salesforce

Embedded Service Chat for Web Developer Guide

Version 60.0, Spring '24

@salesforcedocs

Last updated: January 12, 2024

CONTENTS

CHAPTER 1 Embedded Service Chat for Web Developer Guide

Improve the functionality of your Embedded Service Chat deployment by customizing parameters in the Embedded Service code snippet, adding HTML and JavaScript, or using a Lightning Web Component. See which features are supported for each version of the Embedded Service code snippet.

EDITIONS

Embedded Service setup node is available in: Lightning Experience

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

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Adjust General Parameters in the Code Snippet

Change the general parameters in your Embedded Service code snippet to improve functionality across all chat stages. Choose to resize the chat window or update the font to match branding requirements. Set the default language or change the layout for right-to-left languages. Add the domain name for chat to persist across your site's subdomains.

Change the Embedded Chat Window Appearance and Behavior

Set parameters that affect the appearance and behavior of the chat window at specific stages. Customize the pre-chat image background, logo, waiting state image, and your agent avatar and Einstein Bot pictures. Customize the text that appears on the window when chat is loading, when agents are online, and when agents are offline. Set a routing order and load your branded files for custom chat events.

Customize Embedded Chat

Take full control of the Embedded Chat experience from the static help button to post-chat stages. Use customizable parameters in the code snippet. Expand functionality by passing nonstandard pre-chat details. Set direct-to-button routing and allow pre-chat fields to fill automatically. Implement your own HTML and CSS code and more.

Customize the Channel Menu

Provide your Channel Menu customers with a more personalized experience. Make client-side changes without altering other menu branding parameters for your website.

Embedded Service Customized Components

Simplify the customization process for Embedded Service using HTML or modern JavaScript with Lightning Web Components. Aura Components are still available but less flexible or try multiple components on your web page with Lightning Out.

Embedded Service Code Snippet Versions

See what features are available in previous and current versions of the Embedded Service code snippet.

SEE ALSO:

Embedded Service for Web and Mobile Apps

Embedded Service for Mobile Apps Product Page

Adjust General Parameters in the Code Snippet

Change the general parameters in your Embedded Service code snippet to improve functionality across all chat stages. Choose to resize the chat window or update the font to match branding requirements. Set the default language or change the layout for right-to-left languages. Add the domain name for chat to persist across your site's subdomains.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Set the Width, Height, and Base Font Size

Change the default sizes for your chat window in the code snippet to match your websites requirements.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Keep the following in mind when setting your sizes:

- You can enter values in pixels (px), 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 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 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. 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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

`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 chat window 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

To customize the language for a deployment, including custom labels, set the parameter to a Salesforce supported language. This parameter is empty and must be set in your Embedded Service code snippet for chat.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

For example, enter 'en' or 'en-US' for English. `embedded_svc.settings.language = en_US`

Note: Use only the `http` locale format (like `en-US` or `en`).

Important considerations for language translations:

- If no language is set in the code snippet or is invalid (empty code string, wrong format, or unsupported), the **Site Guest User** language for the site associated with your Embedded Service deployment is used.
- When Translation Workbench is disabled, the language set for the **Site Guest User** of the Site associated with your Embedded Service deployment is used.
- The language on a label is set only with the `embedded_svc.settings.language` parameter or the **Site Guest User** language, and not the chat button.

SEE ALSO:

Supported Languages

Set Right-to-Left Language Layout

This parameter is included in your generated code snippet and set to English for versions 5.0 and later. To enable right-to-left layout enhancements in Chat, set the two-letter value to the supported language in your code snippet.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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

There are other settings to take care of to make sure that translation works properly. Use the Salesforce Help for guidance.

Change the Embedded Chat Window Appearance and Behavior

Set parameters that affect the appearance and behavior of the chat window at specific stages. Customize the pre-chat image background, logo, waiting state image, and your agent avatar and Einstein Bot pictures. Customize the text that appears on the window when chat is loading, when agents are online, and when agents are offline. Set a routing order and load your branded files for custom chat events.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Specify Chat Images

Use your own images for the pre-chat banner, header logo, waiting state image, and the agent and Einstein Bots avatars. We recommend adding your images in Embedded Service setup, but you can use these settings to override what you created in setup.

Display the Default Chat Button

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

Customize the Online, Offline, and Loading Chat Text

Set the text that's displayed to your customers in the chat window when there are agents available, when there aren't agents available, and when the chat is connecting to an agent. We recommend customizing your labels in Embedded Service setup, but you can use these settings to override what you have in Setup.

Set a Routing Order

Set a list of user IDs and button IDs on your Embedded Service 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.

Load Files for Custom Chat Events

Load your own JavaScript and CSS files into Embedded Chat to handle and style custom chat events. Your scripts and styles are loaded only after the agent accepts the chat request.

Modify the Post-Chat

After you add a post-chat URL for your chat button, you can specify whether it automatically opens or not. Valid values are `true` and `false`.

Specify Chat Images

Use your own images for the pre-chat banner, header logo, waiting state image, and the agent and Einstein Bots avatars. We recommend adding your images in Embedded Service setup, but you can use these settings to override what you created in setup.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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

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

Pre-Chat Banner

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

Specify a URL to set an image in the pre-chat form below the header and above the fields.

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

Logo for Header and Minimized Chat

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

Specify a URL to set the logo for the chat header and minimized chat.

Tip: We recommend a logo that's 36x36 pixels.

Waiting State

```
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 250x60 pixels.

Agent Avatar

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

Specify a URL to set the image of the agent that appears during pre-chat and chat. If your chat window 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 chat window so they can start a chat from your web page. Valid values are `true` and `false`.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

`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 chat window when there are agents available, when there aren't agents available, and when the chat is connecting to an agent. We recommend customizing your labels in Embedded Service setup, but you can use these settings to override what you have in Setup.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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 Embedded Service 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.loadingText = "..."`

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.

Note: `embedded_svc.settings.loadingText` replaced `embedded_svc.settings.onlineLoadingText` for version 3.0 and later code snippets.

Set a Routing Order

Set a list of user IDs and button IDs on your Embedded Service 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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

`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 Embedded Service 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.

Note: The language on a label is set only with the `embedded_svc.settings.language` parameter, not the chat button.

SEE ALSO:

Fallback Routing in Pre-Chat Forms

Load Files for Custom Chat Events

Load your own JavaScript and CSS files into Embedded Chat to handle and style custom chat events. Your scripts and styles are loaded only after the agent accepts the chat request.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Upload your files as Static Resources with the cache control set to public. Assign names that are easy to remember and with no spaces. You will reference them by static resource name in Embedded 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`*.

Modify the Post-Chat

After you add a post-chat URL for your chat button, you can specify whether it automatically opens or not. Valid values are `true` and `false`.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

