# Snap-Ins for Web Developer Guide

Version 43.0, Summer '18

@salesforcedocs

Last updated: May 30, 2018

# CONTENTS

# CHAPTER 1 Snap-Ins for Web Developer Guide

Customize your Snap-ins deployment with customizable parameters in the Snap-ins code snippet, JavaScript, and custom Lightning Components. See which features are supported for each version of the Snap-ins code snippet.

EDITIONS

Snap-ins setup node is available in: Lightning Experience

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

SEE ALSO:

Snap-Ins for Web and Mobile Apps

Snap-Ins for Mobile Apps Product Page

## Customize Your Web Snap-In

You can customize certain parameters in the snap-ins code snippet to alter the size, domain, and language for your snap-in.

## Customize General Parameters in the Snap-Ins Code Snippet

You can customize certain parameters to size your snap-in, set the default language, and set the domain so your snap-in can persist across subdomains.

### Set the Width, Height, and Base Font Size

Set the default sizes for your snap-in.

Keep the following in mind when setting sizes.

- You can enter values in pixels (px) or percent (%), or em or rem.
- When you set the width in your code snippet, the max-width is set to none. Similarly, when you set the height, the max-height is set to none. This action prevents the chat window from auto-sizing if the browser window's height or width changes to less than the set height or width of the chat window.
- If the height of the browser window is less than 498px, the height defaults to 90% of the browser window's height.

### Set the width

```
embedded_svc.settings.widgetWidth = "..."
```

To customize the width of the chat window, add this parameter to your code snippet and set the width you want to show in pixels (px) or percent (%). If you don't specify a value, the default size of 320px is used.

### Set the height

```
embedded_svc.settings.widgetHeight = "..."
```

To customize the height of the chat window, add this parameter to your code snippet and set the height you want to show in pixels (px) or percent (%). If you don't specify a value, the default size of 498px is used.

### Set the font size

```
embedded_svc.settings.widgetFontSize = "..."
```

To customize the base font size for the text in the chat window, add this parameter to your code snippet and set the base font size you want to show. If you don't specify a value, the default size of 16px is used.

Tip: We recommend selecting a size no smaller than 12px and no larger than 24px.

## Set a Domain

This parameter is included as a code comment in your generated code snippet for versions 2.0 and up. When you set the domain, visitors can navigate subdomains during a chat session without losing their chat. Make sure that each page where you want to allow chats contains the code snippet.

```
embedded_svc.settings.storageDomain = "..."
```

To specify the domain for your deployment, set the parameter to whatever top-level domain you use for chats.

Important:

- The `storageDomain` parameter is available only for version 2.0 and later code snippets, and it's required if you want your snap-in to persist across subdomains. If you don't set this parameter, chats use the domain of the container page.
- Follow the format mywebsite.com for your domain. Don't include a protocol (`http://mywebsite.com` or `https://mywebsite.com`) or a trailing slash (`mywebsite.com/`).

## Set the Language

This parameter is included in your generated code snippet and set to English for versions 2.0 and up. To customize the language for a deployment, set the parameter to the desired language.

```
embedded_svc.settings.language = "..."
```

You also need to take care of some other settings to make sure that translation works properly. See the Salesforce Help for guidance.

Note: We don't support an underscore format for languages (like `en_US`). Use `http` locale format (like `en-US` or `en`).

SEE ALSO:

Localization and Translation for Snap-Ins Chat

# Customize Snap-Ins Chat

Take full control of the chat experience from pre-chat to post-chat. Use customizable parameters in the Snap-ins code snippet. Expand pre-chat functionality by passing nonstandard pre-chat details, setting up direct-to-button routing, and enabling pre-chat fields to fill automatically for logged-in customers. Connect an automated invitation to your snap-in and implement your own HTML and CSS.

Set Up and Customize an Automated Invitation

Connect an automated chat invitation with your Snap-ins deployment to proactively invite your customers to start a chat with an agent. Your invitation can slide, fade, or appear anywhere on the page based on the criteria you selected in Setup. You can also use your own HTML and CSS to make it match your company's branding. Upgrade your code snippet to version 4.0 to use invitations.

Create Custom Chat Events

Custom chat events let you have your own communication channel with your customers using the agent console to send and receive your own chat events. Create custom events using your own JavaScript and CSS files.

Enhance the Pre-Chat Page for Snap-Ins Chat

Pass nonstandard pre-chat details, set up direct-to-button routing, and enable pre-chat fields to fill automatically for logged-in customers.

## Customize Snap-Ins Chat Parameters

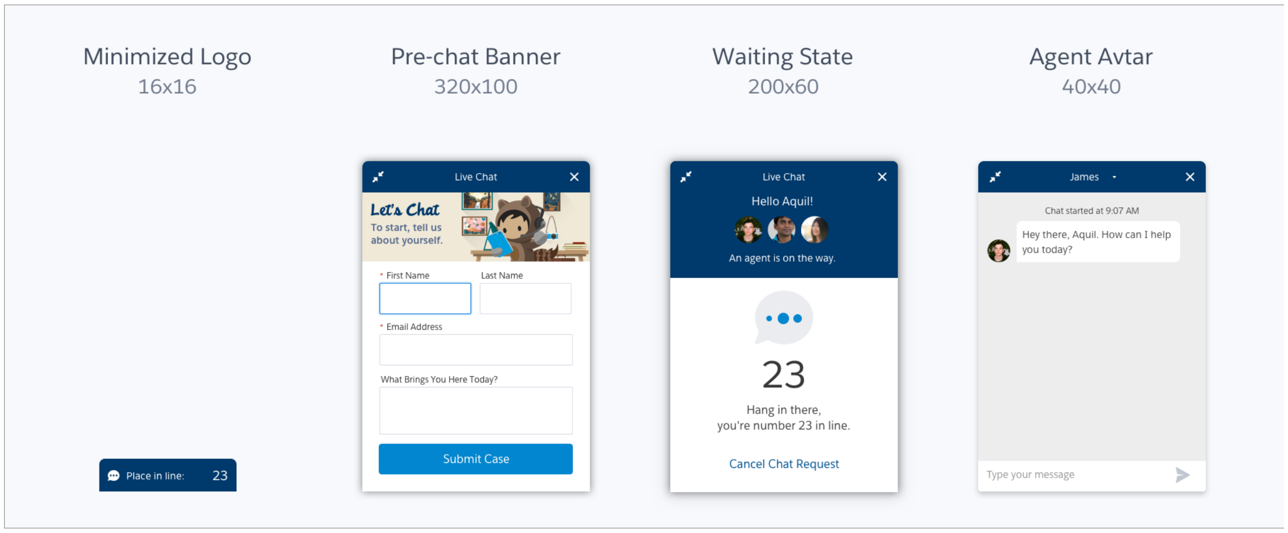
Customize certain parameters that affect the appearance and behavior of the chat snap-in so that the chat experience reflects your company's branding. Use these parameters to customize the pre-chat banner image, logo, waiting state image, and your agent and Einstein Bots avatar pictures. Customize the wording that appears on the chat button and on the snap-in when the chat is loading, when agents are online, and when agents are offline. Set a routing order and load your files for custom chat events.

### Specify Chat Images

Use your own images for the pre-chat banner, logo on the waiting state when the snap-in is minimized, image on the waiting state, and the agent and Einstein Bots avatars.

If the images are hosted in the same repository as the web page where you plan to add the chat snap-in, you can use either relative URL paths and names or full URLs. If the images are hosted elsewhere, use the full URLs for the images.

Before customizing the code, upload the image files that you want to use in the snap-in. Create your images in `.png` format and use the following sizes to ensure that the images don't become distorted during the chat experience.



## Pre-chat banner image

`embedded_svc.settings.prechatBackgroundImgURL = "..."`

Specify a URL to set the image shown in the pre-chat form between the greeting (for example, “Hello Guest!”) and the subtext (for example, “An agent is on the way.”).

Tip: We recommend an image that’s 320x100 pixels.

## Logo for minimized snap-in in waiting state

`embedded_svc.settings.smallCompanyLogoImgURL = "..."`

Specify a URL to set the logo shown when the chat is minimized.

Tip: We recommend an image size that’s 25x25 pixels.

## Waiting state image

`embedded_svc.settings.waitingStateBackgroundImgURL = "..."`

Specify a URL to set the background image when the chat is in a waiting state.

Tip: We recommend an image that’s 200x60 pixels.

## Agent avatar

`embedded_svc.settings.avatarImgURL = "..."`

Specify a URL to set the image of the agent that appears when the agent is chatting. If your snap-in uses an automated invitation, this image appears in the top left of the invitation with the default HTML and CSS in the snippet.

Tip: We recommend an image size that’s 40x40 pixels.

### Einstein Bots avatar

`embedded_svc.settings.chatbotAvatarImgURL = “...”`

Specify a URL to set the avatar image that appears when the customer is chatting with a bot.

Tip: We recommend an image size that’s 40x40 pixels.

## Display the Default Chat Button

The default chat button connects your customers to the snap-in so they can start a chat from your web page. Valid values are `true` and `false`.

`embedded_svc.settings.displayHelpButton = "..."`

To display the default chat button, set this parameter to `true`. The chat button lets your customers start a chat from your web page.

## Customize the Online, Offline, and Loading Chat Text

Set the text that’s displayed to your customers in the snap-in when there are agents available, when there aren’t agents available, and when the chat is connecting to an agent.

### Online text

`embedded_svc.settings.defaultMinimizedText = "..."`

To customize the text that appears in the chat button when agents are online, set the parameter to whatever text you want to show. If you don’t specify a value, the default text “Chat with an Expert” is used.

Important: The `defaultMinimizedText` parameter is available only for version 3.0 and later code snippets. If you’re using an earlier code snippet version, use the `onlineText` parameter.

