Live Assist

Web Developer Guide

Version 1.62

CAFÉ

Updated: 2018-06-18

Document version: 1.62/1

Contact Information

For technical support or other queries, contact CaféX Communications Support at:

support@cafex.com

For our worldwide corporate office address, see:

http://www.cafex.com

Contents

Introduction

This guide describes the **Fusion Live Assist** solution from a developer integration and impact point of view. We assume that the reader is familiar with JavaScript, HTML, and CSS.

Fusion Live Assist provides voice and video calling from a consumer to an agent, along with co-browsing, and document push by the agent, and remote control and annotation of the consumer's screen by the agent. See the ***Live Assist Overview and Installation Guide*** for details of what that means in practice.

For ease of integration and development, **Fusion Live Assist** uses the **Fusion Client SDK** for voice and video support, while exposing a simple API for co-browsing. When developing using the **Live Assist SDK**, you use the **Fusion Client SDK** to set up the call, and the **Live Assist SDK** for co-browsing. You therefore need at least a basic understanding of the **Fusion Client SDK** in order to develop using **Live Assist** (see the ***FCSDK Developer Guide***).

This Developer Guide gives information on integrating the **Fusion Live Assist SDK** into a Web application, and how to use it to provide the co-browsing functions to a consumer.

References:

[1] ***FCSDK Developer Guide***, obtained from CaféX product documentation

[2] ***FCSDK Administration Guide***, obtained from CaféX product documentation

Integration with an Existing Application

The steps needed to integrate **Live Assist** with a Web application is described in the following sections.

Note: On Windows, web-based applications are supported on desktop only, not tablet.

Packaging JavaScript

The **Live Assist JavaScript SDK** is available as part of the **Live Assist** server component, so the web page can include the necessary JavaScript library, which loads the SDK directly from the server. This is the recommended method.

We recommend that you do not include the contents of the `cafex_live_assist_web_consumer_SDK-n.n.n.zip` package in your web application, and use it as the source of the SDK; important updates to the SDK, available on server upgrade, would not be available to the client application without recompilation with the new packages.

Making Pages Supportable

Every page that is to allow support to start or continue must include the `assist.js` file from the **Live Assist SDK**, and have the `<DOCTYPE html>` declaration. Add the following lines should to the HTML page, where `<fas address>` is the host name or IP address of the **Live Assist** server:

```
<DOCTYPE html>
 ...
<script src='<fas address>/assistserver/sdk/web/consumer/assist.js'/>
 ...
```

We suggest that you add these lines to the template for the site, if there is one.

Note: When developing with **Live Assist**, remember that the SDK also requires cookies and JavaScript to be enabled in the browser.

Supporting Iframes

By default, **Live Assist** ignores iframes within the supported page, because it is not possible to include iframes as part of the support session without an additional implementation step.

If you want to include iframe support, add the `assist-iframe.js` script to the body of the iframe's source (that is, the webpage targeted by the iframe must include the `assist iframe.js` script), and initialize `AssistIFrameSDK` with an object containing an `allowedOrigins` element:

```
<script src='<fas address>/assistserver/sdk/web/consumer/js/assist-
iframe.js'/>
 ...
AssistIFrameSDK.init({allowedOrigins: '*'});
 ...
```

The `allowedOrigins` element should be an array of origin domains, including scheme and port, in the form `scheme:host:port` (for example `http://127.0.0.1:8080`), which is typically set to match the origin of the page that includes the iframe. This facilitates safe communication between the iframe and its parent. The special value `"*"` (as above) specifies that the iframe will communicate with a parent from any origin address.

Live Assist supports both local-origin and cross-origin iframes, allowing agents to see the content of iframes; however remote agent interaction with iframes is currently not supported.

allowedIframeOrigins

Including the `allowedOrigins` member in the configuration object passed in to `AssistIFrameSDK.init` enables the programmer to protect the iframe from rogue pages which may attempt to embed the iframe (see the Supporting Iframes section on the previous page). The similar `allowedIframeOrigins` member is a list of pages which embed the iframe (acting as the iframe's parents), passed in to the configuration object when the application calls `startSupport` (see the Session Configuration section on the next page). Set it either to `false` (to disable iframe support in **Live Assist**), or to an array containing either all the URLs which embed the iframe (`['http://192.168.0.1:8080', 'http://www.server.net']`), or the wildcard (`['*']`). The default value (if `allowedIframeOrigins` is not specified) is the wildcard, which allows the iframe to be embedded in any page.

Note:

- The use of the wildcard as the default is a temporary measure to preserve backward compatibility. In a future release it will be removed, so that in order for iframes to be co-browse enabled, the correct origins will need to be supplied both inside the iframe and on the parent page containing the iframe, using the two SDKs (`AssistSDK` and `AssistIframeSDK`).

- When explicitly setting `allowedIframeOrigins` to the wildcard, remember to include it as the only element of an array.

Starting a Support Session

The application starts a support session, normally in response to the user clicking on a **Help** or **Request Support** button, using the `AssistSDK.startSupport` function, passing in a configuration object. To start a simple support session with default values, the application only needs to specify the `destination`:

```
<a title='Live Assist' onclick='AssistSDK.startSupport({destination :
"agent1"})'>Support</a>
```

The above code provides a link which a user can click on for support; when a consumer clicks the link, **Live Assist** starts a call and co-browse session with the support agent named `agent1`.

Typically, customer support services provide a queue, which is serviced by a number of support agents. The `destination` parameter can also specify a queue instead of an individual agent:

```
var config;
config.destination = 'customer-support';
config.videoMode = 'agentOnly';
 ...
AssistSDK.startSupport(config);
```

The configuration object is a JavaScript object with a number of properties which control aspects of the session (see the Session Configuration section below).

Session Configuration

The configuration object passed in to `startSupport` can contain the following properties:

Property	Default Value or Behavior	Description
`destination`		User name of agent or agent group, if that agent or agent group is local to the Web Gateway; otherwise, the full SIP URI of an agent or queue.

Property	Default Value or Behavior	Description
`videoMode`	`full`	Sets whether to show video, and from which parties. Allowed values are: ▪ `full` ▪ `agentOnly` ▪ `none`
`correlationId`	Generated	ID of the co-browsing session.
`auditName`	empty string	Name to identify the consumer in event logs (see the ***Live Assist Overview and Installation Guide*** for more details on event logging).
`url`	Calculated from `src` attribute of `script` tag	Base URL of **Live Assist** server and **FCSDK** Web Gateway, including only scheme, host name or IP address, and port number. Include this if the `assist.js`*JavaScript* file included in the HTML page with the `<script>` tag is on a different host to the **Live Assist** server. URIs of shared documents (see **the Sharing Documents section on page 27**) are also resolved against this URL.
`sdkPath`	Calculated from `src` attribute of `script` tag	URL of the base directory of the consumer SDK. As with the `url` property, include this if the **Live Assist SDK** is not on the same server as the `assist.js` file.