```
embedded_svc.settings.autoOpenPostChat = true; // or false
```

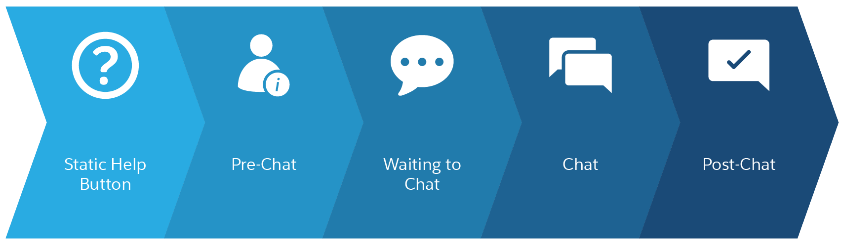
By default, your chat window gives customers the option of closing the chat or opening the post-chat URL when they end a chat. To automatically display the post-chat URL instead, set this parameter to `true`.

Customize Embedded Chat

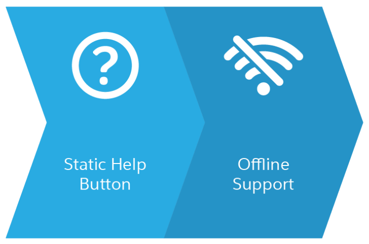
Take full control of the Embedded Chat experience from the static help button to post-chat stages. Use customizable parameters in the code snippet. Expand functionality by passing nonstandard pre-chat details. Set direct-to-button routing and allow pre-chat fields to fill automatically. Implement your own HTML and CSS code and more.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Here are the five stages of Embedded Chat to consider when using these developer guidelines.



An additional stage to consider if you've activated offline support.



Expand the Pre-Chat Stage

Before starting a conversation with your customer at the pre-chat stage, consider adding customized invitations that appear on your website. Improve the pre-chat page or create a snippet settings file for greater flexibility and to provide more information to agents.

Boost Chat Stage

Your customer is chatting with an agent at the chat stage. Consider adding chat event notifications and custom chat events to improve the agent view of their needs.

Add Special APIs

Embedded Chat can be customized across several chat stages, for example, the Start, End and Clear or the Bootstrap APIs.

Expand the Pre-Chat Stage

Before starting a conversation with your customer at the pre-chat stage, consider adding customized invitations that appear on your website. Improve the pre-chat page or create a snippet settings file for greater flexibility and to provide more information to agents.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Enhance the Pre-Chat Page for Embedded Chat

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

Set Up and Customize an Automated Invitation

Connect an automated chat invitation with your Embedded Service 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.

Get Chat Event Notifications

Set up notifications when certain chat events are triggered. Subscribe to these particular events by calling to `embedded_svc.addEventHandler` in your Embedded Chat code snippet.

Create a Snippet Settings File for an Experience Site

Take your snippet-only settings like extra pre-chat configuration or direct-to-button routing to your Experience site. Create a JavaScript file and upload it as a static resource that you reference in your Embedded Chat component settings.

Add Mobile Accessibility for Chat

Provide an accessible Embedded Chat experience on your website for mobile customers.

Enhance the Pre-Chat Page for Embedded Chat

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

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Pass Nonstandard Pre-Chat Details

Further control the pre-chat experience using parameters in your Embedded Service 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.

Follow Pre-Chat Code Examples

During the pre-chat stage: find existing contacts, avoid attaching records to transcripts, attach a record to an existing field, or override a specific field in your org. These examples and more illustrate some common use cases for pre-chat code snippets.

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

Set your chat window 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 Pre-Chat Form Fields to Automatically Populate when Customers Log In

When your customers log in, agents 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 embedded chat windows that are placed outside of Salesforce with Lightning Out (beta). If you use your embedded window in Experience sites, you can enable the contact fields to fill in automatically in the Embedded Chat component settings.

Pass Nonstandard Pre-Chat Details

Further control the pre-chat experience using parameters in your Embedded Service 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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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.

Name	Type	Description
`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

Name	Type	Description
		conduct a search with the `findOrCreate.map` API method. See findOrCreate.map.isExactMatch in the Live Agent Developer Guide.
`saveToTranscript`	String	The name of the transcript field to which to save the record. This field must 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

Follow Pre-Chat Code Examples

During the pre-chat stage: find existing contacts, avoid attaching records to transcripts, attach a record to an existing field, or override a specific field in your org. These examples and more illustrate some common use cases for pre-chat code snippets.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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 Embedded Chat 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 chat window 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 chat window to let them know that no one’s online.

The designated button ID must be from 15 to 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 or later 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 fourth pre-chat field. You would enter the following:

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

Note: The language on a label is set only with the `embedded_svc.settings.language` parameter, not the chat button.

Set Pre-Chat Form Fields to Automatically Populate when Customers Log In

When your customers log in, agents 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 embedded chat windows that are placed outside of Salesforce with Lightning Out (beta). If you use your embedded window in Experience sites, you can enable the contact fields to fill in automatically in the Embedded Chat component settings.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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 Embedded Chat 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"
};
```

Set Up and Customize an Automated Invitation

Connect an automated chat invitation with your Embedded Service 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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Before you write any code, set up your invitation and Embedded Service 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 Chat deployment that you plan to use for your Embedded Service deployment. Then create or edit a Embedded Service deployment and select the Chat deployment and invitation for your Chat Settings.

Keep in mind that there are some differences to how invitations work in Embedded Chat versus a regular Chat deployment:

- The position and animation don't apply to customers using a mobile browser. They see the invitation above the chat button with the "fade" animation type.
- Custom animations aren't supported.
- The same fields that aren't supported for Embedded Chat buttons aren't supported for Embedded Chat 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 Embedded Chat component in your Experience site.

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 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 Embedded Chat deployment and regenerate the snippet. The default invitation uses the font, primary color, and secondary color that you selected in Embedded Service setup.
- When you set an avatar image (set 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 to override in the Embedded Service code snippet:

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

If you're using custom variable rules, 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 version 4.0 and later code snippet when you add an invitation to an Embedded Chat deployment and regenerate the code snippet.

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

```
<!-- 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 -->
```

Get Chat Event Notifications

Set up notifications when certain chat events are triggered. Subscribe to these particular events by calling to `embedded_svc.addEventHandler` in your Embedded Chat code snippet.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Important: Calls to `embedded_svc.addEventHandler` must take place before calls to `embedded_svc.init`. If your code snippet contains calls to `embedded_svc.init`, make sure that you enter your calls in the correct order.

The following events pass a JSON object parameter to the provided callback which contains one attribute:

```
{    liveAgentSessionKey: chasitorData.chatKey    }
```

eventName	Scenario
`onAgentJoinedConference`	Fired when an agent joins the chat conference.

eventName	Scenario
`onAgentLeftConference`	Fired when an agent leaves the chat conference.
`onAgentMessage`	Fired when the agent sends a message.
`onAgentRichMessage`	Fired when the bot sends a rich message. The bot sends a mixture of rich messages and plain messages.
`onChasitorMessage`	Fired when the chat visitor sends a message.
`onChatConferenceEnded`	Fired when the chat conference has ended.
`onChatConferenceInitiated`	Fired when the chat conference is initiated.
`onChatEndedByAgent`	Fired when the agent ends the chat.
`onChatEndedByChasitor`	Fired when the chat visitor ends the chat.
`onChatEndedByChatbot`	Fired when the bot ends a chat.
`onChatReconnectSuccessful`	Fired when the chat reconnects successfully.
`onChatTransferInitiated`	Fired when any transfer request occurs.
`onChatRequestSuccess`	Fired when the chat request is successful.
`onChatTransferInitiated`	Fired when any transfer request occurs.
`onChatTransferSuccessful`	Fired when a chat transfer is successful.
`onConnectionError`	Fired when the connection to the agent is lost.
`onIdleTimeoutClear`	Fired when the visitor responds after an idle timeout warning is visible, which will clear the warning.
`onIdleTimeoutOccurred`	Fired when a chat times out due to the visitor being idle. The chat ends and the visitor sees a message that the chat has ended.
`onIdleTimeoutWarningStart`	Fired when the visitor has not responded to the agent during the time set in the Customer Time-out Warning (seconds) field in Chat Buttons & Invitations Setup.
`onQueueUpdate`	Fired in the following scenarios: • The visitor has requested a chat and is waiting for an agent. • After requesting a chat, the visitor navigates to another page, but is still waiting for an agent. • The visitor had previously lost connection (see `reconnectingState`) and has regained it. • The visitor has advanced in the queue, but is still waiting for an agent to accept the chat request. • A bot or agent has requested a chat transfer and is waiting for another agent to accept. • During a chat transfer, the visitor navigates to another page, but is still waiting to be transferred.

eventName	Scenario
	• The visitor has advanced in the queue, but is still waiting for an agent to accept the chat transfer. Note: This event fires only if queue position is enabled for your Embedded Service deployment and your Embedded Service code snippet is version 5.0 or later.