### Offline text

`embedded_svc.settings.disabledMinimizedText = "..."`

To customize the text that appears in the chat button when agents are offline, set the parameter to whatever text you want to show. If you don’t specify a value, the default text “Agent Offline” is used.

Important: The `disabledMinimizedText` parameter is available only for version 3.0 and later code snippets. If you’re using an earlier code snippet version, use the `offlineText` parameter

### Offline support text

`embedded_svc.settings.offlineSupportMinimizedText = “...”`

To customize the text that appears in the chat button when agents are offline and the Snap-ins deployment has offline support enabled, set the parameter to whatever text you want to show. If you don’t specify a value, the default text “Contact Us” is used.

### Loading text

`embedded_svc.settings.onlineLoadingText = "..."`

To customize the text that appears in the chat button when the chat window is loading, set the parameter to whatever text you want to show. If you don't specify a value, the default text "Loading" is used.

## Set a Routing Order

Set a list of user IDs and button IDs on your Snap-ins deployment to replace the assigned chat button. When a customer requests a chat, it's routed to the first available user or button in the list.

```
embedded_svc.settings.fallbackRouting = ["...", “...”]
```

With version 5.0 and later code snippets, you have the following parameter available as a code comment.

```
//embedded_svc.settings.fallbackRouting = []; //An array of button IDs, user IDs, or
userId_buttonId
```

Accepted values are:

- `userId`
- `buttonId`
- `userId_buttonId`

Tip: This parameter overrides the assigned chat button you've set for the Snap-ins deployment in Setup. If the users and buttons you include in your array aren't available, the chat isn't routed to the assigned button. We recommend including your assigned button's ID at the end of your array.

## Load Files for Custom Chat Events

Load your own JavaScript and CSS files into Snap-ins Chat to handle and style custom chat events. Custom events are supported in console apps created in Salesforce Classic only.

Upload your files as Static Resources with the cache control set to public. Give them names that are easy to remember and type and with no spaces, because you reference them by static resource name in Snap-ins Chat.

### Load a JavaScript file

```
embedded_svc.settings.externalScripts = ["...", “...”]
```

Specify your resources using the static resource name, not the file name itself. For example, if you upload CustomEvent.js and give it the name CustomEvent, enter *`CustomEvent`*.

### Load a CSS file

```
embedded_svc.settings.externalStyles = ["...", “...”]
```

Specify your resources using the static resource name, not the file name itself. For example, if you upload CustomEvent.css and give it the name CustomEventCSS, enter *`CustomEventCSS`*.

SEE ALSO:

Create Custom Chat Events

## Set Up and Customize an Automated Invitation

Connect an automated chat invitation with your Snap-ins deployment to proactively invite your customers to start a chat with an agent. Your invitation can slide, fade, or appear anywhere on the page based on the criteria you selected in Setup. You can also use your own HTML and CSS to make it match your company's branding. Upgrade your code snippet to version 4.0 to use invitations.

Before you write any code, set up your invitation and Snap-ins deployment in Setup. From Salesforce Classic Setup, enter *`Chat Buttons`* in the Quick Find box, then select **Chat Buttons & Invitations**. Connect the invitation to the Live Agent deployment that you plan to use for your Snap-ins deployment. Then create or edit a Snap-ins deployment and select the Live Agent deployment and invitation for your Live Agent Settings.

Keep in mind that there are some differences to how invitations work in Snap-ins Chat versus Live Agent:

- The position and animation don't apply to customers using a mobile browser. They see the invitation above the snap-in with the "fade" animation type.
- Custom animations aren't supported.
- The same fields that aren't supported for Snap-ins chat buttons aren't supported for Snap-ins invitations: Pre-Chat Form Page, Pre-Chat Form URL, Custom Chat Page, Invitation Image, and Site for Resources.
- You can't use invitations with the Snap-ins component in your Community.

When you have an invitation in your deployment and a 4.0 code snippet, there's a section of the code snippet that begins with `<!-- Invitations - Static HTML/CSS/JS -->`. This is where you'll define some behavior for the invitation and add your own HTML and CSS (if you want to).

If you use your own HTML and CSS, remember the following:

- Wrap the HTML properly: `<div  id=”snapins_invite></div>`. The `<div>` must also have a CSS property of `visibility: hidden` to ensure that the animations and rules work as you specified in Setup.
- We generate the default HTML and CSS for you when you add an invitation to your Snap-ins deployment and regenerate the snippet. The default invitation uses the font, primary color, and secondary color that you selected in Snap-ins setup.
- If you set an avatar image by defining `embedded_svc.settings.avatarImgURL` in your code snippet, the image appears in the top left of the invitation with the default HTML and CSS.

When you use invitations, there are two JavaScript functions that you need to override in the Snap-ins code snippet:

- `embedded_svc.inviteAPI.inviteButton.acceptInvite()`
- `embedded_svc.inviteAPI.inviteButton.rejectInvite()`

If you're using custom variable rules, then you should also use this function:

- `embedded_svc.inviteAPI.inviteButton.setCustomVariable()`

## Automated Invitation Code Example

The following code example shows the default HTML, CSS, and JavaScript functions in the code snippet. This code is included in your code snippet when you add an invitation to a Snap-ins deployment and regenerate the code snippet (which upgrades it to version 4.0).

```
<!-- Start of Invitations -->
<div class="embeddedServiceInvitation" id="snapins_invite" aria-live="assertive"
role="dialog" aria-atomic="true">
    <div class="embeddedServiceInvitationHeader" aria-labelledby="snapins_titletext"
aria-describedby="snapins_bodytext">
        <img id="embeddedServiceAvatar">
        <span class="embeddedServiceTitleText" id="snapins_titletext">Need help?</span>
        <button type="button" id="closeInvite" class="embeddedServiceCloseIcon"
aria-label="Exit invitation">&times;</button>
    </div>
    <div class="embeddedServiceInvitationBody">
        <p id="snapins_bodytext">How can we help you?</p>
    </div>
    <div class="embeddedServiceInvitationFooter" aria-describedby="snapins_bodytext">
```

```
        <button type="button" class="embeddedServiceActionButton"
id="rejectInvite">Close</button>
        <button type="button" class="embeddedServiceActionButton" id="acceptInvite">Start
 Chat</button>
    </div>
</div>

<style type='text/css'>
    #snapins_invite { background-color: #FFFFFF; font-family: "Salesforce Sans", sans-serif;
 overflow: visible; border-radius: 8px; visibility: hidden; }
    .embeddedServiceInvitation { background-color: transparent; max-width: 290px; max-height:
 210px; -webkit-box-shadow: 0 7px 12px rgba(0,0,0,0.28); -moz-box-shadow: 0 7px 12px
rgba(0,0,0,0.28); box-shadow: 0 7px 12px rgba(0,0,0,0.28); }
    @media only screen and (min-width: 48em) { /*mobile*/ .embeddedServiceInvitation {
max-width: 332px; max-height: 210px; } }
    .embeddedServiceInvitation > .embeddedServiceInvitationHeader { width: inherit; height:
 32px; line-height: 32px; padding: 10px; color: #FFFFFF; background-color: #222222; overflow:
 initial; display: flex; justify-content: space-between; align-items: stretch;
border-top-left-radius: 8px; border-top-right-radius: 8px; }
    .embeddedServiceInvitationHeader #embeddedServiceAvatar { width: 32px; height: 32px;
border-radius: 50%; }
    .embeddedServiceInvitationHeader .embeddedServiceTitleText { font-size: 18px; color:
#FFFFFF; overflow: hidden; word-wrap: normal; white-space: nowrap; text-overflow: ellipsis;
 align-self: stretch; flex-grow: 1; max-width: 100%; margin: 0 12px; }
    .embeddedServiceInvitationHeader .embeddedServiceCloseIcon { border: none; border-radius:
 3px; cursor: pointer; position: relative; bottom: 3%; background-color: transparent;
width: 32px; height: 32px; font-size: 23px; color: #FFFFFF; }
    .embeddedServiceInvitationHeader .embeddedServiceCloseIcon:focus { outline: none; }
    .embeddedServiceInvitationHeader .embeddedServiceCloseIcon:focus::before { content: "
 "; position: absolute; top: 11%; left: 7%; width: 85%; height: 85%; background-color:
rgba(255, 255, 255, 0.2); border-radius: 4px; pointer-events: none; }
    .embeddedServiceInvitationHeader .embeddedServiceCloseIcon:active,
.embeddedServiceCloseIcon:hover { background-color: #FFFFFF; color: rgba(0,0,0,0.7);
opacity: 0.7; }
    .embeddedServiceInvitation > .embeddedServiceInvitationBody { background-color: #FFFFFF;
 max-height: 110px; min-width: 260px; margin: 0 8px; font-size: 14px; line-height: 20px;
overflow: auto; }
    .embeddedServiceInvitationBody p { color: #333333; padding: 8px; margin: 12px 0; }
    .embeddedServiceInvitation > .embeddedServiceInvitationFooter { width: inherit; color:
 #FFFFFF; text-align: right; background-color: #FFFFFF; padding: 10px; max-height: 50px;
border-bottom-left-radius: 8px; border-bottom-right-radius: 8px; }
    .embeddedServiceInvitationFooter > .embeddedServiceActionButton { font-size: 14px;
max-height: 40px; border: none; border-radius: 4px; padding: 10px; margin: 4px; text-align:
 center; text-decoration: none; display: inline-block; cursor: pointer; }
    .embeddedServiceInvitationFooter > #acceptInvite { background-color: #005290; color:
#FFFFFF; }
    .embeddedServiceInvitationFooter > #rejectInvite { background-color: #FFFFFF; color:
#005290; }
</style>

<script type='text/javascript'>
    (function() {
        document.getElementById('closeInvite').onclick = function() {
embedded_svc.inviteAPI.inviteButton.rejectInvite(); };
```

```
        document.getElementById('rejectInvite').onclick = function() {
embedded_svc.inviteAPI.inviteButton.rejectInvite(); }; // use this API call to reject
invitations
        document.getElementById('acceptInvite').onclick = function() {
embedded_svc.inviteAPI.inviteButton.acceptInvite(); }; // use this API call to start chat
 from invitations
        document.addEventListener('keyup', function(event) { if (event.keyCode == 27) {
embedded_svc.inviteAPI.inviteButton.rejectInvite(); }})
    })();
</script>
<!-- End of Invitations -->
```

Note: The provided code sample uses object field names, org IDs, button IDs, and stylesheets that possibly don't work in your Snap-ins implementation. Make sure that you replace the information with your own when you use this sample.

