Publisher and Quick Action Developer Guide

Salesforce, Winter '27

Last updated: July 22, 2026

© Copyright 2000–2026 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

GET STARTED 1

HOW ARE THE APIS DIFFERENT? 2

API PARITY 4

WORK WITH THE APIS 5

Quick Action API 5

- Considerations 7
- getAvailableActions 8
- getAvailableActionFields 9
- getCustomAction 10
- getSelectedActions 10
- invokeAction 11
- refresh 11
- selectAction 11
- setActionFieldValues 12

Publisher API 13

CUSTOMIZE WITH VISUALFORCE 19

Layout and Appearance 20

Email Action 26

Portal Action 28

Log a Call Action 30

Article Tool 33

Replicate a Case Page 35

Create Custom Actions 37

OTHER RESOURCES 41

GET STARTED WITH THE PUBLISHER AND QUICK ACTION APIS

Create custom components to interact with the actions on pages in Salesforce Classic and Lightning Experience apps. Using Aura components, Visualforce, and Apex, you can customize your app's experience, including the case feed. For example, you can use a custom component to let users send an email with a Knowledge article.

The Salesforce Classic Publisher JavaScript APIs, also known as the Case Feed Publisher APIs, and the Lightning Quick Action JavaScript APIs both interact with page actions. The Publisher APIs work with Visualforce components and pages to interact with publisher actions. The Quick Action APIs are called by the `lightning:quickActionAPI` component to interact with quick actions.

Note: Starting with API version 43.0 of the Publisher API, the methods used in Visualforce components work in Lightning Experience. Just point to the latest version of the Publisher API script in your Visualforce pages.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Group**, **Professional**, **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

To use this guide, it helps if you have a basic familiarity with JavaScript, Visualforce, Apex, Aura components, and the Salesforce user interface.

SEE ALSO:

- How Are the Publisher and Quick Action APIs Different?
- Method Parity Between the Publisher API and the Quick Action API
- Quick Action APIs in Lightning Experience
- Publisher APIs in Salesforce Classic
- Customize Case Feed Actions with Visualforce

HOW ARE THE PUBLISHER AND QUICK ACTION APIS DIFFERENT?

The user interface in your org can dictate which development tools you can use to interact with actions. In Salesforce Classic, you use the Salesforce Classic Publisher JavaScript APIs with Visualforce components to interact with actions. In Lightning Experience, you use the `lightning:quickActionAPI` component to call the Lightning Quick Action JavaScript APIs to interact with actions.

Implementation

Salesforce Classic Publisher JavaScript APIs

To implement, load the publisher script in your Visualforce page or component. For example:

```
<script
  type='text/javascript'
  src='/canvas/sdk/js/43.0/publisher.js'/>
```

Then you can reference the Publisher APIs through the `Sfdc.canvas.publisher` object. For example:

```
Sfdc.canvas.publisher.selectAction({...})
```

Lightning Quick Action JavaScript APIs

To implement, use the component `lightning:quickActionAPI` in your custom Aura component. For example:

```
<aura:component
  implements="flexipage:availableForRecordHome"
  description="My Aura component">
      <lightning:quickActionAPI
        aura:id="quickActionAPI" />
</aura:component>
```

Then you can reference the Quick Action APIs in your controller code.

Supported Actions, Apps, and Pages

Salesforce Classic Publisher JavaScript APIs

Works with any quick action on a record page in Salesforce Classic apps for objects that are feed-enabled. Supports apps with standard navigation and console navigation.

Lightning Quick Action JavaScript APIs

Works with any quick actions on a record page in any Lightning Experience app. Supports apps with standard navigation and console navigation.

Available Methods

Salesforce Classic Publisher JavaScript APIs

Provides the following methods:

- publisher.customActionMessage
- publisher.invokeAction
- refresh
- publisher.selectAction
- publisher.setActionInputValues

Lightning Quick Action JavaScript APIs

Provides the following methods:

- getAvailableActions
- getAvailableActionFields
- getCustomAction
- getSelectedActions
- invokeAction
- refresh
- selectAction
- setActionFieldValues

Lightning Experience and Salesforce Classic Support

Salesforce Classic Publisher JavaScript APIs

Works in Salesforce Classic and Lightning Experience. Starting with API version 43.0 of the Salesforce Classic JavaScript Publisher API, the methods used in Visualforce components and pages work in Lightning Experience. Just point to the latest version of the Publisher API script in your Visualforce pages.

```
<script
  src="/canvas/sdk/js/43.0/publisher.js"
  type="text/javascript">
</script>
```

The `portalPostFields` input value is not supported in Lightning Experience.

Lightning Quick Action JavaScript APIs

Works only in Lightning Experience. Before implementing, review the Quick Action API Considerations.

METHOD PARITY BETWEEN THE PUBLISHER API AND THE QUICK ACTION API

The Lightning Quick Action JavaScript API allows you to interact with actions within Aura components similar to how the Salesforce Classic Publisher JavaScript API allows you to interact with publisher actions within Visualforce pages.

This table shows which Quick Action API methods map to Publisher API methods.

Quick Action API Method (in Aura Component)	Publisher API Method (in Visualforce)
getAvailableActions	N/A
getAvailableActionFields	N/A
getCustomAction	customActionMessage
getSelectedActions	N/A
invokeAction	invokeAction
refresh	refresh
selectAction	selectAction
setActionFieldValues	setActionInputValues

WORK WITH THE QUICK ACTION AND PUBLISHER APIS

The Lightning Quick Action JavaScript API and the Salesforce Classic Publisher JavaScript API both let you interact with actions. If you're building out Aura components in Lightning Experience, use the Quick Action API. This API can interact with all quick actions on a record page. If you're writing Visualforce pages in Salesforce Classic, use the Publisher API. This API can interact with any quick actions on record pages in Salesforce Classic apps for objects that are feed-enabled.

IN THIS SECTION:

Quick Action APIs in Lightning Experience

A `lightning:quickActionAPI` component allows you to access methods for programmatically controlling quick actions on record pages. This component is supported in Lightning Experience and supports utility pop-out. This component requires API version 43.0 and later.

Publisher APIs in Salesforce Classic

The Salesforce Classic Publisher JavaScript API lets your Visualforce pages and components interact with actions you've added to a record page in a Salesforce Classic app for objects that are feed-enabled. The Publisher API works in Salesforce Classic apps with standard navigation and console navigation. For example, you could develop a component that generates customized, pre-written text, adds that text to a new post in the Case Feed portal action, and submits the post to the portal, all with one click.

Quick Action APIs in Lightning Experience

A `lightning:quickActionAPI` component allows you to access methods for programmatically controlling quick actions on record pages. This component is supported in Lightning Experience and supports utility pop-out. This component requires API version 43.0 and later.

For example, if you have a custom Aura component that displays Knowledge articles, you can use the `lightning:quickActionAPI` component to attach and send a Knowledge article from your custom component using the Email quick action on the case record page.

To access these methods, create an instance of the `lightning:quickActionAPI` component inside your Aura component or page and assign an `aura:id` attribute to it.

EDITIONS

Available in: Lightning Experience

Available in: **Group**, **Professional**, **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

```
<lightning:quickActionAPI aura:id="quickActionAPI"/>
```

This component provides similar functionality to the Publisher APIs in Salesforce Classic.

Sample Code

This example creates two buttons that interact with the Update Case quick action on a case record page in Lightning Experience. The controller code uses the following Quick Action API methods: selectAction, setActionFieldValues, and invokeAction.

Component code:

```
<aura:component implements="flexipage:availableForRecordHome" description="My Lightning
Component">
    <lightning:quickActionAPI aura:id="quickActionAPI" />
```

```
    <div>
        <lightning:button label="Select Update Case Action"
onclick="{!c.selectUpdateCaseAction}"/>
        <lightning:button label="Update Case Status Field"
onclick="{!c.updateCaseStatusAction}"/>
    </div>
</aura:component>
```

Controller code:

```
({
    selectUpdateCaseAction : function( cmp, event, helper) {
        var actionAPI = cmp.find("quickActionAPI");
        var args = { actionName :"Case.UpdateCase" };
        actionAPI.selectAction(args).then(function(result) {
            // Action selected; show data and set field values
        }).catch(function(e) {
            if (e.errors) {
                // If the specified action isn't found on the page,
                // show an error message in the my component
            }
        });
    },

    updateCaseStatusAction : function( cmp, event, helper ) {
        var actionAPI = cmp.find("quickActionAPI");
        var fields = { Status : { value : "Closed"},
                       Subject : { value : "Sets by lightning:quickActionAPI component"
},
                       accountName : { Id : "accountId" } };
        var args = { actionName : "Case.UpdateCase",
                     entityName : "Case",
                     targetFields : fields };
        actionAPI.setActionFieldValues(args).then(function() {
            actionAPI.invokeAction(args);
        }).catch(function(e) {
            console.error(e.errors);
        });
    }
})
```

IN THIS SECTION:

Quick Action API Considerations

Before working with the Lightning Quick Action JavaScript API methods, review some considerations that might impact your implementation.

getAvailableActions

Allows custom components to get a list of the available actions on a record page.

getAvailableActionFields

Allows custom components to get a list of the available fields for a specific action on a record page.

getCustomAction

Allows custom components to access a custom quick action and pass data or messages to it.

getSelectedActions
Allows custom components to access selected quick actions on a record page.

invokeAction
Allows custom components to save or submit the quick action on a record page.

refresh
Refreshes the current record page.

selectAction
Allows custom components to select and focus on a quick action on a record page.

setActionFieldValues
Allows custom components to select a quick action on a record page and then specify field values for that action.