The following general application-level events to the client are not specific to a Chat session, but can be used during Chat. These events do not broadcast any data.

eventName	Scenario
`afterMaximize`	Fired when the Embedded Service Chat application is maximized by the end user clicking the minimized state.
`afterDestroy`	Fired when Embedded Service Chat has ended and the application is closed.
`afterMinimize`	Fired when the Embedded Service Chat application is minimized by the end user clicking the minimize button in the chat window header.
`onClickSubmitButton`	Click handler for the "Submit" button in the offline support UI component.
`onHelpButtonClick`	Callback to fire when the help button is clicked.
`onInvitationResourceLoaded`	Fired after the automated chat invitation static resource is loaded.
`onInviteAccepted`	Accepts the automated chat invitation.
`onInviteRejected`	Rejects the automated chat invitation.

The following general application-level events to the client are not specific to a Chat session, but can be used during Chat. These events do broadcast data.

eventName	Scenario
`onAvailability`	Fires in a loop every 60 seconds and indicates if the agent is online. The data parameter includes: • isAgentAvailable • id
`onSettingsCallCompleted`	Fired after the `getSettings` call is completed. The call happens on page load and retrieves settings for the chat button requested. Includes the initial agent availability status of the button on the snippet. The data parameter includes: • isAgentAvailable

Code Example

The following code is an example of how these events can be used in your Embedded Service Chat code snippet.

```
embedded_svc.addEventHandler("onHelpButtonClick", function(data) {
    console.log("onHelpButtonClick event was fired.");
});

embedded_svc.addEventHandler("onChatRequestSuccess", function(data) {
    console.log("onChatRequestSuccess event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatEstablished", function(data) {
    console.log("onChatEstablished event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChasitorMessage", function(data) {
    console.log("onChasitorMessage event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onAgentMessage", function(data) {
    console.log("onAgentMessage event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatConferenceInitiated", function(data) {
    console.log("onChatConferenceInitiated event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onAgentJoinedConference", function(data) {
    console.log("onAgentJoinedConference event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onAgentLeftConference", function(data) {
    console.log("onAgentLeftConference event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatConferenceEnded", function(data) {
    console.log("onChatConferenceEnded event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatTransferSuccessful", function(data) {
    console.log("onChatTransferSuccessful event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatEndedByChasitor", function(data) {
    console.log("onChatEndedByChasitor event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onChatEndedByAgent", function(data) {
    console.log("onChatEndedByAgent event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onQueueUpdate", function(data) {
    console.log("onQueueUpdate event was fired. liveAgentSessionKey was " +
data.liveAgentSessionKey + "and queuePosition was " + data.queuePosition);
});

embedded_svc.addEventHandler("onIdleTimeoutOccurred", function(data) {
    console.log("onIdleTimeoutOccurred event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onConnectionError", function(data) {
    console.log("onConnectionError event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onClickSubmitButton", function(data) {
    console.log("onClickSubmitButton event was fired.  liveAgentSessionKey was " +
data.liveAgentSessionKey);
});

embedded_svc.addEventHandler("onInviteAccepted", function() {
    console.log("onInviteAccepted event was fired.");
});

embedded_svc.addEventHandler("onInviteRejected", function() {
    console.log("onInviteRejected event was fired.");
});

embedded_svc.addEventHandler("onInvitationResourceLoaded", function() {
    console.log("onInvitationResourceLoaded event was fired.");
});

embedded_svc.addEventHandler("onSettingsCallCompleted", function(data) {
    console.log("onSettingsCallCompleted event was fired. Agent availability status is "
+ data.isAgentAvailable ? "online": "offline");
});

embedded_svc.addEventHandler("onAvailability", function(data) {
    console.log("onAvailability event was fired. Agent availability status is " +
data.isAgentAvailable ? "online": "offline");
});
```

Create a Snippet Settings File for an Experience Site

Take your snippet-only settings like extra pre-chat configuration or direct-to-button routing to your Experience site. Create a JavaScript file and upload it as a static resource that you reference in your Embedded Chat component settings.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: Automated chat invitations are available in Experience sites using static resources (see Chat Code Settings to Experience Sites).

Note: In this file, you can't set pre-chat fields to fill in for logged-in users. Use the Fill-in pre-chat fields option in your Embedded Chat component settings.

1. Create a Snippet Settings File Using JavaScript

In your JavaScript file, define the settings you want to use in your site's chat window.

Table 1: Available Settings

Setting Name	To Specify in Your File
Auto-open for post chat	`embedded_svc.snippetSettingsFile.autoOpenPostChat = true; // or false`
Avatar image URL	`embedded_svc.snippetSettingsFile.avatarImgURL = '';`
Chatbot (Einstein Bots) image URL	`embedded_svc.snippetSettingsFile.chatbotAvatarImgURL = '';`
Company logo image URL	`embedded_svc.snippetSettingsFile.smallCompanyLogoImgURL = '';`
Direct-to-button routing	`embedded_svc.snippetSettingsFile.directToButtonRouting = function(prechatFormData) {};`
External scripts	`embedded_svc.snippetSettingsFile.externalScripts = [];`
External styles	`embedded_svc.snippetSettingsFile.externalStyles = [];`
Extra pre-chat form details	`embedded_svc.snippetSettingsFile.extraPrechatFormDetails = [];`
Extra pre-chat information	`embedded_svc.snippetSettingsFile.extraPrechatInfo = [];`
Pre-chat background image URL	`embedded_svc.snippetSettingsFile.prechatBackgroundImgURL = '';`
Routing order	`embedded_svc.snippetSettingsFile.fallbackRouting = [];`
Waiting state background image URL	`embedded_svc.snippetSettingsFile.waitingStateBackgroundImgURL = '';`

Important: Your settings are applied first from the Embedded Chat component settings in the Experience Builder, then from your JavaScript file, then from Setup. For example:

- If you set an avatar image in your JavaScript file but leave the setting blank in Setup, the image you set in your JavaScript file displays.
- Let's say you include an agent avatar in Setup but don't specify one in your Embedded Chat component settings or in your JavaScript file. Then the image you included in Setup displays.

2. Upload Your File to Static Resources

Upload your file as a static resource in Setup. If you can't access Setup, ask your Salesforce admin for help.

When you upload your file, make sure that you:

- Select Public for the cache control.
- Give it a name that's easy to remember and type. You use the static resource name, not the file name, to reference the file from the Embedded Chat component settings.

3. Reference Your File in Your Embedded Chat Component Settings

In the Experience Builder, enter the static resource name (not the file name) in the **Snippet Settings File** field. If you can't access the Experience Builder, ask your Salesforce admin for help.

For example, if your JavaScript file is called `SnapInCodeSnippetSettings.js` and you named it `SnippetSettings` in your static resources, enter *`SnippetSettings`* in the **Snippet Settings File** field.