## Create Custom Chat Events

Custom chat events let you have your own communication channel with your customers using the agent console to send and receive your own chat events. Create custom events using your own JavaScript and CSS files.

To create your events, you need to use the following.

- `sendCustomEvent()`—Sends a custom event to the client-side chat window for a chat with a specific chat key.
- `onCustomEvent()`—Registers a function to call when a custom event takes place during a chat.
- `embedded_svc.liveagentAPI.sendCustomEvent()`—Sends a custom event to the agent console of the agent who is currently chatting with a customer.
- `embedded_svc.liveagentAPI.getCustomEvents()`—Retrieves a list of custom events from both agent and chasitor that have been received during this chat session.
- `embedded_svc.liveagentAPI.addCustomEventListener()`—Registers a function to call when a custom event is received in the chat window.

You don't have to add your own styling (our example doesn't include any), but we recommend it.

Note: We include two samples, one for sending an event from the agent to the customer and one for sending an event from the customer to the agent. If you want to create both events for your snap-in, combine them into one JavaScript file.

### Send an Event from an Agent to a Customer

The following example shows how you can create a custom event that's sent from the agent to the customer. This example creates a link on a Visualforce page that, when clicked, sends an event to the customer.

1. Create a Visualforce page to send an event from an agent to a customer.

   In this example, a custom event of type `agent_to_customer_event_type` is sent to the customer, with the data `data from the agent`.

```
<apex:page>
 <apex:includeScript value="/support/console/42.0/integration.js" />

 <a href="#" onClick="sendEventFromAgentToCustomer();">Send an event from an agent to
a customer</a>

 <script type="text/javascript">
```

```
  function sendEventFromAgentToCustomer() {
   var chatKey = undefined; // Provide a chat key here
   var eventType = "agent_to_customer_event_type";
   var eventData = "data from the agent";

   sforce.console.chat.sendCustomEvent(chatKey, eventType, eventData, function(result)
 {
    if (!result || !result.success) {
     console.log("Sending an event from an agent to a customer failed");
     return;
    }

    console.log("The customEvent (" + eventType + ") has been sent");
   });
  }
 </script>
</apex:page>
```

2. Provide a chat key using an external JavaScript file.

   In this example, the JavaScript file is called `CustomEvents_fromAgentToCustomer.js`.

```
function customEventReceived(result) {
 var eventType;
 var eventData;

 if (!result || !result.success) {
  console.log("customEventReceived failed");
  return;
 }

 eventType = result.type;
 eventData = JSON.stringify(result.data);
 console.log("A custom event of type '" + eventType + "' has been received with the
following data: " + eventData);
}

function wireUpCustomEventListeners() {
 embedded_svc.liveAgentAPI.addCustomEventListener("agent_to_customer_event_type",
customEventReceived);
}

wireUpCustomEventListeners();
```

3. Upload your file as a static resource in Salesforce. Give it a name that's easy to remember and doesn't include spaces.

   In this example, the static resource name for the JavaScript file is `CustomEvents_fromAgentToCustomer`.

4. Add your file to the Snap-ins code snippet (make sure you're using version 5.0 or later).

   Enter the static resource name, not the file name.

```
embedded_svc.settings.externalScripts = ["CustomEvents_fromAgentToCustomer"];
```

5. Add the Visualforce page to the console.

   From the console, add a tab and paste the URL of the preview page of the Visualforce page your created in step 1.

6. Test your new event.

   In this example, there's a "Send an event from an agent to a customer" link in your Visualforce page. Reload your Visualforce page console tab and snap-in, start a chat, and click the link in your Visualforce page console tab. The link triggers a call to `sendEventFromAgentToCustomer()` and sends the event to the customer.

## Send an Event from a Customer to an Agent

The following example shows how you can create a custom event that's sent from the customer to the agent. This example creates an event listener that, when the customer's chat message field matches "trigger", sends an event to the agent.

1. Create a Visualforce page to send an event from a customer to an agent.

   In this example, a custom event of type `customer_to_agent_event_type` is sent to the customer, with the data `data from the customer`.

```
<apex:page>
 <apex:includeScript value="/support/console/42.0/integration.js" />

 <script type="text/javascript">
  function registerListeners() {
   var chatKey = undefined; // Provide a chat key here
   var eventType = "customer_to_agent_event_type";

   sforce.console.chat.onCustomEvent(chatKey, eventType, function(result) {
    if (!result || !result.success) {
     console.log("onCustomEvent (" + eventType
      + ") was not successful");
     return;
    }

    console.log("A new custom event has been received of type "
     + result.type + " and with data: " + result.data);
   });
  }

  registerListeners();
 </script>
</apex:page>
```

2. Provide a chat key using an external JavaScript file.

   In this example, the JavaScript file is called `CustomEvents_fromCustomerToAgent.js`.

```
function wireTextChangeListner() {
 // Find the chasitor's chat input field
 var obj = document.getElementsByClassName('chasitorText');

 // Wire up the event listener
 obj[0].oninput = function() {
  switch(this.value) {
   // When the chasitor types "trigger", an event will be fired to the agent
   case "trigger":
    embedded_svc.liveAgentAPI.sendCustomEvent(
     "Customer_to_agent_event_type",
     "data from the customer");
    break;

   default:
    break;
  }
 };
}

wireTextChangeListner();
```

3. Upload your file as a static resource in Salesforce. Give it a name that's easy to remember and doesn't include spaces.

   In this example, the static resource name for the JavaScript file is `CustomEvents_fromCustomerToAgent`.

4. Add your file to the Snap-ins code snippet (make sure you're using a version 5.0 or later).

   Enter the static resource name, not the file name.

   ```
   embedded_svc.settings.externalScripts = ["CustomEvents_fromCustomerToAgent"];
   ```

5. Add the Visualforce page to the console.

   From the console, add a tab and paste the URL of the preview page of the Visualforce page you created in step 1.

6. Test your new event.

   Reload your Visualforce page and snap-in, start a chat, and enter *`trigger`* in the chat message field as the customer. The event listener you created watches for changes in the chat message field, and the event is sent when the field matches *`trigger`*.

Snap-Ins Chat Custom Event APIs
There are three APIs that let you create custom chat events with Snap-ins Chat. Available using Snap-ins code snippet version 5.0 and later.

SEE ALSO:

Load Files for Custom Chat Events

Snap-Ins Chat Custom Event APIs

## Snap-Ins Chat Custom Event APIs

There are three APIs that let you create custom chat events with Snap-ins Chat. Available using Snap-ins code snippet version 5.0 and later.

### `embedded_svc.liveagentAPI.sendCustomEvent()`

Sends a custom event to the agent console of the agent who is currently chatting with a customer.

**Table 1: Arguments**

| Name | Type | Description |
|---|---|---|
| `data` | string | Additional data you want to send to the agent console along with the custom event. |

| Name | Type | Description |
|---|---|---|
| `type` | string | The name of the custom event to send to the agent console. |

### `embedded_svc.liveagentAPI.getCustomEvents()`

Retrieves a list of custom events from both agent and chasitor that have been received during this chat session.

**Table 2: Arguments**

| Name | Type | Description |
|---|---|---|
| `callback` | function | JavaScript method called upon completion of the method. It passes a JSON formatted string of the events. |

### `embedded_svc.liveagentAPI.addCustomEventListener()`

Registers a function to call when a custom event is received in the chat window.

**Table 3: Arguments**

| Name | Type | Description |
|---|---|---|
| `callback` | function | JavaScript method called upon completion of the method. It passes a JSON formatted string of the events. |
| `type` | string | The type of custom event you want to listen for. |

SEE ALSO:

Create Custom Chat Events

Load Files for Custom Chat Events

## Enhance the Pre-Chat Page for Snap-Ins Chat

Pass nonstandard pre-chat details, set up direct-to-button routing, and enable pre-chat fields to fill automatically for logged-in customers.

Pass Nonstandard Pre-Chat Details

Further control the pre-chat experience using parameters in your Snap-ins code snippet. Two parameters relate to the pre-chat experience: `extraPrechatFormDetails` and `extraPrechatInfo`. With these parameters, you can send information to the agent and to your org beyond what's shown on the pre-chat form.

Pre-Chat Code Examples

The following examples illustrate some common use cases for pre-chat code snippets.

Route Chats Based on Pre-Chat Responses with Direct-to-Button Routing

Set your snap-in to route chats to different chat buttons based on the customer's pre-chat response on any and all of your pre-chat fields. Available when you upgrade your code snippet to version 4.0.

Set Certain Pre-Chat Form Fields to Automatically Populate when Customers are Logged In

When your users are logged in, you already know basic information like their name and email address. Use this array in your 4.0 code snippet to populate relevant pre-chat fields for them. You can mix and match fields for different record types. This information is for snap-ins that are placed outside of Salesforce with Lightning Out (beta). If you use your snap-in inside Communities, you can enable the contact fields to fill in automatically in the Snap-ins Chat component settings.

## Pass Nonstandard Pre-Chat Details

Further control the pre-chat experience using parameters in your Snap-ins code snippet. Two parameters relate to the pre-chat experience: `extraPrechatFormDetails` and `extraPrechatInfo`. With these parameters, you can send information to the agent and to your org beyond what's shown on the pre-chat form.