SEE ALSO:

Lightning Component Library: lightning:quickActionAPI
Lightning Components Developer Guide

Quick Action API Considerations

Before working with the Lightning Quick Action JavaScript API methods, review some considerations that might impact your implementation.

Tip: The Lightning Quick Action JavaScript APIs can only interact with quick actions that are targetable on the page. Review the following support.

- Targetable: An action that displays in the highlights panel, including the dropdown action overflow
- Targetable: An action that displays in the publisher, including the More overflow
- Targetable: An action that's nested in an accordion component section or tab that's expanded by default
- Not targetable: An action that's nested in an accordion component section or tab that's not expanded by default*

  *The action becomes targetable after a user opens the accordion section or tab containing the action.

If you use the Lightning Quick Action JavaScript APIs in custom code in a Lightning app, the targeted quick actions must be visible on the page. If you target an action that isn't visible on the page, it fails.

The Quick Action APIs work with most action types.

Action Type	Supported?	Notes
Create a Record	Yes	Supported in all Lightning apps, on any object.
Custom Visualforce	Yes	Supported in all Lightning apps, on any object. To work with these action types, use the `getCustomAction` method. Other methods aren't supported for this action type.
Flow	No	Results in error.
Log a Call	Yes	Supported in all Lightning apps, on any object.

Action Type	Supported?	Notes
Aura Component	Yes	Supported in all Lightning apps, on any object. To work with these action types, use the `getCustomAction` method. Other methods aren't supported for this action type.
Send Email	Yes	Supported in all Lightning apps, on any object.
Update a Record	Yes	Supported in all Lightning apps, on any object.

The `lightning:quickActionAPI` component supports utility popout. However, the `getCustomAction` method doesn't work with utility popout yet. The Salesforce Classic Publisher APIs also support utility popout if you place them in a Visualforce page that's used in the utility bar. The `customActionMessage` doesn't support utility popout either.

The Quick Action APIs don't support the following items.

- Opportunity products
- Knowledge articles
- Crew Size field on the Service Crew object
- Social quick action in the case feed publisher provided with Social Customer Service
- Experience Cloud sites—the `lightning:quickActionAPI` component doesn't work in Experience Cloud sites

getAvailableActions

Allows custom components to get a list of the available actions on a record page.

Arguments

None.

Sample Code

```
getAvailableActions : function( cmp, event, helper) {
        var actionAPI = cmp.find("quickActionAPI");
        actionAPI.getAvailableActions().then(function(result){
            //All available actions shown;
        }).catch(function(e){
            if(e.errors){
                //If the specified action isn't found on the page, show an error message
in the my component
            }
        });
    }
```

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: true,
actions:
    {actionName: "Case._LightningUpdateCase", recordId: "recordId", type: "QuickAction"}
    {actionName: "FeedItem.TextPost", recordId: "recordId", type: "QuickAction"}
    {actionName: "Case.LogACall", recordId: "recordId", type: "QuickAction"}
    {actionName: "Case.SendEmail", recordId: "recordId", type: "QuickAction"}
errors: []
```

getAvailableActionFields

Allows custom components to get a list of the available fields for a specific action on a record page.

Arguments

Name	Type	Description
`actionName`	string	The name of the quick action that you want to access.

The `actionName` parameter starts with the Salesforce object, followed by the quick action name. For example:

```
actionName: "Case.LogACall"
```

Sample Code

```
getAvailableActionFields : function( cmp, event, helper) {
        var actionAPI = cmp.find("quickActionAPI");
        var args = {actionName :"Case.LogACall", entityName:"Case" };
        actionAPI.getAvailableActionFields(args).then(function(result){
            //All available action fields shown for Log a Call
        }).catch(function(e){
            if(e.errors){
                //If the specified action isn't found on the page, show an error message
in the my component
            }
        });
    }
```

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: true,
fields:
    {fieldName: "Subject", type: "textEnumLookup"}
    {fieldName: "Description", type: "TextArea"}
```

```
    {fieldName: "WhoId", type: "Lookup"},
errors: []
```

getCustomAction

Allows custom components to access a custom quick action and pass data or messages to it.

Arguments

Name	Type	Description
`actionName`	string	The name of the quick action that you want to access.

The `actionName` parameter starts with the Salesforce object, followed by the quick action name. For example:

```
actionName: "Case.MyCustomAction"
```

Sample Code

```
actionApi.getCustomAction(args).then(function(customAction) {
  if (customAction) {
    customAction.subscribe(function(data) {
      // Handle quick action message
    });
    customAction.publish({
      message : "Hello Custom Action",
      Param1 : "This is a parameter"
    });
  }
}).catch(function(error) {
  // We can't find that custom action.
});
```

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: boolean,
customAction: {
  subscribe: function,
  publish: function,
  unsubscribe: function
},
unavailableAction: boolean,
errors: []
```

getSelectedActions

Allows custom components to access selected quick actions on a record page.

Arguments

None.

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: boolean,
actions: [ {
    actionName: "UpdateCase",
    recordId: "recordId",
    type: "QuickAction"
} ],
errors: []
```

invokeAction

Allows custom components to save or submit the quick action on a record page.

Arguments

Name	Type	Description
`actionName`	string	The name of the quick action that you want to access.

The `actionName` parameter starts with the Salesforce object, followed by the quick action name. For example:

```
actionName: "Case.UpdateCase"
```

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

refresh

Refreshes the current record page.

Arguments

None.

selectAction

Allows custom components to select and focus on a quick action on a record page.

Arguments

Name	Type	Description
`actionName`	string	The name of the quick action that you want to access.

The `actionName` parameter starts with the Salesforce object, followed by the quick action name. For example:

```
actionName: "Case.UpdateCase"
```

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: boolean,
unavailableAction: boolean,
targetableFields: [{
    fieldName: "Status",
    type: "PickList"
}],
actionName: string,
errors: []
```

setActionFieldValues

Allows custom components to select a quick action on a record page and then specify field values for that action.

Because this method also selects the quick action, you don't need to use the `selectAction` method. To submit the quick action updates, pass `submitOnSuccess` as `true`.

Arguments

Name	Type	Description
`actionName`	string	The name of the quick action that you want to access.
`parentFields`	Object	Optional. The fields that you want to update on the current record. For example, if you want to set field values on the Email quick action on the case record page, the case object is the parent record.
`targetFields`	Object	The fields that you want to update on the quick action.
`submitOnSuccess`	boolean	Optional. Set to true if you want to save and submit the quick action after setting the field values. Default is false.

The `actionName` parameter starts with the Salesforce object, followed by the quick action name. For example:

```
actionName: "Case.UpdateCase"
```

The `parentFields` and `targetFields` objects contain a list of field names with values for each field. Each field can optionally specify the insertion behavior using the `insertType` key, which can be `replace` (default), `cursor`, or `begin`. For example:

```
var parentFields = { Status: {value: "Closed"},
                     Subject: {value: "Case subject", insertType: "cursor"} }
var targetFields = { ToAddress: {value: "to@to.com"},
                     TextBody: {value: "the text body", insertType: "cursor"} }
```

We recommend that you don't use this API with the following items:

- Read-only fields
- Encrypted fields
- Fields within social actions

Response

Returns a Promise. Success resolves to a response object. The Promise is rejected on error response.

```
success: boolean,
actionName: "LogACall",
unavailableAction: boolean,
targetFieldErrors: [{
  Status: {message: "error"},
  Subject: {message: "error",
}],
errors: []
```

Publisher APIs in Salesforce Classic

The Salesforce Classic Publisher JavaScript API lets your Visualforce pages and components interact with actions you've added to a record page in a Salesforce Classic app for objects that are feed-enabled. The Publisher API works in Salesforce Classic apps with standard navigation and console navigation. For example, you could develop a component that generates customized, pre-written text, adds that text to a new post in the Case Feed portal action, and submits the post to the portal, all with one click.

Use the `publish` method on the `Sfdc.canvas.publisher` object to allow console components to interact with quick actions.

Tip: Starting with API version 43.0 of the Salesforce Classic JavaScript Publisher API, the methods used in Visualforce components work in Lightning Experience. You can use Visualforce pages in Lightning Experience through custom quick actions, or by adding it to the page in the Lightning App Builder. Just point to the 43.0 version of the Publisher API script in your Visualforce pages.