Example of a Snippet Settings File

Note: This example doesn't use real URLs, file names, user IDs, button IDs, or email addresses. Enter your own information if you copy parts of this example for your own snippet settings file.

```
window._snapinsSnippetSettingsFile = (function() {
console.log("Snippet settings file loaded."); // Logs that the snippet settings file was
loaded successfully

embedded_svc.snippetSettingsFile.avatarImgURL = 'https://yourwebsite.here/avatar.jpg';
embedded_svc.snippetSettingsFile.smallCompanyLogoImgURL =
'https://yourwebsite.here/company_logo.png';
embedded_svc.snippetSettingsFile.prechatBackgroundImgURL =
'https://yourwebsite.here/prechat_background.jpg';
embedded_svc.snippetSettingsFile.waitingStateBackgroundImgURL =
'https://yourwebsite.here/waiting_background.png';
embedded_svc.snippetSettingsFile.headerBackgroundURL =
'https://yourwebsite.here/header_background.jpg';
embedded_svc.snippetSettingsFile.chatbotAvatarImgURL =
'https://yourwebsite.here/bot_avatar.jpg';
embedded_svc.snippetSettingsFile.autoOpenPostChat = true;

embedded_svc.snippetSettingsFile.externalScripts = ['my_scripts'];
embedded_svc.snippetSettingsFile.externalStyles = ['my_styles'];

embedded_svc.snippetSettingsFile.directToButtonRouting = function(prechatFormData) {
if(prechatFormData[1].value === "Computer") {
```

```
console.log("direct to button routing initiated.");
alert("Alert: direct to button routing initiated!");
return "BUTTONIDHERE";
}
}

embedded_svc.snippetSettingsFile.fallbackRouting = ['USERIDHERE', 'BUTTONIDHERE',
'USERID_BUTTONID'];


embedded_svc.snippetSettingsFile.extraPrechatFormDetails =
[{"label":"FirstName","value":"John","displayToAgent":true},
{"label":"LastName","value":"Doe","displayToAgent":true},
{"label":"Email","value":"john.doe@salesforce.com","displayToAgent":true}];

embedded_svc.snippetSettingsFile.extraPrechatInfo = [{
"entityName": "Contact",
"showOnCreate": true,
"linkToEntityName": "Case",
"linkToEntityField": "ContactId",
"saveToTranscript": "ContactId",
"entityFieldMaps" : [{
"doCreate":true,
"doFind":true,
"fieldName":"FirstName",
"isExactMatch":true,
"label":"First Name"
}, {
"doCreate":true,
"doFind":true,
"fieldName":"LastName",
"isExactMatch":true,
"label":"Last Name"
}, {
"doCreate":true,
"doFind":true,
"fieldName":"Email",
"isExactMatch":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"
}]
}];
})();
```

Updating Your Snippet Settings File

When you change your JavaScript file, follow these steps. They help you to ensure that you don't have to regenerate your Embedded Service code snippet or edit your Embedded Chat component settings.

1. Go to Static Resources in Setup.
2. Next to your static resource for your old JavaScript file, click **Edit**.
3. Upload your updated JavaScript file.
4. Click **Save**.

Your static resource name doesn't change, so you don't have to modify any additional settings.

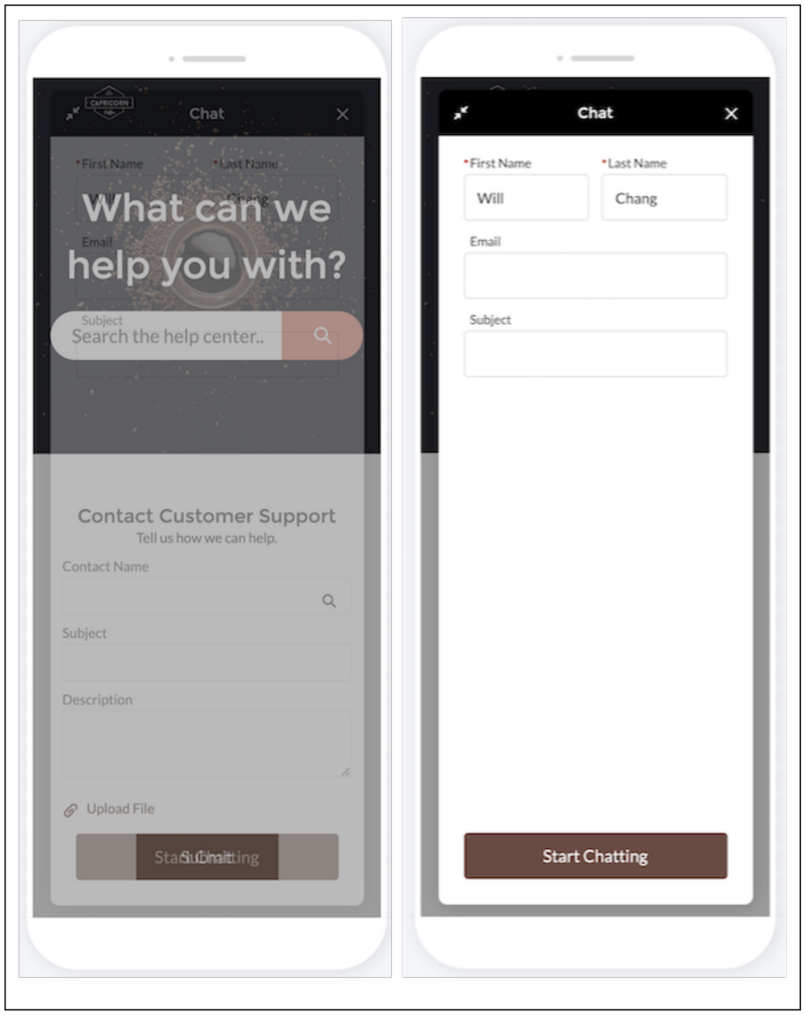
Add Mobile Accessibility for Chat

Provide an accessible Embedded Chat experience on your website for mobile customers.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Most browsers don't support hiding background content for screen readers on mobile phones when the Chat modal opens. By adding hooks to the code, you streamline the experience for these users to focus on Chat. Once the window is closed, the full web page content returns.

The phone on the left shows what the screen reader accesses when the modal opens. The one on the right demonstrates the experience with the inert function added.



This solution provides code that temporarily hides document body children and keep only the Embedded Service Web widget accessible and visible to assistive technology.

Note: This solution is not available for Experience sites.

Start with a base page and our code snippet with an empty `onBodyLoaded` JavaScript method. The body page content is wrapped with a div with the id #body-content and the body has an `onload` handler specified. You'll use the CSS selector div.#body-content and implement the JavaScript method `onBodyLoaded()` after adding your snippet.

Base Page Code with Embedded Chat Code Snippet

```
<html>

<head>
   <meta name="viewport" content="width=device-width, initial-scale=1, minimum-scale=1,
maximum-scale=1, user-scalable=0">
   <script type='text/javascript'>
        function onBodyLoaded() {
            // You’ll fill in this function during the next step.
        }
   </script>
</head>

<body onload="onBodyLoaded()">
   <div id="body-content">
       <h1>Welcome to the best website!</h1>

       <style type='text/css'>
           .embeddedServiceHelpButton .helpButton .uiButton {
               background-color: #A70BA5;
               font-family: "Comic Sans MS", sans-serif;
           }
           .embeddedServiceHelpButton .helpButton .uiButton:focus {
               outline: 1px solid #A70BA5;
           }
           .previews img {
               max-width: 100%;
               border-radius: 10px;
           }
           .chat-content a {
               color: #fff;
           }
       </style>
   </div>

   <script type='text/javascript'
src='https://service.force.com/embeddedservice/5.0/esw.min.js'></script>
   <script type='text/javascript'>
       var initESW = function (gslbBaseURL) {
           embedded_svc.settings.displayHelpButton = true; //Or false
           embedded_svc.settings.language = ''; //For example, enter 'en' or 'en-US'
           embedded_svc.settings.enabledFeatures = ['LiveAgent'];
           embedded_svc.settings.entryFeature = 'LiveAgent';

           embedded_svc.init(
               // params for your org
           );
       };

       if (!window.embedded_svc) {
           var s = document.createElement('script');
           s.setAttribute('src', 'https://<yourCoreURL>/embeddedservice/5.0/esw.min.js');

           s.onload = function () {
               initESW(null);
           };
           document.body.appendChild(s);
       } else {
           initESW('https://service.force.com');
       }
   </script>
</body>

</html>
```

Next implement the JavaScript method `onBodyLoaded` to disable the div with id body-content whenever Embedded Chat is opened and re-enable body-content whenever Embedded Chat is minimized/closed. Define the JavaScript method `onBodyLoaded` in this step. For more help on adding event handlers for maximize, destroy, and minimize, see "Get Chat Event Notifications."