### `extraPrechatFormDetails`

This parameter allows you to send the agent more information and add information to the chat transcript.

The following sample code illustrates the most common fields for `extraPrechatFormDetails`.

```
embedded_svc.settings.extraPrechatFormDetails = [{
  "label": "First Name",
  "value": "John",
  "displayToAgent": true
}, {
  "label": "Last Name",
  "value": "Doe",
  "displayToAgent": true
}, {
  "label": "Email",
  "value": "john.doe@salesforce.com",
  "displayToAgent": true
}];
```

The following properties can be used in this JSON parameter.

| Name | Type | Description |
|---|---|---|
| `label` | String | The label for this field. |
| `value` | String | The value for this field. |
| `displayToAgent` | Boolean | Whether to display this label and value to the agent. |
| `transcriptFields` | Array of Strings | Names of the fields on the chat transcript record to which to save this value. |

### `extraPrechatInfo`

This parameter lets you map the pre-chat form details from the `extraPrechatFormDetails` parameter to fields in existing or new records. The information in this parameter is merged with the information already specified from your org's setup.

Important: If the label name is the same in the pre-chat setup in your org and in the `extraPrechatInfo` parameter, the information in the `extraPrechatInfo` parameter takes precedence.

The following sample code illustrates the most common fields for `extraPrechatInfo`.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate": false,
    "doFind": true,
    "fieldName": "LastName",
    "isExactMatch": true,
    "label": "Last Name"
  }, {
    "doCreate": false,
    "doFind": true,
    "fieldName": "Email",
    "isExactMatch": true,
    "label": "Email"
  }],
  "entityName": "Contact"
}];
```

The following properties can be used in this JSON parameter.

| Name | Type | Description |
|---|---|---|
| `entityName` | String | The type of record to search for or create. |
| `entityFieldMaps` | Array of Objects | The list of fields within the record type specified in `entityName`. |
| `entityFieldMaps.fieldName` | String | The name of the field. |
| `entityFieldMaps.label` | String | The label of the field in the pre-chat form. This value *must* match the label in the `extraPrechatFormDetails` parameter. |
| `entityFieldMaps.doCreate` | Boolean | Whether to create if it doesn't exist. |
| `entityFieldMaps.doFind` | Boolean | Whether to search for the field if not an exact match. |
| `entityFieldMaps.isExactMatch` | Boolean | Whether a field value must match the field value in an existing record when you conduct a search with the `findOrCreate.map` API method. See findOrCreate.map.isExactMatch in the Live Agent Developer Guide. |

| Name | Type | Description |
|---|---|---|
| `saveToTranscript` | String | The name of the transcript field to which to save the record. This field needs to be a standard lookup field or a custom field with a lookup relationship. If you don't want to attach contact records to the transcript, set `saveToTranscript` to an empty string. |
| `linkToEntityName` | String | The name of the related object you want to link this object to. If you don't want the default link between a contact and a case, set `linkToEntityName` to an empty string. |
| `linkToEntityField` | String | The name of the field (within the related object specified in `linkToEntityName`) that you want to link this object to. |

SEE ALSO:

Customize the Pre-Chat Form

Find and Create Records Automatically with the Pre-Chat APIs

## Pre-Chat Code Examples

The following examples illustrate some common use cases for pre-chat code snippets.

### Find contacts but don't create new ones

In this example, we don't want to create contact records — we only want to find them. To disable creation of a record, set `doCreate` to `false` for all the required fields for the record. This code disables a common default behavior of creating a contact record with each chat session.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate":false,
    "doFind":true,
    "fieldName":"LastName",
    "isExactMatch":true,
    "label":"Last Name"
  }, {
    "doCreate":false,
    "doFind":true,
    "fieldName":"FirstName",
    "isExactMatch":true,
    "label":"First Name"
  }, {
    "doCreate":false,
    "doFind":true,
    "fieldName":"Email",
    "isExactMatch":true,
    "label":"Email"
  }],
  "entityName":"Contact"
}];
```

### Don't attach records to the chat transcript

If you don't want to attach contact records to the transcript, set `saveToTranscript` to an empty string.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate":true,
    "doFind":true,
    "fieldName":"LastName",
    "isExactMatch":true,
    "label":"Last Name"
  }, {
    "doCreate":true,
    "doFind":true,
    "fieldName":"FirstName",
    "isExactMatch":true,
    "label":"First Name"
  }, {
    "doCreate":true,
    "doFind":true,
    "fieldName":"Email",
    "isExactMatch":true,
    "label":"Email"
  }],
  "entityName":"Contact",
  "saveToTranscript": ""
}];
```

### Attach a record to a custom field on the chat transcript

If you want to attach the created case or contact record to a custom field on the transcript, use `saveToTranscript`.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate": true,
    "doFind": true,
    "fieldName": "LastName",
    "isExactMatch": true,
    "label": "Last Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "FirstName",
    "isExactMatch": true,
    "label": "First Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "Email",
    "isExactMatch": true,
    "label": "Email"
  }],
  "entityName": "Contact",
  "saveToTranscript": "customContact__c"
}];
```

## Don't show the created record to the agent

If you don't want to show the created record when the chat session starts, set `showOnCreate` to `false`.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate": true,
    "doFind": true,
    "fieldName": "LastName",
    "isExactMatch": true,
    "label": "Last Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "FirstName",
    "isExactMatch": true,
    "label": "First Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "Email",
    "isExactMatch": true,
    "label": "Email"
  }],
  "entityName": "Contact",
  "showOnCreate": false
}];
```

## Override fields specified in your org's setup

This example overrides the first name, last name, and subject passed in from the pre-chat form. To test this code, select the service scenario in your org's setup and add this code to your snippet.

```
embedded_svc.settings.extraPrechatFormDetails = [{
  "label": "First Name",
  "value": "John",
  "displayToAgent": true
}, {
  "label": "Last Name",
  "value": "Doe",
  "displayToAgent": true
}, {
  "label": "Email",
  "value": "john.doe@salesforce.com",
  "displayToAgent": true
}, {
  "label": "issue",
  "value": "Overriding your setup",
  "displayToAgent": true
}];

embedded_svc.settings.extraPrechatInfo = [{
  "entityName": "Contact",
  "showOnCreate": true,
  "linkToEntityName": "Case",
  "linkToEntityField": "ContactId",
  "saveToTranscript": "ContactId",
  "entityFieldMaps": [{
    "isExactMatch": true,
    "fieldName": "FirstName",
    "doCreate": true,
    "doFind": true,
    "label": "First Name"
  }, {
    "isExactMatch": true,
    "fieldName": "LastName",
    "doCreate": true,
    "doFind": true,
    "label": "Last Name"
  }, {
    "isExactMatch": true,
    "fieldName": "Email",
    "doCreate": true,
    "doFind": true,
    "label": "Email"
  }]
}, {
  "entityName": "Case",
  "showOnCreate": true,
  "saveToTranscript": "CaseId",
  "entityFieldMaps": [{
    "isExactMatch": false,
    "fieldName": "Subject",
    "doCreate": true,
    "doFind": false,
    "label": "issue"
  }, {
    "isExactMatch": false,
    "fieldName": "Status",
    "doCreate": true,
    "doFind": false,
    "label": "Status"
  }, {
    "isExactMatch": false,
    "fieldName": "Origin",
    "doCreate": true,
    "doFind": false,
    "label": "Origin"
  }]
}]
```

## Create a new record from a different Salesforce object

If your business needs a record from a Salesforce object that isn't available in the standard scenarios, you can define the object in `extraPrechatInfo`. For example, you can create an account when a chat session starts.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityName": "Contact",
  "entityFieldMaps": [{
    "isExactMatch": true,
    "fieldName": "FirstName",
    "doCreate": true,
    "doFind": true,
    "label": "firstName"
  }, {
    "isExactMatch": true,
    "fieldName": "LastName",
    "doCreate": true,
    "doFind": true,
    "label": "LastName"
  }, {
    "isExactMatch": true,
    "fieldName": "Email",
    "doCreate": true,
    "doFind": true,
    "label": "Email"
  }]
}, {
  "entityName": "Case",
  "entityFieldMaps": [{
    "isExactMatch": false,
    "fieldName": "Subject",
    "doCreate": true,
    "doFind": false,
    "label": "issue"
  }, {
    "isExactMatch": false,
    "fieldName": "Status",
    "doCreate": true,
    "doFind": false,
    "label": "Status"
  }, {
    "isExactMatch": false,
    "fieldName": "Origin",
    "doCreate": true,
    "doFind": false,
    "label": "Origin"
  }]
}, {
  "entityName": "Account",
  "entityFieldMaps": [{
    "isExactMatch": true,
    "fieldName": "Name",
    "doCreate": true,
    "doFind": true,
    "label": "LastName"
  }]
}];
embedded_svc.settings.extraPrechatFormDetails = [{
  "label": "firstName",
  "value": "John",
  "displayToAgent": true
}, {
  "label": "LastName",
  "value": "Doe",
  "displayToAgent": false
}, {
  "label": "Email",
  "value": "john.doe@salesforce.com",
  "displayToAgent": true
}, {
  "label": "issue",
  "value": "Do the work",
  "displayToAgent": true
}];
```

## Link to another Salesforce object

If you want to link the case record created by Snap-ins to the account record you created from `extraPrechatInfo`, use `linkToEntityName` and `linkToEntityFieldName`.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityName": "Account",
  "linkToEntityName": "Case",
  "linkToEntityField": "AccountId",
  "entityFieldMaps": [{
    "isExactMatch": true,
    "fieldName": "Name",
    "doCreate": true,
    "doFind": true,
    "label": "LastName"
  }]
}];

embedded_svc.settings.extraPrechatFormDetails = [{
  "label": "firstName",
  "value": "Jane",
  "displayToAgent": true
}, {
  "label": "LastName",
  "value": "Doe",
  "displayToAgent": false
}, {
  "label": "Email",
  "value": "jane.doe@gmail.com",
  "displayToAgent": true
}, {
  "label": "issue",
  "value": "Do the work",
  "displayToAgent": true
}];
```

## Don’t attach a contact to a case

If you don’t want the default link between a contact and a case, set `linkToEntityName` to an empty string.

