてイベントをテストします。

- 取引先責任者名: `force:navigateToSObject` が起動され、取引先責任者レコードページでビューが更新されます。取引先責任者名は、次のコンポーネントに対応します。

```
<ui:outputText value="{!v.contact.Name}" click="{!!c.gotoRecord}"/>
```

- 取引先責任者の編集アイコン: `force:editRecord` が起動され、レコードの編集ページが開きます。取引先責任者の編集アイコンは、次のコンポーネントに対応します。

```
<div onclick="{!!c.editRecord}>

</div>
```

- 住所: `force:navigateToURL` が起動され、指定した url の Google マップが開きます。住所は、次のコンポーネントに対応します。

```
<div class="col-sm-4">

    <ui:outputTextArea aura:id="address" value="{!!v.contact.MailingStreet}" click="{!!c.navigate}"/>

</div>
```

- `contactsController.js` を開きます。`hideSpinner` コントローラの後に次のコードを入力して保存します。

```
createRecord : function (component, event, helper) {

    // Open the create record page

    var createRecordEvent = $A.get("e.force:createRecord");

    createRecordEvent.setParams({
        "entityApiName": "Contact"
    });

    createRecordEvent.fire();

},

select : function(component, event, helper) {

    // Get the selected value of the ui:inputSelect component

    var selectCmp = component.find("selection");

    var selectVal = selectCmp.get("v.value");
```

```
// Display all primary contacts or all contacts

if (selectVal==="All Primary") {

    var action = component.get("c.getPrimary");

    action.setCallback(this, function(response) {

        var state = response.getState();

        if (component.isValid() && state === "SUCCESS") {

            component.set("v.contacts", response.getReturnValue());

        }

    });

    $A.enqueueAction(action);

}

else {

    // Return all contacts

    helper.getContacts(component);

}

}
```

ページをプルダウンして離すと、ページのビューのすべてのデータが更新されます。この動作は、`$A.get('e.force:refreshView').fire();` を実行する場合と似ています。

これで、「Salesforce1 のコンポーネントの作成」(ページ 36)で強調表示された領域をクリックして、コンポーネントをテストできます。

Salesforce1 とは関係なく使用できるスタンドアロンアプリケーションの作成例については、「[スタンドアロン Lightning アプリケーションを作成する](#)」(ページ 9)を参照してください。

コンポーネントの作成

第3章 コンポーネント

トピック:

- 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
- 表示ラベルの使用
- ローカライズ
- Lightning コンポーネントの有効化
- Salesforce1へのLightning コンポーネントの追加
- Lightning ページとLightning App Builderのコンポーネントの設定

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。

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。

関連トピック:

- 開発者コンソールの使用

コンポーネント

- アプリケーションへのコンポーネントの追加
- 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ます。そのため、`<div>` や``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。

`aura:component` には次の省略可能な属性があります。

属性	型	説明
<code>access</code>	<code>String</code>	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、 <code>public</code> (デフォルト)と <code>global</code> です。
<code>controller</code>	<code>String</code>	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 <code>namespace.myController</code> となります。
<code>description</code>	<code>String</code>	コンポーネントの説明。

属性	型	説明
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。

関連トピック:

- [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](#)
- [DOMへのクライアント側表示](#)
- [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。

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、組織の默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
- Apex で動的に作成するコンポーネント

 **メモ:** 動的に作成するコンポーネントに名前空間を指定するときは、必要に応じて、`c` 名前空間を使用できます。

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 55) 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
- Apex で動的に作成するコンポーネント

 **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 55) 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を登録する手順は、次のとおりです。

1. [設定] で、[作成] > [パッケージ] をクリックします。
2. [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、[次へ] 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 および Apex のコンポーネントの動的作成

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	<pre><aura:attribute name="newExpense" type="Expense__c" default="{ 'sobjectType': 'Expense__c', 'Name': '', 'Amount__c': 0, ... }" /></pre>
式のカスタム項目	<pre><ui:inputNumber value="{!!v.newExpense.Amount__c}" label="..." /></pre>
JavaScript 関数のカスタム項目	<pre>updateTotal: function(component) { ... for(var i = 0 ; i < expenses.length ; i++){ var exp = expenses[i]; total += exp.Amount__c; } ... }</pre>
JavaScript 関数で動的に作成されたコンポーネント	<pre>var myCmp = \$A.services.component.newComponent({ componentDef : { descriptor : "markup://c:myComponent" } });</pre>
JavaScript 関数のインターフェース比較	<code>aCmp.newInstanceOf("c:myInterface")</code>
イベントの登録	<code><aura:registerEvent type="c:updateExpenseItem" name="..." /></code>
イベントハンドラ	<code><aura:handler event="c:updateExpenseItem" action="..." /></code>
明示的な連動関係	<code><aura:dependency resource="markup://c:myComponent" /></code>

参照される項目	例
JavaScript 関数のアプリケーションイベント	<pre>var updateEvent = \$A.get("e.c:updateExpenseItem");</pre>
Apex で動的に作成されたコンポーネント	<pre>Cmp.expenseList expList = new Cmp.expenseList(); Cmp.c.expenseList expList = new Cmp.c.expenseList();</pre> <p>上記の 2 行のコードは同等です。デフォルトの名前空間がサポートされていますが、この場合は省略可能です。</p>

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	<code><yournamespace:myComponent /></code>
システム属性で使用されるコンポーネント	<code><aura:component extends="yournamespace:myComponent"> <aura:component implements="yournamespace:myInterface"></code>
Apex コントローラ	<code><aura:component controller="yournamespace.ExpenseController"></code>
属性データ型のカスタムオブジェクト	<code><aura:attribute name="expenses" type="yournamespace.Expense__c[]" /></code>
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	<code><aura:attribute name="newExpense" type="yournamespace.Expense__c" default="{ 'sobjectType': 'yournamespace__Expense__c', 'Name': '', 'yournamespace__Amount__c': 0, ... }" /></code>
式のカスタム項目	<code><ui:inputNumber value="{!v.newExpense.yournamespace__Amount__c}" label="..." /></code>
JavaScript 関数のカスタム項目	<code>updateTotal: function(component) { ... }</code>

参照される項目	例
JavaScript 関数で動的に作成されたコンポーネント	<pre>for(var i = 0 ; i < expenses.length ; i++){ var exp = expenses[i]; total += exp.yournamespace__Amount__c; } ... }</pre>
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");
Apex で動的に作成されたコンポーネント	Cmp.yournamespace.expenseList expList = new Cmp.yournamespace.expenseList();

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 (ページ 50) aura:application (ページ 225)
CSS スタイル	sample.css	コンポーネントのスタイル。	コンポーネント内の CSS (ページ 61)
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ためのクライアント側コントローラメソッド。	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 125)
設計	sample.design	Lightning App Builder または Lightning ページで使用されるコンポーネントでは必須。	Lightning ページと Lightning App Builder のコンポーネントの設定 (ページ 77)
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 80)
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。	DOM へのクライアント側表示 (ページ 169)
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 166)
SVG ファイル	sample.svg	Lightning App Builder 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 App Builder のコンポーネントの設定 (ページ 77)

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。

ローカル ID

ローカル ID はコンポーネント内で一意であり、範囲はそのコンポーネント内のみです。

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。

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

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。

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。

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 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。

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>
```

JavaScript で getGlobalId() 関数を使用して、コンポーネントのグローバル ID を取得できます。

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

逆の操作も可能で、グローバル ID があればコンポーネントを取得できます。

```
var comp = $A.getCmp(globalId);
```

 **メモ:** 詳細は、<https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app> (<mySalesforceInstance> は、na1 など、組織をホストするインスタンスの名前) の JavaScript API を参照してください。

関連トピック:

[ID によるコンポーネントの検索](#)

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、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。

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 `` 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 360)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](#)」(ページ 83)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 159)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、事实上 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.

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

最上位要素は THIS クラスと一致し、灰色の背景で表示されます。

<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、 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。

属性名の先頭文字は、英字またはアンダースコア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。

 **メモ:** 式にはハイフンを含む属性は使用できません。たとえば、`cmp.get("v.name-withHyphen")` はサポートされますが、`<ui:button label=" {!v.name-withHyphen} "/>` はサポートされません。

ここで、`?whom=you` を URL に追加して、ページを再読み込みします。クエリ文字列の値によって、`whom` 属性の値が設定されます。コンポーネントを要求するときにクエリ文字列で属性値を指定すると、そのコンポーネントの属性を設定することができます。

 **警告:** これは文字列型の属性でのみ機能します。

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。

属性の検証

`helloAttributes.app` に有効な属性のセットを定義したので、フレームワークでは、有効な属性のみがそのコンポーネントに渡されることが自動的に検証されます。

クエリ文字列 `?fakeAttribute=fakeValue` を追加して `helloAttributes.app` の要求を試してみてください。`helloAttributes.app` に `fakeAttribute` 属性がないというエラーが表示されます。

関連トピック:

[サポートされる aura:attribute の型](#)

[式](#)

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 `docsample:helloHTML` と `docsample: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 `docsample: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。

```
<!--docsample: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になります。

```
<!--docsample: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。

```
<!--docsample:nestedComponents-->  
  
<aura:component>  
  
    Observe! Components within components!  
  
    <docsample:helloHTML/>  
  
    <docsample: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、`docsample` 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 `docsample:helloHTML` です。

`nestedComponents cmp` が `docsample: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 `docsample:helloAttributes` の属性値として渡されます。

```
<!--docsample:nestedComponents2-->

<aura:component>

    <aura:attribute name="passthrough" type="String" default="passed attribute"/>

    Observe! Components within components!

    <docsample:helloHTML/>

    <docsample: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。

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 `docsample:helloAttributes` を追加します。 `docsample: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。

```
<!--docsample:nestedComponents3-->

<aura:component>

    <aura:attribute name="passthrough" type="String" default="passed attribute"/>

    Observe! Components within components!

    <docsample:helloHTML/>

    <docsample:helloAttributes whom=" {!v.passthrough} "/>

    <docsample: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>

    <ui:button label="Push Me"/>

    <!--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。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>

        <ui:button label="Push Me"/>

        <!--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。

```
<ui:panel>

    Hello world!

</ui:panel>
```

これは次の指定を簡易にしたものです。

```
<ui:panel>

  <aura:set attribute="body">
    Hello World!
  </aura:set>

</ui:panel>
```

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。

コンポーネントのソース

```
<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。コンポーネント `helloFacets.cmp` でこれらの属性を設定します。

コンポーネントのソース

```
<aura:component>

    See how we set the header facet.<br/>

    <auradocs:facetHeader>

        Nice body!

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

</aura:component>
```

`aura:set` は、スーパーコンポーネントから継承した属性の値を設定しますが、`v.body` の値を設定する場合は `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ときは、`<aura:if>` または `<aura:renderIf>` タグを使用します。または、JavaScript ロジックでマークアップを条件に応じて設定することもできます。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> か <aura:renderIf> か

<aura:if> は、そのボディまたは `else` 属性のマークアップのみを作成して表示するため、<aura:renderIf> よりも軽量です。条件付きマークアップが必要なときは、常にまず <aura:if> を試します。

<aura:renderIf> は、`true` と `false` の両方の状態に対してマークアップを表示する場合にのみ使用を検討します。最初に表示されないコンポーネントを作成するには、サーバに往復処理を要求することになります。

<aura:if> と <aura:renderIf> の簡単な比較を次に示します。

	<aura:if>	<aura:renderIf>
表示	1つのブランチのみを作成して表示	両方のブランチを作成するが1つのみ表示
条件の切り替え	現在のブランチを非表示にして破棄。他のブランチを作成して表示。	現在のブランチを非表示にして他のブランチを表示
空のブランチ	DOM プレースホルダを作成	DOM プレースホルダを作成

条件付きマークアップの代替方法の検討

<aura:if> または <aura:renderIf> の代替方法を検討すべきいくつかの使用事例を次に示します。

表示を切り替える

<aura:if> または <aura:renderIf> タグを使用して、マークアップの表示を切り替えないでください。代わりに、CSS を使用します。「マークアップの動的な表示または非表示」(ページ 188)を参照してください。

条件付きロジックをネストするか、反復内で条件付きロジックを使用する必要がある

<aura:if> または <aura:renderIf> タグを使用すると、多数のコンポーネントが作成され、パフォーマンスが劣化する可能性があります。また、マークアップで条件付きロジックを過度に使用すると、マークアップが雑然として管理しにくくなることがあります。

代わりに、`init` イベントハンドラで JavaScript ロジックを使用するなどの代替方法を検討します。「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 182)を参照してください。

関連トピック:

[条件式](#)

表示ラベルの使用

このフレームワークでは、表示ラベルがサポートされており、コードと項目表示ラベルを分離できます。

このセクションの内容:

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。

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。

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。

```
<ui:inputNumber label="Pick a Number:" value="54" />
```

表示ラベルは、入力項目の左側に配置され、`labelClass="assistiveText"` を設定することで非表示にできます。`assistiveText` は、アクセシビリティをサポートするために使用されるグローバルスタイルクラスです。

関連トピック:

アクセシビリティのサポート

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"/>

    <ui:button label="Set Label" aura:id="button1" press="{'!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"/>

    <ui:inputTextarea aura:id="textarea"
        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。

コンポーネントは、ブラウザのロケール情報を取得し、それに従って日時を表示します。次の例では、デフォルトの `langLocale` および `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。

[コンポーネントのソース](#)

```
<aura:component>

    <ui:outputDateTime value="2013-05-07T00:17:08.997Z" timezone="Europe/Berlin"
        langLocale="de"/>
```

```
</aura:component>
```

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

さらに、グローバル値プロバイダ \$Locale を使用してブラウザのロケール情報を取得できます。このフレームワークは、デフォルトでブラウザのロケールを使用しますが、グローバル値プロバイダを利用して他のロケールを使用するように設定できます。

ローカライズサービスの使用

フレームワークのローカライズサービスでは、日付、時刻、数値、通貨のローカライズを管理できます。

次の例では、\$Locale とローカライズサービスを使用して、設定済みの形式で日時を設定します。

```
var dateFormat = $A.get("$Locale.dateFormat");  
  
var dateString = $A.localizationService.formatDateTime(new Date(), dateFormat);
```

ブラウザの日付情報を取得しない場合は、独自の日付形式を指定できます。次の例では、日付形式を指定し、ブラウザの言語ロケール情報を使用します。

```
var dateFormat = "MMMM d, yyyy h:mm a";  
  
var userLocaleLang = $A.get("$Locale.langLocale");  
  
return $A.localizationService.formatDate(date, dateFormat, userLocaleLang);
```

次の例では、2つの日付を比較して、一方がもう一方より後の日付かどうかを確認します。

```
if( $A.localizationService.isAfter(StartTime,EndTime) ) {  
  
    //throw an error if StartTime is after EndTime  
  
}
```

関連トピック:

[グローバル値プロバイダ](#)

Lightning コンポーネントの有効化

組織で Lightning コンポーネントを有効化するように選択する必要があります。

- [設定] で、[開発] > [Lightning コンポーネント] をクリックします。
- [Lightning コンポーネントを有効化] チェックボックスをオンにします。



警告: このベータバージョンの Lightning コンポーネントでは、Salesforce1 の Force.com Canvas アプリケーションはサポートされていません。Lightning コンポーネントを有効化すると、組織のすべての Force.com Canvas アプリケーションが Salesforce1 で動作しなくなります。

3. [保存]をクリックします。

Salesforce1へのLightningコンポーネントの追加

作成したLightningコンポーネントを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する必要があります。

エディション

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インターフェースを使用すると、コンポーネントをSalesforce1のナビゲーションメニューで使用できます。

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. 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]で、[作成]>[タブ]をクリックします。
 - 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. カスタムタブに表示する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。
- メモ:** カスタムタブの作成を前提条件として、Salesforce1ナビゲーションメニューでコンポーネントを有効化できますが、Salesforce フルサイトからの 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、[モバイル管理]>[モバイルナビゲーション] をクリックします。
 - 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 ページと Lightning App Builder のコンポーネントの設定

Lightning ページまたは Lightning App Builder でカスタム Lightning コンポーネントを使用する前に、2つの調整を行う必要があります。

新規インターフェースをコンポーネントに追加する

コンポーネントを Lightning App Builder または Lightning ページに表示するには、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。

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">

    <aura:attribute name="greeting" type="String" default="Hello" />
    <aura:attribute name="subject" type="String" default="World" />

    <div style="box">
        <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
```

```
</aura:component>
```

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

Lightning コンポーネントを Lightning ページおよび Lightning App Builder で使用できるようにするには、コンポーネントバンドルにデザインリソースを含める必要があります。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。

Lightning コンポーネントの属性をシステム管理者が Lightning App Builder 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 App Builder で自動的にユーザに表示されます。コンポーネント定義内のデフォルト値が設定された必須属性と必須とマークされていない属性は、デザインリソースで指定する必要があります。指定しないとユーザには表示されません。

「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>
```

項目を選択リストとして表示するには、次のように、`datasource` をデザインリソースの属性に追加します。

```
<design:attribute name="Name" datasource="value1,value2,value3" />
```

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。Lightning App Builder がどのように表示されるかは、コンポーネントで定義するデータ型によって異なります。次に例を示します。

- `<aura:attribute name="Name" type="String" />` は、選択リストとして表示されます。
- `<aura:attribute name="Name" type="String[]" />` は、複数選択リストとして表示されます。

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 App Builder 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。リソースをコンポーネントバンドルに追加するだけです。

「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:

[コンポーネントのバンドル](#)

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:** 標準で付属するすべてのコンポーネントの詳細は、
<https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app>
(<mySalesforceInstance> は、na1 など、組織をホストするインスタンスの名前です)にある 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。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 5)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](#)

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。
	<ul style="list-style-type: none"> • 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
<code><aura:component></code>	<code><aura:component description="Represents a button element"></code>
<code><aura:attribute></code>	<code><aura:attribute name="langLocale" type="String" description="The language locale used to format date value." /></code>
<code><aura:event></code>	<code><aura:event type="COMPONENT" description="Indicates that a keyboard key has been pressed and released"/></code>
<code><aura:interface></code>	<code><aura:interface description="A common interface for date components"/></code>
<code><aura:registerEvent></code>	<code><aura:registerEvent name="keydown" type="ui:keydown" description="Indicates that a key is pressed"/></code>

ドキュメントの表示

作成したドキュメントは、

`https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`
(`<mySalesforceInstance>` は、`na1` など、組織をホストするインスタンスの名前です) で表示できます。

関連トピック:

[リファレンスの概要](#)

第4章 式

トピック:

- 式の動的出力
- 条件式
- 値プロバイダ
- 式の評価
- 式の演算子のリファレンス
- 式の関数のリファレンス

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。

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。

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、テキスト値を单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](#)

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。

最も一般的な値プロバイダは、`v` と `c` で、ビューやコントローラで使用されます。

値プロバイダ	説明
<code>v</code>	コンポーネントの属性セット
<code>c</code>	固有のアクションおよびイベントハンドラを含むコンポーネントのコントローラ

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。

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。

 **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。

項目および関連オブジェクトへのアクセス

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](#)

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は次のとおりです。

- `globalID` — 「[コンポーネントの ID](#)」 (ページ 60)を参照してください。
- `$Browser` — 「[\\$Browser](#)」 (ページ 86)を参照してください。
- `$Locale` — 「[\\$Locale](#)」 (ページ 87)を参照してください。

\$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提供します。

属性	説明
<code>formFactor</code>	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 <code>FormFactor</code> Enum 値を返します。 <ul style="list-style-type: none"> • <code>DESKTOP</code>: デスクトップクライアント • <code>PHONE</code>: 電話 (ブラウザ対応の携帯電話やスマートフォンを含む) • <code>TABLET</code>: タブレットクライアント (<code>isTablet</code> が <code>true</code> を返します)
<code>isAndroid</code>	ブラウザが Android デバイス上で実行されているか (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isIOS</code>	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 (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isIPad</code>	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 (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isiPhone</code>	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 (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isPhone</code>	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 (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isTablet</code>	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 (<code>true</code>)、否か (<code>false</code>) を示します。
<code>isWindowsPhone</code>	ブラウザが Windows Phone 上で実行されているか (<code>true</code>)、否か (<code>false</code>) 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:** 次の例では、コンポーネントを表示しているブラウザのオペレーティングシステムおよびデバイスに応じて、true または false の値が返されます。

コンポーネントのソース