`onBodyLoaded` Code

```
function onBodyLoaded() {
   /**
     * Toggles the inert attribute on background content for the page.
     * The inert attribute needs to be set on parent level DOM
     * nodes. Inert will set the DOM node aria-hidden attribute and
     * and set tab-index="-1" on all children so that screen readers
     * can't access the content.
     */
   function toggleInert(disabled) {
       let bodyElem = document.getElementById("body-content");
       if(bodyElem) {
           bodyElem.inert = disabled;
       }
   }

   // Add hooks to toggle inert on our page when Embedded Chat
   // changes state.
   function addA11yHooks() {
       embedded_svc.addEventHandler('afterMaximize', function() {
           toggleInert(true);
       });
       embedded_svc.addEventHandler('afterDestroy', function() {
           toggleInert(false);
       });
       embedded_svc.addEventHandler('afterMinimize', function() {
           toggleInert(false);
       });
   }

   // Early out for desktop.
   if(embedded_svc.isDesktop()) {
       return;
   }

   // Add hooks for mobile.
   addA11yHooks();
}
```

Here's the combined code for the steps you completed.

Mobile Accessible Embedded Chat Example Code

```
<html>

<head>
   <meta name="viewport" content="width=device-width, initial-scale=1, minimum-scale=1,
maximum-scale=1, user-scalable=0">
```

```
   <script type='text/javascript'>
       function onBodyLoaded() {
           /**
             * Toggles the inert attribute on background content for the page.
             * The inert attribute just needs to be set on parent level DOM
             * nodes. Inert will set the DOM node aria-hidden attribute and
             * and set tab-index="-1" on all children so that screen readers
             * can't access the content.
             */
           function toggleInert(disabled) {
               let bodyElem = document.getElementById("body-content");
               if(bodyElem) {
                   bodyElem.inert = disabled;
               }
           }

           // Add hooks to toggle inert on our page when Embedded Chat
           // changes state.
           function addA11yHooks() {
               embedded_svc.addEventHandler('afterMaximize', function(data) {
                   toggleInert(true);
               });
               embedded_svc.addEventHandler('afterDestroy', function(data) {
                   toggleInert(false);
               });
               embedded_svc.addEventHandler('afterMinimize', function(data) {
                   toggleInert(false);
               });
           }

           // Early out for desktop.
           if(embedded_svc.isDesktop()) {
               return;
           }

           // Add hooks for mobile.
           addA11yHooks();
       }
   </script>
</head>

<body onload="onBodyLoaded()">
   <div id="body-content">
       <h1>Welcome to the best website!</h1>

       <style type='text/css'>
           .embeddedServiceHelpButton .helpButton .uiButton {
               background-color: #A70BA5;
               font-family: "Comic Sans MS", sans-serif;
           }
           .embeddedServiceHelpButton .helpButton .uiButton:focus {
               outline: 1px solid #A70BA5;
           }
           .previews img {
               max-width: 100%;
               border-radius: 10px;
           }
           .chat-content a {
               color: #fff;
           }
       </style>
   </div>

   <script type='text/javascript'
src='https://service.force.com/embeddedservice/5.0/esw.min.js'></script>
   <script type='text/javascript'>
       var initESW = function (gslbBaseURL) {
           embedded_svc.settings.displayHelpButton = true; //Or false
           embedded_svc.settings.language = ''; //For example, enter 'en' or 'en-US'
           embedded_svc.settings.enabledFeatures = ['LiveAgent'];
           embedded_svc.settings.entryFeature = 'LiveAgent';

           embedded_svc.init(
               // params for your org
           );
       };

       if (!window.embedded_svc) {
           var s = document.createElement('script');
           s.setAttribute('src', 'https://<yourCoreURL>/embeddedservice/5.0/esw.min.js');

           s.onload = function () {
               initESW(null);
           };
           document.body.appendChild(s);
       } else {
           initESW('https://service.force.com');
       }
   </script>
</body>

</html>
```

SEE ALSO:

Get Chat Event Notifications

Boost Chat Stage

Your customer is chatting with an agent at the chat stage. Consider adding chat event notifications and custom chat events to improve the agent view of their needs.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Embedded Chat Custom Events

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

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.

Embedded Chat Custom Events

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

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

`embedded_svc.liveagentAPI.sendCustomEvent()`

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

Table 2: Arguments

Name	Type	Description
`type`	string	The name of the custom event to send to the agent console.
`data`	string	Extra data you want to send to the agent console along with the custom event.

`embedded_svc.liveagentAPI.getCustomEvents()`

Retrieves a list of custom events from both the agent and chat visitor that are received during this chat session.

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.

`embedded_svc.liveagentAPI.addCustomEventListener()`

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

Table 4: Arguments

Name	Type	Description
`type`	string	The type of custom event you want to listen for.
`callback`	function	JavaScript method called upon completion of the method. It passes an object that has two attributes: type and data.

SEE ALSO:

Create Custom Chat Events

Load Files for Custom Chat Events

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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

To create your events, 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()` on page 39—Sends a custom event to the agent console of the agent who is currently chatting with a chat visitor.
- `embedded_svc.liveagentAPI.getCustomEvents()` on page 39—Retrieves a list of custom events from both the agent and chat visitor that are received during a chat session.
- `embedded_svc.liveagentAPI.addCustomEventListener()` on page 39—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 chat window, 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 Embedded Service 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 you are 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 chat window, 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 is 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 Embedded Service 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_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 chat window, 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`*.

   By listening to `onChatStateLoaded` and `onCustomScriptsLoaded` events (added as part of the code snippet), you can determine when the scripts are loaded to send or receive custom events with the agent. These events will also include whether or not the scripts are loaded on the primary tab.

SEE ALSO:

Load Files for Custom Chat Events

Embedded Chat Custom Events

`sendCustomEvent()`

`onCustomEvent()`

Add Special APIs

Embedded Chat can be customized across several chat stages, for example, the Start, End and Clear or the Bootstrap APIs.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Bootstrap Embedded Chat

This API provides developers with a quick Chat setup that skips the static help button stage, loading all the necessary dependencies, bootstraps, and opens the chat application on your website with one call. You can then quickly add a custom chat button, for example, by replacing the default button with more code changes.

Start, End, and Clear Embedded Chat Sessions

Three APIs allow you to start, end, and clear sessions for Embedded Chat. Use the Embedded Service code snippet version 5.0 and later. The APIs are not available for Experience sites with Lightning Locker enabled.

Bootstrap Embedded Chat

This API provides developers with a quick Chat setup that skips the static help button stage, loading all the necessary dependencies, bootstraps, and opens the chat application on your website with one call. You can then quickly add a custom chat button, for example, by replacing the default button with more code changes.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: Use the Embedded Service code snippet version 5.0 and later. The API can only be used on a page that includes the snippet. This API isn't available for Experience sites.

`embedded_svc.bootstrapEmbeddedService(attributes);`

This function loads scripts and dependencies and creates the Embedded Chat application. It returns a Promise object that resolves after the Embedded Chat application is created. The Promise rejects on any error thrown during the bootstrapping.

Use the optional `attributes` JSON object parameter to pass information to the Embedded Chat Bootstrap API to launch the chat window.

Your `attributes` parameter should include the attributes from the table.