```
embedded_svc.settings.extraPrechatInfo = [{
  "entityFieldMaps": [{
    "doCreate": true,
    "doFind": true,
    "fieldName": "LastName",
    "isExactMatch": true,
    "label": "Last Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "FirstName",
    "isExactMatch": true,
    "label": "First Name"
  }, {
    "doCreate": true,
    "doFind": true,
    "fieldName": "Email",
    "isExactMatch": true,
    "label": "Email"
  }],
  "entityName": "Contact",
  "linkToEntityName": ""
}];
```

## Disable pre-chat and pass along the user's name

If you want to avoid the pre-chat form but still have the user’s name show up in the Waiting state, you can:

1. Find the object that contains `"label" : "First  Name"`.
2. Add a property `"name" : "FirstName"`.
3. Set the value of the `"value"` property to the desired first name value. For example, `"value" : "Jane"`.

```
embedded_svc.settings.extraPrechatFormDetails = [{
    "label":"First Name",
    "name":"FirstName",
    "value":"Jane",
    "displayToAgent":true
}, {
    "label":"Last Name",
    "value":"Doe",
    "displayToAgent":true
}, {
    "label":"Email",
    "value":"jane.doe@salesforce.com",
    "displayToAgent":true
}, {
    "label":"issue",
    "value":"Sales forecasts",
    "displayToAgent":true
}];

embedded_svc.settings.extraPrechatInfo = [{
    "entityName":"Contact",
    "showOnCreate":true,
    "linkToEntityName":"Case",
    "linkToEntityField":"ContactId",
    "saveToTranscript":"ContactId",
    "entityFieldMaps": [{
        "isExactMatch":true,
        "fieldName":"FirstName",
        "doCreate":true,
        "doFind":true,
        "label":"First Name"
    }, {
        "isExactMatch":true,
        "fieldName":"LastName",
        "doCreate":true,
        "doFind":true,
        "label":"Last Name"
    }, {
        "isExactMatch":true,
        "fieldName":"Email",
        "doCreate":true,
        "doFind":true,
        "label":"Email"
    }]
}, {
    "entityName":"Case",
    "showOnCreate":true,
    "saveToTranscript":"CaseId",
    "entityFieldMaps": [{
        "isExactMatch":false,
        "fieldName":"Subject",
        "doCreate":true,
        "doFind":false,
        "label":"issue"
      }, {
        "isExactMatch":false,
        "fieldName":"Status",
        "doCreate":true,
        "doFind":false,
        "label":"Status"
      }, {
        "isExactMatch":false,
        "fieldName":"Origin",
        "doCreate":true,
        "doFind":false,
        "label":"Origin"
    }]
}];
```

### Set custom fields on the transcript object

If you want to set custom fields on the transcript object, pass those fields in as transcript fields in `prechatFormDetails`. This code assumes you created a custom field called `CartValue` on the chat transcript object.

```
embedded_svc.settings.extraPrechatFormDetails = [{
     "label":"CartValue",
     "value":”200”,
     "transcriptFields":[ "CartValue__c" ],
     "displayToAgent":true
}];
```

### Save pre-chat form values into a custom field on the transcript object

If you want to save dynamic values from the pre-chat form directly into a custom field in the transcript object, you can:

1. Create a custom field on the transcript object (for example, `LastName__c`).

   Tip: For the transcript field, the following data types work best: text, number, email, checkbox, and phone.

2. Pass the transcript fields in the extra pre-chat form details without passing the value property.

The value is calculated from the pre-chat form and saved into the custom field.

```
embedded_svc.settings.extraPrechatFormDetails = [{"label":"Last Name", "transcriptFields":
 ["LastName__c"]}];
```

Note: We don’t support passing pre-chat form values to standard transcript fields.

## Route Chats Based on Pre-Chat Responses with Direct-to-Button Routing

Set your snap-in to route chats to different chat buttons based on the customer’s pre-chat response on any and all of your pre-chat fields. Available when you upgrade your code snippet to version 4.0.

You can set a specific chat button for each option in a picklist or even for certain keywords in text fields. For example, if your customer can choose which product they own from a picklist and they select “Laptop,” you can send that chat request to a button that’s linked to the “Laptop” agent skill or Omni-Channel queue. Similarly, if your customer can describe their reason for requesting a chat in a text field, you can have it route to your “Laptop” agents if the customer enters “laptop” in that field.

If no agents are available for the button you’ve defined for a particular pre-chat field, the customer sees a dialog in the snap-in to let them know that no one’s online.

The designated button ID must be 15–18 characters and start with the correct prefix. If the button ID you provide doesn’t meet these requirements, the chat routes to your default button for the deployment. If the button ID you provide follows these requirements but doesn’t identify a button record in your org, the customer isn’t able to start the chat.

With a 4.0 snippet, the following is included and inactive:

```
//embedded_svc.settings.directToButtonRouting = function(prechatFormData) {
// Dynamically changes the button ID based on what the visitor enters in the pre-chat form.
//Returns a valid button ID.
//};
```

After you've entered your function, remove the comment indicators to activate the function.

Let's look at an example. Say that you want to route chats to a button with the ID `5733000000000Gq` if the user selects "Other" from a picklist in the 4th pre-chat field. You would enter the following:

```
embedded_svc.settings.directToButtonRouting = function(prechatFormData) {
 if (prechatFormData[3].value === "other")
  return "5733000000000Gq";
}
```

## Set Certain Pre-Chat Form Fields to Automatically Populate when Customers are Logged In

When your users are logged in, you already know basic information like their name and email address. Use this array in your 4.0 code snippet to populate relevant pre-chat fields for them. You can mix and match fields for different record types. This information is for snap-ins that are placed outside of Salesforce with Lightning Out (beta). If you use your snap-in inside Communities, you can enable the contact fields to fill in automatically in the Snap-ins Chat component settings.

The parameter you can use to set the fields that you want to pre-populate is included with your version 4.0 code snippet as a code comment.

`embedded_svc.settings.prepopulatedPrechatFields = {...}`

If you use snap-ins outside of Salesforce (with Lightning Out), you can set any pre-chat field to populate automatically using the fields' API names. Find the field API names on the object pages for the objects you use with pre-chat. Then you can set the value of `embedded_svc.settings.prepopulatedPrechatFields` with a JavaScript object that contains the customer's information.

The following sample code for `embedded_svc.settings.prepopulatedPrechatFields` populates the First Name, Last Name, Email, and Subject fields in the pre-chat form.

```
embedded_svc.settings.prepopulatedPrechatFields = {
    FirstName: “John”,
    LastName: “Doe”,
    Email: “john.doe@salesforce.com”,
    Subject: “Hello”
};
```

# Custom Lightning Components for Snap-Ins for Web

Use custom Lightning Components to customize the user interface for your snap-in.

Customize the Pre-Chat Page UI with Lightning Components
Customize the fields, layout, buttons, images, validation, or any other part of the user interface for pre-chat using a custom Lightning component.

Custom Pre-Chat Component Code Samples
You can use Aura or plain JavaScript to create your pre-chat components.

Customize the Minimized Snap-In UI with Lightning Components
Customize the user interface for the snap-in when it's minimized on your web page using a custom Lightning component.

Custom Minimized Component Code Samples
The following code sample contains examples of the component, controller, and helper code for a custom minimized snap-in component using Aura.

Get Settings from the Snap-Ins Code Snippet
Get settings for use with your Snap-ins Lightning components. You can get the Live Agent button ID or deployment ID assigned to your Snap-ins deployment and the agent and chatbot avatar image URLs.

## Customize the Pre-Chat Page UI with Lightning Components

Customize the fields, layout, buttons, images, validation, or any other part of the user interface for pre-chat using a custom Lightning component.

Before you start, make sure that you have a Snap-ins deployment with pre-chat already set up. Next, go to the Developer Console and click **File** > **New** > **Lightning Component**. Enter a name and description for your component and click **Submit**.

1. Implement the pre-chat interface.

   Change the opening aura component tag to:

   ```
   <aura:component implements="lightningsnapin:prechatUI">
   ```

   This code implements the `lightningsnapin:prechatUI` interface, which makes the component available to select as your pre-chat page from Snap-ins Setup.

2. Create the pre-chat API component.

   ```
   <lightningsnapin:prechatAPI aura:id="prechatAPI"/>
   ```

   This code provides an API that you can use to customize the user interface for the pre-chat page.

3. Add an initialize aura handler.

   This action gets called when the component is initialized.

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

   Note: There's a separate handler for when the component renders.

4. Add your markup.

   Create your buttons, images, validation, or whatever else you want to create. You have full control over the layout using the fields you specified in Snap-ins Setup.

5. Add an initialize action in your component controller.

   ```
   /**
    * On initialization of this component, set the prechatFields attribute and render
   prechat fields.
    *
    * @param cmp - The component for this state.
    * @param evt - The Aura event.
    * @param hlp - The helper for this state.
    */
   ```

```
onInit: function(cmp, evt, hlp) {
   // Get prechat fields defined in setup using the prechatAPI component.
   var prechatFields = cmp.find("prechatAPI").getPrechatFields();

   // Render your fields
},
```

**6.** Add a handler for starting a chat.

Add a click handler to a button element. The customer uses this button to request a chat.

```
/**
 * Function to start a chat request (by accessing the chat API component)
 *
 * @param cmp - The component for this state
 */
onStartButtonClick: function(cmp) {
    // Make an array of field objects for the library.
    var fields = // Get your prechat fields.