```
<script src="/canvas/sdk/js/43.0/publisher.js"
type="text/javascript"></script>
```

If you use the JavaScript Publisher API methods in custom code in a Lightning app, the targeted quick actions must be visible on the page. If you target an action that isn't visible on the page, it fails.

EDITIONS

Available in: Salesforce Classic (not available in all orgs)

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

publisher.selectAction

Description	Payload Values	Available Versions
Selects the specified action and puts it in focus.	`actionName`—The action to select. Supported values are: • *`action_name`*–A create, log a call, or custom Visualforce quick action. For example, *`action_name`* for a create contact action might be `create_contact`. • `Case.CaseComment`—Case Feed portal action • `Case.ChangeStatus`—Case Feed change status action • `Case.Email`—Case Feed email action • `Case.LogACall`—Case Feed log a call action • `FeedItem.TextPost`—Standard Chatter post action (Available in API versions 32.0 and later) • ***`SocialPostAPIName`***`.SocialPost`—Social post action (Available in API versions 32.0 and later)	Available in API versions 29.0 and later.

Code Sample

This code snippet selects the email action and puts it in focus.

```
Sfdc.canvas.publisher.publish({name:"publisher.selectAction",payload:{actionName:"Case.Email"}});
```

publisher.setActionInputValues

Description	Payload Values	Available Versions
Specifies which fields on the action should be populated with specific values, and what those values are.	`actionName`—The action on which fields should be populated. The available field values depend on which action you specify. • `emailFields`–Available on `Case.Email`; the standard available fields on the Case Feed email action: – `to` – `cc` – `bcc` – `subject` – `body` – `template` • `portalPostFields`–Available on `Case.CaseComment`; the standard available fields on the Case Feed portal action: – `body` – `sendEmail` (boolean)	Available in API versions 29.0 and later.

Description	Payload Values	Available Versions
	• `targetFields`–Available on `Case.ChangeStatus` , `Case.LogACall`, `FeedItem.TextPost`, and the Social action; the standard available fields on those actions. – On `Case.ChangeStatus`: `commentBody` – On `Case.LogACall`: `description` – On `FeedItem.TextPost`: `body` Attributes on `body` are `value` and `insertType` (optional). Valid values for `insertType` are `begin`, `end`, `cursor`, and `replace`. The default value is `replace`. (Available in API versions 32.0 and later) – On ***`SocialPostAPIName`***`.SocialPost`: `content` and `insertType` (optional). Valid values for `insertType` are `begin`, `end`, `cursor`, and `replace` . The default value is `replace`. (Available in API versions 32.0 and later) • `parentFields`—Available on `Case.ChangeStatus`, `Case.Email`, and `Case.LogACall`; standard and custom fields on case. Lookup fields aren't supported.	

Code Sample

This code snippet populates the fields on an email message with predefined values, and sets the status of the associated case to Closed.

```
Sfdc.canvas.publisher.publish({name:"publisher.setActionInputValues",
    payload:{actionName:"Case.Email",parentFields: {Status:{value:"Closed"}},
   emailFields: {to:{value:"customer@company.com"},cc:{value:"customer2@company.com"},

    bcc:{value:"supervisor@company.com"},
subject:{value:"Your Issue Has Been Resolved"},
    body:{value:"Thank you for working with our support department.
        We've resolved your issue and have closed this ticket, but
        please feel free to contact us at any time if you encounter this
        problem again or need other assistance."}}}});
```

This code snippet inserts the phrase "Hello World" in the body of the Post action at the current cursor position.

```
Sfdc.canvas.publisher.publish({name:"publisher.setActionInputValues",
payload:{actionName:"FeedItem.TextPost", targetFields:{body:{value:"Hello World",
insertType:"cursor"}}}});
```

publisher.invokeAction

Description	Payload Values	Available Versions
Triggers the submit function (such as sending an email or posting a portal comment) on the specified action.	`actionName`—The action on which to trigger the submit function. Supported actions are: • `Case.Email` • `Case.CaseComment` • `Case.ChangeStatus` • `Case.LogACall` • `FeedItem.TextPost` (Available in API versions 32.0 and later) • ***`SocialPostAPIName`***`.SocialPost` (Available in API versions 32.0 and later)	Available in API versions 29.0 and later.

Code Sample

This code snippet triggers the submit function on the email action, sending an email message and generating a related feed item.

```
Sfdc.canvas.publisher.publish({name:"publisher.invokeAction",
payload:{actionName:"Case.Email"}});
```

publisher.customActionMessage

Description	Payload Values	Available Versions
Passes a custom event to a custom action. Supported for Visualforce-based custom actions only.	`actionName`—The Visualforce custom action to pass the event to. `message`–The event to pass to the custom action.	Available in API versions 29.0 and later.

Code Sample

This code snippet passes the Hello world event to the action my_custom_action.

```
Sfdc.canvas.publisher.publish({name:"publisher.customActionMessage",
payload:{actionName:"my_custom_action", message:"Hello world"}});
```

This code snippet is what my_custom_action uses to listen to the Hello world event.

```
Sfdc.canvas.publisher.subscribe([{name : "publisher.customActionMessage", onData :
function(e) {alert(e.message);}}]);
```

publisher.refresh

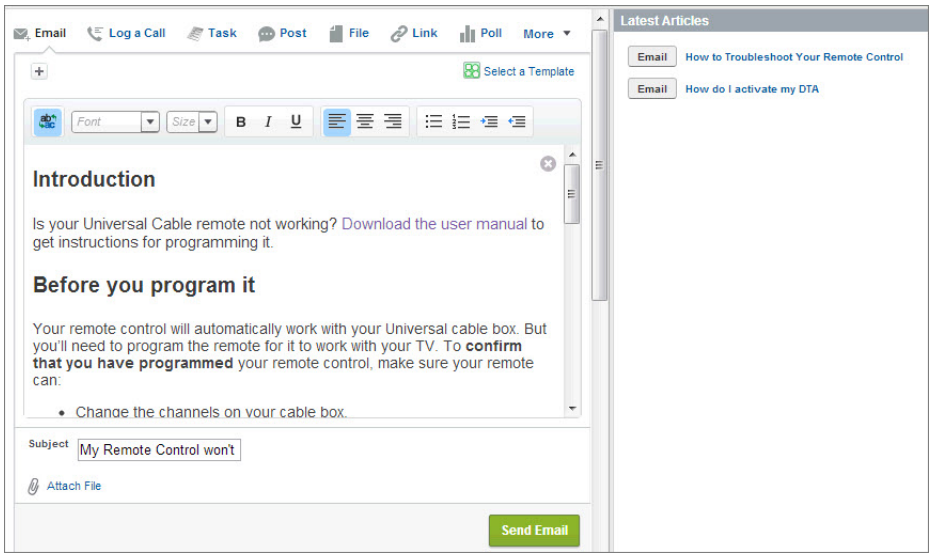
Refreshes the current record page. This method has no arguments.

Use Case and Sample Code

Example: Universal Cable serves millions of phone and cable customers throughout the United States, with 4000 support agents in call centers of varying sizes around the country. Universal wanted to make it easy for agents to access the company's extensive collection of articles in Salesforce Knowledge and share them with customers through email to help keep support costs in check.

Universal used the events on `publish` to create a custom console component that:

- Displays a list of Knowledge articles, from most recently published to oldest.
- Lets agents view an article by clicking its title.
- Lets agents add the full, formatted text of an article to a message in the Case Feed email action by clicking the **Email** button in the console component.



This code sample shows an Apex class containing a custom controller used by the Visualforce page below.

```
public with sharing class KBController {
   public List<FAQ__kav> articles {get; set;}

    public KBController() {
      articles = [select knowledgearticleid, id, title, content__c from FAQ__kav where

      publishstatus = 'Online' and language='en_US' order by lastpublisheddate];
   }
}
```

This code sample shows the Visualforce page that's used as the custom console component in the use case above.