Name	Type	Description
`baseCoreURL`	string	The URL of an organization's instance.
`communityEndpointURL`	string	The URL representing a community endpoint.

The `attributes` parameter is optional. If you don't provide this parameter, the function uses what is stored in `embedded_svc.settings`.

The result of using this API is the same for a customer clicking the standard "Chat With an Expert" button. If you have pre-chat enabled, it surfaces the pre-chat form. If you've disabled pre-chat, it sends a chat request and surface the waiting state.

Note: See Start, End, and Clear Embedded Chat Sessions for an API related to this topic.

Add a Custom Chat Button

Here's an example of how to apply this API. Hide the default Salesforce chat button and replace it with a button that matches your company's brand.

To hide the standard button, include this line in your embedded code snippet: `embedded_svc.settings.displayHelpButton = false;`.

On click of your company's branded button, invoke `embedded_svc.bootstrapEmbeddedService();` to surface the chat experience for your end user.

To show a basic button that invokes the `bootstrapEmbeddedService()` function:

```
<script type='text/javascript'>
 function bootstrapChat() {
  embedded_svc.bootstrapEmbeddedService();
 }
</script>
<button onclick="bootstrapChat()">
 Click me to show the Embedded Chat experience.
</button>
```

Start, End, and Clear Embedded Chat Sessions

Three APIs allow you to start, end, and clear sessions for Embedded Chat. Use the Embedded Service code snippet version 5.0 and later. The APIs are not available for Experience sites with Lightning Locker enabled.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: Embedded Chat must be a menu item on a Channel Menu deployment for Chat APIs to work with a Channel Menu. This API isn't available for Experience sites.

`embedded_svc.liveAgentAPI.startChat(attributes);`

The Start Chat API launches the widget and initiates a chat request. Use the `attributes` object to pass more information (contact or case details) to the API. For example, directly route to another `buttonId` or a specific `agentId` based on logic you determine. Any parameters must be passed in as an object. Your code might look like this:

```
embedded_svc.liveAgentAPI.startChat({
 directToAgentRouting: {
 buttonId: "573xx0000000000",
 // userId: "",
 fallback: true
},
extraPrechatInfo: [],
extraPrechatFormDetails: []
});
```

To start a chat, the `attributes` object contains any of these Embedded Service parameters from the table.

Name	Type	Description
`directToAgentRouting`	object	Direct agent routing supported only in Start Chat. • `buttonId` (string) – The button ID to request a chat (required). • `userId` (string) – The agent ID to directly route from specified button (optional, defaults to snippet buttonId). • `fallback` (boolean) – If set as true, follows the button's `fallbackRouting` rules if the button or agent is not available (required, defaults to false).
`prepopulatedPrechatFields`	object	See: Set Certain Pre-Chat Form Fields to Automatically Populate Note: If Pre-Chat is disabled, this setting is not supported for Start Chat.
`extraPrechatInfo`	array	See: Pass Nonstandard Pre-Chat Details
`extraPrechatFormDetails`	array	See: Pass Nonstandard Pre-Chat Details
`fallbackRouting`	array	See: Set Routing Order
`directToButtonRouting`	function	See: Route Chats Based on Pre-Chat Responses

Important: Using the API overrides the Embedded Service code snippet settings. If you don't pass in settings to the API, you override any existing settings on the snippet.

Limitations

- Start Chat maximizes the widget if it's minimized. Calling the API when the end user is in an active chat request or session doesn't initiate another chat request.
- Start Chat attributes are not supported for Offline Support and don't pre-populate offline support forms or submit cases.
- To avoid opening the widget in No Agents Available dialog or Offline Support states, use the `OnSettingsCallCompleted` event to ensure that agents are available first.

`embedded_svc.liveAgentAPI.endChat();`

The End Chat API ends the current chat (if active) and close the chat widget. It ends active chat requests or ongoing chat sessions (as ended by the visitor) and clears embedded service data related to the ongoing chat session. No parameters are required.

Note: End Chat is not supported for the Post-Chat state.

`embedded_svc.liveAgentAPI.clearSession();`

The Clear Sessions API ends any chat connections (if active), close the chat widget, and delete session data pertaining to all client-side chat sessions. No parameters are required.

SEE ALSO:

Create Custom Chat Events

Bootstrap Embedded Chat

Customize the Channel Menu

Provide your Channel Menu customers with a more personalized experience. Make client-side changes without altering other menu branding parameters for your website.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

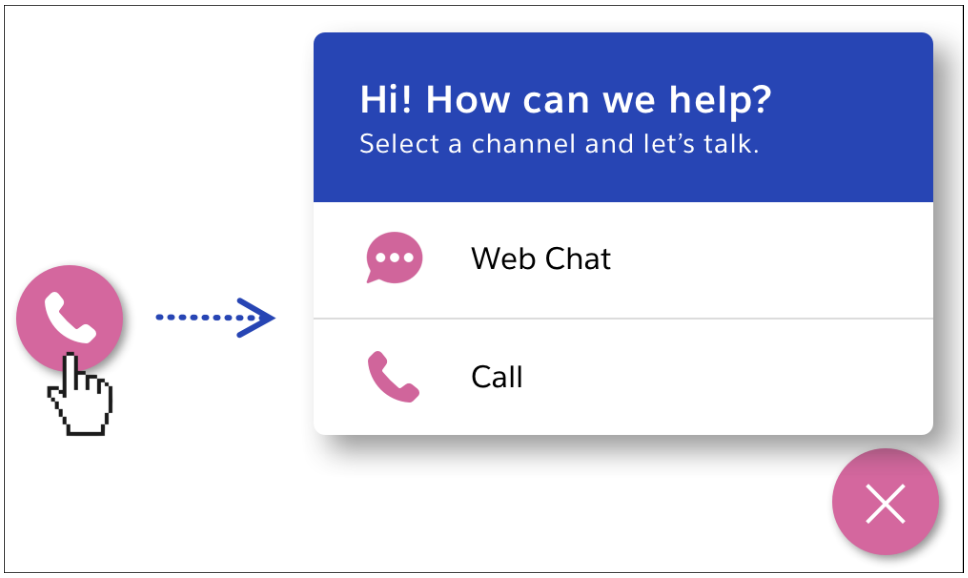
Channel Menu Reordering

Show, hide, or reorder your Channel Menu selections as customers browse your website for a more dynamic experience. Change the floating action button and channel options offered, depending on the user's page location or how long they're on the page.

Channel Menu Reordering

Show, hide, or reorder your Channel Menu selections as customers browse your website for a more dynamic experience. Change the floating action button and channel options offered, depending on the user's page location or how long they're on the page.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.



This API can reorder any channels configured in setup (`EmbeddedServiceMenuItem`), and hide/show items as needed based on the user's interaction on the web page.

To create a dynamic deployment:

1. Create a New Deployment or use an existing deployment on your Channel Menu Setup page. Decide which menu items to initially display on page load and which menu items to dynamically surface after a particular user action on your website.
2. In Menu Setup, uncheck the Show this menu item box under Default Display for items to appear later. Write down the menu item names.
3. At runtime, you invoke the Reordering API by calling `embedded_svc.menu.reorder` with a list of your menu item names. The items are listed in order on the same web page as the Channel Menu code snippet.

   ```
   embedded_svc.menu.reorder(["YourMenuItemName"]);
   ```

   Note: If you leave a menu item's name out of the array, the menu item will not be displayed after reordering.

4. A menu item array example (replace with your exact menu item names):

   ```
   var newOrder = ["ServiceChat", "PriorityPhoneCall", "CallCenter"];
   ```

For example, if you decide to show only a priority-channel phone number when an expensive item is added to the shopping cart. You call `embedded_svc.menu.reorder(["PriorityPhoneCall"]);` inside your `addToCart` JavaScript logic for the item price or other criteria met. Now, adding an expensive item to the cart triggers a channel menu reorder and surface a priority channel to the end user.