```
<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、ブラウザのロケールに関する情報を返します。

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

属性	説明	サンプル値
country	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	「.」
grouping	桁区切り記号	「,」
language	言語コード	「en」、「de」、「zh」
langLocale	ロケール ID	「en_US」、「en_GB」

属性	説明	サンプル値
timezone	タイムゾーン ID	「America/Los_Angeles」、 「America/New_York」
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」

 **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](#)

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。一部のコンポーネントは、<ui:button> の press 属性など、他の属性を使用してユーザインターフェースイベントへのアクションの割り当てを簡略化します。

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、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 84) を参照してください。

```
<ui:button aura:id="likeBtn"
  label="{! (v.likeId == null) ? 'Like It' : 'Unlike It'}"
  press="{! (v.likeId == null) ? c.likeIt : c.unlikeIt}"
/>
```

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。

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
整数	2	整数は小数点や指数のない数字です。
浮動小数	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

文字列リテラルは单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
文字列	'hello world'	文字列リテラルは单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: <ul style="list-style-type: none"> • \t (タブ) • \n (改行) • \r (行頭復帰) エスケープ文字: <ul style="list-style-type: none"> • \" (リテラル") • \' (リテラル') • \\ (リテラル\\)
Unicode	'\u#####	

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

演算子	代替方法	使用方法	説明
<code>!=</code>	<code>ne</code>	<code>1 != 2</code> <code>1 != true</code> <code>1 != '1'</code> <code>null != false</code> <code>1 ne 2</code>	は、一貫性なくクライアントあるいはサーバで評価するため、信頼できません。
<code><</code>	<code>lt</code>	<code>1 < 2</code> <code>1 lt 2</code>	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、 <code>true</code> を返します。<演算子を <code>&lt;</code> 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、 <code>lt</code> 演算子を使用できます。
<code>></code>	<code>gt</code>	<code>42 > 2</code> <code>42 gt 2</code>	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、 <code>true</code> を返します。
<code><=</code>	<code>le</code>	<code>2 <= 42</code> <code>2 le 42</code>	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、 <code>true</code> を返します。<=演算子を <code>&lt;=</code> 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、 <code>le</code> 演算子を使用できます。
<code>>=</code>	<code>ge</code>	<code>42 >= 42</code> <code>42 ge 42</code>	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、 <code>true</code> 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。

演算子	使用方法	説明
<code>&&</code>	<code>isEnabled && hasPermission</code>	両方のオペランドが <code>true</code> の場合に、 <code>true</code> を返します。 <code>&&</code> 演算子を <code>&amp;&amp;</code> 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、 <code>and()</code> 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、 <code>and(isEnabled, hasPermission)</code> です。

演算子	使用方法	説明
	hasPermission isRequired	いずれかのオペランドが true の場合に、true を返します。
!	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。	+ (二項)
sub	subtract	sub(10, 2)	最初の引数から2つ目の引数を減算します。	- (二項)
mult	multiply	mult(2, 10)	最初の引数を2つ目の引数で乗算します。	* (二項)
div	divide	div(4, 2)	最初の引数を2つ目の引数で除算します。	/ (二項)
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。	% (二項)
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、負の場合に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。	+ (二項)

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)	<p>引数が空の場合に <code>true</code> を返します。空の引数とは、<code>undefined</code>、<code>null</code>、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。</p> <p>メモ: この関数は、<code>String</code>、<code>Array</code>、<code>Object</code>、<code>List</code>、<code>Map</code>、<code>Set</code> のいずれかの型の引数で機能します。</p> <p>ヒント: <code>{! !empty(v.myArray)}</code> は <code>{!v.myArray && v.myArray.length > 0}</code> よりも評価が速いため、パフォーマンスを向上させるためには <code>empty()</code> の使用をお勧めします。</p> <p>JavaScript の <code>\$A.util.isEmpty()</code> メソッドは、マークアップの <code>empty()</code> 式と同じです。</p>

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 <code>true</code> 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。	<code>==</code> または <code>eq</code>
notequals	notequals(1,2)	指定した引数が等しくない場合に <code>true</code> 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。	<code>!=</code> または <code>ne</code>
lessthan	lessthan(1,5)	最初の引数の数値が 2 つ目の引数 <code><</code> または <code>lt</code> より小さい場合に <code>true</code> を返します。	
greaterthan	greaterthan(5,1)	最初の引数の数値が 2 つ目の引数 <code>></code> または <code>gt</code> より大きい場合に <code>true</code> を返します。	
lessthanorequal	lessthanorequal(1,2)	最初の引数の数値が 2 つ目の引数 <code><=</code> または <code>le</code> 以下の場合に <code>true</code> を返します。	

関数	使用方法	説明	対応する演算子
greaterthanorequal	greaterthanorequal(2,1)	最初の引数の数値が2つ目の引数 <code>>=</code> または <code>ge</code> 以上の場合に <code>true</code> を返します。	

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 <code>true</code> の場合に <code>&&</code> <code>true</code> を返します。	
or	or(hasPermission, hasVIPPass)	いずれかの引数が <code>true</code> の場合に <code> </code> <code>true</code> を返します。	
not	not(isNew)	引数が <code>false</code> の場合に <code>true</code> <code>!</code> 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。 <code>true</code> 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。	<code>? :</code> (3 項)

第5章 UI コンポーネント

フレームワークには、共通のユーザ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。

使用可能なすべてのコンポーネントについては、

<https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app>

(`<mySalesforceInstance>` は、na1 など、組織をホストするインスタンスの名前です) のコンポーネント参照をご覧ください。

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	<code>ui:message</code>	さまざまな重要度レベルのメッセージ通知
メニュー	<code>ui:menu</code>	表示を制御するトリガを含むドロップダウンリスト
	<code>ui:menuList</code>	メニュー項目のリスト
	<code>ui:actionMenuItem</code>	アクションをトリガするメニュー項目
	<code>ui:checkboxMenuItem</code>	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
	<code>ui:radioMenuItem</code>	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
	<code>ui:menuItemSeparator</code>	メニュー項目の視覚的な分離線
	<code>ui:menuItem</code>	<code>ui:menuList</code> 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
	<code>ui:menuTrigger</code>	メニューを展開したり折りたたんだりするトリガ
	<code>ui:menuTriggerLink</code>	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 <code>ui:menuTrigger</code> 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 ui:outputCheckbox	複数選択をサポートする選択可能なオプション 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 ui:inputSelectOption	オプションを含むドロップダウンリスト 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 ui:outputCurrency	通貨を入力するための入力項目 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 ui:outputEmail	メールアドレスを入力するための入力項目 クリック可能なメールアドレスを表示します
日時	ui:inputDate ui:inputDateTime ui:outputDate	日付を入力するための入力項目 日時を入力するための入力項目 デフォルトまたは指定された形式で、日付を表示します

型	主要コンポーネント	説明
	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
	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](#)
- [コンポーネントのバンドル](#)

UI イベント

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取得して、こうした操作を処理しやすくなります。

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、「[リファレンスドキュメントアプリケーション](#)」(ページ 225)を参照してください。

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:** 使用可能なすべてのコンポーネント属性およびイベントについては、[リファレンスドキュメントアプリケーション](#) (ページ 225)のコンポーネント参照をご覧ください。

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="assitiveText"` 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);  
  
}
```

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。

日時項目

日時項目では、クライアント側のローカライズ、日付ピッカー、共通のキーワードイベントとマウスイベントがサポートされます。このような項目コンポーネントから出力を表示する場合は、それぞれの `ui:output` コンポーネントを使用します。たとえば、`ui:inputDate` コンポーネントの出力を表示するには、`ui:outputDate` を使用します。

日時項目は、次のコンポーネントで表されます。

データ型	説明	関連コンポーネント
日付	<code>text</code> 型の日付を入力するための入力項目。	<code>ui:inputDate</code> <code>ui:outputDate</code>
日時	<code>text</code> 型の日時を入力するための入力項目。	<code>ui:inputDateTime</code> <code>ui:outputDateTime</code>

日時項目の使用

次に、日付ピッカーを使用した日付項目の基本設定を示します。

```
<ui:inputDate aura:id="dateField" label="Birthday" value="2000-01-01" displayDatePicker="true"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputDate">

  <label class="uiLabel-left uiLabel">

    <span>Birthday</span>

  </label>

  <input placeholder="MMM d, yyyy" type="text" class="uiInput uiInputDate">

  <a class="datePicker-openIcon" aria-haspopup="true">

    <span class="assistiveText">Date Picker</span>

  </a>

  <div class="uiDatePicker">

    <!--Date picker set to visible when icon is clicked-->

  </div>

</div>
```

```
</div>  
</div>
```

項目値のバインド

{!v.myAttribute.Name} または {!v.myAttribute.namespace__MyField__c} などの式を使用し、Apex コントローラで入力値を保存することで、オブジェクトの項目に項目値をバインドできます。「[スタンダードアロン Lightning アプリケーションを作成する](#)」(ページ 9)の例を参照してください。

日時項目のスタイル設定

日時項目の外観と、コンポーネントの CSS リソースでの出力のスタイルを設定できます。

次の例は、myStyle セレクタを使用して ui:inputDateTime コンポーネントにスタイルを設定します。

```
<!-- Component markup -->  
  
<ui:inputDateTime class="myStyle" label="Date" displayDatePicker="true"/>  
  
/* CSS */  
  
.THIS .myStyle {  
  
    border: 1px solid #dce4ec;  
  
    border-radius: 4px;  
  
}
```

関連トピック:

- [入力コンポーネントの表示ラベル](#)
- [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](#)
- [ローカライズ](#)
- [コンポーネント内の CSS](#)

数値項目

数値項目には、数値を含めることができます。数値項目では、クライアント側の書式設定、ローカライズ、共通のキーワードイベントとマウスイベントがサポートされます。

このような項目コンポーネントから出力を表示する場合は、それぞれの ui:output コンポーネントを使用します。たとえば、ui:inputNumber コンポーネントの出力を表示するには、ui:outputNumber を使用します。

数値項目は、次のコンポーネントで表されます。

型	関連コンポーネント	説明
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
通貨	ui:inputCurrency	通貨を入力するための入力項目
	ui:outputCurrency	通貨を表示します

数値項目の使用

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 uiInputText uiInputNumber">
  <label class="uiLabel-left uiLabel">
    <span>Enter age</span>
  </label>
  <input aria-describedby placeholder type="text"
    class="uiInput uiInputText uiInputNumber">
</div>
```

項目値のバインド

{!v.myAttribute.Name} または {!v.myAttribute.namespace__MyField__c} などの式を使用し、Apex コントローラで入力値を保存することで、オブジェクトの項目に項目値をバインドできます。「[スタンダードアロン Lightning アプリケーションを作成する](#)」(ページ 9)の例を参照してください。

有効な数値を返す

ui:inputNumber コンポーネントの値には、有効な数値が予期され、カンマは使用できません。カンマを含めるには、type="String" の代わりに 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}"/>
```

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}" />
```

数値項目の書式設定とローカライズ

`format` 属性は、数値入力の形式を決定します。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。次のコードは、指定された `format` 属性に基づいて 10,000.00 を表示する、数値項目の基本設定です。

```
<ui:label label="Cost" for="costField"/>  
  
<ui:inputNumber aura:id="costField" format="#,\#\#0,000.00#" value="10000"/>
```

数値項目のスタイル設定

数値項目および出力の外観のスタイルを設定できます。コンポーネントの CSS ファイルで、対応するクラスセレクタを追加します。次のクラスセレクタは、数値の文字列表示のスタイルを指定します。たとえば、`ui:inputCurrency` コンポーネントのスタイルを設定するには、`.THIS.uiInputCurrency` を使用するか、最上位要素の場合は `.THIS.uiInputCurrency` を使用します。

次の例は、`myStyle` セレクタを使用して `ui:inputNumber` コンポーネントにスタイルを設定します。

```
<!-- Component markup -->  
  
<ui:inputNumber class="myStyle" label="Amount" placeholder="0" />  
  
/* CSS */  
  
.THIS .myStyle {  
  
    border: 1px solid #dce4ec;  
  
    border-radius: 4px;  
  
}
```

関連トピック:

- [入力コンポーネントの表示ラベル](#)
- [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](#)
- [ローカライズ](#)
- [コンポーネント内の CSS](#)

テキスト項目

テキスト項目には、英数字と特殊文字を含めることができます。共通のキーボードイベントとマウスイベントを使用できます。このような項目コンポーネントから出力を表示する場合は、それぞれの `ui:output` コンポーネントを使用します。たとえば、`ui:inputPhone` コンポーネントの出力を表示するには、`ui:outputPhone` を使用します。

テキスト項目は、次のコンポーネントで表されます。

型	関連コンポーネント	説明
メール	<code>ui:inputEmail</code>	メールアドレスを入力するための入力項目
	<code>ui:outputEmail</code>	クリック可能なメールアドレスを表示します
パスワード	<code>ui:inputSecret</code>	秘密のテキストを入力するための入力項目
電話番号	<code>ui:inputPhone</code>	電話番号を入力するための入力項目
	<code>ui:outputPhone</code>	クリック可能な電話番号を表示します
リッチテキスト	<code>ui:inputRichText</code>	リッチテキストを入力するための入力項目
	<code>ui:outputRichText</code>	リッチテキストを表示します
テキスト	<code>ui:inputText</code>	1行のテキストを入力するための入力項目
	<code>ui:outputText</code>	テキストを表示します
テキストエリア	<code>ui:inputTextArea</code>	複数行のテキストを入力するための入力項目
	<code>ui:outputTextArea</code>	参照のみのテキストエリアを表示します
URL	<code>ui:inputURL</code>	URLを入力するための入力項目
	<code>ui:outputURL</code>	クリック可能な URLを表示します

テキスト項目の使用

通常、テキスト項目はフォームで使用されます。たとえば、次の例はメール項目の基本設定です。

```
<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 uiInputText uiInputEmail">

  <label class="uiLabel-left uiLabel">
    <span>Email</span>
  </label>

  <input placeholder="abc@email.com" type="email" class="uiInput uiInputText uiInputEmail">
</div>
```

 **メモ:** force:navigateToURL イベントを使用して、要素を URL リンクのように動作させることもできます。詳細は、「[force:navigateToURL](#)」(ページ 342) を参照してください。

項目値のバインド

{!v.myAttribute.Name} または {!v.myAttribute.namespace__MyField__c} などの式を使用し、Apex コントローラで入力値を保存することで、オブジェクトの項目に項目値をバインドできます。「[スタンドアロン Lightning アプリケーションを作成する](#)」(ページ 9) の例を参照してください。

テキスト項目のスタイル設定

テキスト項目および出力の外観のスタイルを設定できます。コンポーネントの CSS ファイルで、対応するクラスセレクタを追加します。

たとえば、ui:inputPhone コンポーネントのスタイルを設定するには、.THIS .uiInputPhone を使用するか、最上位要素の場合は .THIS.uiInputPhone を使用します。

次の例は、myStyle セレクタを使用して ui:inputText コンポーネントにスタイルを設定します。

```
<!-- Component markup-->

<ui:inputText class="myStyle" label="Name" />

/* CSS */

.THIS .myStyle {
  border: 1px solid #dce4ec;
  border-radius: 4px;
```

```
}
```

関連トピック:

- [リッチテキスト項目](#)
- [入力コンポーネントの表示ラベル](#)
- [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](#)
- [ローカライズ](#)
- [コンポーネント内の CSS](#)

リッチテキスト項目

ui:inputRichText は、リッチテキストを入力するための入力項目です。次のコードは、テキストエリアとボタンとして表示される、このコンポーネントの基本実装を示します。ボタンをクリックすると、ui:outputRichText コンポーネントの入力値を返すクライアント側のコントローラアクションが実行されます。この場合、値は「Aura」を太字で返し、「input rich text demo」を赤色で返します。

```
<!--Rich text demo-->

<ui:inputRichText isRichText="false" aura:id="inputRT" label="Rich Text Demo"
    cols="50" rows="5" value="Aura", <span style='color:red'>input
    rich text demo</span>"/>

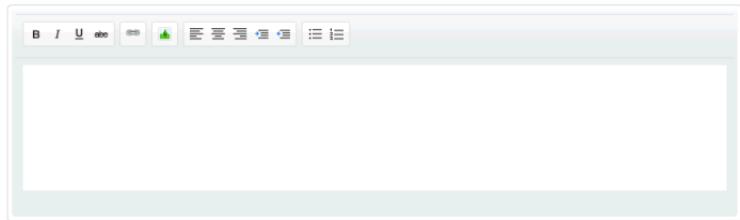
<ui:button aura:id="outputButton"
    buttonTitle="Click to see what you put into the rich text field"
    label="Display" press="{!!c.getInput}"/>

<ui:outputRichText aura:id="outputRT" value=" " />

/*Client-side controller*/

getInput : function(cmp) {
    var userInput = cmp.find("inputRT").get("v.value");
    var output = cmp.find("outputRT");
    output.set("v.value", userInput);
}
```

このデモでは、isRichText="false" 属性によってコンポーネントが ui:inputTextArea コンポーネントに置き換えられます。この属性が設定されていない場合、以下のような WYSIWYG リッチテキストエディタが表示されます。



リッチテキストエディタの幅と高さは、`ui:inputTextArea` コンポーネントの幅と高さとは独立しています。`isRichText="false"` に設定した場合にコンポーネントの幅と高さを設定するには、`cols` および `rows` 属性を使用します。それ以外の場合は、`width` および `height` 属性を使用します。

関連トピック:

[テキスト項目](#)

チェックボックス

チェックボックスは、クリックおよびアクションの実行が可能であり、複数の選択肢のグループとして表示できます。チェックボックスは、動作とイベントを `ui:input` から継承する `ui:inputCheckbox` を使用して作成できます。`value` および `disabled` 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。イベントは、チェックボックスごとに別個に使用する必要があります。

次に、チェックボックスを設定する基本的な方法をいくつか示します。

オン

チェックボックスをオンにするには、`value="true"` を設定します。次の例は、チェックボックスの初期値を設定します。

```
<aura:attribute name="check" type="Boolean" default="true"/>
<ui:inputcheckbox value="{!v.check}"/>
```

オフの状態

```
<ui:inputcheckbox disabled="true" label="Select" />
```

前の例の結果、次の HTML になります。

```
<label class="uiLabel-left uiLabel" for="globalId"><span>Select</span></label>
<input disabled="disabled" type="checkbox" id="globalId" class="uiInput uiInputCheckbox">
```

イベントの操作

`ui:inputCheckbox` の共通イベントには、`click` イベントと `change` イベントがあります。たとえば、`click=" {!c.done}"` では、関数名 `done` を使用してクライアント側のコントローラがコールされます。

次のコードは、チェックボックス項目にチェックマークを入れます。

```
<!--The checkbox-->
<ui:inputcheckbox label="Cross this out" click=" {!c.crossout}" class="line" />
```

```
/*The controller action*/  
  
crossout : function(cmp, event) {  
  
    var elem = event.getSource().getElement();  
  
    $A.util.toggleClass(elem, "done");  
  
}
```