Property	Default Value or Behavior	Description
`popupCssUrl`		URL of CSS stylesheet containing styles for the **Live Assist** popup window. This allows you to customize the **Live Assist** user interface (see the **Customizing the Live Assist popup Window section on page 32**).
`popupInitialPosition`		Object containing values for the initial position of the popup window on the screen (see the **Customizing the Live Assist popup Window section on page 32**).
`sessionToken`		Web Gateway session token (if required).
`uui`		The value set is placed in the SIP `User to User` header in hex-encoded form. **Note:** The UUI can only be used when **Anonymous Consumer Access** is set to `trusted` mode. See the ***Live Assist Architecture Guide*** for further information. The UUI is ignored if the session token is provided.
`allowedIframeOrigins`	*	List of pages which will host iframes. See the **Supporting Iframes section on page 7** for details.

Property	Default Value or Behavior	Description
`retryIntervals`	`[1.0,2.0, 4.0,8.0, 16.0,32.0]`	Indicates the number of automatic reconnection attempts, and the time in seconds between each attempt. To disable automatic reconnection, specify an empty array. See the Connection Configuration section on page 34.
`connectionStatusCallbacks`		A set of callback functions which allow the application to control or monitor the status of the current connection. See the Connection Callbacks section on page 34.
`mutationBlacklist`		An object containing lists of element IDs, classes, and animations, changes to which will *not* cause the agent's view of the consumer's screen to update. See the Controlling Updates to the Agent's View section on page 52.

Note: If the configuration object does not include a `sessionToken` property, the **Live Assist SDK** automatically creates a session with the **Fusion Client SDK** server; that session is used for co-browsing and the FCSDK voice and video call (if any); we expect this to be the normal case.

If the `sessionToken` property *is* provided (for instance, if a session token is provided separately using a bespoke security mechanism (see the Consumer Session Creation section on page 54), or the FCSDK initiated a call which is now being escalated to co-browse (see the Escalating a Call to Co-browsing section on the next page)), then the configuration object passed to `startSupport` is used as provided, and the session identified by the session token is used for co-browsing. You must specify any non-default values for the other properties.

Note: If `startSupport` is called programmatically, it will trigger the popup blocker that is built into most browsers; however, if it is called as a direct consequence of a user interaction (such as pressing a button in the UI), it is not.

Escalating a Call to Co-browsing

In most cases, the application calls `startSupport` with an agent name, and allows **Live Assist** to set up a call to the agent and implicitly add **Live Assist** support to that call. However, there may be cases where a call to an agent already exists, and the application needs to add **Live Assist** support capabilities. To do this, you need to supply the **session token** and a **correlation ID** in the configuration object which you supply to `startSupport`; and the agent needs to connect to the same session. The **Live Assist** server provides some support for doing this.

1. The application connects to a specific URL on the **Live Assist** server, to request a **short code** (error handling omitted):

```
var request = new XMLHttpPRequest();
request.onreadystatechange = function() {
   if (request.readyState == 4) {
      if (request.status == 200) {
         var shortcode = JSON.parse(request.responseText).shortCode;
         start(shortcode);
      }
   }
}
request.open('PUT', '<fas address>/assistserver/shortcode/create',
true);
request.send();
```

2. The application uses the short code in another call to a URL on the **Live Assist** server, and receives a JSON object containing a session token and a correlation ID:

```
var start = function(shortcode) {
   var request = new XMLHttpRequest();
   request.onreadystatechange = function() {
      if (request.readyState == 4) {
         if (request.status == 200) {
            var response = JSON.parse(request.responseText);
             ...
         }
      }
```

```
        }
    }
    request.open('GET', '<fas
    address>/assistserver/shortcode/consumer?appkey=' + shortcode, true);
    request.send();
```

3. The application includes those values in the configuration object and passes it to `startSupport`:

   ```
   var configuration;
   configuration.sessionToken = response['session-token'];
   configuration.correlationId = response.cid;
    ...
   AssistSDK.startSupport(configuration);
   ```

   More configuration can be set in the configuration object.

4. The agent uses the same short code to get a JSON object containing the session token and correlation ID, which it can then use to connect to the same **Live Assist** support session (see the ***Live Assist Agent Console Developer Guide***). Informing the agent of the short code is a matter for the application. It could be something as simple as having it displayed on the consumer's screen and having the consumer read it to the agent on the existing call (this is how the sample application does it).

The sample application supplied with the SDK includes a JavaScript file called `short-code-assist.js`, which contains a function called `ShortCodeAssist.startSupport`, which contains the necessary code and takes a callback function and a configuration object:

```
ShortCodeAssist.startSupport(function() {
    ...
},
configuration);
```

The SDK calls the callback function when the support session starts successfully. You can take this code and adjust it as you need for your own purposes.

Note:

- When escalating an existing call, the `destination` property should *not* be set on the configuration object; in this case, the destination is known implicitly from the existing call.
- The short code expires after 5 minutes, or when it has been used by both agent and consumer to connect to the same session.
- If you wish to define an audit name to identify the consumer in event logs (see the ***Live Assist Overview and Installation Guide*** for more details on event logging), include an `auditName` parameter

in the URL which creates the short code:

```
/assistserver/shortcode/create?auditName=consumer
```

Ending a Support Session

When voice and video is enabled, the user can end the session using the default UI that **Live Assist** adds; the application can also end the session programmatically using the `AssistSDK.endSupport` function. In co-browse-only mode, **Live Assist** does not add a default UI, so the application must call `AssistSDK.endSupport` to end the support session.

During a Co-browsing Session

While a co-browsing session is active (after the application has called `startSupport` successfully, and before either it calls `endSupport` or receives the `onEndSupport` notification to indicate that the agent has ended the support session), the application may:

- Accept an agent into, or expel the agent from, the co-browsing session
- Pause and resume the co-browsing session
- Receive a document from the agent
- Push a document to the agent
- Receive an **annotation** (a piece of text or drawing to show on the device's screen, overlaid on the application's view) from the agent
- Have a form on its screen wholly or partly filled in by the agent

Actions which are initiated by the application (such as pushing a document to the agent) require it to call one of the methods on the `AssistSDK` object.