    // If the prechat fields pass validation, start a chat.
    if(cmp.find("prechatAPI").validateFields(fields).valid) {
        cmp.find("prechatAPI").startChat(fields);
    } else {
        console.warn("Prechat fields did not pass validation!");
    }
},
```

**7.** Save your component and select it from Snap-ins Setup.

After you save your component, go to Snap-ins Setup and navigate to your chat settings. Your component should be available to select as a custom component for your pre-chat page.

SEE ALSO:

Lightning Components Developer Guide

## Custom Pre-Chat Component Code Samples

You can use Aura or plain JavaScript to create your pre-chat components.

Custom Pre-Chat Component Sample Using Aura

The following code sample contains examples of the component, controller, and helper code for a custom pre-chat component using Aura.

Custom Pre-Chat Component Sample Using JavaScript

The following code sample contains examples of the component, controller, and helper code for a custom pre-chat component using plain JavaScript.

### Custom Pre-Chat Component Sample Using Aura

The following code sample contains examples of the component, controller, and helper code for a custom pre-chat component using Aura.

This component:

- Uses Aura to create the pre-chat fields and start a chat.
- Uses an initialize function that fetches the pre-chat fields from setup and dynamically creates the pre-chat field components using `$A.createComponents`.
- Creates an array when the user clicks the **Start Chat** button. The array contains pre-chat field information to pass to the `startChat` method on the prechatAPI component.

## Component Code

```
<aura:component implements="lightningsnapin:prechatUI" description="Sample custom prechat
 component for Snapins. Implemented using Aura.">
    <!-- You must implement "lightningsnapin:prechatUI" for this component to appear in
the "Prechat Component" customization dropdown in the Snapins setup. -->

    <!-- Prechat field components to render. -->
    <aura:attribute name="prechatFieldComponents" type="List" description="An array of
objects representing the prechat fields specified in prechat setup."/>

    <!-- Handler for when this component is initialized. -->
    <aura:handler name="init" value="{!this}" action="{!c.onInit}" />

    <!-- For Aura performance. -->
    <aura:locator target="startButton" description="Prechat form submit button."/>

    <!-- Contains methods for getting prechat fields, starting a chat, and validating
fields. -->
    <lightningsnapin:prechatAPI aura:id="prechatAPI"/>

    <h2>Prechat form.</h2>
    <div class="prechatUI">
        <div class="prechatContent">
            <ul class="fieldsList">
                <!-- Look in the controller's onInit function. This component dynamically
 creates the prechat field components. -->
                {!v.prechatFieldComponents}
            </ul>
        </div>
        <div class="startButtonWrapper">
            <ui:button aura:id="startButton" class="startButton"
label="{!$Label.LiveAgentPrechat.StartChat}" press="{!c.handleStartButtonClick}"/>
        </div>
    </div>

</aura:component>
```

## Controller Code

```
({
    /**
     * On initialization of this component, set the prechatFields attribute and render
prechat fields.
     *
     * @param cmp - The component for this state.
     * @param evt - The Aura event.
     * @param hlp - The helper for this state.
     */
 onInit: function(cmp, evt, hlp) {
        // Get prechat fields defined in setup using the prechatAPI component.
  var prechatFields = cmp.find("prechatAPI").getPrechatFields();
        // Get prechat field types and attributes to be rendered.
       var prechatFieldComponentsArray = hlp.getPrechatFieldAttributesArray(prechatFields);


        // Make asynchronous Aura call to create prechat field components.
        $A.createComponents(
            prechatFieldComponentsArray,
            function(components, status, errorMessage) {
                if(status === "SUCCESS") {
                    cmp.set("v.prechatFieldComponents", components);
                }
            }
        );
    },

    /**
     * Event which fires when start button is clicked in prechat.
     *
     * @param cmp - The component for this state.
     * @param evt - The Aura event.
     * @param hlp - The helper for this state.
     */
    handleStartButtonClick: function(cmp, evt, hlp) {
        hlp.onStartButtonClick(cmp);
    }
});
```

## Helper Code

```
({
    /**
     * Event which fires the function to start a chat request (by accessing the chat API
component).
     *
     * @param cmp - The component for this state.
     */
    onStartButtonClick: function(cmp) {
        var prechatFieldComponents = cmp.find("prechatField");
       var apiNamesMap = this.createAPINamesMap(cmp.find("prechatAPI").getPrechatFields());

        var fields;

        // Make an array of field objects for the library.
        fields = this.createFieldsArray(apiNamesMap, prechatFieldComponents);

        // If the prechat fields pass validation, start a chat.
        if(cmp.find("prechatAPI").validateFields(fields).valid) {
            cmp.find("prechatAPI").startChat(fields);
        } else {
            console.warn("Prechat fields did not pass validation!");
        }
    },

    /**
     * Create an array of field objects to start a chat from an array of prechat fields.
     *
     * @param fields - Array of prechat field Objects.
     * @returns An array of field objects.
     */
    createFieldsArray: function(apiNames, fieldCmps) {
        if(fieldCmps.length) {
            return fieldCmps.map(function(fieldCmp) {
                return {
                    label: fieldCmp.get("v.label"),
                    value: fieldCmp.get("v.value"),
                    name: apiNames[fieldCmp.get("v.label")]
                };
            }.bind(this));
        } else {
            return [];
        }
    },

    /**
     * Create map of field label to field API name from the pre-chat fields array.
     *
     * @param fields - Array of prechat field Objects.
     * @returns An array of field objects.
     */
    createAPINamesMap: function(fields) {
        var values = {};

        fields.forEach(function(field) {
            values[field.label] = field.name;
        });

        return values;
    },

    /**
     * Create an array in the format $A.createComponents expects.
     *
     * Example:
     * [["componentType", {attributeName: "attributeValue", ...}]]
     *
     * @param prechatFields - Array of prechat field Objects.
     * @returns Array that can be passed to $A.createComponents
     */
    getPrechatFieldAttributesArray: function(prechatFields) {
        // $A.createComponents first parameter is an array of arrays. Each array contains
```

```
 the type of component being created, and an Object defining the attributes.
        var prechatFieldsInfoArray = [];

        // For each field, prepare the type and attributes to pass to $A.createComponents.

        prechatFields.forEach(function(field) {
           var componentName = (field.type === "inputSplitName") ? "inputText" : field.type;

             var componentInfoArray = ["ui:" + componentName];
             var attributes = {
                 "aura:id": "prechatField",
                 required: field.required,
                 label: field.label,
                 disabled: field.readOnly,
                 maxlength: field.maxLength,
                 class: field.className,
                 value: field.value
             };

             // Special handling for options for an input:select (picklist) component.
             if(field.type === "inputSelect" && field.picklistOptions) attributes.options
= field.picklistOptions;

             // Append the attributes Object containing the required attributes to render
this prechat field.
             componentInfoArray.push(attributes);

             // Append this componentInfoArray to the fieldAttributesArray.
             prechatFieldsInfoArray.push(componentInfoArray);
        });

        return prechatFieldsInfoArray;
    }
});
```

## Custom Pre-Chat Component Sample Using JavaScript

The following code sample contains examples of the component, controller, and helper code for a custom pre-chat component using plain JavaScript.

This component:

- Uses Javascript to create an email input field
- Uses minimal Aura to get pre-chat fields from Snap-ins setup in a render handler
- Provides an example of getting pre-chat field values to pass to the `startChat` Aura method on `lightningsnapin:prechatAPI`

**Important:** This code sample is an example and doesn't create a functioning pre-chat form on its own. You can use this sample as a starting point, but you need to add more pre-chat fields and include your own styling.

## Component Code

```
<aura:component
 description="Sample pre-chat component that uses Aura only when absolutely necessary"
 implements="lightningsnapin:prechatUI">

 <!-- Contains methods for getting prechat fields, starting a chat, and validating fields.
 -->
 <lightningsnapin:prechatAPI aura:id="prechatAPI"/>

 <!-- After this component has rendered, call the controller's onRender function. -->
 <aura:handler name="render" value="{!this}" action="{!c.onRender}"/>

 <div class="prechatUI">
        Prechat Form
  <div class="prechatFields">
   <!-- We'll add pre-chat field HTML elements in the controller's onInit function. -->
  </div>
  <button class="startChatButton" onclick="{!c.onStartButtonClick}">
   Start Chat
  </button>
 </div>

</aura:component>
```

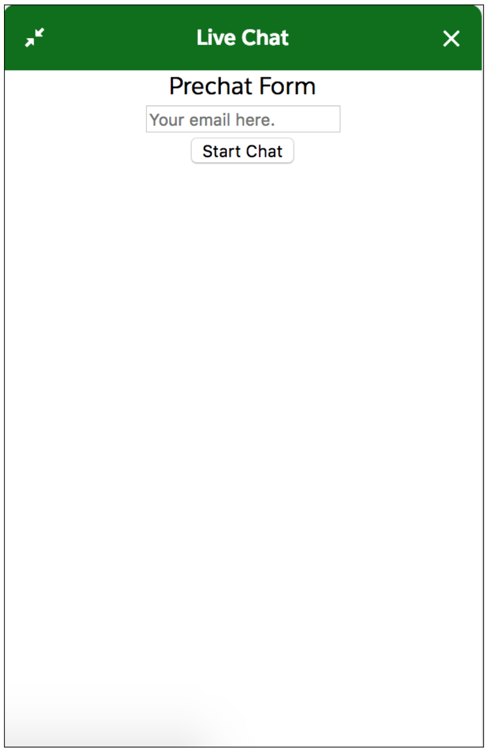
## Controller Code

```
({
 /**
  * After this component has rendered, create an email input field.
  *
  * @param component - This prechat UI component.
  * @param event - The Aura event.
  * @param helper - This component's helper.
  */
 onRender: function(component, event, helper) {
  // Get array of prechat fields defined in setup using the prechatAPI component.
  var prechatFields = component.find("prechatAPI").getPrechatFields();
  // This example renders only the email field using the field info that comes back from
prechatAPI getPrechatFields().
  var emailField = prechatFields.find(function(field) {
   return field.type === "inputEmail";
  });