```
<apex:page sidebar="false" controller="KBController">
   <script type='text/javascript' src='/canvas/sdk/js/publisher.js'/>
   <style>
      .sampleTitle { background-color: #99A3AC;color:#FFFFFF;font-size:1.1em;
      font-weight: bold;padding:3px 6px 3px 6px; }
      .sampleHeader { }
      .sampleArticleList { min-width: 250px; padding: 8px 0 5px 0;}
      .sampleUl { padding: 0; margin: 0; list-style: none;}
      .sampleLi { display: block; position: relative; margin: 0;}
```

```
      .sampleRow { min-height: 16px; padding: 4px 10px;}
      .emailBtn { margin: 1px 1px 1px 3px; padding: 3px 8px; color: #333;
        border: 1px solid #b5b5b5; border-bottom-color: #7f7f7f; background: #e8e8e9;

         font-weight: bold; font-size: .9em; -moz-border-radius: 3px;
         -webkit-border-radius: 3px; order-radius: 3px; }
      .emailBtn:active { background-position: right -60px; border-color: #585858;
         border-bottom-color: #939393; }
      .sampleArticle { padding-left: 4px; padding-bottom: 2px; font-weight: bold;
         font-size: 1em; color: #222; }
      .sampleLink { color: #015ba7; text-decoration: none; font-weight: bold;
         font-size: .9em; }
   </style>
   <script>
      function emailArticle(content) {
         Sfdc.canvas.publisher.publish({name: 'publisher.selectAction',
         payload: { actionName: 'Case.Email'}});
         Sfdc.canvas.publisher.publish({name: 'publisher.setActionInputValues',
         payload: {
            actionName: 'Case.Email',
            emailFields: { body: { value:content, format:'richtext', insert: true}}
         }});
     }
   </script>
   <div style="margin-left:-10px;margin-right:-10px;">
      <div class="sampleTitle">Latest Articles</div>
      <div class="sampleHeader" style=""></div>
      <div class="sampleArticleList">
         <apex:repeat value="{!articles}" var="article">
            <ul class="sampleUl">
               <li class="sampleLi">
                  <div class="sampleRow">
                  <div style="display:none;" id="content_{!article.id}">
                     <apex:outputText value="{!article.content__c}" escape="false"/>
                  </div>
                    <input type="button" title="Email" value="Email" class="emailBtn"

                        onclick="emailArticle(document.getElementById
                           ('content_{!article.id}').innerHTML);"/>
                           <span class="sampleArticle">
                               <a href="/{!article.knowledgearticleid}"
                                   title="{!article.title}" class="sampleLink">
                                  {!article.title}</a>
                           </span>
                  </div>
               </li>
            </ul>
         </apex:repeat>
      </div>
   </div>
</apex:page>
```

CUSTOMIZE CASE FEED ACTIONS WITH VISUALFORCE

The Salesforce-provided Case Feed Visualforce components enable you to create a customized page within a Salesforce Classic app. To create custom Salesforce console components that interact with Case Feed actions, publish the Case Feed-related events using the `publish` method on the `Sfdc.canvas.publisher` object in the Salesforce Classic Publisher JavaScript API.

Important: This section of the guides focuses on customizing the Case Feed in a Salesforce Classic console app. However, you can use the Visualforce components in Salesforce Classic apps with standard navigation that use the case object, too. You can also use the Case Feed Visualforce components in Lightning Experience. However, there are some issues with refresh for certain Visualforce components. We recommend that you use these components in Salesforce Classic only.

Requirements

Before customizing Case Feed in the Salesforce console, make sure that:

- Case Feed, Chatter, and feed tracking on cases are enabled in your organization.
- Your organization has at least one Salesforce console app. For more information, see Set Up a Salesforce Console App in Salesforce Classic.
- You're familiar with developing with Visualforce. Check out the Visualforce Developer Guide for a comprehensive overview.

Note: Lookup field filters aren't supported on any of the Case Feed Visualforce components.

Assigning Custom Pages to Users

Generally, when you create a custom Case Feed page using Visualforce, it's not possible to assign that page only to certain users while allowing other users to see the standard Case Feed page. However, with the `support:CaseFeed` component, you can create a page that replicates the standard Case Feed page, assign that page to certain users, and then create a custom page to assign to a different set of users. See Replicating a Standard Case Feed Page for more information.

Customization Overview

Here are the Case Feed Visualforce components.

Component Name	Description
apex:emailPublisher	Displays and controls the appearance and functionality of the Case Feed Email action.
apex:logCallPublisher	Displays and controls the appearance and functionality of the Case Feed Log a Call action.
support:caseArticles	Displays and controls the appearance and functionality of the Articles tool for cases.
support:CaseFeed	Replicates the standard Case Feed page, including all standard actions, links, and buttons.
support:portalPublisher	Displays and controls the appearance and functionality of the Case Feed Portal action.

In addition, the chatter:feed component has two attributes related to Case Feed.

- `feedItemType`: Lets you specify how feed items are filtered.
- `showPublisher`: Lets you display the Chatter publisher on a page.

Here are the Publisher JavaScript APIs.

Method Name	Description
customActionMessage	Passes a custom event to a custom action. Supported for Visualforce-based custom actions only.
invokeAction	Triggers the submit function (such as sending an email or posting a portal comment) on the specified action.
selectAction	Selects the specified action and puts it in focus.
refresh	Refreshes the current record page.
setActionInputValues	Specifies which fields on the action should be populated with specific values, and what those values are.

Customizing the Layout and Appearance of Case Feed

Creating a customized Case Feed page with Visualforce lets you control the overall layout and appearance, including which actions and tools are shown and where they're located on the page. You can also include other standard and custom console components to enhance the functionality of the page.

In addition to the four case-specific Visualforce components detailed in this guide, you can also use the `chatter:feed` component to customize Case Feed. The table below lists its attributes.

`chatter:feed` Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	Entity ID of the record for which to display the feed; for example, Contact.Id	Yes	25.0	
`feedItemType`	String	The feed item type on which the Entity or UserProfileFeed is filtered. See the `Type` field on the FeedItem object listing in the API Object Reference Guide for accepted values.		25.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		20.0	global
`onComplete`	String	The Javascript function to call after a post or comment is added to the feed		25.0	
`rendered`	Boolean	A Boolean value that specifies whether the additional fields defined in the action layout are displayed.		20.0	global

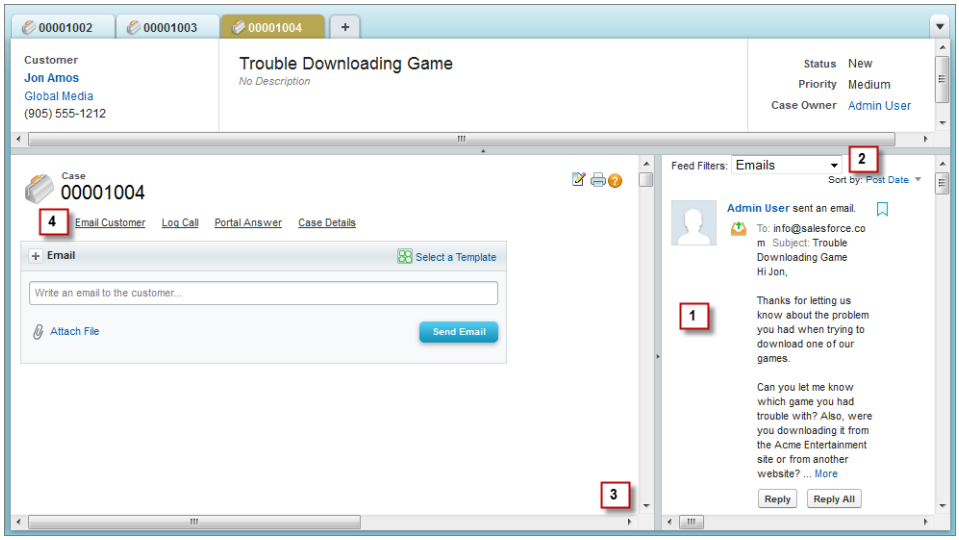
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
`showPublisher`	Boolean	Displays the Chatter publisher.		25.0	

Use Case

Acme Entertainment creates online games used by more than a million people on multiple platforms. Acme's 1500 support agents use desktop computers, laptops, and tablets, and the company wanted to customize the Case Feed page to standardize its look and feel across different devices. They also wanted to make it easier for agents to track case activities using filters.

Acme used these steps to create a customized Case Feed page:

1. Using the chatter:feed component, they positioned the feed in the sidebar so the publisher and other Case Feed tools are always in the center of the page.
2. They repositioned the feed filter and auto-selected default filters depending on case origin:
   - If the case origin is email,. the default filter is Emails.
   - If the case origin is phone, the default filter is Call Logs.
   - If the case origin is Web, the default filter is Portal Answers.
3. In `apex:emailPublisher`, `apex:logCallPublisher`, and `support:portalPublisher`, they made the width percentage-based so the publisher expands and contracts as the size of the page changes, making its appearance more consistent across different screen sizes.
4. They changed the orientation of the publisher action tabs from their standard left-side vertical arrangement to a horizontal arrangement at the top of the page.



Code Sample

This code sample shows a Visualforce page with custom Email, Portal, Log a Call, and Case Details tabs.