Actions initiated by the agent (such as annotating the consumer's screen) can in general be allowed to proceed without interference from the application, as the **Live Assist SDK** manages them, overlaying the user's screen with its own user interface where necessary.

However, the application can receive notifications of these events by defining one or more of the callback functions on the `AssistSDK` object:

```
window.AssistSDK = {
   onEndSupport: function() {
       ...
   };
}
```

Callbacks

When it calls `startSupport`, the consumer application can provide callback functions to the consumer web SDK. The SDK calls these functions to notify the application of events; the application can respond to the events, and in some cases can tell the SDK what to do next.

onConnectionEstablished

A consumer application can implement the `onConnectionEstablished` callback to receive notification when an agent first joins a **Live Assist** session. Once this has happened, the agent may request permission to co-browse.

```
AssistSDK.onConnectionEstablished = function() {
   console.log("Connection Established");
};
```

Note: By default, **Live Assist** presents the request for permission to the user; however, the application can override this behavior; see the onScreenshareRequest section below.

onWebcamUseAccepted

When **Live Assist** establishes a voice and video call, it prompts the consumer to allow the application to use their webcam; the application receives this callback after the user has given permission. You might use it to update the user interface to remove a warning message, or to update a progress indicator:

```
AssistSDK.onWebcamUseAccepted = function() {
   // Hide the warning
   hideWebcamWarning();
};
```

onScreenshareRequest

The `onScreenshareRequest` callback notifies the application when an agent asks to co-browse the consumer's screen. It gives the application an opportunity (by returning `true`) to allow the screenshare without asking the consumer (for example, there could be a flag in the application's configuration which gives permanent permission for screen sharing):

```
AssistSDK.onScreenshareRequest = function() {
   if (screenshareAllowed) {
      return true;
   }
    ...
   return false;
};
```

By default, **Live Assist** displays a dialog box when an agent requests co-browsing, allowing the consumer to accept or reject the co-browse.

onInSupport

The application can receive an `onInSupport` callback when it accepts a screenshare, and the agent has joined the co-browse. It gives the application an opportunity to change its own UI to reflect the fact that a co-browsing session is active, or to log events.

```
AssistSDK.onInSupport = function() {
   // Show user extra UI as they’re in a Live Assist session
   showCobrowseUI();
};
```

onPushRequest

When the agent pushes a document to the consumer, by default **Live Assist** displays a dialog box, allowing the user to accept or reject the document; if they accept it, it shows the document to the consumer. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

The application developer can override this behavior using the `onPushRequest` callback. The SDK calls this function when the agent pushes a document to the consumer, before it displays it (note that the application does *not* receive this callback when the agent pushes a URI). The callback function receives two functions: an `allow` function and a `deny` function. The callback function should call the `allow` function to show the pushed document to the consumer, or the `deny` function to reject the pushed document:

```
AssistSDK.onPushRequest = function(allow, deny) {
   var result = confirm("The agent wants to send you a document or image. 
   Would you like to view it?");
   if (result)
      allow();
   else
      deny();
}
```

The above function's behavior is very similar to the default behavior. It shows a confirmation prompt on the screen, and lets the user click **OK** or **Cancel**, depending on whether they want to view the document. To always show documents without prompting:

```
AssistSDK.onPushRequest = function(allow, deny) {
   allow();
}
```

Document Callbacks

By default, after it receives a document, **Live Assist** opens a window to display the shared document; if there is an error loading or parsing the shared document file, it displays an error window. It does this without any interaction from the application.

If it successfully load, parses, and displays the shared document, the SDK calls the `onDocumentReceivedSuccess` callback function; the function receives a `sharedDocument` object (described below). If an error occurs while trying to load or parse the shared document, it calls the `onDocumentReceivedError` callback function, which also receives a `sharedDocument` object. The two callback functions are optional - the SDK does nothing if you do not supply them.

The `sharedDocument` object that the SDK passes to the `onDocumentReceivedSuccess` and `onDocumentReceivedError` functions contains an `id` property, which is a unique identifier for the document; and may contain a `metadata` property, which contains additional information about the document supplied by the agent. It also has a `close` method, which the application can call to close the shared document window or error window. Additionally, the application may add an `onClosed` handler to the `sharedDocument` object, to receive notification when the window closes due to a user (consumer or agent) closing it from the UI.

The following code creates `onDocumentReceivedSuccess` and `onDocumentReceivedError` callbacks, adds an `onClosed` handler to the `sharedDocument` object, and sets a timer to call `sharedDocument.close`:

```
AssistSDK.onDocumentReceivedSuccess = function(sharedDocument) {
   console.log("*** shared item opened successfully: " + sharedDocument.id);
   sharedDocument.onClosed = function(actor) {
      alert("Shared document window has closed by " + actor + ".");
   };
   console.log("Setting shared item " + sharedDocument.id + " to close in 15 
secs.");
   setTimeout(function() {
      console.log("*** Closing shared item " + sharedDocument.id);
      sharedDocument.close();
   }, 15 * 1000);
};

AssistSDK.onDocumentReceivedError = function(sharedDocument) {
   console.log("*** shared item opened with error: " + sharedDocument.id);
   sharedDocument.onClosed = function(actor) {
      alert("Shared document error window has been closed by " + actor +
      ".");
   };
   setTimeout(function() { sharedDocument.close();}, 5 * 1000);
};
```

Note: The application does not receive these callbacks if the agent pushes a URI.

Annotation Callbacks

There are two callbacks whichnotify the application when an agent draws on a shared screen:

- `onAnnotationAdded(annotation, sourceName)`

  Called when an annotation is received from an agent. The `annotation` object contains the following properties:

Property	Description
`stroke`	The color of the annotation
`strokeOpacity`	A number between 0.0 and 1.0 indicating the opacity of the annotation
`strokeWidth`	A number giving the width of the line of the annotation
`points`	An array of points representing the path of the annotation. By default, **Live Assist** draws a line with the color, opacity, and width, following these points as its path.

- `onAnnotationsCleared()`

  Called when an agent clears the annotations.ons.

You can implement these callbacks to control the display and clearing of annotations, or simply to record what the agent has sent:

```
   AssistSDK.onAnnotationAdded = function(annotation, sourceName) {
      console.log("Annotation added by " + sourceName);
   };

   AssistSDK.onAnnotationsCleared = function() {
      console.log("Annotations cleared");
   }
```

See the Annotations section on page 30 for more details.