チェックボックスのスタイル設定

ui:inputCheckbox コンポーネントは、通常の CSS スタイル設定を使用してカスタマイズできます。この例では、次の画像を含むチェックボックスを示します。



```
<ui:inputCheckbox labelClass="check"  
                  label="Select?" value="true" />
```

次の CSS スタイルは、指定された画像を持つデフォルトのチェックボックスに置き換わります。

```
.THIS input[type="checkbox"] {  
    display: none;  
}  
  
.THIS .check span {  
    margin: 20px;  
}  
  
.THIS input[type="checkbox"]+label {  
    display: inline-block;  
    width: 20px;  
    height: 20px;
```

```
vertical-align: middle;  
  
background: url('images/checkbox.png') top left;  
  
cursor: pointer;  
  
}  
  
.THIS input[type="checkbox"]:checked+label {  
  
background:url('images/checkbox.png') bottom left;  
  
}
```

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](#)

ラジオボタン

ラジオボタンは、クリックおよびアクションの実行が可能であり、グループとして表示された場合は1つのみ選択できます。ラジオボタンは、動作とイベントを `ui:input` から継承する `ui:inputRadio` を使用して作成できます。`value` および `disabled` 属性は、ラジオボタンの状態を制御し、`click` や `change` などのイベントはその動作を決定します。イベントは、ラジオボタンごとに別個に使用する必要があります。

メニューでラジオボタンを使用する場合は、代わりに `ui:radioMenuItem` を使用します。

ラジオボタンを設定するいくつかの基本的な方法を次に示します。

選択済み

ラジオボタンを選択するには、`value="true"` に設定します。

```
<ui:inputRadio value="true" label="Select?" />
```

オフの状態

```
<ui:inputRadio label="Select" disabled="true" />
```

前の例の結果、次の HTML になります。

```
<label class="uiLabel-left uiLabel" for="globalId"><span>Select</span></label>  
<input type="radio" id="globalId" class="uiInput uiInputRadio">
```

属性を使用した表示ラベルの指定

属性を使用して、表示ラベルの値を初期化することもできます。次の例では、属性を使用してラジオボタンの表示ラベルを入力し、ラジオボタンが選択または選択解除されたときにクライアント側のコントローラアクションに結び付けます。

```
<!--docsample:labelsAttribute-->

<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.doSomething}"/>
    </aura:iteration>
</aura:component>
```

イベントの操作

`ui:inputRadio` の共通イベントには、`click` イベントと `change` イベントがあります。たとえば、`click=" {!c.showItem}"` では、関数名 `showItem` を使用してクライアント側のコントローラアクションがコールされます。

次のコードは、ラジオボタンがクリックされたときにコンポーネントの CSS クラスを更新します。

```
<!--The radio button-->

<ui:inputRadio click=" {!c.showItem}" label="Show Item"/>

/* The controller action */

showItem : function(cmp, event) {
    var elem = cmp.find('myCmp');
    $A.util.toggleClass(elem, "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](#)

ボタン

ボタンは、クリックおよびアクションの実行が可能であり、テキスト表示ラベル、画像、またはその両方を設定できます。ボタンは次の3通りの方法で作成できます。

- テキストのみのボタン

```
<ui:button label="Find" />
```

- 画像のみのボタン

```
<!-- 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。

- テキストと画像を含むボタン

```
<!-- Component markup -->

<ui:button label="Find" />

/** CSS **/


THIS.uiButton {
    background: url(/path/to/img) no-repeat;
}
```

HTML 表示

テキストと画像を含むボタンのマークアップの結果、次の HTML になります。

```
<button class="default uiBlock uiButton" accesskey type="button">
```

```
<span class="label bBody truncate" dir="ltr">Find</span>  
</button>
```

クリックイベントの操作

ui:button コンポーネントの press イベントは、ユーザがボタンをクリックすると起動されます。次の例の press=" {!c.getInput}" は、入力テキスト値を出力する関数名 getInput を使用して、クライアント側のコントローラアクションをコールします。

```
<aura:component>  
  
    <ui:inputText aura:id="name" label="Enter Name:" placeholder="Your Name" />  
  
    <ui:button aura:id="button" label="Click me" press=" {!c.getInput}"/>  
  
    <ui:outputText aura:id="outName" value="" class="text"/>  
  
</aura:component>  
  
/* Client-side controller */  
  
{  
  
    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);  
  
    }  
}
```

ボタンのスタイル設定

ui:button コンポーネントは、通常の CSS スタイル設定を使用してカスタマイズできます。コンポーネントの CSS リソースで、次のクラスセレクタを追加します。

```
.THIS.uiButton {  
  
    margin-left: 20px;  
  
}
```

ボタンコンポーネントが最上位レベルの要素である場合、.THIS.uiButton セレクタでスペースは追加されません。

アプリケーションですべての `ui:button` コンポーネントのスタイル設定を上書きするには、アプリケーションの CSS リソースで次のクラスセレクタを追加します。

```
.THIS .uiButton {  
    margin-left: 20px;  
}
```

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](#)

ドロップダウンリスト

ドロップダウンリストには、選択可能なオプションを含むドロップダウンメニューが表示されます。単一選択と複数選択の両方がサポートされています。ドロップダウンリストを作成するには、`ui:input` から動作とイベントを継承する `ui:inputSelect` を使用します。

ドロップダウンリストを設定するいくつかの基本的な方法を次に示します。

複数選択の場合、`multiple` 属性を `true` に設定します。

単一選択

```
<ui:inputSelect>  
  
    <ui:inputSelectOption text="Red"/>  
  
    <ui:inputSelectOption text="Green" value="true"/>  
  
    <ui:inputSelectOption text="Blue"/>  
  
</ui:inputSelect>
```

複数選択

```
<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>
```

デフォルトの選択値は、`value="true"` で指定されます。各オプションは `ui:inputSelectOption` で表されます。

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、条件に応じてオプションを表示します。

```
<aura:attribute name="contacts" type="String[]" default="All Contacts,Others"/>

<ui:inputSelect>

  <aura:iteration items="{!!v.contacts}" var="contact">

    <aura:if isTrue="={!contact == 'All Contacts'}">

      <ui:inputSelectOption text=" {!contact}" label=" {!contact}"/>

      <aura:set attribute="else">

        <ui:inputSelectOption text="All Primary" label="All Primary"/>

        <ui:inputSelectOption text="All Secondary" label="All Secondary"/>

      </aura:set>

    </aura:if>

  </aura:iteration>

</ui:inputSelect>
```

動的なオプションの生成

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。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", selected: "false" }
    ];

    cmp.set("v.options", opts);
  }
});
```

```

        { class: "optionClass", label: "Option2", value: "opt2" },
        { class: "optionClass", label: "Option3", value: "opt3" }

    ];
    cmp.find("InputSelectDynamic").set("v.options", opts);
}
)

```

 メモ: `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語です。二重引用符で囲んだ `"class"` を使用することをお勧めします。

上記のデモの `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。

`InputOption` オブジェクトには次のパラメータがあります。

パラメータ	型	説明
<code>label</code>	<code>String</code>	ユーザインターフェースに表示するオプションの表示ラベル。
<code>name</code>	<code>String</code>	オプションの名前。
<code>selected</code>	<code>boolean</code>	オプションが選択されているかどうかを示します。
<code>value</code>	<code>String</code>	このオプションの値。

イベントの操作

`ui:inputSelect` の共通イベントには、`change` イベントと `click` イベントがあります。たとえば、`change=" {!c.onSelectChange} "` では、ユーザが選択を変更したときに、関数名 `onSelectChange` を使用してクライアント側のコントローラアクションがコールされます。

項目レベルのエラーのスタイル設定

`ui:inputSelect` コンポーネントは、通常の CSS スタイル設定を使用してカスタマイズできます。次の CSS サンプルは、ドロップダウンメニューに固定幅を追加します。

```
.THIS.uiInputSelect {
    width: 200px;
    height: 100px;
```

```
}
```

または、`class` 属性を使用して独自の CSS クラスを指定します。

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](#)
- [コンポーネント内の CSS](#)

項目レベルのエラー

項目レベルのエラーは、ユーザ入力後に項目で検証エラーが発生した場合に表示されます。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` が作成され、`click` や `mouseover` などの基本イベントが提供されます。詳細は、「[項目の検証](#)」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](#)
- [コンポーネント内の CSS](#)

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。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>
```

オブジェクトから項目のリストを表示できます。次の例では、`aura:iteration` を使用してメニューに取引先責任者名のリストを表示します。

```
<aura:component>

    <aura:attribute name="contacts" type="String[]" default="All,Primary,Secondary"/>

    <ui:menu>

        <ui:menuTriggerLink label="Select Contact"/>

        <ui:menuList>

            <aura:iteration var="contact" items="{!v.contacts}">

                <ui:actionMenuItem label="{!!contact}"/>

            </aura:iteration>

        </ui:menuList>

    </ui:menu>

</aura:component>
```

次のコンポーネントは、`ui:menu` にネストされています。

コンポーネント	説明
<code>ui:menu</code>	表示を制御するトリガを含むドロップダウンリスト
<code>ui:menuList</code>	メニュー項目のリスト
<code>ui:actionMenuItem</code>	アクションをトリガするメニュー項目
<code>ui:checkboxMenuItem</code>	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
<code>ui:radioMenuItem</code>	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
<code>ui:menuItemSeparator</code>	メニュー項目の視覚的な分離線
<code>ui:menuItem</code>	<code>ui:menuList</code> 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
<code>ui:menuTrigger</code>	メニューを展開したり折りたたんだりするトリガ
<code>ui:menuTriggerLink</code>	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 <code>ui:menuTrigger</code> を拡張します

第6章 アクセシビリティのサポート

トピック:

- アクセシビリティの考慮事項
- ボタンの表示ラベル
- ヘルプとエラー メッセージ
- 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

これらのコンポーネントまたはそのサブ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。アプリケーションで使用可能なコンポーネントについては、「[UIコンポーネント](#)」を参照してください。

アクセシビリティの考慮事項

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ー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。

関連トピック:

- [アクセシビリティのサポート](#)
- [コンポーネント](#)
- [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](#)

ボタンの表示ラベル

ボタンは、テキストのみ、画像とテキスト、またはテキストがない画像のみが表示されるように設計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、`ui: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。表示ラベルのテキストは支援技術では使用できますが、画面には表示されません。

```
<ui:button label="Search"  
iconImgSrc="/auraFW/resources/aura/images/search.png"/>
```

`ui:button` を使用する場合、空でない文字列を表示ラベル属性に割り当てます。次の例に、`ui:button` をどのように表示するかを示します。

```
<!-- Good: using alt attribute to provide a invisible label -->  
  
<button>  
  
      
  
</button>
```

```
<!-- Good: using span/assistiveText to hide the label visually, but show it to screen  
readers -->  
  
<button>  
  
    ::before  
  
    <span class="assistiveText">Search</span>
```

```
</button>
```

関連トピック:

[ボタン](#)

ヘルプとエラーメッセージ

`ui:inputDefaultError` コンポーネントを作成および処理するために入力コンポーネントを使用すると、エラーメッセージで `ariaDescribedby` 属性が自動的に取得されます。ただし、アクションを手動で管理する場合は、`ui:inputDefaultError` コンポーネントと関連出力を関連付ける必要があります。

コードの実行に失敗した場合は、`ariaDescribedby` が欠落しているかどうかを確認してください。コンポーネントは次の例のようになります。

```
<!-- Good: aria-describedby is used to associate error message -->

<label for="fname">Contact name</label>

<input name="" type="text" id="fname" aria-describedby="msgid">

<ul class="uiInputDefaultError" id="msgid">

  <li>Please enter the contact name</li>

</ul>
```

関連トピック:

[項目の検証](#)

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。

`type="file"` の場合を除き、`ui:input` を拡張する入力コンポーネントを使用します。たとえば、複数行のテキスト入力の場合は `<textarea>` タグではなく `ui:inputTextarea` を使用し、`<select>` タグではなく `ui:inputSelect` コンポーネントを使用します。

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。

```
<!-- 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。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。

イベントとの通信

第7章 イベント

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
- 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

JavaScript または Java Swing を使用した開発の経験があれば、イベント駆動型プログラムの概念を理解されていることと思います。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。イベントタイプは、`aura:event` タグの `type="COMPONENT"` または `type="APPLICATION"` のいずれかで設定されます。

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

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。

各アクション関数は、コントローラが属するコンポーネント、アクションで処理するイベント、ヘルパー(使用する場合)の3つのパラメータを取得します。クライアント側のコントローラは、名前-値ペアの対応付けが含まれる JSON オブジェクトであることを示すために角括弧と中括弧で囲まれます。

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、`componentNameController.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

HTML タグの実装が異なるイベントを見てみましょう。次のサンプルコンポーネントでは、3つの異なるボタンを作成しますが、3つのボタンのうち最後の2つのボタンのみが適切に機能します。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/>

    <input type="button" value="Hybrid HTML Button" onclick="{!!c.handleClick}"/>

    <br/>

    <ui:button label="Framework Button" press="{!!c.handleClick}"/>

    <br/>

    { !v.text}

</aura:component>
```

クライアント側コントローラのソース

```
{  
  
    handleClick : function(component, event) {  
  
        var attributeValue = component.get("v.text");  
  
        aura.log("current text: " + attributeValue);  
  
  
        var target;  
  
        if (event.getSource) {  
  
            // handling a framework component event  
  
            target = event.getSource(); // this is a Component object  
  
            component.set("v.text", target.get("v.label"));  
  
        } else {  
  
            // handling a native browser event  
  
            target = event.target.value; // this is a DOM element  
  
            component.set("v.text", event.target.value);  
  
        }  
  
    }  
  
}
```

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原因是ブラウザイベントのみです。コンポーネントの任意の JavaScript は無視されます。

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、このフレームワークには独自のイベントシステムがあるため、この「Flawed」ボタンは機能しません。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。

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。

コントローラの `handleClick` アクションを呼び出す「Hybrid」ボタンの `onclick` 属性を見ていきましょう。 「Framework」ボタンで使用される構文は、`press` 属性のある `<ui:button>` コンポーネントの構文と同じです。

この簡単なサンプルでは、「Framework」ボタンの操作と「Hybrid」HTML ボタンの操作に機能的な違いはほとんどありません。ただし、コンポーネントは、障害者や補助技術を使用するユーザもアプリケーションを使用できるように、アクセシビリティを考慮して設計されます。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ログ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。

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。

`component.get("v.attributeName")` では、`attributeName` 属性の値が返されます。`aura.log()` ユーティリティ関数は、ブラウザコンソールの検出を試みて、そこに属性値を記録します。

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、ヘルパー関数を使用して、`helper.someFunction(component)` で呼び出します。ヘルパリソースには、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。

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、コンポーネント自体、またはそのコンポーネントをインスタンス化するあるいは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。

カスタムコンポーネントイベントを作成する

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できます。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、`docsample:compEvent` コンポーネントイベントには `message` 属性が1つ設定されています。

```
<!--docsample: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。

コンポーネントイベントを登録する

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="docsample: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。

コンポーネントイベントを起動する

JavaScript でコンポーネントイベントへの参照を取得するには、`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");

// set some data for the event (also known as event shape)

// compEvent.setParams(...);

compEvent.fire();
```

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="docsample:compEvent"/>

<aura:handler name="sampleComponentEvent" 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。

インスタンス化されたコンポーネントのコンポーネントイベントを処理する

イベントを登録するコンポーネントは、イベントの `name` 属性を宣言します。たとえば、

`<docsample:eventsNotifier>` コンポーネントには、`<aura:registerEvent>` タグが含まれます。

```
<aura:registerEvent name="sampleComponentEvent" type="docsample:compEvent"/>
```

`<docsample:eventsNotifier>` を別のコンポーネントでインスタンス化するときは、`<aura:registerEvent>` タグの `name` 属性の値を使用してハンドラを登録します。たとえば、`<docsample:eventsHandler>` コンポーネントのマークアップに `<docsample:eventsNotifier>` が含まれる場合は、`eventsHandler` が `eventsNotifier` をインスタンス化し、`eventsNotifier` が生成したすべてのイベントを処理できます。`<docsample:eventsHandler>` が `<docsample:eventsNotifier>` をインスタンス化する方法を次に示します。

```
<docsample:eventsNotifier sampleComponentEvent="{'!c.handleComponentEventFired'}"/>
```

`sampleComponentEvent` は、`<docsample:eventsNotifier>` にある `<aura:registerEvent>` タグの `name` 属性の値と一致します。

コンポーネントイベントを動的に処理する

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。「[イベントハンドラの動的な追加](#)」(ページ 187)を参照してください。

コンポーネントイベントのソースを取得する

JavaScript の `evt.getSource()` を使用して、どのコンポーネントがコンポーネントイベントを起動したかを確認します。この `evt` はイベントへの参照です。

関連トピック:

[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。

この例のイベントおよびコンポーネントは、`docsample` 名前空間にあります。この名前空間は特別なものではありませんが、コードの数か所で参照されます。必要に応じて、別の名前空間を使用するようにコードを変更します。

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。

```
<!--docsample: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

`ceNotifier cmp` 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、`press` 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。

```
<!--docsample:ceNotifier-->

<aura:component>

  <aura:registerEvent name="cmpEvent" type="docsample:ceEvent"/>

  <h1>Simple Component Event Sample</h1>

  <p><ui:button
    label="Click here to fire a component event"
    press=" {!c.fireComponentEvent} " />
  </p>
```

```
</aura:component>
```

ceNotifierController.js

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。

```
{
    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

`ceHandler.cmp` ハンドラコンポーネントには、`<docsample:ceNotifier>` コンポーネントが含まれ、`<docsample:ceNotifier>` にある `<aura:registerEvent>` タグから、`name` 属性の値 `cmpEvent` を使用してハンドラを登録します。

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 `handleComponentEvent` アクションが呼び出されます。

```
<!--docsample:ceHandler-->
<aura:component>

    <aura:attribute name="messageFromEvent" type="String"/>
    <aura:attribute name="numEvents" type="Integer" default="0"/>

    <!-- handler contains the notifier component
    Note that cmpEvent is the value of the name attribute in aura:registerEvent
```

```
in ceNotifier.cmp -->

<docsample:ceNotifier cmpEvent="{!c.handleComponentEvent}" />

<p>{ !v.messageFromEvent }</p>
<p>Number of events: { !v.numEvents }</p>

</aura:component>
```

ceHandlerController.js

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。

```
{
  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

このコードをテストする場合は、リソースをサンプルアプリケーションに追加して、ハンドラコンポーネントに移動します。たとえば、docsample アプリケーションがある場合は、次のアドレスに移動します。

<http://<mySalesforceInstance>/<namespace>/docsample/ceHandler.cmp> (mySalesforceInstance は、na1.salesforce.com など、組織をホストするインスタンスの名前です)。

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。

カスタムアプリケーションイベントを作成する

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できます。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 `docsample:appEvent` アプリケーションイベントには `message` 属性が 1 つ設定されています。

```
<!--docsample: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を登録する

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="docsample:appEvent"/>
```

アプリケーションイベントを起動する

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。`fire()` を使用して、イベントを起動します。

```
var appEvent = $A.get("e.docsample:appEvent");

// set some data for the event (also known as event shape)

//appEvent.setParams({ ... });

appEvent.fire();
```

アプリケーションイベントを処理する

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。`<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:handler event="docsample:appEvent" action="{!!c.handleApplicationEvent}" />
```