```
<apex:page standardController="Case">

    <!-- Repositions publisher tabs to a horizontal arrangement on top of the page -->
    <ul class="demoNav" style="list-style: none; overflow: hidden">
        <li style="float:left">
            <a id="custom_email_tab" class="selected" href="javascript:void(0);"
                onclick="getDemoSidebarMenu().selectMenuItem('custom_email_tab');">
                <span class="menuItem">Email Customer</span>
            </a>
        </li>
        <li style="float:left">
            <a id="custom_log_call_tab" href="javascript:void(0);"
                onclick="getDemoSidebarMenu().selectMenuItem('custom_log_call_tab');">
                <span class="menuItem">Log Call</span>
            </a>
        </li>
        <li style="float:left">
            <a id="custom_portal_tab" href="javascript:void(0);"
                onclick="getDemoSidebarMenu().selectMenuItem('custom_portal_tab');">
                <span class="menuItem">Portal Answer</span>
            </a>
         </li>
        <li style="float:left">
            <a id="custom_detail_tab" href="javascript:void(0);"
                onclick="getDemoSidebarMenu().selectMenuItem('custom_detail_tab');">
                <span class="menuItem">Case Details</span>
            </a>
        </li>
    </ul>

    <!-- Email action -->
    <div id="custom_email_pub_vf">
        <apex:emailPublisher entityId="{!case.id}"
             width="80%"
             emailBodyHeight="10em"
             showAdditionalFields="false"
             enableQuickText="true"
             toAddresses="{!case.contact.email}"
             toVisibility="readOnly"
             fromAddresses="support@cirrus.com"
             onSubmitSuccess="refreshFeed();" />
    </div>

    <!-- Log call action -->
    <div id="custom_log_call_vf" style="display:none">
        <apex:logCallPublisher entityId="{!case.id}"
            width="80%"
            logCallBodyHeight="10em"
            reRender="demoFeed"
            onSubmitSuccess="refreshFeed();" />
    </div>
```

```
<!-- Portal action -->
<div id="custom_portal_vf" style="display:none">
    <support:portalPublisher entityId="{!case.id}"
        width="80%"
        answerBodyHeight="10em"
        reRender="demoFeed"
        answerBody="Dear {!Case.Contact.FirstName},
            \n\nHere is the solution to your case.\n\nBest regards,\n\nSupport"
        onSubmitSuccess="refreshFeed();" />
</div>

<!-- Case detail page -->
<div id="custom_detail_vf" style="display:none">
    <apex:detail inlineEdit="true" relatedList="true" rerender="demoFeed" />
</div>

<!-- Include library for using service desk console API -->
<apex:includeScript value="/support/console/25.0/integration.js"/>

<!-- Javascript for switching publishers -->
<script type="text/javascript">
    function DemoSidebarMenu() {
        var menus = {"custom_email_tab" : "custom_email_pub_vf",
                     "custom_log_call_tab" : "custom_log_call_vf",
                     "custom_portal_tab" : "custom_portal_vf",
                     "custom_detail_tab" : "custom_detail_vf"};

        this.selectMenuItem = function(tabId) {
            for (var index in menus) {
                var tabEl = document.getElementById(index);
                var vfEl = document.getElementById(menus[index]);

                if (index == tabId) {
                    tabEl.className = "selected";
                    vfEl.style.display = "block";
                } else {
                    tabEl.className = "";
                    vfEl.style.display = "none";
                }
            }
         };
    }
    var demoSidebarMenu;
    var getDemoSidebarMenu = function() {
        if (!demoSidebarMenu) {
            demoSidebarMenu = new DemoSidebarMenu();
        }
        return demoSidebarMenu;
    };
</script>

<!-- Javascript for firing event to refresh feed in the sidebar -->
<script type="text/javascript">
```

```
        function refreshFeed() {
            sforce.console.fireEvent
                ('Cirrus.samplePublisherVFPage.RefreshFeedEvent', null, null);
        }
    </script>
</apex:page>
```

The following sample shows an Apex class containing a controller extension to be used with the Visualforce page above.

```
public class MyCaseExtension {
    private final Case mycase;
    private String curFilter;

    public MyCaseExtension(ApexPages.StandardController stdController) {
        this.mycase = (Case)stdController.getRecord();

        // initialize feed filter based on case origin
        if (this.mycase.origin.equals('Email')) {
            curFilter = 'EmailMessageEvent';
        } else if (this.mycase.origin.equals('Phone')) {
            curFilter = 'CallLogPost';
        } else if (this.mycase.origin.equals('Web')) {
            curFilter = 'CaseCommentPost';
        }
    }

    public String getCurFilter() {
        return curFilter;
    }

    public void setCurFilter(String c) {
        if (c.equals('All')) {
            curFilter = null;
        } else {
            curFilter = c;
        }
    }

    public PageReference refreshFeed() {
        return null;
    }
}
```

This sample shows a Visualforce page with custom feed filters and Chatter feed for cases. You can use this page in the sidebar of a Salesforce console.

```
<apex:page standardController="Case" extensions="MyCaseExtension">

    <!-- Feed filter -->
    <div>
        <span>Feed Filters:</span>
        <select onchange="changeFilter(this.options[selectedIndex].value);"
            id="custom_filterSelect">
            <option value="All" id="custom_all_option">All</option>
            <option value="EmailMessageEvent"
```

```
            id="custom_email_option">Emails</option>
        <option value="CaseCommentPost"
            id="custom_web_option">Portal Answers</option>
        <option value="CallLogPost"
            id="custom_phone_option">Call Logs</option>
    </select>
</div>

<apex:form >
    <!-- actionFunction for refreshing feed when the feed filter is updated -->
    <apex:actionFunction action="{!refreshFeed}" name="changeFilter"
        reRender="custom_demoFeed" immediate="true" >
        <apex:param name="firstParam" assignTo="{!curFilter}" value="" />
    </apex:actionFunction>

    <!-- actionFunction for refreshing feed when there is an event fired for
        updating the feed -->
    <apex:actionFunction action="{!refreshFeed}" name="updateFeed"
        reRender="custom_demoFeed" immediate="true" />
</apex:form>

<!-- Chatter feed -->
<chatter:feed entityId="{!case.id}" showPublisher="false"
    feedItemType="{!curFilter}" id="custom_demoFeed" />

<!-- Include library for using service desk console API -->
<apex:includeScript value="/support/console/25.0/integration.js"/>

<!-- Javascript for adding event listener for refreshing feed -->
<script type="text/javascript">

    var listener = function (result) {
        updateFeed();
    };

    // add a listener for the 'Cirrus.samplePublisherVFPage.RefreshFeedEvent'
        event type
    sforce.console.addEventListener('Cirrus.samplePublisherVFPage.RefreshFeedEvent',
        listener);
</script>

<!-- Javascript for initializing select option based on case origin -->
<script type="text/javascript">
    window.onload = function() {
        var caseOrigin = "{!case.origin}";
        if (!caseOrigin) {
            caseOrigin = "all";
        } else {
            caseOrigin = caseOrigin.toLowerCase();
        }
        var selectElem = document.getElementById('custom_' + caseOrigin + '_option');