Zoom Callbacks

The application can receive notifications when the zoom window opens or closes (see the Zoom section on page 29):

```
AssistSDK.onZoomStarted = function() {
    ...
  pushDocumentButton.disabled = true;
});

AssistSDK.onZoomEnded = function() {
    ...
  pushDocumentButton.disabled = false;
});
```

You might want to use these callbacks to update the user interface to prevent user interaction which will not work.

The application will receive these callbacks whether the consumer or agent application opens or closes the zoom window.

Co-browsing Callbacks

As well as using the CSS class mechanism to customize its user interface (see the Customizing the Live Assist popup Window section on page 32), there are two callback functions which the consumer web application can implement to define what happens when co browsing starts and ends:

```
AssistSDK.onCobrowseActive = function() {
  // Display indicator, log, etc.
}
AssistSDK.onCobrowseInactive = function() {
  // Remove indicator, log, etc.
}
```

If an application does not provide these callbacks, the SDK provides a default implementation, displaying a banner at the top of the browser window, stating ***This page is currently being shared***.

Agent Callbacks

The application can implement the following callbacks to receive notification when agents join and leave the co-browsing session::

- `onAgentJoinedSession(agent)`

  This callback indicates that an agent has answered the support call and joined the support session; this occurs before the agent either requests or initiates co-browsing. The callback allows the developer to pre-approve the agent into the co-browse, before the agent makes the request.

- `onAgentRequestedCobrowse(agent)`

  This callback notifies the developer that the agent has specifically requested to co-browse. There is no specific requirement for the application to allow or disallow co-browsing at this point, but it is an obvious point to do so.

- `onAgentJoinedCobrowse(agent)`

  This callback indicates when the Agent joins the co-browse session.

- `onAgentLeftCobrowse(agent)`

  This callback occurs when the agent leaves the co-browse session, and can no longer see the consumer’s screen. Leaving the co-browse also resets the agent's co-browse permission; the agent may subsequently request co-browse access again.

- `onAgentLeftSession(agent)`

  This callback notifies the application that the agent has left the overall support session.

The `agent` parameter to all these callbacks is a *JavaScript* object which can be passed in to the `AssistSDK.allowCobrowseForAgent` or `AssistSDK.disallowCobrowseForAgent` functions. See the Allow and Disallow Co-browse for an Agent section on page 24.

These callbacks allow the developer to maintain a list of agents that are in the co-browse and dynamically allow them in and out of the co-browsing session at any time. To do this the developer can hold on to the agent references that they receive during the `onAgentJoinedSession` callback, which will remain valid, and can then admit and eject agents during the co-browsing session on whatever basis the application determines.

The default implementation displays a dialog box on the consumer's device, asking whether to allow co-browsing or not. If the consumer allows co-browsing, it allows any agent into the co-browsing session whenever they request it. Implementing this interface can give the application more control over which agents are allowed into the co-browsing session, and when.

onEndSupport

When a **Live Assist** session terminates (for example when the call ends, or the consumer application calls `AssistSDK.endSupport`), the application can receive notification in the `onEndSupport` function.

```
AssistSDK.onEndSupport = function() {
    ...
};
```

This callback provides a place for the application to reset its user interface to indicate that it is no longer in a support session. **Live Assist** removes its own UI automatically, so the application only needs to restore any changes it has made itself.

onError

The application can handle error events that cause the failure of a **Live Assist** session using the `onError` callback:

```
AssistSDK.onError = function(error) {
    ...
}
```

The `error` object received by the callback is a *JavaScript* object with properties `code` and `message`. The `message` property is a free-form text message. The following error codes may be received:

Error Code	Value	Received by:		Meaning
		Agent	Consumer	
CONNECTION_ LOST	0	Yes	Yes	Failed to connect after retry count. No retry inter- vals specified, will not attempt to reconnect.
PERMISSION	1	Yes	Yes	Received a permission change message on a topic with no permissions. Error trying to leave a topic. The message will include topic ID and error message.
SOCKET	2	Yes	Yes	Low level socket error. The message will include the socket error code.
CALL_FAIL	3	Yes	Yes	Tried to share a document when co-browsing is not active.

Error Code	Value	Received by:		Meaning
		Agent	Consumer	
				Tried to allow or disallow co-browsing for an agent when support is not active.
POPUP	4	No	Yes	Couldn't reconnect to popup.
SESSION_IN_PROGRESS	5	Yes	Yes	There is already a session in use.
SESSION_CREATION_FAILURE	6	No	Yes	Error connecting to server. The message will include the server URL.

Note: Not all errors can be received by both parties.

Allow and Disallow Co-browse for an Agent

You may wish to remove a specific agent from the co-browsing session. To do this, call:

```
AssistSDK.disallowCobrowseForAgent(agent)
```

passing in the agent object received in the `onAgentRequestedCobrowse` callback (see the Agent Call-backs section on page 21).

If the agent is already in the co-browse session, they are removed from it; if they are not in the co-browse session, they will not be admitted until the application calls

```
AssistSDK.allowCobrowseForAgent(agent);
```

When the application calls `allowCobrowseForAgent`, the specified agent joins the co-browse immediately.