  // Append an input element to the prechatForm div.
  helper.renderEmailField(emailField);
 },

 /**
  * Handler for when the start chat button is clicked.
  *
  * @param component - This prechat UI component.
  * @param event - The Aura event.
  * @param helper - This component's helper.
  */
 onStartButtonClick: function(component, event, helper) {
  var prechatInfo = helper.createStartChatDataArray();

  if(component.find("prechatAPI").validateFields(prechatInfo).valid) {
   component.find("prechatAPI").startChat(prechatInfo);
  } else {
   // Show some error.
  }
 }
});
```

## Helper Code

```
({
 /**
  * Create an HTML input element, set necessary attributes, add the element to the DOM.
  *
  * @param emailField - Email prechat field object with attributes needed to render.
  */
 renderEmailField: function(emailField) {
  // Dynamically create input HTML element.
  var input = document.createElement("input");

  // Set general attributes.
  input.type = "email";
  input.class = emailField.label;
  input.placeholder = "Your email here.";

  // Set attributes required for starting a chat.
  input.name = emailField.name;
  input.label = emailField.label;

  // Add email input to the DOM.
  document.querySelector(".prechatFields").appendChild(input);
 },

 /**
  * Create an array of data to pass to the prechatAPI component's startChat function.
     */
 createStartChatDataArray: function() {
  var input = document.querySelector(".prechatFields").childNodes[0];
  var info = {
   name: input.name,
   label: input.label,
   value: input.value
  };

  return [info];
 }
});
```

The sample creates the following unstyled pre-chat form.



## Customize the Minimized Snap-In UI with Lightning Components

Customize the user interface for the snap-in when it's minimized on your web page using a custom Lightning component.

Before you start, make sure that you have a Snap-ins deployment already set up. Next, go to the Developer Console and click **File** > **New** > **Lightning Component**. Enter a name and description for your component and click **Submit**.

1. Implement the minimized interface.

   Change the opening aura component tag to:

   ```
   <aura:component implements="lightningsnapin:minimizedUI">
   ```

   This code implements the `lightningsnapin:minimizedUI` interface, which makes the component available to select as your minimized component from Snap-ins Setup.

2. Create the minimized API component.

   ```
   <lightningsnapin:minimizedAPI aura:id="minimizedAPI"/>
   ```

   This code provides an API that you can use to customize the user interface for the minimized component.

3. Add an initialize aura handler.

   This action gets called when the component is initialized.

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

4. Add your markup.

Create your buttons, images, validation, or whatever else you want to create. You have full control over the layout using the fields you specified in Snap-ins Setup.

Make sure to add a maximize container action so your customers can open the snap-in. We recommend you add the following as a click handler on a button, for example.

```
<button onclick="{!c.handleMaximize}">
    {!v.message}
</button>
```

5. Add an initialize action in your component controller.

```
/**
 * On initialization of this component, register the generic event handler for all the
 minimized events..
 *
 * @param cmp - The component for this state.
 * @param evt - The Aura event.
 * @param hlp - The helper for this state.
 */
onInit: function(cmp, evt, hlp) {
   cmp.find("minimizedAPI").registerEventHandler(hlp.minimizedEventHandler.bind(hlp,
cmp));
},
```

6. Add a handler for maximizing chat from the minimized component.

   Add a click handler to a button element. The customer uses this button to maximize chat.

```
/**
 * Function to handle maximizing the chat.
 *
 * @param cmp - The component for this state
 * @param evt - The Aura event.
 * @param hlp - The helper for this state.
 */
handleMaximize: function(cmp, evt, hlp) {
    cmp.find("minimizedAPI").maximize();
},
```

7. Add a minimized event generic handler to your helper.

```
/**
 * Function to handle maximizing the chat.Function to start a chat request (by accessing
 the chat API component)
 *
 * @param cmp - The component for this state
 * @param eventName - The name of the event fired.
 * @param eventData - The data associated with the event fired.
 */
minimizedEventHandler: function(cmp, eventName, eventData) {
    switch(eventName) {
        case "prechatState":
            cmp.set("v.message", "Chat with an Expert!");
            Break;
        // Handle other events here!
        default:
            cmp.set("v.message", eventData.label);
    }
}
```

8. Save your component and select it from Snap-ins Setup.

   After you save your component, go to Snap-ins Setup and navigate to your chat settings. Your component should be available to select as a custom component for the minimized snap-in.

Events for the Minimized Snap-In

Use the following events in `eventHandlerFunction` in your minimized snap-in Lightning component.

SEE ALSO:

Lightning Components Developer Guide

## Events for the Minimized Snap-In

Use the following events in `eventHandlerFunction` in your minimized snap-in Lightning component.

`eventHandlerFunction` is called with two positional arguments, `eventName` and `eventData`.

Below are the possible values for `eventName` and the corresponding scenario

Note: Some events are fired multiple times per scenario. Avoid writing logic that can't be safely executed multiple times.

| eventName | Scenario |
|---|---|
| `chatEndedState` | Fired when the chat has ended for any reason. |
| `chatState` | Fired in the following scenarios:<br>• The visitor's chat request has been accepted and they're chatting with an agent.<br>• After starting a chat, the visitor navigates to another page and resumes chatting.<br>• A chat transfer has completed (see `chatTransferringState`), and the visitor is now chatting with the new agent.<br>• The visitor had previously lost connection (see `reconnectingState`) and has regained it. |
| `chatTimeoutUpdate` | Fired when the visitor idle timeout has started, and every additional second during which the visitor is still idle. |
| `chatTransferringState` | Fired when the a chat transfer has been initiated. |
| `chatUnreadMessage` | Fired every time the visitor has received a message from the agent, but the visitor hasn't read it yet. |
| `offlineSupportState` | Fired when the offline support form has loaded. |

| eventName | Scenario |
|---|---|
| `prechatState` | Fired when the pre-chat form has loaded. |
| `postchatState` | Fired when the post chat form has loaded. |
| `queueUpdate` | Fired in the following scenarios: <br>• The visitor has requested a chat and is waiting for an agent. <br>• After requesting a chat, the visitor navigates to another page but is still waiting for an agent. <br>• The visitor had previously lost connection (see `reconnectingState`) and has regained it. <br>• The visitor has advanced in the queue but is still waiting for an agent. <br>Note: This event fires only if queue position is enabled for your Snap-ins deployment and your Snap-ins code snippet is version 5.0 or later. |
| `reconnectingState` | Fired when the visitor has lost connection. |
| `waitingEndedState` | Fired when the visitor's chat request has failed for any reason. |
| `waitingState` | Fired in the following scenarios: <br>• The visitor has requested a chat and is waiting for an agent. <br>• After requesting a chat, the visitor navigates to another page but is still waiting for an agent. <br>• The visitor had previously lost connection (see `reconnectingState`) and has regained it. <br>Note: This event fires only if either queue position is disabled for your Snap-ins deployment or your Snap-ins code snippet is version is earlier than 5.0. |

The `eventData` is an object that contains the default localized label for the event. For some events, it contains additional values. Below are the possible additional values for `eventData`.

| eventName | Property | Type | Description |
|---|---|---|---|
| `chatEndedState` | `reason` | string | A description of why the chat ended. Possible values include: <br>• `agentEndedChat`– The agent ended the chat. <br>• `visitorConnectionError`– The visitor lost connection for too long. <br>• `visitorEndedChat`– The visitor ended the chat. |

| eventName | Property | Type | Description |
|---|---|---|---|
| | | | • `visitorTimeout`– The visitor was idle too long and the chat timed out. |
| `chatState` | `agentName` | string | The name of the agent. |
| | `agentType` | string | The type of agent handling the chat. Possible values are `agent` (support agent) and `chatbot` (Einstein Bots). |
| `chatTimeoutUpdate` | `timeoutSecondsRemaining` | int | The number of seconds remaining until the chat times out due to the visitor being idle for too long. |
| `chatUnreadMessage` | `unreadMessageCount` | int | The number of unread messages. |
| | `agentType` | string | The type of agent handling the chat. Possible values are `agent` (support agent) and `chatbot` (Einstein Bots). |
| `queueUpdate` | `queuePosition` | int | The visitor's current place in line. |
| `waitingEndedState` | `reason` | string | A description of why the visitor's chat request failed. Possible values include: • `agentsUnavailable`– All agents were unavailable, or an agent declined the chat request. • `connectionError`– The chat request failed for any other reason. • `visitorBlocked`– The visitor's IP address has been blocked. |

## Custom Minimized Component Code Samples

The following code sample contains examples of the component, controller, and helper code for a custom minimized snap-in component using Aura.

This component:

- Uses Aura to create a ui:button component, and binds the button click to maximize the snap-in

- Uses an initialize function that registers the generic event handler for minimized events that will update the message depending on the event name
- Includes optional CSS styling

## Component Code

```
<aura:component implements="lightningsnapin:minimizedUI" description="Custom Minimized
UI">
    <aura:handler name="init" value="{!this}" action="{!c.onInit}"/>
    <aura:attribute name="message" type="String" default="Chat with us!"/>

 <!-- For registering our minimized event handler and maximizing. -->
 <lightningsnapin:minimizedAPI aura:id="minimizedAPI"/>

 <button class="minimizedContainer"
  press="{!c.handleMaximize}"
  aura:id="minimizedContainer">
        <div class="messageContent">
            {!v.message}
        </div>
 </button>
</aura:component>
```

## Controller Code

```
({
 onInit: function(cmp, evt, hlp) {
        // Register the generic event handler for all the minimized events.
       cmp.find("minimizedAPI").registerEventHandler( hlp.minimizedEventHandler.bind(hlp,
 cmp));
 },

    handleMaximize: function(cmp, evt, hlp) {
        cmp.find("minimizedAPI").maximize();
    }
})
```

## Helper Code