        if (selectElem) {
            selectElem.selected = true;
```

```
            }
        }
    </script>

</apex:page>
```

Customizing the Email Action

The Email action in Case Feed lets support agents connect with customers via email. With the `apex:emailPublisher` component, you can:

- Customize the dimensions of the Email action.
- Define defaults and visibility for fields.
- Define the visibility and label of the send button.
- Define onSubmit functionality.
- Support email templates and attachments in the action.

Note: The `apex:emailPublisher` component closes a task in Open Activities created by Email-to-Case inbound email.

`apex:emailPublisher` Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`autoCollapseBody`	Boolean	A Boolean value that specifies whether the email body collapses to a small height when it is empty.		25.0	
`bccVisibility`	String	The visibility of the BCC field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
`ccVisibility`	String	The visibility of the CC field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
`emailBody`	String	The default text value of the email body.		25.0	
`emailBodyFormat`	String	The format of the email body can be 'text', 'HTML', or 'textAndHTML'.		25.0	
`emailBodyHeight`	String	The height of the email body in em.		25.0	
`enableQuickText`	Boolean	A Boolean value that specifies whether the Quick Text autocomplete functionality is available in the action.		25.0	
`entityId`	id	Entity ID of the record for which to display the Email action. In the current version, only Case record ids are supported.	Yes	25.0	
`expandableHeader`	Boolean	A Boolean value that specifies whether the header is expandable or fixed.		25.0	
`fromAddresses`	String	A restricted set of from addresses.		25.0	

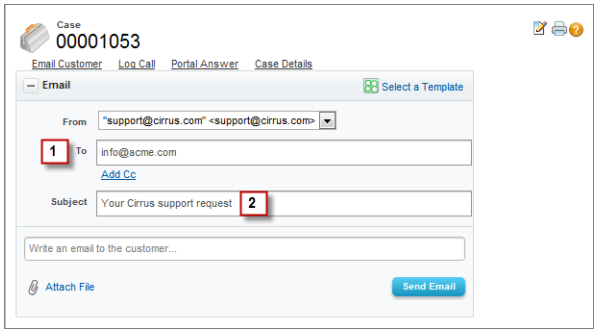
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`fromVisibility`	String	The visibility of the From field can be 'selectable' or 'hidden'.		25.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		25.0	Global
`onSubmitFailure`	String	The JavaScript invoked if the email is not successfully sent.		25.0	
`onSubmitSuccess`	String	The JavaScript invoked if the email is successfully sent.		25.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		25.0	Global
`reRender`	Object	The ID of one or more components that are redrawn when the email is successfully sent. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
`sendButtonName`	String	The name of the send button in the Email action.		25.0	
`showAdditionalFields`	Boolean	A Boolean value that specifies whether the additional fields defined in the action layout are displayed.		25.0	
`showAttachments`	Boolean	A Boolean value that specifies whether the attachment selector is displayed.		25.0	
`showSendButton`	Boolean	A Boolean value that specifies whether the send button is displayed.		25.0	
`showTemplates`	Boolean	A Boolean value that specifies whether the template selector is displayed.		25.0	
`subject`	String	The default value of the Subject.		25.0	
`subjectVisibility`	String	The visibility of the Subject field can be 'editable', 'readOnly', or 'hidden'.		25.0	
`submitFunctionName`	String	The name of a function that can be called from JavaScript to send the email.		25.0	
`title`	String	The title displayed in the Email action header.		25.0	
`toAddresses`	String	The default value of the To field.		25.0	
`toVisibility`	String	The visibility of the To field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
`width`	String	The width of the action in pixels (px) or percentage (%).		25.0	

Use Case

Cirrus Computers, a multinational hardware company with technical support agents in 10 support centers throughout the world, wanted to customize the Email action to increase standardization in outgoing messages and to limit the fields agents could edit.

Cirrus used the `apex:emailPublisher` component to create an Email action that:

- Has read-only To and Subject fields.
- Pre-populates those fields, ensuring consistency and increasing agents' efficiency when writing email messages.



Code Sample

```
<apex:page standardController="Case" >
  <apex:emailPublisher entityId="{!case.id}"
      fromVisibility="selectable"
      subjectVisibility="readOnly"
      subject="Your Cirrus support request"
      toVisibility="readOnly"
      toAddresses="{!case.contact.email}"
      emailBody=""/>
</apex:page>
```

Customizing the Portal Action

The Portal action makes it easy for support agents to compose and post messages to customers on portals. With the `support:portalPublisher` component, you can:

- Customize the dimensions of the Portal action.
- Define a default value for the portal message text.
- Define the visibility and label of the submit button.
- Define onSubmit functionality.

`support:portalPublisher` Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`answerBody`	String	The default text value of the answer body.		25.0	

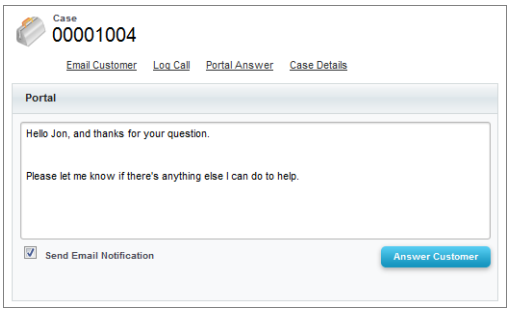
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`answerBodyHeight`	String	The height of the answer body in ems (em).		25.0	
`autoCollapseBody`	Boolean	A Boolean value that specifies whether the answer body is collapsed when it is empty.		25.0	
`entityId`	id	Entity ID of the record for which to display the Portal action. In the current version, only Case record ids are supported.	Yes	25.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		25.0	Global
`onSubmitFailure`	String	The JavaScript invoked if the answer failed to be published to the portal.		25.0	
`onSubmitSuccess`	String	The JavaScript invoked if the answer was successfully published to the portal.		25.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		25.0	Global
`reRender`	Object	The ID of one or more components that are redrawn when the answer is successfully published. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
`showSendEmailOption`	Boolean	A Boolean value that specifies whether the option to send email notification should be displayed.		25.0	
`showSubmitButton`	Boolean	A Boolean value that specifies whether the submit button should be displayed.		25.0	
`submitButtonName`	String	The name of the submit button in the portal action.		25.0	
`submitFunctionName`	String	The name of a function that can be called from JavaScript to publish the answer.		25.0	
`title`	String	The title displayed in the portal action header.		25.0	
`width`	String	The width of the action in pixels (px) or percentage (%).		25.0	

Use Case

The Wellness Group is a healthcare company with 300 support agents in three tiers of support. Wellness wanted to customize the Portal action to reduce the amount of standard text, such as greetings and closings, agents had to type when replying to customers, which would help increase agents' efficiency and improve the standardization of portal communications.

Wellness used the `support:portalPublisher` component to create a Portal action that:

- Pre-populates the message body with a standard opening ("Hello {name}, and thanks for your question.") and a standard closing ("Please let me know if there's anything else I can do to help.").
- Lets agents edit the pre-populated text if needed.



Code Sample

```
<apex:page standardController="Case">
    <support:portalPublisher entityId="{!case.id}" width="800px"
        answerBody="Hello {!Case.Contact.FirstName}, and thanks for your question.
            \n\nPlease let me know if there's anything else I can do to help.">
    </support:portalPublisher>
</apex:page>
```

Customizing the Log a Call Action

The Log a Call action lets support agents record notes and information about customer calls. With the `apex:logCallPublisher`, you can:

- Customize the appearance and dimensions of the Log a Call action.
- Specify which fields are displayed in the action.
- Define the visibility and label of the submit button.
- Define onSubmit functionality.

`apex:logCallPublisher` Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`autoCollapseBody`	Boolean	A Boolean value that specifies whether the Log a Call body is collapsed when it is empty.		25.0	
`entityId`	id	Entity ID of the record for which to display the Log a Call action. In the current version, only Case record ids are supported.	Yes	25.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		25.0	Global

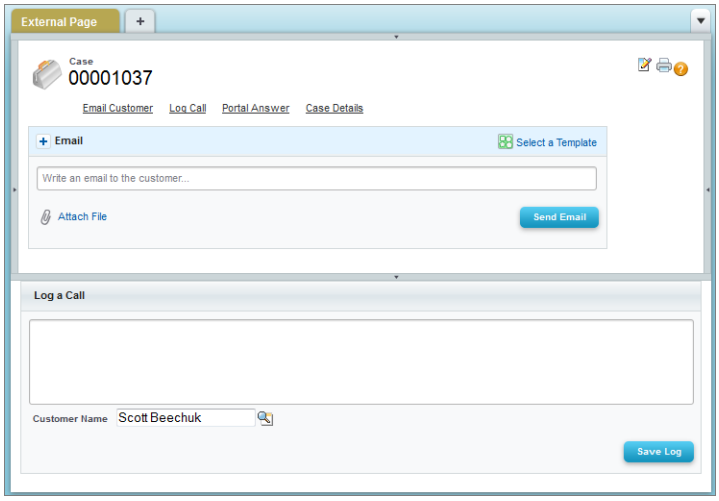
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`logCallBody`	String	The initial text value of the Log a Call body when the action is rendered.		25.0	
`logCallBodyHeight`	String	The height of the Log a Call body in em.		25.0	
`onSubmitFailure`	String	The JavaScript invoked if the call is not successfully logged.		25.0	
`onSubmitSuccess`	String	The JavaScript invoked if the call is successfully logged.		25.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		25.0	Global
`reRender`	Object	The ID of one or more components that are redrawn when the call is successfully logged. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
`showAdditionalFields`	Boolean	A Boolean value that specifies whether the additional fields defined in the action layout should be displayed.		25.0	
`showSubmitButton`	Boolean	A Boolean value that specifies whether the submit button should be displayed.		25.0	
`submitButtonName`	String	The name of the submit button in the Log a Call action.		25.0	
`submitFunctionName`	String	The name of a function that can be called from JavaScript to publish the call log.		25.0	
`title`	String	The title displayed in the Log a Call action header.		25.0	
`width`	String	The width of the action in pixels (px) or percentage (%).		25.0	

Use Case

Stellar Wireless is a mobile phone provider with several high-volume call centers, where agents are rewarded both for solving customers' issues quickly and for keeping detailed, accurate records of customer interactions. Stellar wanted to customize the Log a Call action so it was open and available to agents at all times, even when they were working with another action, giving them a quick and easy way of taking notes about incoming calls.

Stellar used the `apex:logCallPublisher` component to create a Log a Call action that:

- Appears in the footer of the page, replacing the standard interaction log.
- Is open and available by default each time a support agent opens a case.



Code Sample