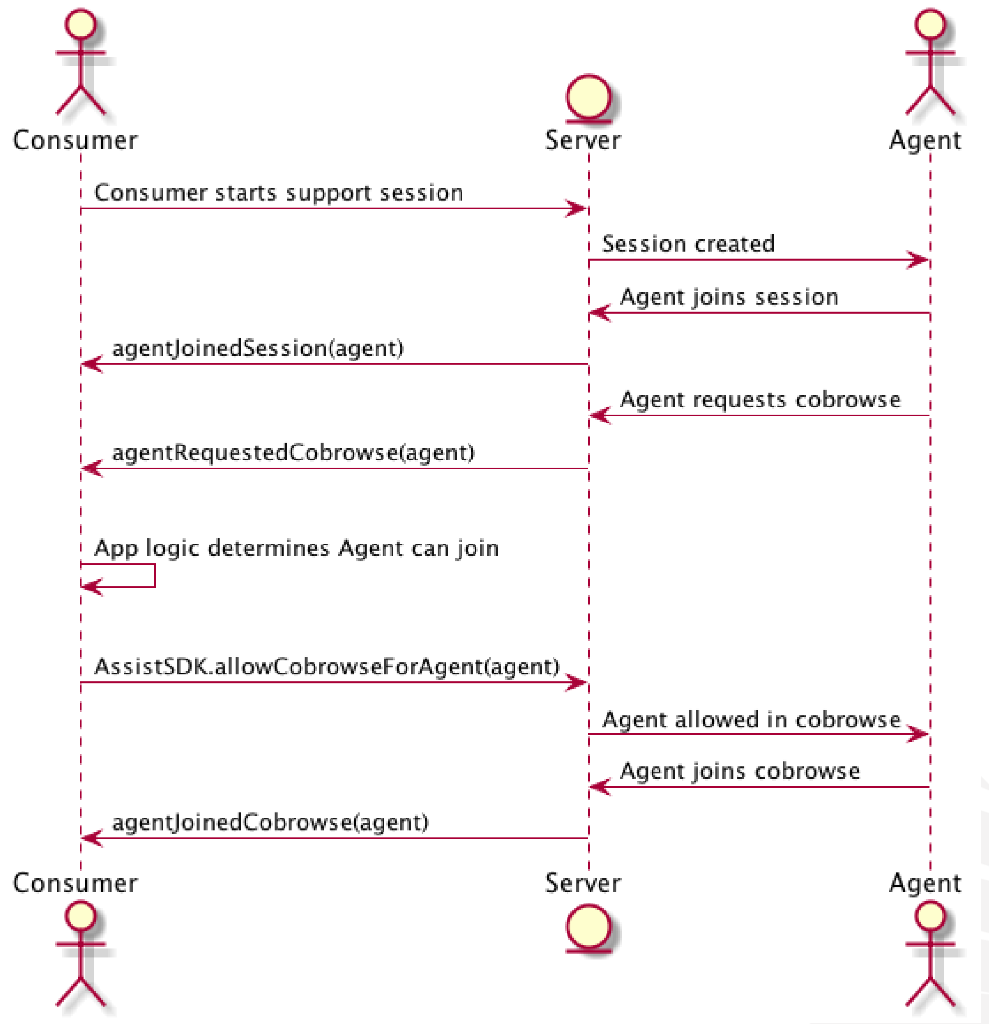
Web-specific considerations

On the web, when the consumer navigates to a new support enabled page during a support session, the co-browse, and indeed the entire support session, is torn down and recreated on the new page. This means that any agents will re-join the session on each page without any permission to access the co-browse, and permission will need to be re-granted to the appropriate agents in order for the co-browse to continue without interruption.

Agent accepted into co-browse

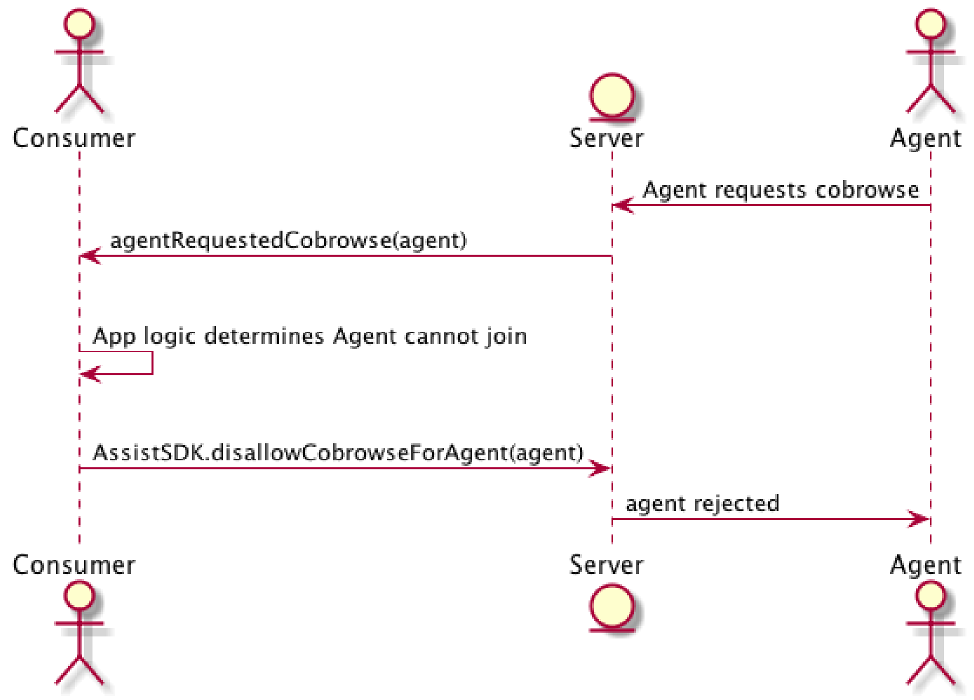
When an agent is accepted into the co-browse, the following occurs:

1. Consumer starts support session.
2. Agent joins session.
3. `agentJoinedSession` callback fired in the consumer application.
4. Agent requests co-browse.
5. `agentRequestedCobrowse` callback fired in the consumer application.
6. The consumer application has logic that decides the agent is allowed access to the co-browse. This logic could be based on permissions.
7. Agent is accepted into the co-browse.
8. Agent joins the co-browse.
9. `agentJoinedCobrowse` callback fired in the consumer application.
10. Agent can see the consumer's screen.


Consumer
Server
Agent
Consumer starts support session
Session created
Agent joins session
agentJoinedSession(agent)
Agent requests cobrowse
agentRequestedCobrowse(agent)
App logic determines Agent can join
AssistSDK.allowCobrowseForAgent(agent)
Agent allowed in cobrowse
Agent joins cobrowse
agentJoinedCobrowse(agent)
Consumer
Server
Agent

Agent rejected from co-browse

1. Agent requests co-browse.
2. `agentRequestedCobrowse` callback fired in the consumer application, with the agent's permissions.
3. Consumer application checks the agent permissions, and finds they do not have the required permissions to view the current part of the application.
4. Consumer application rejects the agent’s request to join the co-browse, and the agent is unable to see the consumer’s screen.


Consumer
Server
Agent
Agent requests cobrowse
agentRequestedCobrowse(agent)
App logic determines Agent cannot join
AssistSDK.disallowCobrowseForAgent(agent)
agent rejected
Consumer
Server
Agent

Pausing and Resuming a Co-browsing Session

The application can temporarily pause a co-browse session with the agent by calling:

```
AssistSDK.pauseCobrowse();
```

While paused, the connection to the **Live Assist** server remains open, but the co-browse session is disabled, disabling annotations, document sharing, and so on as a consequence. When the application wishes to resume the co-browsing session, it should call:

```
AssistSDK.resumeCobrowse();
```

When the application pauses a co-browse, **Live Assist** notifies the Agent Console, which can present a notification or message to the agent to indicate what has happened.