Access List of Original Channel Menu Items from Setup

Once the reorder is called, the menu displays the new order from the API. If you want to return the user to all original channels defined in Channel Menu Setup, access the value: `embedded_svc.menu.menuConfig.menuItems`.

If you want to return the user to the channels defined in Channel Menu Setup configured for their operating system, access the value: `embedded_svc.menu.menuConfig.configuredChannels`.

API Considerations:

- The API is fully client-side. No additional server calls are made.
- The API updates a single item action button to a menu with multiple channel items configured.

- This API does not dynamically update branding, labels, or other generic menu settings at runtime.
- If you pass in an empty array (`i.e. embedded_svc.menu.reorder([]);`), this action hides the current Channel Menu. You can also call `embedded_svc.menu.hideButtonAndMenu()` to do the same.

Tip: If you don't see an expected channel displayed, set `` `embedded_svc.menu.settings.devMode = true;' `` in the code snippet. You can see the console logs which menu items are reordered or hidden. Validation errors show up even if `devMode` is false.

Embedded Service Customized Components

Simplify the customization process for Embedded Service using HTML or modern JavaScript with Lightning Web Components. Aura Components are still available but less flexible or try multiple components on your web page with Lightning Out.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page.

Note: Embedded Service web components aren't supported on the login page of Experience sites.

Lightning Web Components for Embedded Service
Build custom chat components using Lightning Web Components in Embedded Service, which is supported in Lightning Out, Experience Cloud, and Essentials. Leverage HTML and JavaScript for a modern experience.

Custom Aura Components for Embedded Service
Use Aura components to adapt the user interface for your embedded components. Aura components are the original components and don't offer the same flexibility and ease of use compared to Lightning Web Components.

Create Multiple Components on a Web Page with Lightning Out
Lightning Out allows you to create Salesforce components outside of the Salesforce domain.

Lightning Web Components for Embedded Service

Build custom chat components using Lightning Web Components in Embedded Service, which is supported in Lightning Out, Experience Cloud, and Essentials. Leverage HTML and JavaScript for a modern experience.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: Embedded Service web components aren't supported on the login page of Experience sites.

Base Chat Header

`lightningsnapin-base-chat-header` enables customizations of the chat window header. Your custom chat header component must import `BaseChatHeader` from the `lightningsnapin/baseChatHeader` module, extend `BaseChatHeader`, and specify the `lightningSnapin__ChatHeader` target in the `js-meta.xml` configuration file.

For more detailed information, see Base Chat Header in the Components Reference Guide.

Base Chat Message

`lightningsnapin-base-chat-message`enables customizations of the user interface for chat messages. Your custom chat message component must import the `BaseChatMessage` from the `lightningsnapin/baseChatMessage` module, extend `BaseChatMesssage`, and specify the `lightningSnapin__ChatMessage` target in the `js-meta.xml` configuration file.

For more detailed information, see Base Chat Message in the Lightning Web Components Reference Guide.

Base Pre-Chat

`lightningsnapin-base-prechat` enables customization of the user interface for the pre-chat form. Your pre-chat component imports `BasePreChat` from the `lightningsnapin/basePrechat` module, extends`BasePreChat`, and specifies the `lightningSnapin__PreChat` target in the `js-meta.xml` configuration file.

For more detailed information, see Pre-Chat Message in the Lightning Web Components Reference Guide.

Minimized

`lightningsnapin-minimized` enables customization of the user interface for the chat minimized state. Your minimized component uses `assignHandler` and `minimize` functions from the `lightningsnapin/minimized` module and should specify the `lightningSnapin__Minimized` target in the `je-meta.xml` configuration file.

For more detailed information, see Minimized Chat Message in the Lightning Web Components Reference Guide.

Custom Aura Components for Embedded Service

Use Aura components to adapt the user interface for your embedded components. Aura components are the original components and don't offer the same flexibility and ease of use compared to Lightning Web Components.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Important: Use Lightning Web Components (LWC) instead of Aura components. All new Embedded Service features will only work with LWC and support for Aura components may deprecate in the future.

Note: Embedded Service web components aren't supported on the login page of Experience sites.

To create an Aura component, go to the Developer Console and click **File** > **New** > **Lightning Component**.

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

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

Customize the Minimized Embedded Service UI with Aura Components
Customize the user interface for the embedded component when it's minimized on your web page using a custom Aura component.

Custom Minimized Component Code Samples

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

Get Settings from the Embedded Service Code Snippet

Get settings for use with your Embedded Service Aura components. You can get the Chat button ID or deployment ID assigned to your Embedded Service deployment and the agent and chatbot avatar image URLs.

Customize the Pre-Chat Page UI with Aura Components

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

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Before you start, make sure that you have an Embedded Service 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 Embedded Service 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 Embedded Service 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 Embedded Service Setup.

After you save your component, go to Embedded Service 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 Aura Components Developer Guide

Custom Pre-Chat Component Code Samples

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

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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 pre-chat
 component for Embedded Chat. Implemented using Aura.">
    <!-- You must implement "lightningsnapin:prechatUI" for this component to appear in
the "Pre-chat Component" customization dropdown in the Embedded Service setup -->

    <!-- Pre-chat field components to render -->
    <aura:attribute name="prechatFieldComponents" type="List" description="An array of
objects representing the pre-chat fields specified in pre-chat 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="Pre-chat form submit button."/>

    <!-- Contains methods for getting pre-chat 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 pre-chat 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
pre-chat 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 pre-chat fields defined in setup using the prechatAPI component
  var prechatFields = cmp.find("prechatAPI").getPrechatFields();
        // Get pre-chat field types and attributes to be rendered
       var prechatFieldComponentsArray = hlp.getPrechatFieldAttributesArray(prechatFields);


        // Make asynchronous Aura call to create pre-chat 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 pre-chat
     *
     * @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

```
({
 /**
  * Map of pre-chat field label to pre-chat field name (can be found in Setup)
  */
 fieldLabelToName: {
        "First Name": "FirstName",
        "Last Name": "LastName",
        "Email": "Email",
        "Phone": "Phone",
        "Fax": "Fax",
        "Mobile": "MobilePhone",
        "Home Phone": "HomePhone",
        "Other Phone": "OtherPhone",
        "Asst. Phone": "AssistantPhone",
        "Title": "Title",
        "Lead Source": "LeadSource",
        "Assistant": "AssistantName",
        "Department": "Department",
        "Subject": "Subject",
        "Case Reason": "Reason",
        "Type": "Type",
        "Web Company": "SuppliedCompany",
        "Web Phone": "SuppliedPhone",
        "Priority": "Priority",
        "Web Name": "SuppliedName",
        "Web Email": "SuppliedEmail",
        "Company": "Company",
        "Industry": "Industry",
        "Rating": "Rating"
    },

 /**
  * 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 fields;

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

        // If the pre-chat 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 pre-chat fields
  *
  * @param fields - Array of pre-chat field Objects.
  * @returns An array of field objects.
  */
 createFieldsArray: function(fields) {
  if(fields.length) {
   return fields.map(function(fieldCmp) {
    return {
     label: fieldCmp.get("v.label"),
     value: fieldCmp.get("v.value"),
     name: this.fieldLabelToName[fieldCmp.get("v.label")]
    };
   }.bind(this));
  } else {
   return [];
  }
 },

    /**
     * Create an array in the format $A.createComponents expects
     *
     * Example:
     * [["componentType", {attributeName: "attributeValue", ...}]]
     *
  * @param prechatFields - Array of pre-chat 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 pre-chat 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.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

This component:

- Uses Javascript to create an email input field
- Uses minimal Aura to get pre-chat fields from Embedded Service 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 must 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 pre-chat 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">
   <!-- 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

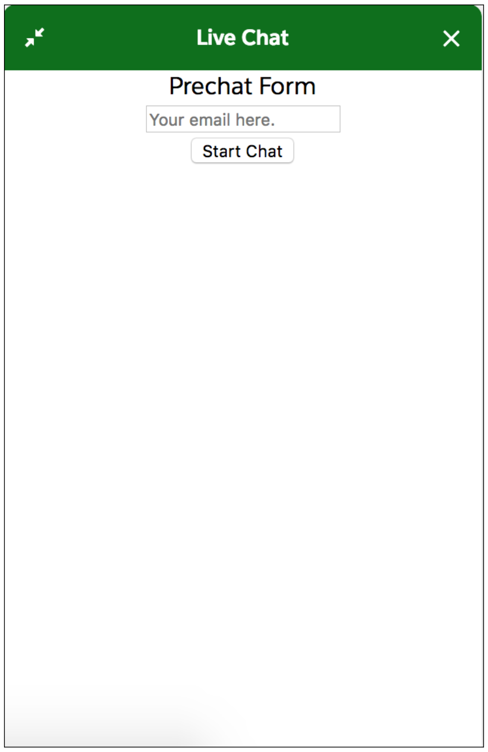
```
({
 /**
```