```
<apex:page standardController="Case">
  <apex:logCallPublisher entityId="{!case.id}"
      width="100%"
      title="Log a Call"
      autoCollapseBody="false"
      showAdditionalFields="false"
      submitButtonName="Save Log" />
</apex:page>
```

After you create a Visualforce page with this code, follow these steps to use the Log a Call action you create as a replacement for the standard interaction log:

1. From the object management settings for cases, go to Page Layouts.
2. Select the layout you're using from the Page Layouts for Case Feed Users list, and then select **Edit detail view**.
3. Click the **Custom Console Components** link at the top of the page.
4. In the Subtab Components section, use the lookup to select the page you created as the component to use for the bottom sidebar.
5. Specify the height of the action.
6. Click **Save**.
7. In the page layout editor, click **Layout Properties**.
8. Uncheck `Interaction Log`.
9. Click **OK**.
10. Click **Save**.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Customizing the Articles Tool

The Articles tool lets support agents browse Salesforce Knowledge articles, see whether articles are attached to a case, and share relevant articles with customers. With the `support:caseArticles` component, you can:

- Customize the appearance and dimensions of the Articles tool.
- Define how the tool's search function works, including which article types and keywords are used by default and whether advanced search is available.
- Specify whether agents can attach articles to emails.

`support:caseArticles` Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`articleTypes`	String	Article types to be used to filter the search. Multiple article types can be defined, separated by commas.		25.0	
`attachToEmailEnabled`	Boolean	A Boolean value that specifies whether articles can be attached to emails.		25.0	
`bodyHeight`	String	The height of the body in pixels (px) or 'auto' to automatically adjust to the height of the currently displayed list of articles.		25.0	
`caseId`	id	Case ID of the record for which to display the case articles.	Yes	25.0	
`categories`	String	Data categories to be used to filter the search. The format of this value should be: 'CatgeoryGroup1:Category1' where CategoryGroup1 and Category1 are the names of a Category Group and a Category respectively. Multiple category filters can be specified separated by commas but only one per category group.		25.0	
`defaultKeywords`	String	The keywords to be used when the `defaultSearchType` attribute is 'keyword'. If no keywords are specified, the Case subject is used as a default.		25.0	
`defaultSearchType`	String	Specifies the default query of the article search form when it is first displayed. The value can be 'keyword', 'mostViewed', or 'lastPublished'.		25.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		25.0	Global
`language`	String	The language used for filtering the search if multilingual Salesforce Knowledge is enabled.		25.0	

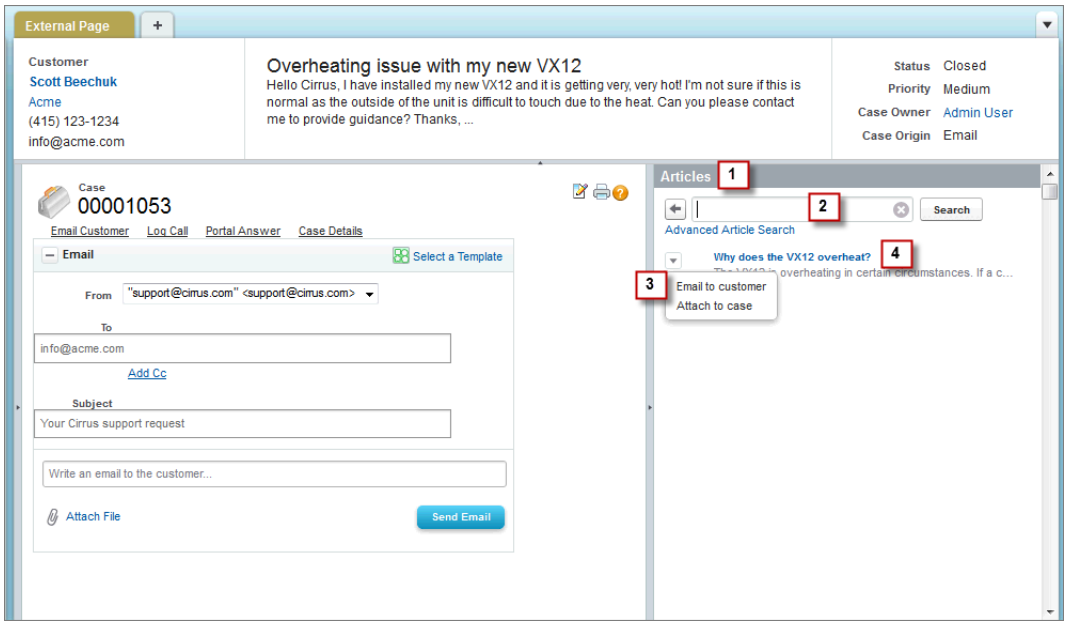
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`logSearch`	Boolean	A Boolean value that specifies whether keyword searches should be logged.		25.0	
`mode`	String	Specifies whether the component displays articles currently attached to the case, an article search form, or both. The value can be 'attached', 'search', 'attachedAndSearch', or 'searchAndAttached'.		25.0	
`onSearchComplete`	String	The JavaScript invoked after an article search has completed.		25.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		25.0	Global
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
`searchButtonName`	String	The display name of the search button.		25.0	
`searchFieldWidth`	String	The width of the keyword search field in pixels (px).		25.0	
`searchFunctionName`	String	The name of a function that can be called from JavaScript to search for articles if the widget is currently in search mode.		25.0	
`showAdvancedSearch`	Boolean	A Boolean value that specifies whether the advanced search link should be displayed.		25.0	
`title`	String	The title displayed in the component's header.		25.0	
`titlebarStyle`	String	The style of the title bar can be 'expanded', 'collapsed', 'fixed', or 'none'.		25.0	
`width`	String	The width of the component in pixels (px) or percentage (%).		25.0	

Use Case

Cirrus Computers wanted to customize the Case Feed articles tool so agents could more easily find articles to help resolve customers' issues.

Cirrus used the `support:caseArticles` component to create an articles tool that:

1. Appears in the right sidebar of the page and is open by default on all case pages.
2. Uses search-as-you-type functionality to show suggested articles quickly.
3. Lets agents attach articles to messages they write with the email action.
4. Displays the most recently published articles when no articles are attached to a case.



Code Sample

```
<apex:page standardController="Case">
    <div style="margin-left:-10px;margin-right:-10px;">
        <div style="background-color: #99A3AC;color:#FFFFFF;font-size:1.1em;font-weight:
            bold;padding:3px 6px 3px 6px;">Articles</div>
        <support:caseArticles caseId="{!case.id}"
            bodyHeight="auto"
            titlebarStyle="none"
            searchButtonName="Search"
            searchFieldWidth="200px"
            defaultSearchType="lastPublished"
        />
    </div>
</apex:page>
```

Replicating a Standard Case Feed Page

The `support:CaseFeed` component includes all of the elements of the standard Case Feed page:

- Email, Portal, Log a Call, and Case Note actions
- Case activity feed
- Feed filters
- Highlights panel
- Case following icon
- Case followers list
- Layout, print, and help links

`support:CaseFeed` Attributes

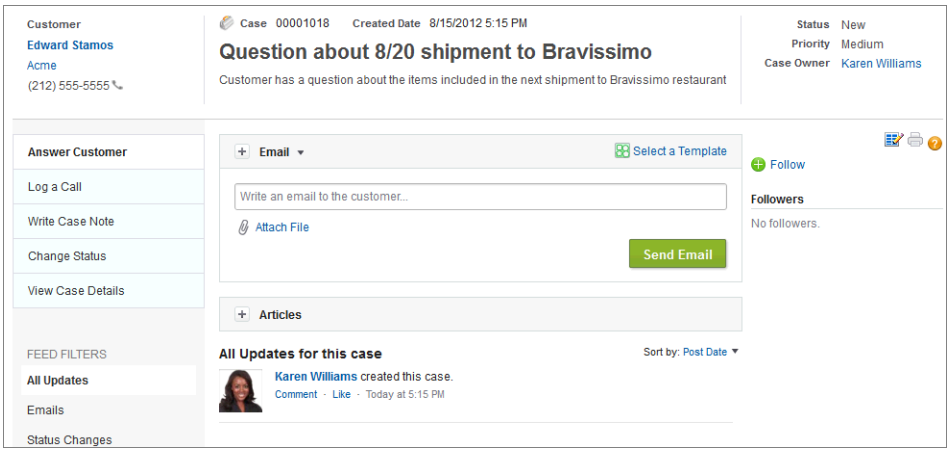
Attribute Name	Attribute Type	Description	Required?	API Version	Access
`caseId`	id	ID of the case record to display in Case Feed.	Yes	26.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		26.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		26.0	global

Use Case

National Foods is a food service company supplying restaurants and corporate cafeterias throughout the United States. National's support operations includes both call center agents who work primarily on desktop computers and field agents who work mainly on mobile devices. The company wanted a simplified Case Feed page that would be easy for its field agents to use, and also wanted to give its call center agents access to the full Case Feed functionality.

National used the `support:CaseFeed` component to recreate the standard Case Feed page for its call center agents working on desktops, and created a custom page for its field agents working on mobile devices.

Standard Case Feed page created with `support:CaseFeed`



Code Sample

```
<apex:page standardController="Case"
    extensions="CasePageSelectorExtension" showHeader="true" sidebar="false">
    <apex:dynamicComponent componentValue="{!casePage}"/>
</apex:page>
```

The following sample shows an Apex class containing a controller extension to be used with the Visualforce page above.