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。

アプリケーションイベントのソースを取得する

`evt.getSource()` はアプリケーションイベントでは機能せず、コンポーネントイベントでのみ機能します。コンポーネントイベントは通常、`cmp.getEvent('myEvt').fire()` のようなコードによって起動されるため、誰がイベントを起動したかが明白です。他方、どのコンポーネントがアプリケーションイベントを起動したかはやや不明瞭です。アプリケーションイベントは、`$A.getEvt('myEvt').fire();` のようなコードによって起動されます。アプリケーションイベントのソースを確認する必要がある場合は、`evt.setParams()` を使用して、イベントを起動する前にイベントデータにソースコンポーネントを設定します。たとえば、`evt.setParams("source" : sourceCmp)` を使用します。この `sourceCmp` はソースコンポーネントへの参照です。

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。最後に、すべての表示が完了したこと

を示すために、aura:doneWaiting が起動されてから aura:doneRendering が起動されます。詳細は、「[表示ライフサイクル中に起動されたイベント](#)」(ページ 149) 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](#)
- [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?](#)

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。

この例のイベントおよびコンポーネントは、docsample 名前空間にあります。この名前空間は特別なものではありませんが、コードの数か所で参照されます。必要に応じて、別の名前空間を使用するようにコードを変更します。

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。

```
<!--docsample: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、`press` 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。

```
<!--docsample:aeNotifier-->

<aura:component>

    <aura:registerEvent name="appEvent" type="docsample:aeEvent"/>

    <h1>Simple Application Event Sample</h1>

    <p><ui:button

        label="Click here to fire an application event"

        press="{!!c.fireApplicationEvent}" />

    </p>

</aura:component>
```

aeNotifierController.js

クライアント側コントローラが、`$A.get("e.docsample: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。

```
{
    fireApplicationEvent : function(cmp, event) {
        // Get the application event by using the
        // e.<namespace>.<event> syntax
        var appEvent = $A.get("e.docsample: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。

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。

```
<!--docsample:aeHandler-->

<aura:component>

    <aura:attribute name="messageFromEvent" type="String"/>

    <aura:attribute name="numEvents" type="Integer" default="0"/>

    <aura:handler event="docsample:aeEvent" action="{!c.handleApplicationEvent}" />

    <p>{ !v.messageFromEvent }</p>
    <p>Number of events: { !v.numEvents }</p>
</aura:component>
```

aeHandlerController.js

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。

```
{
    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。

```
<!--docsample:aeContainer-->

<aura:component>

    <docsample:aeNotifier/>

    <docsample: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、リソースをサンプルアプリケーションに追加して、コンテナコンポーネントに移動します。たとえば、docsample アプリケーションがある場合は、次のアドレスに移動します。

<http://<mySalesforceInstance>/<namespace>/docsample/aeContainer.cmp> (mySalesforceInstance は、na1.salesforce.com など、組織をホストするインスタンスの名前です)。

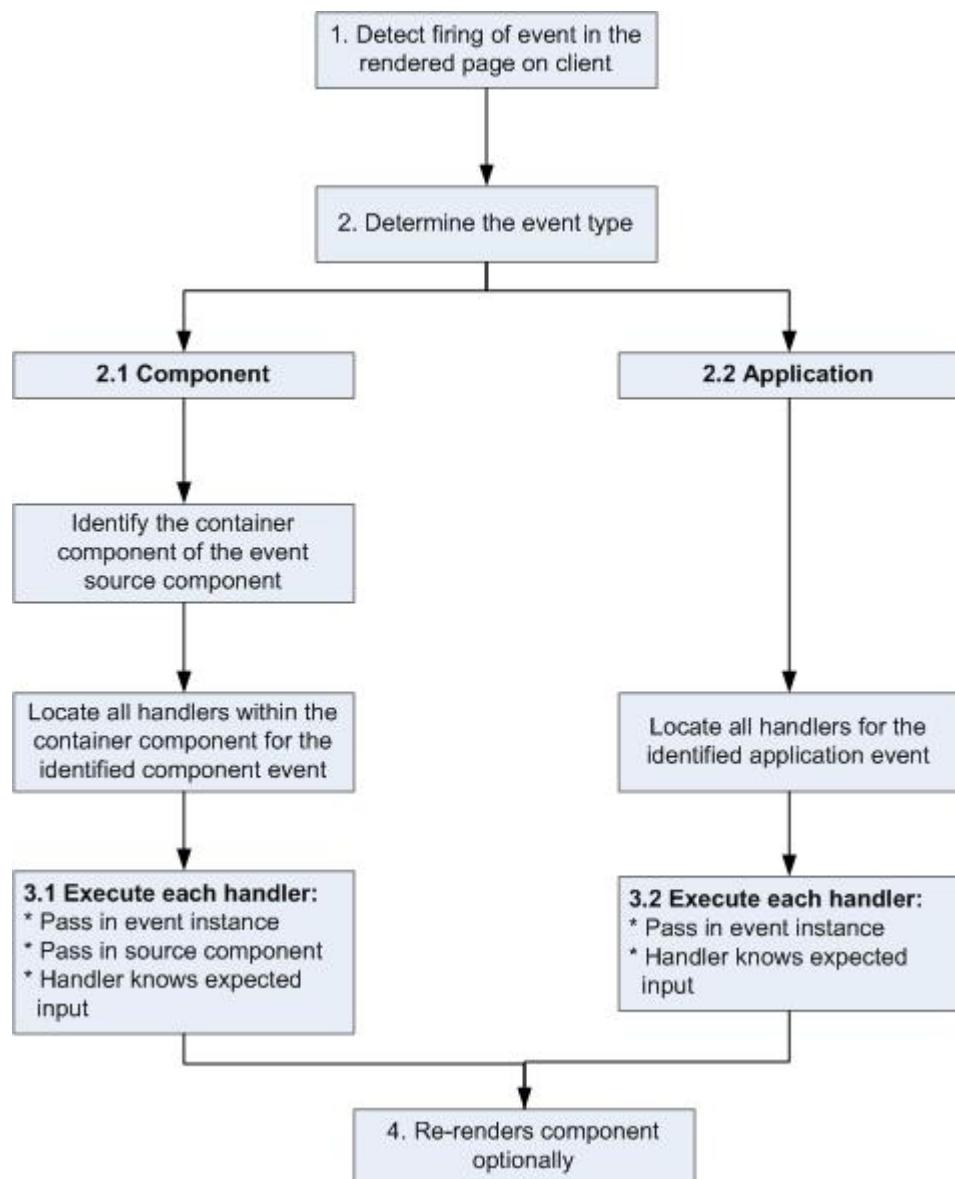
サーバ上のデータにアクセスする場合は、この例を拡張して、ハンドラのクライアント側コントローラからサーバ側コントローラをコールします。

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自動的に再表示することができます。

関連トピック:

[DOMへのクライアント側表示](#)

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。
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)が含まれます。

リソース	リソース名	使用方法
コンテナコンポーネント	eventsContainer.cmp	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。 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になります。

```
<!--docsample: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になります。

```
<!--docsample: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 press ブラウザイベントが含まれます。

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` 属性は必須ですが、アプリケーションイベントでは空のままにします。

`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。

```
<!--docsample:eventsNotifier-->

<aura:component>

    <aura:attribute name="parentName" type="String"/>

    <aura:registerEvent name="componentEventFired" type="docsample:compEvent"/>

    <aura:registerEvent name="appEvent" type="docsample: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><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 */

.docsampleEventsNotifier {

    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.docsample: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、ノーティファイアコンポーネントまたは `<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。

```
<!--docsample: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="docsample:appEvent" action="{!c.handleApplicationEventFired}" />




    <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>

        <docsample:eventsNotifier parentName="{!!v.name}" componentEventFired="{!!c.handleComponentEventFired}" />

    </div>

</aura:component>
```

CSS ソース

CSS は eventsHandler.css にあります。

```
/* eventsHandler.css */  
  
.docsampleEventsHandler {  
  
    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になります。

```
<!--docsample:eventsContainer-->  
  
<aura:component>  
  
    <docsample:eventsHandler name="eventsHandler1"/>  
  
    <docsample: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。

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ー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。

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 182)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:

- [DOMへのクライアント側表示](#)
- [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 の子を通して、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を作成します。

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

- 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 182)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、「DOMへのクライアント側表示」(ページ 169)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、`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。
5. フレームワークは、再表示が必要なコンポーネントがあるかどうか確認し、属性値の更新を反映する場合など、「汚れた」(更新が必要な) コンポーネントがあれば再表示します。この再表示確認は、汚れたコンポーネントや値が存在しない場合でも行われます。
6. 最後に、`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。

`ui:button` コンポーネントがサーバから返されたときに何が起こるかと、ボタンがクリックされてその表示ラベルが更新されたときに行われる再表示について見てみましょう。

```
<!-- The uiExamples:buttonExample container component -->

<aura:component>

    <aura:attribute name="num" type="Integer" default="0"/>

    <ui:button aura:id="button" label="{!v.num}" press=" {!c.update}"/>

</aura:component>

/** Client-side Controller **/


({
    update : function(cmp, evt) {
        cmp.set("v.num", cmp.get("v.num") + 1);
    }
})
```

 **メモ:** `ui:button` のソースを参照すると、表示されるコンポーネント定義を理解するのに役立ちます。 詳細は、<https://github.com/forcedotcom/aura/blob/master/aura-components/src/main/components/ui/button/button.cmp> を参照してください。

初期化の後、`render()` がコールされて `ui:button` が表示されます。`ui:button` にはカスタムレンダラはなく、`render()` という基本実装を使用します。この例では、`render()` が次の順序で 8 回コールされます。

コンポーネント	説明
uiExamples:buttonExample	ui:button コンポーネントが含まれる最上位コンポーネント
ui:button	最上位コンポーネントに含まれる ui:button コンポーネント
aura:html	<button> タグを表示します。
aura:if	ui:button の最初の aura:if タグ。ボタンには画像が含まれていないため、何も表示されません。
aura:if	ui:button の2つ目の aura:if タグ
aura:html	<button> タグにネストされた、ボタンの表示ラベル用の タグ
aura:expression	v.num 式
aura:expression	空の v.body 式

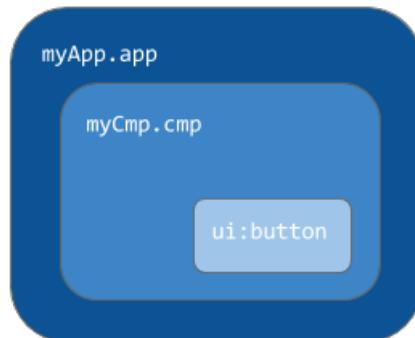
マークアップの HTML タグは、生成される HTML タグを定義する tag 属性が含まれた aura:html に変換されます。この例では、表示が完了すると、これらのコンポーネント定義について afterRender() が8回コールされます。doneWaiting イベントが起動され、その後に doneRendering イベントが起動されます。

ボタンをクリックするとその表示ラベルが更新され、更新が必要なコンポーネントがチェックされます。rerender() が起動されて、それらのコンポーネントが再表示され、その後に doneRendering イベントが起動されます。この例では、rerender() が8回コールされます。変更された値はすべて表示サービスのリストに保存され、更新が必要なコンポーネントがあれば再表示されます。

 **メモ:** カスタムレンダラでイベントを起動することはお勧めしません。詳細は、「[イベントのアンチパターン](#)」を参照してください。

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。`doneWaiting` イベントが同じ順序で起動されます。最後に、`doneRendering` イベントも同じ順序で起動されます。

関連トピック:

[DOMへのクライアント側表示](#)

[システムイベントの参照](#)

第8章

Salesforce1 イベント

Lightning コンポーネントは、イベントを介して Salesforce1 と連動します。

次のイベントを起動できます。これらは Salesforce1 によって自動的に処理されます。これらのイベントを Salesforce1 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は、必要に応じてイベントを処理する必要があります。

イベント名	説明
force:createRecord	指定した entityApiName (「Account」や「myNamespace__MyObject__c」など) の新しい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: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	メッセージをポップアップに表示します。

これらのイベントについての詳細は、「[イベントの参照](#)」(ページ 337) を参照してください。

Salesforce1 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 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ダ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 Salesforce1 イベントをインスタンス化すると、コンポーネントが Salesforce1 またはスタンドアロンアプリケーションのどちらで実行されているか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 およびスタンドアロンアプリケーション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 用にパラメータを設定し、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ことができます。

```
displayToast : function (component, event, helper) {  
  
    var toast = $A.get("e.force:showToast");  
  
    if (toast) {  
  
        //fire the toast event in Salesforce1  
  
        toast.setParams({  
  
            "title": "Success!",  
  
            "message": "The component loaded successfully."  
        });  
  
        toast.fire();  
  
    } else {  
  
        //your toast implementation for a standalone app here  
    }  
}
```

第9章 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 Salesforce 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またはルートコンポーネントの初期表示が完了したことを示します。
aura:doneWaiting	アプリケーションまたはコンポーネント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 <code>aura:waiting</code> イベントがあります。
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。
aura:systemError	エラーが発生したことを示します。
aura:valueChange	値が変更されたことを示します。
aura:valueDestroy	値が破棄処理中であることを示します。
aura:valueInit	値が初期化されたことを示します。
aura:waiting	アプリケーションまたはコンポーネントでサーバ要求への応答を待機していることを示します。

これらのイベントについての詳細は、「[システムイベントの参照](#)」(ページ352)を参照してください。

アプリケーションの作成

第 10 章 アプリケーションの基本

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
- 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。

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://<mySalesforceInstance>.lightning.force.com/<namespace>/<appName>.app`
(<mySalesforceInstance> は、na1 など、組織をホストするインスタンスの名前です)。

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、`<aura:application>` タグで開始する .app リソースにマークアップを挿入します。

「[スタンダードアロン Lightning アプリケーションを作成する](#)」で作成した `accounts.app` リソースを見てみましょう。

```
<aura:application>

  <h1>Accounts</h1>

  <div class="container">

    <!-- Other components or markup here -->

  </div>

</aura:application>
```

`accounts.app` には HTML タグとコンポーネントのマークアップが含まれます。`<div class="container">` のような HTML タグを使用して、アプリケーションのレイアウトを設計できます。

関連トピック:

[aura:application](#)

コンテンツセキュリティポリシーの概要

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。

[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 162)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。

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。

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。違反は次のように記録されます。

```
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。

第11章 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。

このセクションの内容:

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。

関連トピック:

[コンポーネント内の CSS](#)

[Salesforce1 への Lightning コンポーネントの追加](#)

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。フレームワークは現在、Visualforce で使用可能な \$Resource グローバル変数をサポートしていません。

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。

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「静的リソースとは?」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](#)

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;  
}
```

第 12 章 JavaScript の使用

トピック:

- DOM へのアクセス
- 外部 JavaScript ライブラリの使用
- JavaScript での属性値の操作
- JavaScript でのコンポーネントのボディの操作
- コンポーネントのバンドル内の JavaScript コードの共有
- DOM へのクライアント側表示
- 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
- 項目の検証
- エラーの発生および処理
- API コールの実行

クライアント側のコードには JavaScript を使用します。Aura オブジェクトは、JavaScript フレームワークのコードの最上位オブジェクトです。Aura クラスで使用できるすべてのメソッドについては、

<https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app> にある JavaScript API を参照してください (<mySalesforceInstance> は na1 など、組織をホストするインスタンスの名前です)。

\$A は、JavaScript コードでの Aura オブジェクトの短縮名です。

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。

DOM へのアクセス

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。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、DOM のコンポーネントを更新します。

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。

コンポーネントまたはアプリケーションから DOM にアクセスする場合、次の 2 つのガイドラインが非常に重要です。

- レンダラの外部で DOM を変更しないでください。ただし、レンダラの外部で DOM を読み取ることはできます。
- できる限り、直接 DOM 要素を設定せず、式を使用してください。

レンダラの使用

表示サービスは、DOM を更新するためのフレームワークのブリッジです。クライアント側のコントローラから DOM を変更する場合、コンポーネントのレンダラの動作によっては、コンポーネントを表示するときにその変更が上書きされる可能性があります。

式の使用

多くの場合、マークアップで式を使用すれば、カスタムレンダラを作成せずに済みます。

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。

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「静的リソースとは?」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。フレームワークは現在、Visualforce で使用可能な \$Resource グローバル変数をサポートしていません。

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。

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。

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

スクリプトが同一のコンポーネント内または異なるコンポーネント間の複数の`<ltng:require>` タグに指定されている場合、一度だけ読み込まれます。

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。

関連トピック:

[コンテンツセキュリティポリシーの概要](#)

[外部 CSS の使用](#)

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"/>
    <ui:button aura:id="button1" label="Button 1"/>
    { !v.buttonLabel }

    <ui:button label="Get Label" press="{!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");
```

Boolean 属性値を取得する

コンポーネントの `myString` 属性の `boolean` 値を取得するには、次のように記述します。

```
var myString = $A.util.getBooleanValue(cmp.get("v.myString"));
```

たとえば、次の属性が `$A.util.getBooleanValue()` に渡されると `true` が返されます。

```
<aura:attribute name="myString" type="String" default="my string"/>
```

属性が Boolean 型の場合、`cmp.get("v.myBoolean")` で `boolean` 値が返されるため、`$A.util.getBooleanValue()` は必要ありません。

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](#)

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 JSON オブジェクトであることを示すために角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。

ヘルパーの作成

ヘルパリソースは、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。

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

  }

})
```

ヘルパー・メソッドを使用してレンダラをカスタマイズする例については、[「DOMへのクライアント側表示」](#) を参照してください。

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。

```
({

    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。

```
({

    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);

    }

})
```

関連トピック:

- [DOMへのクライアント側表示](#)
- [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](#)

DOM へのクライアント側表示

このフレームワークの表示サービスでは、メモリ内のコンポーネントの状態を取得し、ドキュメントオブジェクトモデル(DOM)のコンポーネントを更新します。

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。

レンダラの外部で DOM を変更しないでください。ただし、レンダラの外部で DOM を読み取ることはできます。

表示ライフサイクル

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表示および再表示が自動的に処理されます。次に、表示ライフサイクルの概要を示します。

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. イベントでのデータ更新がすべて処理されると、フレームワークによって、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表示されます。

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。

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、`render()`、`rerender()`、`afterRender()`、`unrender()` 関数の基本実装があります。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の一部としてコールされます。このトピックでは、これらについて詳細に説明します。基本表示関数は、カスタムレンダラで上書きできます。

 **メモ:** 新しいコンポーネントを作成すると、フレームワークによって `init` イベントが起動されます。これにより、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。デフォルトのレンダラである `render()` では、コンポーネントのボディを取得し、表示サービスを使用して表示します。

レンダラの作成

通常、カスタムレンダラを作成する必要はありませんが、表示動作をカスタマイズする場合は、コンポーネントのバンドルにクライアント側のレンダラを作成できます。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。

別のコンポーネントのレンダラを再利用するには、代わりに `aura:component` の `renderer` システム属性を使用します。たとえば、次のコンポーネントでは、

`auradocs/sampleComponent/sampleComponentRenderer.js` の `auradocs.sampleComponent` の自動的に結び付けられたレンダラが使用されます。

```
<aura:component  
    renderer="js://auradocs.sampleComponent">  
  
    ...  
  
</aura:component>
```

-  **メモ:** コンポーネントのバンドルの自動的に結び付けられたレンダラがすでに存在する場合に、別のコンポーネントのレンダラを再利用すると、自動的に結び付けられたレンダラのメソッドにアクセスできなくなります。保守性を確保するためにコンポーネントのバンドル内のレンダラを使用し、必要な場合にのみ外部レンダラを使用することをお勧めします。