Sharing Documents

As well as receiving shared documents from the agent (see the onPushRequest section on page 18), applications can use the **Live Assist SDK** to share documents with the agent during a co-browsing session. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

Documents shared in this way appear the same as documents pushed by the agent: PDFs are full screen; images are in windows that can be dragged, re-sized, or moved.

Note: Sharing a document does not actually send the document to the agent, but simply displays the document on the local device, so that both the consumer and the agent can see and co-browse the document.

The application shares a document by calling:

```
AssistSDK.shareDocument(document, onLoad, onError);
```

Where

- `document` is a PDF document or image to be shared, expressed as one of the following:
  - A string URL pointing to the PDF document or image to share
  - A JavaScript file or Blob object containing the PDF document or image
- `onLoad` is a callback function that takes no arguments, and is called when the document is successfully loaded.
- `onError` is a callback function that is called when an error occurs loading the document, it is passed the following arguments:
  - an error code
  - an error message.

The error codes are the same as the agent-side error codes for document push, and may be one of the following:

Error Code	Value	Received By:	
		Agent	Consumer
SHARED_DOCUMENT_ERROR_CONNECTION_ERROR	1	Yes	Yes
SHARED_DOCUMENT_ERROR_HTTP_ERROR	2	Yes	Yes
SHARED_DOCUMENT_ERROR_UNSUPPORTED_MIME_TYPE	3	Yes	Yes
SHARED_DOCUMENT_ERROR_FILE_PARSING_ERROR	4	No	Yes
SHARED_DOCUMENT_ERROR_NO_DATA_RECEIVED	5	Yes	No
SHARED_DOCUMENT_ERROR_CO_BROWSE_NOT_ACTIVE	6	No	Yes

Not all values are possible in either case, for example, the agent never receives error code 6, and the consumer never receives error code 5.

The error message is a text string describing the error; it is intended for debugging and logging, rather than for displaying to an end user.

Zoom

Either agent or consumer can open a zoom window on the consumer's device. While the zoom window is open, it displays a magnified version of the content of the consumer's screen where it is positioned.

The zoom window contains controls to change the magnification and to close the window. Either party can use these controls, or move the window about the consumer's screen by dragging it (so the agent can move the zoom window to a part of the consumer's screen they want to look at, or the consumer can move it to a part of the screen they want the agent to look at).

You can change the appearance of the zoom window in CSS by adding styles for the element with ID `assist-zoom-window`; for example:

```
#assist-zoom-window {
  border: 3px solid red;
}
```

to give the zoom window a red border for greater visibility.

Opening the Zoom Window

The application can open the zoom window by calling the `AssistSDK.startZoom` function:

```
zoom: function() {
  AssistSDK.startZoom();
   ...
}
```

You would normally assign the `zoom` function to the `onclick` handler of a button.

There is an equivalent `AssistSDK.endZoom` function, but you will not normally need to call this explicitly; normally, one of the users closes the zoom window with the its close button, and if it is open when the **Live Assist** session ends, **Live Assist** closes it automatically.

Note: Document sharing and zooming are mutually exclusive. If the zoom window is open when you call `AssistSDK.shareDocument`, it has no effect (apart from logging a message to the console). Similarly, if a shared document is open when you call `AssistSDK.startZoom`, it does nothing.

Annotations

By default the **Live Assist SDK** displays any annotations which the application receives on an overlay, so that the consumer can see them together with their own screen. Normally an application needs to do nothing further, but if it needs to receive notifications when an annotation arrives or is removed, it can implement one of the annotation callbacks (see the Annotation Callbacks section on page 20).

Setting the z-index of the annotation layer

Elements in HTML pages may have a `z-index` property, which specifies the order to display them. Elements with a high `z-index` appear in front of elements with a lower `z-index`, potentially hiding the lower `z-index` elements.

Some sites may have a high `z-index` on some elements, leading to annotations appearing behind them. Using CSS, you can set the `z-index` value of the `glass-pane` so that it is high enough to overlay all the elements on the page:

```
#glass-pane {
  z-index:XXXXX !important; // Set to appropriate value
}
```

Note:

- Legitimate values for `z-index` are `auto`, `initial`, `inherit`, or a number (negative numbers are allowed), but if you need to set it, you will probably want to set it to a positive number in order to bring the annotation layer to the top. The other values seem to be less useful in this case.
- `z-index` only works on positioned elements (`position:absolute`, `position:relative`, or `position:fixed`).
- `!important` is necessary in order to override the `z-index` setting of other objects.

Form Filling

One of the main reasons for a consumer to ask for help, or for an agent to request a co-browse, is to enable the agent to help the consumer to complete a form which is displayed on their device. The agent can do this whenever a **Live Assist** co-browse session is active, without further intervention from the application, but there are some constraints on how forms should be designed.

The **Live Assist SDK** automatically detects form fields represented by `input` elements, and relays these forms to the agent so that the agent can fill in values for the user. You must provide each element with a unique label in the HTML, in one of the following ways:

- providing a `label` for the field and including the `for` attribute:
  ```
  <label for="otherloans_id">Other Loans: </label>
  <input id="otherloans_id" type="text"/>
  ```
- setting the `title` attribute of the `input` element:
  ```
  <input type="text" title="Other Loans"/>
  ```
- setting the `name` attribute of the `input` element:
  ```
  <input type="text" name="Other Loans"/>
  ```
- setting the `id` attribute of the `input` element:
  ```
  <input type="text" id="other_loans"/>
  ```
- setting the `value` attribute of the input element, *if* the `input` is of type `radio`:
  ```
  <input type="radio" name="bedrooms" value="studio"/>
  <input type="radio" name="bedrooms" value="one"/>
  <input type="radio" name="bedrooms" value="two"/>
  ```

The SDK looks for a label to present to the user in the order above; if it does not find a `<label>` element for the field, it will look for a `title` attribute; if it does not find a `title` attribute either, it will look for a `name` attribute; and so on.

The SDK automatically prevents the agent from performing form fill if the `type` is `password`.

Note: While the SDK prevents these fields from being presented to the agent as fillable form data, it does not prevent them from being visible as part of the co-browse. You can hide them by adding the appropriate class or permission to the element (see the Excluding Elements from Co-browsing section below).