```
   * 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 pre-chat 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 pre-chat 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 Embedded Service UI with Aura Components

Customize the user interface for the embedded component when it's minimized on your web page using a custom Aura component.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Before you start, make sure that you have an Embedded Service 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 Embedded Service 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 Embedded Service Setup.

   Make sure to add a maximize container action so your customers can open the embedded component. 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 Embedded Service Setup.

   After you save your component, go to Embedded Service Setup and navigate to your chat settings. Your component should be available to select as a custom component for the minimized embedded component.

Events for the Minimized Chat Window

Use the following events in `eventHandlerFunction` in your minimized Embedded Service Aura component.

SEE ALSO:

Lightning Aura Components Developer Guide

Events for the Minimized Chat Window

Use the following events in `eventHandlerFunction` in your minimized Embedded Service Aura component.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

`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
`chatConferenceState`	Fired in the following scenarios: • An agent joins a one-on-one chat, which becomes a chat conference. • An agent joins a chat conference. • An agent leaves a chat conference. • There are two agents in a chat conference. One agent leaves the conference, which becomes a one-on-one chat.
`chatEndedState`	Fired when the chat has ended for any reason.
`chatState`	Fired in the following scenarios: • The visitor's chat request has been accepted and they're chatting with an agent. • After starting a chat, the visitor navigates to another page and resumes chatting. • A chat transfer has completed (see `chatTransferringState`), and the visitor is now chatting with the new agent. • 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.
`prechatState`	Fired when the pre-chat form has loaded.
`postchatState`	Fired when the post chat form has loaded.
`queueUpdate`	Fired in the following scenarios: • The visitor has requested a chat and is waiting for an agent. • After requesting a chat, the visitor navigates to another page, but is still waiting for an agent. • The visitor had previously lost connection (see `reconnectingState`) and has regained it.

eventName	Scenario
	• The visitor has advanced in the queue, but is still waiting for an agent to accept the chat request. • A bot or agent has requested a chat transfer and is waiting for another agent to accept. • During a chat transfer, the visitor navigates to another page, but is still waiting to be transferred. • The visitor has advanced in the queue, but is still waiting for an agent to accept the chat transfer. Note: This event fires only if queue position is enabled for your Embedded Service deployment and your Embedded Service 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: • The visitor has requested a chat and is waiting for an agent. • After requesting a chat, the visitor navigates to another page but is still waiting for an agent. • The visitor had previously lost connection (see `reconnectingState`) and has regained it. Note: This event fires only if either queue position is disabled for your Embedded Service deployment or your Embedded Service 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
`chatConferenceState`	`agentsInChat`	array of strings	A list of the agents in the chat conference.
	`agentName`	string	The name of the agent who is joining or leaving the chat conference.
	`isAgentJoining`	boolean	A boolean indicating whether an agent is joining (`true`) or leaving (`false`) the chat conference.

eventName	Property	Type	Description
`chatEndedState`	`reason`	string	A description of why the chat ended. Possible values include: • `agentEndedChat`– The agent ended the chat. • `visitorConnectionError`– The visitor lost connection for too long. • `visitorEndedChat`– The visitor ended the chat. • `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 embedded component using Aura.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

This component:

- Uses Aura to create an ui:button component, and binds the button click to maximize the embedded component
- Uses an initialize function that registers the generic event handler for minimized events that updates 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"
  onclick="{!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 pre-chat 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 or
refresh, OR
  * 2) the visitor was previously in reconnecting, and they've regained connection.
  *
  * Only one of "waitingState" and "queueUpdate" is ever fired - never both.
  * - "waitingState" is fired if EITHER queue position is DISABLED, OR snippet version
under 5.0.
  * - "queueUpdate" is fired if queue position is ENABLED, AND snippet version is 5.0 or
later.
  */

 /**
  * "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 or
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 Embedded Service Code Snippet

Get settings for use with your Embedded Service Aura components. You can get the Chat button ID or deployment ID assigned to your Embedded Service deployment and the agent and chatbot avatar image URLs.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

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 Chat button is used with the chat window.

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 an Embedded Service deployment already set up. You should also have a working Embedded Service Aura component before you make any changes based on the Embedded Service deployment settings.

To get started, go to the Developer Console and open one of your Embedded Service Aura 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 Chat 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 Chat button is used by the current chat window 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 Chat button for which you want to customize your pre-chat fields

        var ANONYMOUS_BUTTON_ID = "(your button id here)";

        // Fetch the ID of the Chat 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 Aura Components Developer Guide

Create Multiple Components on a Web Page with Lightning Out

Lightning Out allows you to create Salesforce components outside of the Salesforce domain.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Note: Embedded Service web components aren't supported on the login page of Experience sites.

To create these components with Lightning Out on the same web page as an Embedded Service component, use `"embeddedService:sidebarApp"` as your application tag in your call to `$Lightning.use`.

In this code sample, we create a `` `HelloWorldComp` `` component in the body of the HTML document where an Embedded Service component is rendered.

Component Code

```
$Lightning.use(
    "embeddedService:sidebarApp",
    function () {
        $Lightning.createComponent(
            "c:HelloWorldComp",
            { … },
            document.body,
            function (cmp) {
                console.log("callback");
            }
        );
    },
    “communityEndpointUrl”,
    ...
);
```

Embedded Service Code Snippet Versions

See what features are available in previous and current versions of the Embedded Service code snippet.

Important: The legacy chat product is in maintenance-only mode, and we won't continue to build new features. You can continue to use it, but we no longer recommend that you implement new chat channels. Instead, you can modernize your customer communication with Messaging for In-App and Web. Messaging offers many of the chat features that you love plus asynchronous conversations that can be picked back up at any time.

Feature Availability by Snippet Version

Code Snippet Version	Available Features
2.2	• Chat persistence across subdomains • Localization of labels
3.1	The features available in version 2.2, plus: • Passing data into post-chat • Embedded Appointment Booking (Pilot)
4.1	The features available in version 3.1, plus: • Automated chat invitations

Code Snippet Version	Available Features
	• Auto-fill fields in pre-chat with Experience or Lightning Out • Direct-to-button routing
5.0	The features available in version 4.1, plus: • Offline support • Custom chat events • Routing order • Display the customer's place in line • Group chat conferencing • Embedded Flows • Appointment Management for Lightning Scheduler • Appointment Management for Field Service