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。

通常、`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。

-  **メモ:** 表示をカスタマイズする場合、次のガイドラインが非常に重要です。

- レンダラでは、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。
- レンダラで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。

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。データを更新していないが、コンポーネントを再表示する場合は、`rerender()` を明示的にコールします。

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エーニングされます。

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。

ライブラリを使用して DOM にアクセスする場合は、`afterRender()` 内で使用します。

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。

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。これは、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。

確実なクライアント側での表示

デフォルトでは、フレームワークによってサーバ側のデフォルトレンダラがコールされます。クライアント側のレンダラがある場合はそのレンダラがコールされます。最上位コンポーネントを確実にクライアント側で表示するには、`render="client"` を `aura:component` タグに追加します。最上位コンポーネントでこれを設定すると、フレームワークの検出ロジックよりも優先され、運動関係が考慮されます。これは、直接ブラウザでコンポーネントをテストし、テストを読み込むときにクライアント側のフレームワークを使用してコンポーネントを調査する場合に特に便利です。通常は必要ありませんが、テストコンポーネントで `render="client"` を設定すると、クライアント側のフレームワークが確実に読み込まれるようになります。

表示の例

ボタンコンポーネントを見て、基本的な表示動作をカスタマイズする方法を確認しましょう。マークアップの各タグ（標準 HTML タグを含む）に基盤となるコンポーネント表現があることを理解しておくことが重要です。この基盤のコンポーネント表現があるため、フレームワークの表示サービスでは、作成する標準 HTML タグまたはカスタムコンポーネントが同じプロセスを使用して表示されます。

`ui:button` のソースを表示します。ボタンコンポーネントには、`Boolean` でコンポーネントの無効化の状況を追跡する `disabled` 属性があります。

```
<aura:attribute name="disabled" type="Boolean" default="false"/>
```

`button.cmp` では、`onclick` は `{!c.press}` に設定されます。

ボタンコンポーネントのレンダラは、`buttonRenderer.js` です。ボタンコンポーネントにより、デフォルトの `render()` 関数が上書きされます。

```
render : function(cmp, helper) {  
  
    var ret = this.superRender();  
  
    helper.updateDisabled(cmp);  
  
    return ret;  
  
},
```

最初の行で `superRender()` 関数がコールされ、デフォルトの表示動作が呼び出されます。

`helper.updateDisabled(cmp)` コールにより、ヘルパー関数が呼び出され、表示がカスタマイズされます。

`buttonHelper.js` の `updateDisabled(cmp)` 関数を見ていきましょう。

```
updateDisabled: function(cmp) {  
  
    if (cmp.get("v.disabled")) {  
  
        var disabled = $A.util.getBooleanValue(cmp.get("v.disabled"));  
  
        var button = cmp.find("button");  
  
        if (button) {  
  
            var element = button.getElement();  
  
            if (element) {  
  
                if (disabled) {  
  
                    element.setAttribute('disabled', 'disabled');  
  
                } else {  
  
                    element.removeAttribute('disabled');  
  
                }  
  
            }  
  
        }  
  
    }  
  
}
```

`updateDisabled(cmp)` 関数では、Boolean `disabled` 値が HTML で想定される値に変換されます。ここでは、属性は存在していないか、`disabled` に設定されます。

`cmp.find("button")` を使用して、一意のコンポーネントを取得します。`button.cmp` では、`aura:id="button"` を使用して、コンポーネントを一意に識別します。`button.getElement()` は、DOM 要素を返します。

`buttonRenderer.js` の `rerender()` 関数は、`render()` 関数に非常に似ています。この関数で、`updateDisabled(cmp)` もコールします。

```
rerender : function(cmp, helper) {
    this.superRerender();
    helper.updateDisabled(cmp);
}
```

コンポーネントの表示は、フレームワークのライフサイクルの一部ですが、他の概念よりも若干説明しづらい概念です。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、この概念について考える必要はありません。

関連トピック:

[DOMへのアクセス](#)

[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](#)

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

`$A.run()` を使用して、通常の表示ライフサイクル外のコンポーネントを変更するコードをラップします。`$A.run()` 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。

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.run()` 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、クライアント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.run()` 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(function () {
    $A.run(function() {
        if (cmp.isValid()) {
            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.run()` 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:** タイムアウトなど非同期コードのコンポーネントを参照する場合は、常に `isValid()` チェックを追加します。

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](#)

[非 Lightning コードからの Lightning イベントの起動](#)

[イベント](#)

項目の検証

JavaScript を使用して、項目を検証できます。通常、ユーザ入力を検証してエラーを特定し、エラーメッセージを表示します。

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。次の例に、フレームワークで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。

コンポーネントのソース

```
<aura:component>  
  
    Enter a number: <ui:inputNumber aura:id="inputCmp"/> <br/>  
  
    <ui:button label="Submit" press=" {!c.doAction} "/>  
  
</aura:component>
```

クライアント側コントローラのソース

```
{  
  
    doAction : function(component) {  
  
        var inputCmp = component.find("inputCmp");  
  
        var value = inputCmp.get("v.value");  
    }  
}
```

```
// Is input numeric?  
  
if (isNaN(value)) {  
  
    // Set error  
  
    inputCmp.setValid("v.value", false);  
  
    inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);  
  
} else {  
  
    // Clear error  
  
    inputCmp.setValid("v.value", true);  
  
}  
  
}  
}
```

値を入力して[送信]ボタンをクリックすると、コントローラの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に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。コントローラでは、`setValid(false)` を使用して入力値を無効化し、`setValid(true)` を使用してエラーをクリアします。`addError()` を使用して、エラーメッセージを入力値に追加できます。

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` 属性を使用してエラーを処理できます。これらの属性は、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。入力コンポーネントは、`ui:updateError` イベントを使用して、デフォルトのエラーコンポーネント `ui:inputDefaultError` を更新できます。

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。

コンポーネントのソース

```
<aura:component>  
  
    Enter a number: <ui:inputNumber aura:id="inputCmp" onError="{!!c.handleError}"  
    onClearErrors="{!!c.handleError}"/> <br/>  
  
    <ui:button label="Submit" press="{!!c.doAction}"/>  
  
</aura:component>
```

クライアント側コントローラのソース

```
{  
  
    doAction : function(component, event) {  
  
        var inputCmp = component.find("inputCmp");  
  
        var value = inputCmp.get("v.value");  
  
        // is input numeric?  
  
        if (isNaN(value)) {  
  
            // fire event that will set error  
  
            var errorEvent = inputCmp.getEvent("onError");  
  
            errorEvent.setParams({ "errors" : [{message:"Input not a number: " + value}] });  
  
            errorEvent.fire();  
  
        } else {  
  
            // fire event that will clear error  
  
            var clearErrorEvent = inputCmp.getEvent("onClearErrors");  
  
            clearErrorEvent.fire();  
  
        }  
  
    },  
  
    handleError: function(component, event){  
  
        var inputCmp = component.find("inputCmp");  
  
        var errorsObj = event.getParam("errors");  
  
        /* do any custom error handling  
         * logic desired here */  
  
        // set error using default error component
```

```
        inputCmp.setValid("v.value", false);

        inputCmp.addErrors(errorsObj);

        var updateErrorEvent = inputCmp.getEvent("updateError");

        updateErrorEvent.fire();

    },

handleClearError: function(component, event) {

    var inputCmp = component.find("inputCmp");

    /* do any custom error handling
     * logic desired here */

    // clear error using default error component

    inputCmp.setValid("v.value", true);

    var updateErrorEvent = inputCmp.getEvent("updateError");

    updateErrorEvent.fire();

}

}
```

値を入力して[送信]ボタンをクリックすると、コントローラの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、コンポーネントの `onError` 属性を使用して、カスタムエラーハンドラを提供する必要があります。検証に失敗すると、`doAction` が `setParams()` を使用してエラーメッセージを追加し、カスタムエラーハンドラを起動します。カスタムイベントハンドラ `handleError` で、`getParam()` をコールしてエラーを取得し、`setValid(false)` を使用して入力値を無効化します。`updateError` イベントを起動して、デフォルトのエラーコンポーネントを更新できます。

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 コードで柔軟に対応できます。たとえば、`action.setCallback()` を使用して、サーバ側の応答の処理時にこれらのエラーを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、`$A.error("error message here")` を使用します。これには、ページのスタック追跡が表示されます。

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エラーの発生およびキャッチを示します。

```
<!--docsample:recoverableError-->

<aura:component>

    <p>Click the button to trigger the controller to throw an error.</p>
    <div aura:id="div1"></div>

    <ui:button label="Throw an Error" press="{!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.createComponent([
            "ui:message", {
                "title" : "Sample Thrown Error",
                "severity" : "error",
            }],
            [
                "ui:outputText", {
                    "value" : e.message
                }]
            ],
            function(components, status) {
                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);
                }
            }
        );
    }
})
```

コントローラコードが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](#)

API コールの実行

API コールはクライアント側のコードから行うことはできません。代わりに、Salesforce API コールなどの API コールはサーバ側のコントローラから行います。

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。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るため、CSP では JavaScript コードからの API コールは許可されません。

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 206) 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](#)

第 13 章 JavaScript Cookbook

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
- データ変更の検出
- ID によるコンポーネントの検索
- コンポーネントの動的な作成
- イベントハンドラーの動的な追加
- マークアップの動的な表示または非表示
- スタイルの追加と削除

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。

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](#)

[DOMへのクライアント側表示](#)

[コンポーネントの属性](#)

[データ変更の検出](#)

データ変更の検出

自動的にイベントを起動する

コンポーネントのいずれかの属性値が変更されたときにクライアント側のコントローラアクションを自動的に呼び出すようにコンポーネントを設定できます。値が変更されると、`valueChange.evt` イベントが自動的に起動します。`valueChange.evt` は、`value` と `index` の2つの属性を取得する `type="VALUE"` のイベントです。

手動でイベントを起動する

別のコンポーネントおよびアプリケーションイベントは、クライアント側のコントローラの `event.fire()` で手動で起動します。

たとえば、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.items}" action=" {!c.itemsChange}" />
```

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) {
        var v = evt.getParam("value");
        if (v === cmp.get("v.items")) {
            //do something
        }
    }
})
```

`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。詳細は、「[aura:valueChange](#)」(ページ 356)を参照してください。

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](#)

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。たとえば、`ui:button` コンポーネントに `button1` というローカル ID を追加できます。

```
<ui: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。

また、実行時に生成されるグローバル ID によってもコンポーネントを取得できます。

```
var comp = $A.getCmp(globalId);
```

たとえば、`ui:button` コンポーネントは、次のマークアップにより HTML ボタン要素として表示されます。

```
<button class="default uiButton" data-aura-rendered-by="30:463;a">...</button>
```

`$A.getCmp("30:463;a")` を使用してコンポーネントを取得します。

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。

 **メモ:** 廃止された `newComponent()`、`newComponentAsync()`、`newComponentDeprecated()` メソッドの代わりに、`createComponent()` メソッドを使用します。

構文は次のとおりです。

```
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。
3. `callback` — コンポーネントの作成後に呼び出すコールバック。新しいコンポーネントがパラメータとしてコールバックに渡されます。

すべてのパラメータについての詳細は、「[リファレンスドキュメントアプリケーション](#)」のJavaScript API リファレンスを参照してください。

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。

```
<!--docsample: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、press イベントのローカルIDとハンドラを含むボタンを作成します。docsample:createComponent の body にボタンが追加されます。

```
/*createComponentController.js*/
({
    doInit : function(cmp) {
        $A.createComponent(
            "ui:button",
            {
                "aura:id": "findableAuraId",
                "label": "Press Me",
                "press": cmp.getReference("c.handlePress")
            },
            function(newButton) {
                //Add the new button to the body array
                if (cmp.isValid()) {
                    var body = cmp.get("v.body");
                    body.push(newButton);
                    cmp.set("v.body", body);
                }
            }
        );
    },
    handlePress : function(cmp) {
        console.log("button pressed");
    }
})
```

 **メモ:** docsample: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。

作成した新しいボタンを取得するには、body[0] を使用します。

```
var newbody = cmp.get("v.body");
var newCmp = newbody[0].find("findableAuraId");
```

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.createComponent()` 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.createComponent([
    ["ui:message", {
        "title" : "Sample Thrown Error",
        "severity" : "error",
    }],
    ["ui:outputText", {
        "value" : e.message
    }]
],
function(components, status){
    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);
    }
});
});
```

連動関係の宣言

このフレームワークでは、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。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。これにより、必要に応じて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。

使用方法についての詳細は、「[「aura:dependency」](#) (ページ 226) を参照してください。

サーバ側の連動関係

`createComponent()` 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。サーバ側の連動関係が見つからない場合、このメソッドは同期して実行されます。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。

 **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。

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。

サーバ側の連動関係があるコンポーネントは、事前に読み込まれている場合でも、そのサーバで作成されます。サーバ側の連動関係がなく、連動関係の事前の読み込みまたは宣言によってその定義がすでにクライアン

トに存在している場合、サーバコールは実行されません。サーバ要求を強制するには、`forceServer` パラメータを `true` に設定します。

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](#)

[イベントハンドラの動的な追加](#)

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。コンポーネントは、クライアント側で動的に作成することも、実行時にサーバから取得することもできます。

次のサンプルコードでは、イベントハンドラを `docsample:sampleComponent` のインスタンスに追加します。

```
addNewHandler : function(cmp, event) {  
  
    var cmpArr = cmp.find({ instancesOf : "docsample:sampleComponent" });  
  
    for (var i = 0; i < cmpArr.length; i++) {  
  
        var outputCmpArr = cmpArr[i];  
  
        outputCmpArr.addHandler("someAction", cmp, "c.someAction");  
  
    }  
  
}
```

`$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。

`addHandler()` により、イベントハンドラがコンポーネントに追加されます。

コンポーネントから起動しないイベントを、強制的に起動することはできません。`c.someAction` は、コンポーネントの階層の1つのコントローラに含まれるアクションです。`someAction` はイベント名、`cmp` は値の取得元を参照します。`someAction` は、`aura:registerEvent` タグや `aura:handler` タグの `name` 属性の値と一致している必要があります。メソッドおよび引数の完全なリストについては、JavaScript API リファレンス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](#)

[コントローラのサーバ側ロジックの作成](#)

[DOMへのクライアント側表示](#)

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を切り替えるときは CSS を使用します。 `<aura:if>` または `<aura:renderIf>` タグを使用することもできますが、より標準的なアプローチである CSS の使用をお勧めします。

次の例では、`$A.util.toggleClass(element, 'class')` を使用してマークアップの表示を切り替えます。

```
<!--docsample:toggleCss-->

<aura:component>

    <ui:button label="Toggle" press=" {!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

実行時に要素の CSS スタイルを追加または削除できます。

次のデモでは、要素の CSS スタイルを追加および削除する方法を示します。

コンポーネントのソース

```
<aura:component>

    <div aura:id="changeIt">Change Me!</div><br />

    <ui:button press="{!c.applyCSS}" label="Add Style" />

    <ui:button press="{!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 el = cmp.find('changeIt');

        $A.util.addClass(el.getElement(), 'changeMe');

    },
    removeCSS: function(cmp, event) {
        var el = cmp.find('changeIt');

        $A.util.removeClass(el.getElement(), '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 スタイルを要素に追加するには、`$A.util.addClass(element, 'class')` を使用します。同様に、クラスを削除するには、コントローラで `$A.util.removeClass(element, 'class')` を使用します。`cmp.find()` でローカル ID(このデモでは `aura:id="changeIt"`) を使用して要素を特定します。

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(element, 'class')` を使用します。`element` パラメータは、HTML 要素またはコンポーネントの場合があります。

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 188)を参照してください。

要素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 element in arr) {  
  
        $A.util.toggleClass(arr[element], cssClass);  
  
    }  
  
}
```

 **メモ:** `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。このような場合は、HTML 要素よりも `cmp` コンポーネントを渡すことをお勧めします。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 149)を参照してください。

DOM 要素を操作するその他のユーティリティ関数については、JavaScript API リファレンス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](#)

[コンポーネント内の CSS](#)

[コンポーネントのバンドル](#)

第 14 章 Apex の使用

トピック:

- コントローラのサーバ側ロジックの作成
- コンポーネントの操作
- Salesforce レコードの操作
- Apex コードのテスト
- Apex からの API コールの実行

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。

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 194)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、クライアント側からもサーバ側からも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。

[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バック](#)

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バックに配置します。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。

[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バックに入っているときや、サーバからまだ返されていない場合に中止可能にすることができます。これは、キューバックにより新しい中止可能アクションがあるときにアクションを中止する場合に便利です。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。

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、クライアント側からもサーバ側からも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。

 **ヒント:** コントローラ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属性を保存します。

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 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);  
  
    }  
  
}
```

関連トピック:

[サーバ側のアクションのコード](#)

[Apex サーバ側コントローラの作成](#)

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. あなたの名前>[開発者コンソール]をクリックします。
2. [ファイル]>[新規]>[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。
 **メモ:** getter や setter などクライアント側またはサーバ側で公開するメソッドに `@AuraEnabled` アノテーションを付加します。つまり、アノテーションを明示的に付加したメソッドのみが公開されます。
6. [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" >
```

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。

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"/>

    <ui:button label="Call server" press="{!!c.echo}" />

</aura:component>
```

次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。クライアント側コントローラに、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。この場合、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。action.setParams({ firstName : cmp.get("v.firstName") }); は、コンポーネントから firstName 属性を取得して、サーバ側コントローラの serverEcho メソッドに firstName 引数の値を設定します。

```
({

    "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();

            // This callback doesn't reference cmp. If it did,
```

```
// you should run an isValid() check

//if (cmp.isValid() && state === "SUCCESS") {

  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 (cmp.isValid() && state === "ERROR") {

    else if (state === "ERROR") {

      var errors = response.getError();

      if (errors) {

        $A.logf("Errors", errors);

        if (errors[0] && errors[0].message) {

          $A.error("Error message: " +

                    errors[0].message);

        }

      } else {

        $A.error("Unknown error");

      }

    }

  });

}

// optionally set abortable 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。この構文は、クライアント側コントローラのアクションを呼び出すためにマークアップで使用するものと同じです。`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。

`$A.enqueueAction(action)` は、サーバ側コントローラのアクションを、実行されるアクションのキューに追加します。この方法で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、関連する要求をバッチにまとめてからキューのアクションを実行します。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ています。

 **メモ:** コールバックなど非同期コードのコンポーネントを参照する場合は、常に `isValid()` チェックを追加します。

アクションの状態

アクションの有効な状態は次のとおりです。

NEW (新規)

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 (実行中)

アクションを処理中です。

SUCCESS (成功)

アクションが正常に実行されました。

ERROR (エラー)

サーバからエラーが返されました。

ABORTED (中止)

アクションが中止されました。

 **メモ:** `setCallback()` には、コールバックを呼び出すアクション状態を登録する 3 つ目のパラメータがあります。`setCallback()` に 3 つ目の引数を指定しないと、デフォルトで `SUCCESS` および `ERROR` 状態

が登録されます。ABORTED など別の状態のコールバックを設定するには、3つ目の引数でアクションの状態を明示的に設定した `setCallback()` を複数回コールできます。次に例を示します。