Excluding Elements from Co-browsing

When an agent is co-browsing a form, you may not want the agent to see every control on the form. Some may be irrelevant, and some may be private to the consumer.

To do this, add a CSS class (`assist-no-show`) to HTML elements, which instructs the **Live Assist SDK** to mask those areas:

```
<div id="sensitive-details" class="assist-no-show">content</div>
```

By default, **Live Assist** shows excluded elements as black boxes that occupy the same space on the page as the original element; you can specify the color of the box using the `color` attribute of the special `assist-no-show-agent-console` CSS class in your stylesheet (the `color` attribute is the only attribute of the `assist-no-show-agent-console` class that has any effect). The `color` attribute only affects the rendering of the boxes on the agent console, and does not affect the display of the elements on the consumer's pages. For example, the following CSS code makes elements marked with the `assist-no-show` class display as orange boxes in the agent console:

```
.assist-no-show-agent-console {
  color: orange;
}
```

You can make them not appear at all:

```
.assist-no-show-agent-console {
  color: transparent;
}
```

For more detailed control over element visibility, see the Permissions section on page 38.

Co-browsing Visual Indicator

The SDK provides a means to customize the visual indication displayed during screen sharing. The default implementation displays a banner at the top of the window. During screen sharing, the main window of the application has the CSS class `assist-cobrowsing` (in addition to any other CSS classes it may have). You can customize the visual indication by defining this class in your style-sheet and adding properties to it.

Customizing the Live Assist popup Window

You can customize the colors, fonts, and images of the **Live Assist** popup window by creating a CSS file with styles for the body tag, and for elements with the `#title`, `#logo`, and `#status` IDs. When you call `startSupport`, include the CSS file URL as the `popupCssUrl` member of the configuration object:

```
var config = {
  destination: "agent1",
  popupCssUrl: "/assistsample/css/popup.css"
};
 ...
AssistSDK.startSupport(config);
```

To customize the background of the window, specify background attributes for the body tag:

```
body {
   background-color: #0000FF;
   background-image: url('/assistsample/img/foo.jpg');
}
```

To customize the **Live Assist** logo, specify a background image for the `#logo` ID, along with `width` and `height` attributes:

```
#logo {
   background-image: url('/assistsample/img/newlogo.png');
   width: 64px;
   height: 64px;
}
```

Customize fonts by specifying `font` attributes for the `#title` and `#status` IDs.

Popup window position

The default position of the **Live Assist** popup window may obscure an important part of the consumer's screen. The application can control the position of the window by including the `popupInitialPosition` property in the configuration object passed to `startSupport`. The value should be an object containing two properties, `top` and `left`, which control the position (in pixels) of the popup window:

```
var config = {
   popupInitialPosition = {
      top: 200,
      left: 500
   },
};
 ...
AssistSDK.startSupport(config);
```

Note: If you use negative numbers in for the `top` and `left` values, the **Live Assist** popup window appears at `0,0` on the consumer's screen.

WebSocket Reconnection Control

When a co-browse session disconnects due to technical issues, the default behavior is to attempt to reconnect six times at increasing intervals. You can control this behavior by passing in one or both of the following when the application calls `startSupport` (see the Session Configuration section on page 9):

- Connection configuration
- A set of callbacks for connection events, allowing an application to perform its own reconnection

handling, or to simply inform the user of the status of the current connection

Connection Configuration

You can use the optional `retryIntervals` property of the connection object to control reconnection behavior (see the Session Configuration section on page 9):

```
var configuration;
configuration.destination = 'agent1';
configuration.retryIntervals = [5.0,10.0,15.0];
 ...
AssistSDK.startSupport(configuration);
```

If the WebSocket connection to the **Live Assist** server goes down, **Live Assist** will try to re-establish the connection to the server the number of times specified in the array, with the specified time in seconds between them. In the above example, **Live Assist** would try to reconnect 5 seconds after the initial disconnection; then, if that fails, it would try 10 seconds after that; then, if that fails, it would try 15 seconds after that; and if that reconnection attempt fails, it will give up and not try again.

Note: If you do not specify `retryIntervals` in the connection object, **Live Assist** will use its default values, which are `[1.0, 2.0, 4.0, 8.0, 16.0, 32.0]`. If you specify an empty array, **Live Assist** will make no reconnection attempts.

Connection Callbacks

If the default reconnection behavior of **Live Assist** is not what you want, even after specifying the retry intervals, you can implement a set of connection callbacks and pass them to **Live Assist** in the `connectionStatusCallbacks` property of the configuration object:

```
var callbacks = {
  onDisconnect: function(error, connector) {},
  onConnect: function() {},
  onTerminated: function(error) {},
  willRetry: function(inSeconds, retryAttemptNumber, maxRetryAttempts,
  connector) {}
};
var config = {destination: 'agent1', connectionStatusCallbacks: callbacks};
 ...
AssistSDK.startSupport(config);
```

The `connectionStatusCallbacks` property is itself an object with the properties `onDisconnect`, `onConnect`, `onTerminated`, and `willRetry`. These must all be functions defined in the JavaScript.

Note:

- These callbacks need to be defined and added to the configuration explicitly as above. It is not enough to define them on the appropriate object, as it is with other callbacks.
- If you do not specify `retryIntervals` in the configuration object, **Live Assist** will use its default reconnection behavior; if you specify `retryIntervals` , **Live Assist** will use its default reconnection behavior using those values. You can turn off the default reconnection behavior, and take full control of reconnection, by specifying an empty list for `retryIntervals`.

When implementing your own reconnection logic, the most important notifications you receive are `onDisconnect` (called whenever the connection is lost) and `willRetry` (called when automatic reconnection is occurring, and there are more reconnection attempts to come). Both these methods include a `connector` object in their arguments; use the `connector` object to make a reconnection attempt, or to terminate all reconnection attempts.

Callback	Description
`onDisconnect`	Called for the initial WebSocket failure, and for every failed reconnection attempt (including the last one). This method is called regardless of whether `retryIntervals` is specified (that is, whether automatic reconnection is active or not). The `connector` object allows the implementing class to control reconnection, even if reconnection is automatic. For example, an application might decide to give up reconnection attempts even if more reconnection attempts would normally occur; or to try the next reconnection attempt immediately without waiting until the next retry interval has passed.
`onConnect`	Called when a reconnection attempt succeeds. This may be useful to clear an error indication in the application UI, or for canceling reconnection attempts if the application is managing its own reconnections.
`willRetry`	Called under the following conditions: ▪ when the WebSocket connection is lost; or ▪ when a reconnection attempt fails and automatic reconnections are occurring

Callback	Description
	(`retryIntervals` is a non-empty array) *and* there are more automatic reconnection attempts to be made. This method is called after the `onDisconnect` method. Use the `connector` object to override reconnection behavior. For example, to make a reconnect attempt immediately.
`onTerminated`	Called under the following conditions: ▪ when all reconnection attempts have been made and failed, or ▪ when either the `Connector.disconnect` or the `AssistSDK.endSupport` function is called.