```
({
 minimizedEventHandler: function(cmp, eventName, eventData) {
        switch(eventName) {
   case "prechatState":
    this.onPrechatState(cmp, eventData);
    break;
   case "offlineSupportState":
    this.onOfflineSupportState(cmp, eventData);
    break;
   case "waitingState":
    this.onWaitingState(cmp, eventData);
    break;
   case "queueUpdate":
    this.onQueueUpdate(cmp, eventData);
    break;
   case "waitingEndedState":
    this.onWaitingEndedState(cmp, eventData);
    break;
   case "chatState":
    this.onChatState(cmp, eventData);
    break;
   case "chatTimeoutUpdate":
    this.onChatTimeoutUpdate(cmp, eventData);
    break;
   case "chatUnreadMessage":
    this.onChatUnreadMessage(cmp, eventData);
    break;
   case "chatTransferringState":
    this.onChatTransferringState(cmp, eventData);
    break;
   case "chatEndedState":
    this.onChatEndedState(cmp, eventData);
    break;
   case "reconnectingState":
    this.onReconnectingState(cmp, eventData);
    break;
   case "postchatState":
    this.onPostchatState(cmp, eventData);
    break;
   default:
    throw new Error("Received unexpected minimized event '" + eventName + "'.");
  }
 },

 /**
  * "prechatState" event handler. This fires when prechat state is initialized.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onPrechatState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "offlineSupportState" event handler. This fires when offline support state is
initialized.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onOfflineSupportState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },
 /**
  * "waitingState" and "queueUpdate" are fired when EITHER
  * 1) waiting state is initialized, either with a new session or via page
navigation/refresh, OR
  * 2) the visitor was previously in reconnecting, and they've regained connection.
  *
  * Only one of "waitingState" and "queueUpdate" will ever be fired - never both.
  * - "waitingState" will be fired if EITHER queue position is DISABLED, OR snippet version
 < 5.0.
  * - "queueUpdate" will be fired if queue position is ENABLED, AND snippet version >=
5.0.
  */

 /**
  * "waitingState" event handler. See above doc.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onWaitingState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "queueUpdate" event handler. See above doc.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - Event data. For this event, this contains label and
queuePosition.
  */
 onQueueUpdate: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label + " " + eventData.queuePosition);
 },

 /**
  * "waitingEndedState" event handler. This fires in waiting state when the chat request
fails.
  *
  * @param {SampleCustomMinimizedUI.SampleCustomMinimizedUIComponent} cmp - This component.

  * @param {Object} eventData - Event data. For this event, this contains label and reason.
 We don't use reason though.
  */
 onWaitingEndedState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "chatState" event handler. This fires when EITHER
  * 1) chat state is initialized, either with a new session or via page navigation/refresh,
 OR
  * 2) the visitor was previously in chat transfer, and they've been connected to a new
```

```
agent, OR
  * 3) the visitor was previously in reconnecting, and they've regained connection.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onChatState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "chatTimeoutUpdate" event handler. This fires when the visitor idle timeout has started.

  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - Event data. For this event, this contains label and
timeoutSecondsRemaining.
  */
 onChatTimeoutUpdate: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "chatUnreadMessage" event handler. This fires when the agent sends a message but the
visitor hasn't seen it
  * yet, either because they are scrolled up in the chat message area, or because the
widget is minimized.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - Event data. For this event, this contains label and
unreadMessageCount.
  */
 onChatUnreadMessage: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "chatTransferringState" event handler. This fires when a chat transfer has been
initiated.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onChatTransferringState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "chatEndedState" event handler. This fires in chat state when the chat ends for any
reason.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - Event data. For this event, this contains label and reason.

  */
 onChatEndedState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "reconnectingState" event handler. This fires in both waiting and chat state when the
 visitor loses connection.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onReconnectingState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * "postchatState" event handler. This fires when the visitor enters post chat by clicking
 "Give Feedback".
  *
  * @param {Aura.Component} cmp - This component.
  * @param {Object} eventData - The data associated with the event. Always contains a
"label" property.
  */
 onPostchatState: function(cmp, eventData) {
  this.setMinimizedContent(cmp, eventData.label);
 },

 /**
  * Update the contents of the sample minimized component.
  *
  * @param {Aura.Component} cmp - This component.
  * @param {String} message - The text to display.
  */
 setMinimizedContent: function(cmp, message) {
  cmp.set("v.message", message);
 }
});
```

## CSS Code

```
.THIS {
 position: fixed;
 left: auto;
 bottom: 0;
 right: 12px;
 margin: 0;
  min-width: 12em;
 max-width: 14em;
 height: 46px;
 width: 192px;
 max-height: 100%;
 border-radius: 8px 8px 0 0;
 text-align: center;
 text-decoration: none;
 display: flex;
 flex-direction: center;
justify-content: center;
box-shadow: none;
 pointer-events: all;
 overflow: hidden;
 background-color: rgb(0, 112, 210);
 font-size: 16px;
}

.THIS.minimizedContainer:focus,
.THIS.minimizedContainer:hover {
color: rgb(255, 255, 255);
text-decoration: underline;
outline: none;
background-color: rgb(0, 95, 178);
box-shadow: 0 0 12px 0 rgba(0, 0, 0, 0.5);
}

.THIS .messageContent {
 display: block;
 padding: 0 8px;
 height: 100%;
 color: rgb(255, 255, 255);
}
```

## Get Settings from the Snap-Ins Code Snippet

Get settings for use with your Snap-ins Lightning components. You can get the Live Agent button ID or deployment ID assigned to your Snap-ins deployment and the agent and chatbot avatar image URLs.

Use the Aura method `getLiveAgentSettings()` to grab the settings that you want to use: `liveAgentButtonId`, `liveAgentDeploymentId`, `chatbotAvatarImgURL`, `avatarImgURL`.

This example shows you how to use the lightningsnapin:settingsAPI component with a custom pre-chat component. The pre-chat component uses different fields and a CSS class when a specific Live Agent button is used with the snap-in.

Important: This example isn't of a complete pre-chat component. We've marked where the rest of your code should go in the code comments.

Before you start, make sure that you have a Snap-ins deployment already set up. You should also have a working Snap-ins Lightning component before you make any changes based on the Snap-ins deployment settings.

To get started, go to the Developer Console and open one of your Snap-ins Lightning components.

1. Add a line in your component markup file to create the settings API component.

```
<!-- Your pre-chat component's markup file -->
<aura:component implements="lightningsnapin:prechatUI">

    <!-- Contains a method for fetching Live Agent settings -->
    <lightningsnapin:settingsAPI aura:id="settingsAPI"/>

    <!-- The rest of your custom pre-chat component goes here -->
</aura:component>
```

2. In your `init` handler, use the Aura method `getLiveAgentSettings()` to grab the settings you want.

   This example customizes the First Name and Last Name fields when a certain Live Agent button is used by the current snap-in by:

   - Pre-populating the visitor's name as "Anonymous Visitor"
   - Making the fields read-only
   - Adding a CSS class called `anonymousField`

```
// Your pre-chat component's controller file
({
    // Your pre-chat component's init handler
    onInit: function(cmp, evt, hlp) {
        // The ID of the Live Agent button for which you want to customize your pre-chat
 fields
        var ANONYMOUS_BUTTON_ID = "(your button id here)";

        // Fetch the ID of the Live Agent button currently in use
        var buttonId = cmp.find("settingsAPI").getLiveAgentSettings().liveAgentButtonId;


        // Get your pre-chat fields. This example assumes that your pre-chat form includes
 First Name and Last Name fields.
        var prechatFields = cmp.find("prechatAPI").getPrechatFields();
        var prechatFieldComponents = prechatFields.map(function(field) {
            // If the specified button is currently in use, customize the First Name
and Last Name fields
            if (buttonId === ANONYMOUS_BUTTON_ID) {
                if (field.label === "First Name") {
                    // Pre-populate the value, make the field read-only, and add a CSS
 class
                    field.value = "Anonymous";
                    field.readOnly = true;
                    field.className += " anonymousField";
                } else if (field.label === "Last Name") {
                    field.value = "Visitor";
                    field.readOnly = true;
                    field.className += " anonymousField";
                }
            }

            return [
                "ui:inputText",
                {
                    "aura:id": "prechatField",
                    required: field.required,
                    label: field.label,
                    disabled: field.readOnly,
                    maxlength: field.maxLength,
                    class: field.className,
                    value: field.value
                }
            ];
        });

        $A.createComponents(prechatFieldComponents, function(components, status) {
            if (status === "SUCCESS") {
                cmp.set("v.prechatFieldComponents", components);
            }
        });
    },

    // The rest of your component's controller goes here
})
```

3. In your component CSS file, add a CSS rule for our new `anonymousField` CSS class.

```
/* Your pre-chat component's CSS file */

.THIS .anonymousField {
    background-color: rgba(255,0,0,0.3);
}

/* The rest of your component's CSS goes here */
```

4. Save your component.

SEE ALSO:

Lightning Components Developer Guide

## Snap-Ins Code Snippet Versions

See what features are available in previous and current versions of the Snap-ins code snippet.

### Feature Availability by Snippet Version

| Code Snippet Version | Available Features |
|---|---|
| 2.2 | • Chat persistence across subdomains<br>• Localization of labels |
| 3.1 | The features available in version 2.2, plus:<br>• Passing data into post-chat<br>• Snap-ins Appointment Booking (Pilot) |
| 4.1 | The features available in version 3.1, plus:<br>• Automated chat invitations<br>• Auto-fill fields in pre-chat with Communities or Lightning Out |

| Code Snippet Version | Available Features |
|---|---|
| | • Direct-to-button routing |
| 5.0 | The features available in version 4.1, plus: • Snap-Ins Appointment Management • Offline support • Custom chat events • Routing order • Display the customer’s place in line |