```
action.setCallback(this, function(response) { ...}, "ABORTED");
```

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](#)

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。

イベントの処理でイベントハンドラがさらなるイベントを起動する場合は、イベントツリーを生成できます。フレームワークがイベントツリーを処理し、サーバで実行する必要のある各アクションをキューに追加します。

イベントツリーおよびすべてのクライアント側のアクションが処理されると、キューのアクションが1つのメッセージにまとめられ、そのメッセージがサーバに送信されます。メッセージは実質的にアクションのリストを囲むラッパーです。

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や、サーバからまだ返されていない場合に中止可能にすることができます。これは、キューにより新しい中止可能アクションがあるときにアクションを中止する場合に便利です。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。

1つのトランザクションのためのアクションのセット(クリックコールバックなど)はまとめてサーバへの送信キューに入れられます。ユーザが別のナビゲーション項目をクリックするなどして、別のトランザクションを開始した場合、すべての中止可能なアクションはキューから削除されます。中止されたアクションはサーバには送信されず、その状態は `ABORTED` に設定されます。

中止可能なアクションは、通常はサーバに送信されて実行されます(ただし、その後で中止可能なアクションがキューに追加されたときにサーバからまだ返されていない場合を除きます)。

一部のアクションがサーバに送信され、まだ返されていない場合、そのアクションは完了しますが、実行されるのは `ABORTED` 状態(`action.getState() === "ABORTED"`)に関連付けられたコールバックロジックのみです。これにより、コンポーネントは必要に応じてメッセージをログに記録するか、中止されたアクションがある場合はクリーンアップ処理を実行できます。



メモ: 最新の中止可能アクションが以前の中止可能アクションと同一である必要はありません。必要なのは、最新のアクションが中止可能とマークされていることだけです。

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();
```

`setCallback()` には、コールバックを呼び出すアクション状態を登録する 3 つ目のパラメータがあります。`setCallback()` に 3 つ目の引数を指定しないと、デフォルトで `SUCCESS` および `ERROR` 状態が登録されます。コールバックで中止されたアクションがあるかどうかを確認し、中止されたアクションをログに記録するなど、適切なアクションを実行するには、3 つ目の引数に `ABORTED` 状態を明示的に設定し、`setCallback()` をコールします。次に例を示します。

```
// Process default action states

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());

    }

    // process other action states
});

// Explicitly register callback for ABORTED

action.setCallback(this,

    function(response) {

        alert("The action was aborted");

    },

    "ABORTED"
);
```

急速な連続クリック

ナビゲーションメニューの各アクションからサーバへの要求が遅い場合を考えてみましょう。ユーザがナビゲーション項目を立て続けに何回もクリックして、後続のクリックの前にサーバから応答が戻らなかったとします。すべてのアクションが中止可能とマークされている場合、最後のクリックを除き、コールバックは一切コールされません。これにより、複数のサーバ応答を連続して表示したために発生する画面のちらつきがなくなり、ユーザの操作性が向上します。

段階的な読み込み

最初の項目セットを読み込み、最初のセットの表示が完了した後に後続のデータを読み込む、というようにデータの段階的な読み込みが必要になる場合があります。これを行うには、2回目の一連のアクションを遅延させてコールし、Action オブジェクトの `setParentAction()` メソッドを使用して、2回目のセットの各アクションを最初のセットのいずれかのアクションに関連付けます。これにより、ユーザが他の画面に移動した場合、2回目のアクションのセットは中止されます。

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アクション](#)
- [サーバ側のアクションのコール](#)

コンポーネントの操作

Apex では、シンプルなドット表記を使用してコンポーネントとコンポーネントの属性を操作します。

コンポーネントを参照するには、`Cmp.<myNamespace>.<myComponent>` を使用します(例: `Cmp.ui.button`)。

コンポーネントの属性を参照するには、`Cmp.<myNamespace>.<myComponent>.<myAttribute>` を使用します(例: `Cmp.ui.button.label`)。

Apex でのコンポーネントの作成

コンポーネントを作成するには、次の構文を使用します。

```
Cmp.<myNamespace>.<myComponent> cmpVar = new Cmp.<myNamespace>.<myComponent>();
```

次に例を示します。

```
Cmp.ui.button button = new Cmp.ui.button();
```

コンポーネントを作成するときに、属性を含めることもできます。次に例を示します。

```
Cmp.ui.button button = new Cmp.ui.button(label = 'Click Me');
```

コンポーネントの属性の更新

コンポーネントの属性値を更新するには、新しい値を割り当てます。次に例を示します。

```
Cmp.ui.button button = new Cmp.ui.button(label = 'Click Me');

String buttonLabel = button.label;

button.label = 'Click Me Not';
```

現在のコンポーネントへのアクセス

コンポーネントのコントローラで現在のコンポーネントにアクセスするには、`Aura.getComponent()` を使用します。たとえば、ボタンのコントローラでは、次のようにボタンコンポーネントにアクセスできます。

```
Cmp.ui.button button = Aura.getComponent();
```

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 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;

        update acct;
```

```
    }  
}
```

JavaScript から Apex コードをコールする例については、「[ケイックスタート](#)」(ページ 7)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 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  
  
        ];  
  
        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 194) 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 class MyObjController {
    @AuraEnabled
    public static List<MyObj__c> getMyObjects() {
        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", a.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」\(ページ 7\)](#)を参照してください。

レコードデータの保存

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。たとえば、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>
    <ui:button label="Edit Record" press="{!!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。

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) {
```

```
    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、`<ui:button label="Submit" press=" {!c.createExpense} "/>` と指定します。
- 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インスタンスを取得して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。

```
upsertExpense : function(component, expense, callback) {  
  
    var action = component.get("c.saveExpense");  
  
    action.setParams({  
  
        "expense": expense  
  
    });  
  
    if (callback) {  
  
        action.setCallback(this, callback);  
  
    }  
  
    $A.enqueueAction(action);  
  
}
```

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。
 - 本番組織にリリースするときに、組織の名前空間内のすべての単体テストが実行されます。
 - `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・誤両方の場合や单一データだけ

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、「[スタンダードアロン Lightning アプリケーションを作成する](#)」(ページ 9)で入手可能な経費追跡アプリケーションでコントローラークラスと共に使用される Apex テストクラスを示しています。

```
@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.getExpense(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 API コールを行うことはできません。

Apex からの API コールの実行については、[『Force.com Apex コード開発者ガイド』](#) を参照してください。

第15章 オブジェクト指向開発の使用

トピック:

- 継承とは?
- 継承されるコンポーネントの属性
- 抽象コンポーネント
- インターフェース
- 継承ルール

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。

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](#)

[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。`docsample:super` には、値が「Default description」の `description` 属性があります。

```
<!--docsample: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。

`docsample:sub` は、`<aura:component>` タグで `extends="docsample:super"` を設定することによって `docsample:super` を拡張します。

```
<!--docsample:sub-->

<aura:component extends="docsample: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。

`docsample: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。

```
<!--docsample:superBody-->

<aura:component extensible="true">

    Parent body: {!v.body}

</aura:component>
```

この時点では、`{!v.body}` が `docsample: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、`docsample:superBody` では何も出力されません。

`docsample:subBody` は、`<aura:component>` タグで `extends="docsample:superBody"` を設定することによって `docsample:superBody` を拡張します。

```
<!--docsample:subBody-->

<aura:component extends="docsample:superBody">

    Child body: {!v.body}

</aura:component>
```

`docsample:subBody` の出力は、次のようにになります。

```
Parent body: Child body:
```

つまり、`docsample:subBody` は、`{!v.body}` の値をスーパーコンポーネント `docsample:superBody` で設定します。

`docsample:containerBody` には、`docsample:subBody` への参照が含まれます。

```
<!--docsample:containerBody-->

<aura:component>

    <docsample:subBody>

        Body value

    </docsample:subBody>
```

```
</aura:component>
```

docsample:containerBody で、docsample:subBody の body 属性を Body value に設定します。docsample: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態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。

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。

インターフェースは、`<aura:interface>` タグで始まります。インターフェースには、インターフェースの属性を定義する `<aura:attribute>` タグのみを含めることができます。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:cpl1" 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。これは、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する必要があります。

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
<code>component</code>	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	<code><aura:component></code>
<code>app</code>	拡張可能な 1 つのアプリケーション	N/A	<code><aura:application></code>

要素	拡張	実装	デフォルトの基本要素
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](#)

第 16 章

AppCache の使用

トピック:

- AppCache の有効化
- AppCache を使用したリソースの読み込み

アプリケーションキャッシュ (AppCache) は、変更されたリソースのみをダウンロードすることで、アプリケーションの応答時間を短縮し、サーバ負荷を軽減します。デバイスによっては、ブラウザキャッシュの保持制限に影響を受けるページ読み込みが改善されます。

AppCache は、ブラウザキャッシュが非常に小さい場合がある、モバイルデバイスのアプリケーションを開発する場合に便利です。デスクトップクライアント用に作成されたアプリケーションでは AppCache によるメリットが得られない場合があります。このフレームワークでは、Chrome や Safari など、WebKit ベースのブラウザで AppCache がサポートされます。

 **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](#)

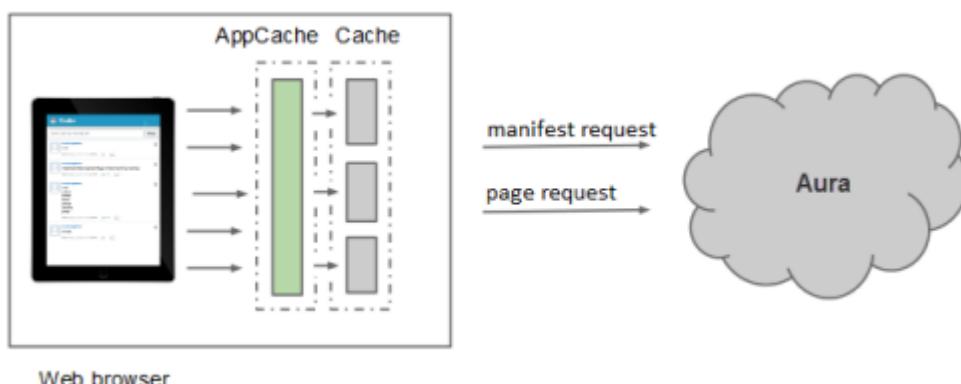
AppCache の有効化

このフレームワークでは、デフォルトで AppCache の使用が無効化されています。

アプリケーションで AppCache を有効化するには、aura:application タグに `useAppcache="true"` システム属性を設定します。開発初期の、まだアプリケーションのリソースが変化している間は、AppCacheを無効にすることをお勧めします。アプリケーションの開発が完了したら、本番環境で使用を開始する前に AppCache を有効にして、AppCache によってアプリケーションの応答時間が改善するかどうかを確認します。

AppCache を使用したリソースの読み込み

キャッシュマニフェストファイルは、AppCache での Web リソースのオフラインキャッシングを定義する単純なテキストファイルです。



Web browser

アプリケーションで AppCache を有効化している場合、キャッシュマニフェストは実行時に自動生成されます。リソースに何らかの変更があると、フレームワークはタイムスタンプを更新して、すべてのリソースの再取得をトリガします。必要な場合にのみリソースを取得すると、サーバとの往復が削減されます。

ブラウザが初めてアプリケーションを要求するとき、応答にマニフェストファイルへのリンクが含まれます。

```
<html manifest="/path/to/app.manifest">
```

マニフェストパスには、現在実行中のアプリケーションのモードとアプリケーション名が含まれます。このマニフェストファイルには、JavaScript コードと CSS だけでなくフレームワークリソースが表示されます。これらは、初めてダウンロードした後にキャッシングされます。URL のハッシュにより、常に最新のリソースを取得できます。

- **メモ:** 表示されるリソースは、実行モードによって異なります。たとえば、`aura_prod.js` は PROD モードで使用でき、`aura_proddbug.js` は PRODDEBUG モードで使用できます。

第17章 アクセスの制御

トピック:

- アプリケーションのアクセス制御
- インターフェースのアクセス制御
- コンポーネントのアクセス制御
- 属性のアクセス制御
- イベントのアクセス制御

このフレームワークでは、これらのタグの `access` 属性を介してアプリケーション、インターフェース、コンポーネント、属性、およびイベントへのアクセスを制御できます。この属性は、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。

タグ	説明
<code>aura:application</code>	アプリケーションを表す
<code>aura:interface</code>	インターフェースを表す
<code>aura:component</code>	コンポーネントを表す
<code>aura:attribute</code>	アプリケーション、インターフェース、コンポーネント、またはイベントの属性を表す
<code>aura:event</code>	イベントを表す

デフォルトでは、すべてのタグで `access` 属性は `public` に設定されているため、リソースは同じ名前空間内で拡張または使用できます。

`access="global"` を設定すると、パッケージの登録者と他の名前空間でバンドルを使用できるようになります。パッケージについての詳細は、「[「アプリケーションとコンポーネントの配布」](#) (ページ 219) を参照してください。

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に拡張できるかどうかを示します。

使用できる値は次のとおりです。

修飾子	説明
<code>global</code>	<code>aura:application</code> タグに <code>extensible="true"</code> が設定されている場合、任意の名前空間で、アプリケーションを別のアプリケーションによって拡張できます。
<code>public</code>	同じ名前空間内でのみ、アプリケーションを別のアプリケーションによって拡張できます。これはデフォルトのアクセスレベルです。

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。

使用できる値は次のとおりです。

修飾子	説明
<code>global</code>	任意の名前空間で、インターフェースを別のインターフェースで拡張したり、コンポーネントで使用したりできます。
<code>public</code>	同じ名前空間内でのみ、インターフェースを別のインターフェースで拡張したり、コンポーネントで使用したりできます。これはデフォルトのアクセスレベルです。

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。

使用できる値は次のとおりです。

修飾子	説明
<code>global</code>	任意の名前空間で、コンポーネントを別のコンポーネントまたはアプリケーションで使用できます。 <code>aura:component</code> タグに <code>extensible="true"</code> が設定されていれば、任意の名前空間で拡張することもできます。

修飾子	説明
public	コンポーネントは同じ名前空間内でのみ、別のコンポーネントで拡張または使用したり、アプリケーションで使用したりできます。これはデフォルトのアクセスレベルです。

 **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示します。

使用できる値は次のとおりです。

アクセス	説明
global	任意の名前空間で属性を使用できます。
public	同じ名前空間内でのみ属性を使用できます。これはデフォルトのアクセスレベルです。
private	属性は、コンテナアプリケーション、インターフェース、コンポーネント、またはイベント内でのみ使用でき、外部からは参照できません。

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または拡張できるかどうかを示します。

使用できる値は次のとおりです。

修飾子	説明
global	任意の名前空間でイベントを使用または拡張できます。
public	同じ名前空間内でのみイベントを使用または拡張できます。これはデフォルトのアクセスレベルです。

第18章 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。

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](#)

デバッグ

第 19 章 デバッグ

トピック:

- [JavaScript コードのデバッグ](#)
- [ログメッセージ](#)
- [警告メッセージ](#)

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がいくつかあります。

たとえば、クライアント側のコードのデバッグには Chrome 開発者ツールを使用します。

- Windows および Linux で開発者ツールを開くには、Google Chrome ブラウザで Ctrl キーと Shift キーを押しながら「!」を押します。Mac の場合は、Option キーと Command キーを押しながら「!」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 の開発者ツール](#)の Web サイトを参照してください。

JavaScript コードのデバッグ

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。

デフォルトでは、Lightning コンポーネントフレームワークは PROD 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、フレームワークはデフォルトで PRODDEBUG モードで実行されます。このモードでは Google Closure Compiler は使用されないため、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。

デバッグモードを有効化する手順は、次のとおりです。

- [設定] で、[開発] > [Lightning コンポーネント] をクリックします。
- [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。
- [保存] をクリックします。

エディション

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

ログメッセージ

クライアント側のコードをデバッグするために、Web 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。

`$A.log(string, [error])` メソッドを使用して、ログメッセージを JavaScript コンソールに書き出します。
最初のパラメータは、ログに記録する文字列です。

2つ目のパラメータ(省略可能)は、エラー オブジェクトであり、より詳細な情報を含めることができます。

メモ:

- `$A` は、JavaScript コードでの Aura オブジェクトの短縮名です。
- `$A.log()` は、PROD または PRODDEBUG モードでは出力を行いません。

たとえば、`$A.log("This is a log message")` は、JavaScript コンソールに次のように出力します。

```
This is a log message
```

クライアント側コントローラの `openNote` というアクションの内部に `$A.log("The name of the action is: " + this.getDef().getName())` を追加した場合、JavaScript コンソールには次のように出力されます。

```
The name of the action is: openNote
```

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。

本番モードでのログ

PROD または PRODDEBUG モードでメッセージをログに記録するために、カスタムログ関数を作成できます。特定の重大度レベルのログメッセージに登録するには、`$A.logger.subscribe(String level, function callback)` を使用する必要があります。

最初のパラメータは、登録される重大度レベルです。有効な値は、次のとおりです。

- ASSERT
- ERROR
- INFO
- WARNING

2つ目のパラメータは、登録された重大度レベルのメッセージがログに記録されるとコールされるコールバック関数です。`$A.log()` は、重大度レベル INFO のメッセージをログに記録します。

クライアント側コントローラのサンプル JavaScript コードを見てみましょう。

```
({  
  
    sampleControllerAction: function(cmp) {  
  
        // subscribe to severity levels  
  
        $A.logger.subscribe("INFO", logCustom);  
  
        // Following subscriptions not exercised here but shown for completeness  
  
        // $A.logger.subscribe("WARNING", logCustom);  
  
        // $A.logger.subscribe("ASSERT", logCustom);  
  
        // $A.logger.subscribe("ERROR", logCustom);  
  
  
        $A.log("log one arg");  
  
        $A.log("log two args", {message: "drat and double drat"});  
  
  
        function logCustom(level, message, error) {  
  
            console.log(getTimestamp(), "logCustom: ", arguments);  
  
        }  
  
  
        function getTimestamp() {  
  
            return new Date().toJSON();  
        }  
    }  
})
```

```
    }  
}  
  
})
```

`$A.logger.subscribe("INFO", logCustom)` は、重大度レベル `INFO` でメッセージがログに記録されると `logCustom()` 関数をコールするように登録します。この場合、`logCustom()` はメッセージをタイムスタンプと一緒にコンソールにログを記録するだけです。

`$A.log()` コールでは、登録と一致する重大度レベル `INFO` のログメッセージがログに記録され、`logCustom()` コールバックが呼び出されます。

警告メッセージ

クライアント側のコードをデバッグするには、`warning()` メソッドを使用して Web ブラウザの JavaScript コンソールに出力を書き出します。

`$A.warning(string)` メソッドを使用して、警告メッセージを JavaScript コンソールに書き出します。このパラメータは、表示するメッセージです。たとえば、`$A.warning("This is a warning message.");` は、「`This is a warning message.`」(これは警告メッセージです。)を JavaScript コンソールに出力します。JavaScript コンソールにスタック追跡も表示されます。

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。

リファレンス

第 20 章 リファレンスの概要

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:application
- aura:dependency
- aura:event
- aura:interface
- aura:set
- コンポーネントの参照
- 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
- サポートされる aura:attribute の型

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。

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には、フレームワークに付随しすぐに使用できるコンポーネントの説明やソースなど、参照情報が含まれています。アプリケーションには、次の URL でアクセスします。

<mySalesforceInstance>.lightning.force.com/auradocs/reference.app
(<mySalesforceInstance> は、na1 など、組織をホストするインスタンスの名前です)。

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ーポイントであり、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。
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。
useAppcache	Boolean	アプリケーションキャッシュを使用するかどうかを指定します。有効なオプションは、true または false です。デフォルトは false です。