```
public class CasePageSelectorExtension {
    boolean isFieldAgent;
```

```
    String caseId;

    public CasePageSelectorExtension(ApexPages.StandardController controller) {
        List<UserRole> roles = [SELECT Id FROM UserRole WHERE Name = 'FieldAgent'];
        isFieldAgent = !roles.isEmpty() && UserInfo.getUserRoleId() == roles[0].Id;
        caseId = controller.getRecord().id;
    }

    public Component.Apex.OutputPanel getCasePage() {
        Component.Apex.OutputPanel panel = new Component.Apex.OutputPanel();
        if (isFieldAgent) {
            Component.Apex.Detail detail = new Component.Apex.Detail();
            detail.subject = caseId;
            panel.childComponents.add(detail);
        } else {
            Component.Support.CaseFeed caseFeed = new Component.Support.CaseFeed();
            caseFeed.caseId = caseId;
            panel.childComponents.add(caseFeed);
        }
        return panel;
    }
}
```

Create Custom Actions

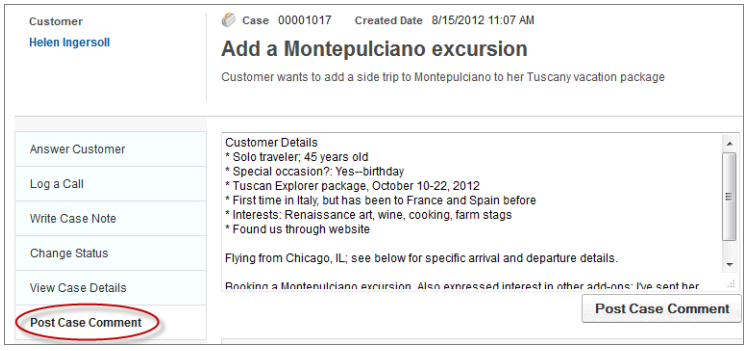
You can create Visualforce pages to use as custom actions in Case Feed. For example, you can create a Map and Local Search action that lets agents look up the customer's location and find nearby service centers.

You can use any Visualforce page that uses the standard case controller as a custom action.

Use Case

Viaggio Italiano is a boutique travel agency specializing in tours of Italy. The company tracks multiple details for each client, including flights, ground transportation specifics, dietary preferences, and itineraries. Viaggio Italiano's agents needed the ability to create long case comments but were limited to 1000 characters for standard case notes. The company wanted a way to bypass this limit.

Viaggio Italiano used Visualforce to create a page that includes the ability to post a case comment, which can be up to 4000 characters long. The company then added the page as a custom action by editing the Case Feed page layout.



Code Samples

The following code sample shows a custom Post Case Comment action for an organization that doesn't have actions in the publisher enabled, or that has actions in the publisher enabled but uses the Case Feed Settings page, not the page layout editor, to choose and configure the actions in the Case Feed publisher.

```
<apex:page standardcontroller="Case"
    extensions="CaseCommentExtension" showHeader="false">
    <apex:includeScript value="/support/api/26.0/interaction.js"/>
    <div>
        <apex:form >
            <!-- Creates a case comment and on complete notifies the Case Feed page
               that a elated list and the feed have been updated -->
            <apex:actionFunction action="{!addComment}" name="addComment" rerender="out"

            oncomplete="sforce.interaction.entityFeed.refreshObject('{!case.id}',
            false, true, true);"/>
            <apex:outputPanel id="out" >
                <apex:inputField value="{!comment.commentbody}" style="width:98%;
                height:160px;" />
            </apex:outputPanel>
        </apex:form><br />
        <button type="button" onclick="addComment();" style="position:fixed; bottom:0px;

        right:2px; padding: 5px 10px; font-size:13px;" id="cpbutton" >Post Case Comment
        </button>
    </div>
</apex:page>
```

This is the code to use for the custom Post Case Comment action if your organization has actions in the publisher enabled and you've opted to use the page layout editor to choose and configure actions in the Case Feed publisher.

```
<apex:page standardcontroller="Case"
    extensions="CaseCommentExtension" showHeader="false">
    <!-- Uses publisher.js rather than interaction.js -->
    <apex:includeScript value="/canvas/sdk/js/28.0/publisher.js"/>
    <div>
        <apex:form >
            <!-- Creates a case comment and on complete notifies the Case Feed page
                that a related list and the feed have been updated -->
            <apex:actionFunction action="{!addComment}" name="addComment" rerender="out"

            <!-- Different oncomplete function using publisher.js -->
            oncomplete="Sfdc.canvas.publisher.publish(
            {name : 'publisher.refresh',  payload :
            {feed: true, objectRelatedLists: {}}});"/>
            <apex:outputPanel id="out" >
                <apex:inputField value="{!comment.commentbody}" style="width:98%;
                height:160px;" />
            </apex:outputPanel>
        </apex:form><br />
        <button type="button" onclick="addComment();" style="position:fixed; bottom:0px;

        right:2px; padding: 5px 10px; font-size:13px;" id="cpbutton" >Post Case Comment
        </button>
```

```
    </div>
</apex:page>
```

The following sample shows an Apex class containing a controller extension to be used with either version of the Visualforce page above.

```
public with sharing class CaseCommentExtension {
    private final Case caseRec;
    public CaseComment comment {get; set;}

    public CaseCommentExtension(ApexPages.StandardController controller) {
        caseRec = (Case)controller.getRecord();
        comment = new CaseComment();
        comment.parentid = caseRec.id;
    }

    public PageReference addComment() {
        insert comment;
        comment = new CaseComment();
        comment.parentid = caseRec.id;
        return null;
    }
}
```

Additional Steps

After creating a Visualforce page, make it available to users.

First, give profiles access to the page:

1. From Setup, enter *Visualforce Pages* in the Quick Find box, then select **Visualforce Pages**.
2. Click **Security** next to the name of the page you created.
3. Choose the profiles you want to be able to access the page.
4. Click **Save**.

Then include the page as a custom action. If you're using the Case Feed Settings page to choose and configure actions:

1. From the object management settings for cases, go to Page Layouts.
2. How you access the Case Feed Settings page depends on what kind of page layout you're working with..
   - For a layout in the Case Page Layouts section, click **Edit**, and then click **Feed View** in the page layout editor.
   - For a layout in the Page Layouts for Case Feed Users section, click the down arrow and choose Edit feed view. (This section appears only for organizations created before Spring '14.)
3. In Custom Actions, click **+ Add a Visualforce page**.
4. Choose the page you want to add.
5. Specify the height of the action. For the best appearance, we recommend a height of 200 pixels.
6. In Select Actions, move the custom action from Available to Selected.
7. Click **Save**.

If you've opted to use the page layout editor to choose and configure actions, you first need to create the custom action:

1. From the object management settings for cases, go to Buttons, Links, and Actions.
2. Click **New Action**.

3. Select `Custom Visualforce`.
4. Select the Visualforce page you created, then specify the height of the action window. (The width is fixed.)
5. Type a label for the action. This is the text users will see for the action in the publisher.
6. If necessary, change the name of the action.
7. Type a description for the action. The description appears on the detail page for the action and in the list on the Buttons, Links, and Actions page. The description isn't visible to your users.
8. Optionally, click **Change Icon** to select a different icon for the action. This icon appears only when you use the action through the API.

Then add the action to a page layout:

1. From the object management settings for cases, go to Page Layouts.
2. How you access the page layout editor depends on what kind of page layout you're working with.
   - For a layout in the Case Page Layouts section, click **Edit**, and then click **Feed View** in the page layout editor.
   - For a layout in the Page Layouts for Case Feed Users section, click the down arrow and choose `Edit detail view`. (This section appears only for organizations created before Spring '14.)
3. Click **Quick Actions** in the palette.
4. Drag the action from the palette to the Quick Actions in the Salesforce Classic Publisher section.
5. Click **Save**.

SEE ALSO:

Salesforce Help: Find Object Management Settings

OTHER RESOURCES

In addition to this guide, there are other resources available for you as you learn how to use the Salesforce Classic Publisher JavaScript API and Lightning Quick Action JavaScript API.

Use these resources to learn more about Aura components, Visualforce, and Case Feed.

- Lightning Aura Components Developer Guide
- Visualforce Developer Guide