`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基本](#)
- [AppCache の使用](#)
- [アプリケーションのアクセス制御](#)

aura:dependency

<aura:dependency> タグでは、フレームワークで簡単に検出できない連動関係を宣言できます。

このフレームワークでは、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、開発時に定義を変更したことが検出されると、フレームワークで自動的に再読み込みされます。ただし、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をインスタンス化する、クライアント側またはサーバ側のプロバイダをコンポーネントで使用する場合は、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、次のタグをコンポーネントに追加すると、aura:placeholder 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://aura:placeholder" />
```

<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	コンポーネントが依存するリソース。たとえば、 resource="markup://sampleNamespace:sampleComponent" は、 sampleNamespace 名前空間の sampleComponent を指します。 ワイルドカード照合では、リソース名でアスタリスク (*) を使用します。たとえば、resource="markup://sampleNamespace:/*" は名前空間のすべてに一致し、resource="markup://sampleNamespace:input*" は名前空間の input で始まるすべてに一致します。
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。全種別のリソースと一致させるには、type="*" を使用します。 最も一般的に使用される値は、次のとおりです。 <ul style="list-style-type: none"> • COMPONENT • APPLICATION

システム属性	説明
	<ul style="list-style-type: none"> EVENT <p>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。</p>

関連トピック:

[コンポーネントの動的な作成](#)

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、 <code>public</code> (デフォルト)と <code>global</code> 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、 <code>extends="namespace:myEvent"</code> 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](#)[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、 <code>public</code> (デフォルト)と <code>global</code> です。
description	String	インターフェースの説明。

属性	型	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、 <code>extends="namespace:intfB"</code> 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: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 `docsample:setTagSuper` コンポーネントです。

```
<!--docsample:setTagSuper-->

<aura:component extensible="true">

    <aura:attribute name="address1" type="String" />

    setTagSuper address1: {!v.address1}<br/>

</aura:component>
```

`docsample:setTagSuper` の出力は、次のようにになります。

```
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 `docsample:setTagSuper` を拡張する `docsample:setTagSub` コンポーネントです。

```
<!--docsample:setTagSub-->

<aura:component extends="docsample:setTagSuper">

    <aura:set attribute="address1" value="808 State St" />
```

```
</aura:component>
```

docsample:setTagSub の出力は、次のようにになります。

```
setTagSuper address1: 808 State St
```

sampleSetTagExdocsample:setTagSub は、スーパーコンポーネント docsample: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 230)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、docsample:setTagSuperRef は docsample:setTagSuper を参照し、aura:set を使用せずに address1 属性を直接設定します。

```
<!--docsample:setTagSuperRef-->

<aura:component>

    <docsample:setTagSuper address1="1 Sesame St" />

</aura:component>
```

docsample:setTagSuperRef の出力は、次のようにになります。

```
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。docsample:myIntfインターフェースの例を見てみましょう。

```
<!--docsample: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。

```
<!--docsample:myIntfImpl-->

<aura:component>

    <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

Salesforce1 または Lightning アプリケーション用に、標準コンポーネントを再利用または拡張します。

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 component attributes here -->  
    <!-- Optional HTML markup -->  
    <div class="container">  
        Hello world!  
        <!-- Other components -->  
    </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。	

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。	
HTMLAttributes	HashMap	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。

次の例は、`isTrue` 式が `true` と評価されているためボディを表示します。

```
<aura:attribute name="display" type="Boolean" default="true"/>

<aura:if isTrue="{!v.display}">

    Show this if true

    <aura:set attribute="else">

        Show this if false

    </aura:set>

</aura:if>
```

属性

属性名	属性型	説明	必須項目
body	ComponentDefRef[]	<code>isTrue</code> が <code>true</code> と評価されたときに表示するコンポーネント。	はい
else	ComponentDefRef[]	<code>isTrue</code> が <code>false</code> 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 <code>aura:set</code> 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 <code>true</code> 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	ArrayList	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

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。`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。

属性

属性名	属性型	説明	必須項目
<code>body</code>	<code>Component[]</code>	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。	
<code>else</code>	<code>Component[]</code>	<code>isTrue</code> が <code>false</code> 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。 <code><aura:set></code> タグを使用して設定します。	
<code>isTrue</code>	<code>Boolean</code>	コンポーネントのボディを表示するために <code>true</code> と評価され はいる必要がある式。	

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
<code>value</code>	<code>String</code>	表示する文字列。	

aura:unescapeHTML

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:unescapeHTML>` 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。	

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抽象コンポーネント。

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。	

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抽象コンポーネント。

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	Object	バインドする Salesforce 項目のデータ値。	

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();
  // Update the component
  helper.getRecords(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。	
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	レコードの保存要求。
recordSaveSuccess	COMPONENT	レコードが正常に保存されたことを示します。

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。ミニレイアウトでは、名前項目および関連する親レコードの項目が表示されます。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="a02D0000006V80v" type="MINI"/>
```

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="type" type="String" default="News" description="The type of feed"
access="GLOBAL"/>
  <aura:attribute name="types" type="String[]" default="Bookmarks,Company,Files,Groups,Home,News,People"
description="A list of feed types"/>
<h1>My Feeds</h1>
```

```
<ui:inputSelect aura:id="typeSelect" change="{!!c.onChangeType}" label="Type"/>
<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 `ui:inputSelect` 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 typeOpts = new Array();

        // Set the feed types on the ui:inputSelect component
        for (var i = 0; i < types.length; i++) {
            typeOpts.push({label: types[i], value: types[i], selected: types[i] === type});
        }
        component.find("typeSelect").set("v.options", typeOpts);
    },

    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",
            {
                componentDef : "markup://forceChatter:feed",
                attributes : {
                    values : {
                        type: type
                    }
                }
            }
        );
    }
})
```

フィードは、Salesforce1 アプリケーションでのみサポートされます。このフィードをコンポーネントに含め、Salesforce1 アプリケーションからアクセスできます。Salesforce1 外で使用すると、リンクの停止、未処理のイベント、スタイルの欠落などの問題が生じる可能性があります。フィード種別のリストは、『Chatter REST API 開発者ガイド』の「[フィードおよびフィード要素の使用](#)」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。	
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ルドの場合、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、News、Home、Record、To です。	

ltng:require

連動関係の順序を維持しながらスクリプトおよびスタイルシートを読み込み、複数のコンポーネントにまたがって1回のみ挿入します。

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。	
scripts	ArrayList	読み込まれる依存性順序で示したスタイルシートのセット。	
styles	ArrayList	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。	

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。

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。

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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。

イベント名	イベントタイプ	説明
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。

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で設定する必要があるのは `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="default uiBlock 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	HTML 入力要素の type 属性を指定します。デフォルト値は「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 はい の value 属性に対応します。	
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	コンポーネントが押されたことを示します。

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。

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:xxxxmenulitem 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
mouseup	COMPONENT	ユーザがマウスボタンを放したこと示します。
mousedown	COMPONENT	ユーザがマウスキーキを押したこと示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したこと示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したこと示します。

イベント名	イベントタイプ	説明
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。

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">

  <label class="uiLabel-left uiLabel">
    <span>Reimbursed?</span>
  </label>

  <input type="checkbox" class="uiInput uiInputCheckbox">

</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}"/>


update : function (cmp, event) {
  var elem = event.getSource().getElement();
  $A.util.toggleClass(elem, "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。	

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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。

イベント名	イベントタイプ	説明
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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 タグとして表示される、通貨としての数値の入力項目を表します。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 label="Amount" class="field" value="50"/>
```

この例の結果、次の HTML になります。

```
<div class="uiInput">

  <label class="uiLabel-left uiLabel">
    <span>Amount</span>
  </label>

  <input class="field" 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取得し、ui:outputCurrency を使用して値を表示します。

```
<aura:component>

    <ui:inputCurrency aura:id="amount" label="Amount" class="field" value="50"/>

    <ui:button class="btn" label="Submit" press="{!!c.setOutput}" />

    <div aura:id="msg" class="hide">
        You entered: <ui:outputCurrency aura:id="oCurrency" value="" />
    </div>
</aura:component>
```

```
({

    setOutput : function(component, event, helper) {
        var el = component.find("msg");
        $A.util.removeClass(el.getElement(), 'hide');

        var amount = component.find("amount").get("v.value");
        var oCurrency = component.find("oCurrency");
        oCurrency.set("v.value", amount);
    }
})
```

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。	
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	BigDecimal	数値の入力値。	

イベント

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがマウスボタンを放したことなどを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことなどを示します。

イベント名	イベントタイプ	説明
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したこと示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたこと示します。

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、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。ブラウザの言語ロケールで指定されたデフォルト形式で値が表示されます。

次に、項目値 1/30/2014 を表示する、日付ピッカーがある日付項目の基本設定を示します。format="MMMd, yyyy" を指定すると、項目値が January 30, 2014 と表示されます。

```
<ui:inputDate aura:id="dateField" label="Birthday" value="2014-01-30"
displayDatePicker="true"/>
```

この例の結果、次のHTMLになります。

```
<div class="uiInput uiInputDate">

  <label class="uiLabel-left uiLabel">
    <span>Birthday</span>
  </label>

  <input placeholder="M/d/yyyy" type="text" class="uiInput uiInputDate">
  <a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
  </a>

  <a class="clearIcon">
    <span class="assistiveText">Clear Button</span>
  </a>

  <div class="uiDatePicker">
    <!--Date picker set to visible when icon is clicked-->
  </div>
</div>
```

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>
```

```
</div>  
</aura:component>
```

```
({  
  
    doInit : function(component, event, helper) {  
  
        var today = new Date();  
  
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"  
+ today.getDate());  
  
        component.set('v.deadline', today);  
  
    },  
  
    setOutput : function(component, event, helper) {  
  
        var el = component.find("msg");  
  
        $A.util.removeClass(el.getElement(), '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。	
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	ui:datePicker が表示されるかどうかを示します。	
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	日時の書式設定に使用される言語ロケール。	
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	Date	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。

イベント名	イベントタイプ	説明
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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、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。ブラウザの言語ロケールで指定されたデフォルト形式で値が表示されます。

次に、7/29/2014 1:11 PM 形式で現在の日時を表示する、日付ピッカーがある日付項目の基本設定を示します。

```
<!-- 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();
    component.set("v.today", today);
```

この例の結果、次の HTML になります。

```
<div class="uiInput uiInputDateTime">

<label class="uiLabel-left uiLabel">
    <span>Expense Date</span>
</label>

<input class="form-control uiInput uiInputDateTime" placeholder="M/d/yyyy h:mm a"
type="text">

<a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
</a>

<a class="clearIcon" href="javascript:void(0);">
    <span class="assistiveText">Clear Button</span>
</a>

<div class="uiDatePicker">
    <!-- Date picker set to visible when icon is clicked -->
</div>
```

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>
```

```
</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 el = component.find("msg");  

    $A.util.removeClass(el.getElement(), '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	ui:datePickerが表示されるかどうかを示します。	
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	日時の書式設定に使用される言語ロケール。	
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	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことと示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことと示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことと示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことと示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことと示します。

イベント名	イベントタイプ	説明
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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、addErrors() を使用して追加できます。入力値を `false` に設定しエラーメッセージを追加することで、デフォルトのエラー処理を使用できます。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.setValue("v.value", false);
        inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
    } else {
        //clear error
        inputCmp.setValid("v.value", true);
    }
}
```

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, evt) {
        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.setValid("v.value", false);
            colorCmp.addErrors("v.value", [{message:"Enter some text"}]);
        }
        else {
            //clear error
            if(!colorCmp.isValid("v.value")){
                colorCmp.setValid("v.value", true);
            }
        }
    }
})
```

```
}
```

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	ArrayList	表示するエラーメッセージのリスト。	

イベント

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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。

```
<div class="uiInput uiInputEmail">

  <label class="uiLabel-left uiLabel">
    <span>Email</span>
  </label>

  <input placeholder="abc@email.com" type="email" class="uiInput uiInputEmail">

</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 el = component.find("msg");
    $A.util.removeClass(el.getElement(), '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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを見ます。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを見ます。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを見ます。

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 タグとして表示される、数値の入力項目を表します。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">
<label class="uiLabel-left uiLabel">
    <span>Age</span>
</label>
<input max="9999999999999" step="1" type="text"
       min="-9999999999999" class="uiInput uiInputNumber">
</div>
```

ui:inputNumber コンポーネントからの出力を表示するには、ui:inputNumber 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:label label="Cost"/>
<ui:inputNumber 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>
    <ui:inputNumber aura:id="inputCmp" label="Enter a number: " /> <br/>
        <ui:button label="Submit" press="={!c.validate}" />
        <ui:outputNumber aura:id="outNum" value="" />
</aura:component>
```

```
{
    validate : function(component, evt) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        var myOutput = component.find("outNum");

        myOutput.set("v.value", value);

        // Check if input is numeric
        if (isNaN(value)) {
```

```

        // Set error message
        inputCmp.setValid("v.value", false);
        inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
    } else {
        // Clear error
        inputCmp.setValid("v.value", true);
    }
}
})

```

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。	
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	BigDecimal	数値の入力値。	

イベント

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーカーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを見ます。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを見ます。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを見ます。

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。

```
<div class="uiInput uiInputPhone">

    <label class="uiLabel-left uiLabel">
        <span>Phone</span>
    </label>

    <input class="uiInput uiInputPhone" 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 el = component.find("msg");
        $A.util.removeClass(el.getElement(), '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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを見ます。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを見ます。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを見ます。

ui:inputRadio

人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">  
  <label class="uiLabel-left 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,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 elem = evt.getSource().getElement();

    var selected = elem.textContent;

    resultCmp = cmp.find("radioResult");

    resultCmp.set("v.value", selected);

},
}

onGroup: function(cmp, evt) {

    var elem = evt.getSource().getElement();

    var selected = elem.textContent;

    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。	

属性名	属性型	説明	必須項目
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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたこと示します。
select	COMPONENT	ユーザが選択を行ったことを示します。

イベント名	イベントタイプ	説明
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことと示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

ui:inputRichText

リッチテキストを入力するための入力項目。

デフォルトでは、ui:inputRichText はリッチテキストを入力するための WYSIWYG エディタを表示します。isRichText="false" 設定では、WYSIWYG エディタではなく ui:inputTextArea コンポーネントを使用します。

リッチテキストエディタの幅と高さは、ui:inputTextArea コンポーネントの幅と高さとは独立しています。isRichText="false" に設定した場合にコンポーネントの幅と高さを設定するには、cols および rows 属性を使用します。それ以外の場合は、width および height 属性を使用します。

ui:outputRichText は、CKEditor でサポートされる HTML タグのリストをサポートします。<script> などのタグは削除されます。

次の例では、テキストエリアと WYSIWYG エディタが表示されます。

```
<aura:component>

<ui:inputRichText aura:id="inputRT" label="Rich Text Demo" labelPosition="hidden" cols="50"
rows="5" value="&lt;b&gt;Aura&lt;/b&gt;, &lt;span style='color:red'&gt;input rich text
demo&lt;/span&gt;" />

<ui:button aura:id="outputButton" buttonTitle="Click to see what you put into the rich
text field" label="Display" press=" {!c.getInput} " />

<ui:outputRichText aura:id="outputRT" value=" " />
```

```
</aura:component>

({
  getInput : function(cmp) {
    var userInput = cmp.find("inputRT").get("v.value");
    var output = cmp.find("outputRT");
    output.set("v.value", userInput);
  }
})
```

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。	
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。	

属性名	属性型	説明	必須項目
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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。

イベント名	イベントタイプ	説明
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことと示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことと示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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">
  <label class="uiLabel-left uiLabel">
    <span>Pin</span>
  </label>
  <input class="field" 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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したこと示します。

イベント名	イベントタイプ	説明
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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>
```

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ます。

```
<aura:attribute name="contacts" type="String[]" default="Primary Contact, Secondary Contact, Other"/>
<ui:inputSelect>
  <aura:iteration items="{!v.contacts}" var="contact">
    <ui:inputSelectOption text="{!contact}" label="{!contact}"/>
  </aura:iteration>
</ui:inputSelect>
```

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。

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, evt) {

        var selectCmp = cmp.find("InputSelectSingle");

        resultCmp = cmp.find("singleResult");

        resultCmp.set("v.value", selectCmp.get("v.value"));

    },

    onMultiSelectChange: function(cmp, evt) {
```

```

var selectCmp = cmp.find("InputSelectMultiple");

resultCmp = cmp.find("multiResult");

resultCmp.set("v.value", selectCmp.get("v.value"));

},

onChange: function(cmp, evt) {

var dynamicCmp = cmp.find("InputSelectDynamic");

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。	
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	ArrayList	aura.components.ui.InputOption のリスト。	
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。	

属性名	属性型	説明	必須項目
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。

イベント名	イベントタイプ	説明
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

ui:inputSelectOption

<ui:inputSelect> 要素内にネストされた 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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。

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 uiInputText">

  <label class="uiLabel-left uiLabel">

    <span>Expense Name</span>

    <span class="required">*</span>

  </label>

  <input required="required" class="uiInput uiInputText" type="text">

</div>
```

次の例は、ui:inputText コンポーネントの値を取得し、ui:outputText を使用して値を表示します。

```
<aura:component>
```

```
<ui:inputText aura:id="name" label="Enter some text" class="field" value="My Text"/>

<ui:button class="btn" label="Submit" press="{!!c.setOutput}" />

<div aura:id="msg" class="hide">
  You entered: <ui:outputText aura:id="oName" value="" />
</div>
</aura:component>
```

```
({
  setOutput : function(component, event, helper) {
    var el = component.find("msg");
    $A.util.removeClass(el.getElement(), 'hide');

    var name = component.find("name").get("v.value");
    var oName = component.find("oName");
    oName.set("v.value", name);
  }
})
```

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
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。

イベント名	イベントタイプ	説明
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことと示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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">
  <label class="uiLabel-left uiLabel">
    <span>Comments</span>
  </label>
  <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 el = component.find("msg");
        $A.util.removeClass(el.getElement(), '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。	

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。	
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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。

イベント名	イベントタイプ	説明
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを見ます。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを見ます。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したことを見ます。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを見ます。

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">
```

```
<label class="uiLabel-left uiLabel">  
  <span>Venue URL</span>  
</label>  
  
<input class="field" 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 el = component.find("msg");  
  
    $A.util.removeClass(el.getElement(), '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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
validationError	COMPONENT	コンポーネントに検証エラーがあることを示します。
paste	COMPONENT	ユーザがコンテンツをクリップボードから貼り付けたことを示します。
change	COMPONENT	コンポーネントのコンテンツまたは状況が変更されたことを示します。
clearErrors	COMPONENT	検証エラーをすべてクリアする必要があることを示します。
cut	COMPONENT	ユーザがコンテンツを切り取ってクリップボードに保存したこと示します。
copy	COMPONENT	ユーザがコンテンツをクリップボードにコピーしたことを示します。

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。menuTriggerLink および menuList コンポーネントを指定する必要があります。

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">

<div style="margin:20px;">

    <div style="display:inline-block; width:50%; vertical-align:top;">

        ui:actionMenuItem

        <ui:menu>

            <ui:menuTriggerLink aura:id="trigger" label="Select your favorite team"/>

            <ui:menuList class="actionMenu" aura:id="actionMenu">

                <ui:actionMenuItem class="actionItem1" aura:id="actionItem1" label="Bayern
Munich" click="{!c.updateTriggerLabel}" />

                <ui:actionMenuItem class="actionItem2" aura:id="actionItem2" label="FC
Barcelona" click="{!c.updateTriggerLabel}" disabled="true"/>

                <ui:actionMenuItem class="actionItem3" aura:id="actionItem3" label="Inter
Milan" click="{!c.updateTriggerLabel}" />

                <ui:actionMenuItem class="actionItem4" aura:id="actionItem4" label="Manchester United" click="{!c.updateTriggerLabel}" />

            </ui:menuList>

        </ui:menu>

    </div>

</div>

<hr/>

<p>

<div style="margin:20px;">

    <div style="display:inline-block; width:50%; vertical-align:top;">

        ui:checkboxMenuItem

        <ui:menu>

            <ui:menuTriggerLink class="checkboxMenuLabel" aura:id="checkboxMenuLabel" label="Select your favorite teams"/>

        </ui:menu>

    </div>

</div>


```

```
<ui:menuList aura:id="checkboxMenu" class="checkboxMenu">

    <ui:checkboxMenuItem class="checkboxItem1" aura:id="checkboxItem1"
label="San Francisco 49ers"/>

    <ui:checkboxMenuItem class="checkboxItem2" aura:id="checkboxItem2"
label="Seattle Seahawks"/>

    <ui:checkboxMenuItem class="checkboxItem3" aura:id="checkboxItem3"
label="St. Louis Rams"/>

    <ui:checkboxMenuItem class="checkboxItem4" aura:id="checkboxItem4"
label="Arizona Cardinals" disabled="true" selected="true"/>

</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>

</div>

</div>

<hr/>

<p/>

<div style="margin:20px;">

    <div style="display:inline-block;width:50%;vertical-align:top;">

        ui:radioMenuItem

        <ui:menu>

            <ui:menuTriggerLink class="radioMenuItemLabel" aura:id="radioMenuItemLabel"
label="Select a team"/>

            <ui:menuList class="radioMenu" aura:id="radioMenu">

                <ui:radioMenuItem class="radioItem1" aura:id="radioItem1" label="San
Francisco"/>

                <ui:radioMenuItem class="radioItem2" aura:id="radioItem2" label="LA
Dodgers"/>

                <ui:radioMenuItem class="radioItem3" aura:id="radioItem3" label="Arizona"/>

            </ui:menuList>
        </ui:menu>
    </div>
</div>
```

```
<ui:radioMenuItem class="radioItem4" aura:id="radioItem4" label="Diego"
disabled="true"/>

<ui:radioMenuItem class="radioItem5" aura:id="radioItem5" label="Colorado"/>

</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>

</div>

</div>

<hr/>

<p/>

<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 teams"/>

        <ui:menuList>

            <ui:actionMenuItem label="Bayern Munich" click="{!!c.updateLabel}"/>
            <ui:actionMenuItem label="FC Barcelona" click="{!!c.updateLabel}"/>
            <ui:actionMenuItem label="Inter Milan" click="{!!c.updateLabel}"/>
            <ui:actionMenuItem label="Manchester United" click="{!!c.updateLabel}"/>
            <ui:menuItemSeparator/>
            <ui:checkboxMenuItem label="San Francisco 49ers"/>
            <ui:checkboxMenuItem label="Seattle Seahawks"/>
            <ui:checkboxMenuItem label="St. Louis Rams"/>
            <ui:checkboxMenuItem label="Arizona Cardinals"/>

        </ui:menuList>
    </ui:menu>
</div>
</div>
```

```
<ui:menuItemSeparator/>

<ui:radioMenuItem label="San Francisco"/>

<ui:radioMenuItem label="LA Dodgers"/>

<ui:radioMenuItem label="Arizona"/>

<ui:radioMenuItem label="San Diego"/>

<ui:radioMenuItem label="Colorado"/>

</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, event) {

    var menuCmp = cmp.find("checkboxMenu");

    var menuItems = menuCmp.get("v.childMenuItems");

    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(","));

},

getRadioMenuItemSelected: function(cmp, event) {

    var menuCmp = cmp.find("radioMenu");

    var menuItems = menuCmp.get("v.childMenuItems");

    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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。

イベント名	イベントタイプ	説明
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。

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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

ui:menuList

メニュー項目の ui:actionMenuItem、ui:checkboxMenuItem、および ui:radioMenuItem と区切り文字コンポーネントの ui:menuItemSeparator を含めることができ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でき、メニュー項目をトリガする ui:menuTriggerLink コンポーネントと一緒に使用できます。

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 に設定されても、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	ArrayList	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
collapse	COMPONENT	コンポーネントが折りたたまれていることを示します。
expand	COMPONENT	コンポーネントが展開されていることを示します。
menuSelect	COMPONENT	ユーザがメニューコンポーネント内のメニュー項目を選択したことを示します。

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:menu component 内で使用されます。

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。	
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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。

イベント名	イベントタイプ	説明
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことと示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことと示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
menuTriggerPress	COMPONENT	メニュートリガがクリックされたことを示します。

ui:menuTriggerLink

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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーカーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを見ます。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。
menuTriggerPress	COMPONENT	メニュートリガがクリックされたことを示します。

ui:message

さまざまな重要度レベルのメッセージを表します。

`severity` 属性は、メッセージの重要度レベルを示し、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、条件が `true` と評価された場合は成功メッセージ、それ以外の場合はエラーメッセージを表示します。

```
<aura:component>

<aura:attribute name="myBool" type="Boolean" default="true"/>

<aura:renderIf isTrue={!v.myBool}>
  <ui:message title="Success" severity="confirm">
    ...
  </ui:message>
</aura:renderIf>
```

```
The operation is successful.  
  
</ui:message>  
  
<aura:set attribute="else">  
  
    <ui:message title="Error" severity="error">  
  
        This is an error.  
  
    </ui:message>  
  
</aura:set>  
  
</aura:renderIf>  
  
</aura:component>
```

次の例では、さまざまな重要度レベル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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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。

```

```

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「checkbox checked」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「checkbox unchecked」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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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>
<ui:inputCurrency aura:id="amount" label="Amount" class="field" value="50"/>
```

```
<ui:button class="btn" label="Submit" press="={!c.setOutput}"/>

<div aura:id="msg" class="hide">
  You entered: <ui:outputCurrency aura:id="oCurrency" value="" />
</div>
</aura:component>
```

```
({
  setOutput : function(component, event, helper) {
    var el = component.find("msg");
    $A.util.removeClass(el.getElement(), 'hide');

    var amount = component.find("amount").get("v.value");
    var oCurrency = component.find("oCurrency");
    oCurrency.set("v.value", amount);
  }
})
```

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。	
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。	

属性名	属性型	説明	必須項目
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	BigDecimal	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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());

        component.set('v.deadline', today);

    },

    setOutput : function(component, event, helper) {
        var el = component.find("msg");

        $A.util.removeClass(el.getElement(), '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。	
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	日付値の形式設定に使用される言語ロケール。	
value	String	日付の出力値。ISO-8601 形式(YYYY-MM-DD) の日付文字列で はい ある必要があります。	

イベント

イベント名	イベントタイプ	説明
mouseup	COMPONENT	ユーザがマウスボタンを放したこと示します。
mousedown	COMPONENT	ユーザがマウスキーを押したこと示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したこと示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したこと示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したこと示します。

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 el = component.find("msg");
    $A.util.removeClass(el.getElement(), '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。	
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	日付値の形式設定に使用される言語ロケール。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。

イベント名	イベントタイプ	説明
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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 el = component.find("msg");
    $A.util.removeClass(el.getElement(), '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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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、ブラウザの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:inputNumber 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>
  <ui:inputNumber aura:id="inputCmp" label="Enter a number: "/> <br/>
    <ui:button label="Submit" press=" {!c.validate} "/>
    <ui:outputNumber aura:id="outNum" value="" />
</aura:component>
```

```
({
  validate : function(component, evt) {
    var inputCmp = component.find("inputCmp");
    var value = inputCmp.get("v.value");

    var myOutput = component.find("outNum");

    myOutput.set("v.value", value);

    // Check if input is numeric
    if (isNaN(value)) {
      // Set error message
      inputCmp.setValid("v.value", false);
      inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
    } else {
      // Clear error
      inputCmp.setValid("v.value", true);
    }
  }
})
```

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	BigDecimal	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーキを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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 el = component.find("msg");  
  
    $A.util.removeClass(el.getElement(), '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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したこと示します。
mousedown	COMPONENT	ユーザがマウスキーを押したこと示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したこと示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したこと示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したこと示します。

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、タグを含むリッチ形式テキストを表示します。

ui:outputRichText コンポーネントは、リッチテキストを表し、ui:inputRichText コンポーネントからの入力の表示に使用できます。

たとえば、ui:inputRichText コンポーネントを介して太字や色付きのテキストを入力し、その値を ui:outputRichText コンポーネントにバインドできます。結果は次の HTML のようになります。

```
<div class="uiOutputRichText">
  <b>Aura</b>, <span style="color:red">input rich text demo</span>
</div>
```

次の例は、ui:inputRichText 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:inputRichText aura:id="inputRT" label="Rich Text Demo" labelPosition="hidden" cols="50"
rows="5" value="&lt;b&gt;Aura&lt;/b&gt;, &lt;span style='color:red'&gt;input rich text
demo&lt;/span&gt;" />

<ui:button aura:id="outputButton" buttonTitle="Click to see what you put into the rich
text field" label="Display" press=" {!c.getInput } "/>

<ui:outputRichText aura:id="outputRT" value=" " />

</aura:component>
```

```
{
  getInput : function(cmp) {

    var userInput = cmp.find("inputRT").get("v.value");

    var output = cmp.find("outputRT");
    output.set("v.value", userInput);
  }
})
```

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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーカーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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}" label="my output"/>
```

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>

    <ui:inputText aura:id="name" label="Enter some text" class="field" value="My Text"/>

    <ui:button class="btn" label="Submit" press=" {!c.setOutput}"/>

    <div aura:id="msg" class="hide">
```

```
You entered: <ui:outputText aura:id="oName" value="" />
</div>
</aura:component>
```

```
({
  setOutput : function(component, event, helper) {
    var el = component.find("msg");
    $A.util.removeClass(el.getElement(), 'hide');

    var name = component.find("name").get("v.value");
    var oName = component.find("oName");
    oName.set("v.value", name);
  }
})
```

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	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
mouseup	COMPONENT	ユーザがマウスボタンを放したこと示します。
mousedown	COMPONENT	ユーザがマウスキーを押したこと示します。

イベント名	イベントタイプ	説明
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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、複数行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。

```
<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 el = component.find("msg");
    $A.util.removeClass(el.getElement(), '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。	
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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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="{!v.myURL}"/>
```

前の例によって次の HTML が表示されます。

```
<a href="http://www.google.com" dir="ltr" class="uiOutputURL">http://www.google.com</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 el = component.find("msg");
        $A.util.removeClass(el.getElement(), '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	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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。

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。

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
mouseup	COMPONENT	ユーザがマウスボタンを放したことを示します。
mousedown	COMPONENT	ユーザがマウスキーを押したことを示します。
mousemove	COMPONENT	ユーザがマウスポインタを移動したことを示します。
dblclick	COMPONENT	コンポーネントがダブルクリックされたことを示します。
mouseout	COMPONENT	ユーザがマウスポインタをコンポーネントから移動したことを示します。
click	COMPONENT	コンポーネントがクリックされたことを示します。

イベント名	イベントタイプ	説明
mouseover	COMPONENT	ユーザがマウスポインタをコンポーネントに移動したことを示します。
keyup	COMPONENT	ユーザがキーボードキーを放したことを示します。
keypress	COMPONENT	ユーザがキーボードキーを押したままにしたことを示します。
select	COMPONENT	ユーザが選択を行ったことを示します。
keydown	COMPONENT	ユーザがキーボードキーを押してから放したことを示します。
focus	COMPONENT	コンポーネントにフォーカスが置かれたことを示します。
blur	COMPONENT	コンポーネントからフォーカスが離れたことを示します。

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、コンポーネントがサーバ応答を待機中はスピナーを表示し、コンポーネントが応答の待機を停止するとスピナーを削除します。

```
<aura:component>

    <aura:handler event="aura:waiting" action="{!c.showSpinner}" />
    <aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}" />

    <center><ui:spinner aura:id="spinner"/></center>

</aura:component>
```

次のクライアント側コントローラでは、適宜スピナーを表示または非表示にします。

```
({
    showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');

        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : true });
        evt.fire();
    },
},
```

```
hideSpinner : function (component, event, helper) {  
    var spinner = component.find('spinner');  
    var evt = spinner.get("e.toggle");  
    evt.setParams({ isVisible : false });  
    evt.fire();  
}  
})
```

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, event) {  
        var spinner = cmp.find('spinner');  
        var evt = spinner.get("e.toggle");  
  
        if (!$A.util.hasClass(spinner.getElement(), '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。	
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
toggleLoadingIndicator	COMPONENT	ui:spinner コンポーネントの表示設定を変更します。

イベントの参照

標準搭載のイベントを使用して、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。

force:createRecord

指定した entityApiName (「Account」や「myNamespace__MyObject__c」など) の新しい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作成ページを表示するには、entityApiName パラメータ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。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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
entityApiName	String	必須。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）。
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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
recordId	String	必須。編集するレコードに関連付けられたレコード ID。

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'];  
  
    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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
listViewId	String	必須。表示するリストビューの ID。
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)。

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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
scope	String	必須。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)。
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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
<code>parentRecordId</code>	String	必須。親レコードの ID。
<code>relatedListId</code>	String	必須。表示する関連リストの API 名 (「Contacts」や「Opportunities」など)。

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。

```
createRecord : function (component, event, helper) {

    var navEvt = $A.get("e.force:navigateToSObject");
    navEvt.setParams({
        "recordId": "00QB000000ybNX",
        "slideDevName": "related"
}
```

```

    });
    navEvt.fire();
}

```

 **メモ:** このイベントは Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
recordId	String	必須。レコード ID。
slideDevName	String	<p>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。</p> <ul style="list-style-type: none"> • <code>detail</code>: レコード詳細スライド。これはデフォルト値です。 • <code>chatter</code>: Chatter スライド。 • <code>related</code>: 関連情報スライド。

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に、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。

 **メモ:** navigateToURL では、標準の URL スキームのみがサポートされます。カスタムスキームにアクセスするには、代わりに `window.location` を使用します。

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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
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 <code>false</code> です。
url	String	必須。対象の URL。

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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

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" onSaveSuccess="{!!c.onSuccess}" />
<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();
}
```

次のクライアント側のコントローラは、別のイベントを起動してコンポーネントを更新することによって `onSaveSuccess` イベントを処理します。

```
onSuccess: function(component) {
    var refresh = $A.get("e.force:refreshView");
    refresh.fire();
}
```

 **メモ:** このイベントは Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

force:showToast

メッセージをポップアップに表示します。

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。

次の例では、トーストメッセージ **Success!** The record has been updated successfully. 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 Salesforce1 でのみサポートされます。このイベントを Salesforce1 外で使用する場合は、適切に処理されません。

属性名	型	説明
title	String	トーストの太字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。
key	String	アイコンの種類を指定します。
duration	Integer	トーストの期間(ミリ秒)。デフォルトは 3,000 ミリ秒です。

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.handleError}"/> <br/>
```

```
<!-- 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 174) 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.addClass}"
        menuExpand=" {!c.removeClass}">
        <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。

```
{
    addClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.addClass(trigger, "myClass");
    },
}
```

```

removeClass : 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.addClass}"
menuExpand="{!!c.removeClass}">

        <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。

```

({
    addClass : function(component, event, helper) {

        var trigger = component.find("trigger");

        $A.util.addClass(trigger, "myClass");

    },

```

```

removeClass : function(component, event, helper) {
    var trigger = component.find("trigger");
    $A.util.removeClass(trigger, "myClass");
}
})

```

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インボ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>
```

```
</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}"/> <br/>  
  
    <!-- 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 174)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

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またはルートコンポーネントの初期表示が完了したことを示します。

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 event="doneRendering"> 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
<code>event</code>	<code>String</code>	イベント名。 <code>aura:doneRendering</code> に設定する必要があります。
<code>action</code>	<code>Object</code>	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

アプリケーションまたはコンポーネント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。

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 event="aura:doneWaiting">` 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、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
$A.log(cmp);
$A.log(event);

})
})
```

`aura:systemError` 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。 <code>aura:systemError</code> 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("message")` 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。

aura:valueChange

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.handleChangeValue}" />
    <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
    }
})
```

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	変更を検出する値。
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。

aura:valueDestroy

値が破棄処理中であることを示します。

このイベントは、属性値が破棄処理中の場合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="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。

```
{
    valueDestroy : function (component, event, helper) {
        var val = event.getParam("value");
        // Do something else here
    }
})
```

たとえば、Salesforce1 アプリケーションで Lightning コンポーネントを表示しているとします。この aura:valueDestroy イベントは、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したときにトリガされ、コンポーネントが破棄されます。この例では、token 属性によって、破棄処理中のコンポーネントが返されます。

destroy 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。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。
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。

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") から取得される破棄処理中の値。

aura:valueInit

値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の初期化時にトリガれます。

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 182)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。
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。

aura:waiting

アプリケーションまたはコンポーネント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。

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 event="aura:waiting"> 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

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、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。

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 `ui: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ログ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、`
` ではなく `
` を使用します。ほとんどの HTML5 タグがサポートされています。

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、次のタグをサポートしていません。

- applet
- base
- basefont
- embed
- font
- frame
- frameset
- isindex
- noframes
- noscript
- object
- param
- svg

関連トピック:

[アクセシビリティのサポート](#)

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
<code>access</code>	<code>String</code>	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、 <code>public</code> (デフォルト)、 <code>global</code> 、 <code>private</code> です。

属性名	型	説明
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。式を使用して、属性のデフォルト値を設定することはできません。動的なデフォルトを設定するには、init を代わりに使用します。 「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 182)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、サポートし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

型	例	説明
		刻項目を含めるには、 <code>DateTime</code> を代わりに使用します。
<code>DateTime</code>	<code><aura:attribute name="lastModifiedDate" type="DateTime" /></code>	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。
<code>Decimal</code>	<code><aura:attribute name="totalPrice" type="Decimal" /></code>	<code>Decimal</code> 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。 <code>java.math.BigDecimal</code> 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、 <code>Double</code> より <code>Decimal</code> のほうが適切です。これは通貨項目に適しています。
<code>Double</code>	<code><aura:attribute name="widthInchesFractional" type="Double" /></code>	<code>Double</code> の値には、小数点以下の値を含めることができます。 <code>java.lang.Double</code> に対応付けられます。通貨項目には、代わりに <code>Decimal</code> を使用します。
<code>Integer</code>	<code><aura:attribute name="numRecords" type="Integer" /></code>	<code>Integer</code> 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 <code>java.lang.Integer</code> に対応付けられます。
<code>Long</code>	<code><aura:attribute name="numSwissBankAccount" type="Long" /></code>	<code>Long</code> 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 <code>java.lang.Long</code> に対応付けられます。 <code>Integer</code> 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。
<code>String</code>	<code><aura:attribute name="message" type="String" /></code>	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[]" />
```

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>  
  
<ui:button press="{!!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、オブジェクト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、オブジェクト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();

```

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" />
```

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
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、 <code>cmp.get("v.sectionLabels")['a']</code> を使用します。
Set	<aura:attribute name="collection" type="Set" default="1,2,3" />	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、"1,2,3" が "3,2,1" と返される可能性があります。

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"/>

<ui:button press=" {!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);
    }
});
```

```

    });
}

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 (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" />
```

コレクションのサポート

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[]" />
```

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	单一のコンポーネント。代わりに <code>Aura.Component[]</code> を使用することをお勧めします。
Aura.Component[]	<p><aura:attribute name="detail" type="Aura.Component[]"/></p> <p><code>type="Aura.Component[]"</code> のデフォルト値を設定するには、 <code>aura:attribute</code> 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。</p> <pre><aura:component> <aura:attribute name="detail" type="Aura.Component[]"> <p>default paragraph1</p> </aura:attribute> Default value is: {!v.detail} </aura:component></pre>	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。 <code>Aura.Component[]</code> という型の属性は <code>facet</code> 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](#)