Example - make a reconnection attempt immediately on disconnection:

In this example, the default reconnection behavior has been disabled, and the application reconnection behavior is dependent on the reason for disconnection.

```
var onDisconnect = function(error, connector) {
   switch(error.code) {
      case -1:
         connector.terminate(error);
         break;
      default:
         connector.reconnect();
         break;
   }
}
```

Example - terminate reconnection attempts in response to user command:

In this example, the default reconnection behavior has not been disabled, but there is a UI control which the user can press to short-circuit the reconnection attempts. If the user has not terminated the connection attempts, automatic reconnection attempts continue.

```
var willRetry = function(retryInSeconds, retryAttemptNumber,
maximumRetryAttempts, connector) {
   if (userHasTerminatedConnection) {
      connector.terminate({code: -1, message: 'User has terminated
      connection'});
   }
}
```

Permissions

You can use permissions to prevent an agent from interacting with, or even seeing, a UI control. Whether an agent can see a particular control or not depends upon both the agent's and the control element's **permissions**.

- **Control element permissions**

  Client applications assign permission markers to UI control elements by calling the `AssistSDK.setPermissionForElement` method:

  ```
  var element = document.getElementById('element_id');
  AssistSDK.setPermissionForElement('permission_X', element);
  ```

  or by setting it on the element in the HTML as a data attribute:

  ```
  <input type='button' id='id_hidden' data-assist-permission='permission_
  X'/>
  ```

  where `permission_X` is the **permission marker** to set on the control.

  Each UI element has at most one permission marker value; elements which do not have a permission marker inherit their parent element’s permission marker; an element which does not have a permission marker either assigned explicitly or inherited from its parent, is assigned the `default` permission marker.

  The `default` permission is explained further in the the Default Permission section on page 44.

- **Agent permissions**

  Agents have two sets of permissions, **viewable permissions** and **interactive permissions**. Each set may contain an arbitrary number of values. Agents which are not assigned any permissions have the `default` permission for both interactive and viewable permission sets.

  Live Assist grants permissions to the agent when the agent presents a **Session Token Description** to the **Live Assist** server (see the ***Live Assist Agent Console Developer Guide*** for more information about setting agent permissions, and under what circumstances the agent can be implicitly assigned the `default` permission).

  The application can determine an agent’s permissions from the `agent` object which it receives in the agent callbacks (see the Agent Callbacks section on page 21). If the application needs to examine this (for instance, to notify the consumer that a particular control will not be visible to the agent), use the `viewablePermissions` and `interactivePermissions` properties of the `agent` object. These properties are arrays of strings representing the permissions an agent has:

```
var permissions = agent.viewablePermissions();
var index = permissions.findIndex(function(element) {
  return element == 'permission_X';
});
if (index >= 0) {
    ...
}
```

Note: If the agent specifies **permissions** in the **Session Token Description**, but leaves both the viewable set and interactive set empty, the agent will end up with no permissions, not even the `default` permission.

The combination of the element's and the agent's permissions determines the visibility of a UI element to an agent. A UI element is visible to a specific agent if, and only if, the agent's set of viewable permissions contains the permission marker assigned to or inherited by that element. Similarly, an agent may interact with a UI element if and only if the agent's set of interactive permissions contains the element's permission marker.

Permissions and permission markers are free-form text, which (apart from the reserved `default` permission) are in the control of the application developer. **Live Assist** will show to the agent those, and only those, elements which the agent has permission to view; but it is up to the application developer to ensure that each agent has the permissions they need, and that the UI elements have corresponding permission markers assigned.

Live Assist assumption: When an agent wishes to establish a co-browse, the permissions the agent should have, as defined by the organization's infrastructure, are known, and can be translated into an equivalent set of permissions in the Session Description.

Agent and Element Permissions

Permissions are compound such that:

Permission marker on element	Agent viewable permission set	Agent interactive permission set	Result
X	["X"]	["X"]	Agent can view and interact with an element marked with X.
X	["X"]	[]	Agent can view the element marked with X but cannot interact with it.
X	[]	["X"]	Agent can neither view nor interact with the element, because it does not have X in its viewable set. (In order to interact with an element, and agent must first be able to view it.)
X	[]	[]	Element marked with X is masked or redacted, as Agent does not have the X permission in its viewable or interactive set.
X	["default"]	["default"]	Element marked with X is masked or redacted, because Agent does not have the X permission in its viewable or interactive set.
X	["default"]	["X"]	Agent can neither view nor interact with the element, because it does not have X in its viewable set.
X	["X"]	["default"]	Agent can view the element, because it has the X permission in its viewable set; it cannot interact with it, because it does not have the X permission in its interactive set.

Permission marker on element	Agent viewable permission set	Agent interactive permission set	Result
B	["X"]	["X"]	Element marked with `B` is masked or redacted, because Agent has `X` permission and not `B` in their permission set.
	["X","default"]	["X","default"]	Agent can view and interact with the element because they have the `default` permission in their viewable and interactive sets, and the element implicitly has the `default` permission.
	["X"]	["X"]	Element is masked or redacted, because Agent's sets do not contain the `default` permission
	["default"]	["default"]	Agent can view and interact with the element, because they have the `default` permission set for their viewable and interactive set.
	[]	[]	Element is masked or redacted, because Agent's sets do not contains `default` permission
	["default"]	["X"]	Agent can see the element because they have the `default` permission in their viewable set. They cannot interact with it because they do not have the `default` permission in their interactive set.
B	["X"]	["B"]	Element is masked or redacted because the agent's viewable set does not contain `B`. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.

Permission marker on element	Agent viewable permission set	Agent interactive permission set	Result
B	["B"]	["X"]	Element is viewable, because the agent's viewable set contains `B`; the element is not interactive, as the agent's interactive set does not contain `B`.

An agent is granted a permission if a permission (such as `A`, `B`, or `X`) configured in their Session Description matches the permission-marker of the UI element in the application.

Note: In some circumstances an agent can be granted the `default` permission *implicitly*, but that ***is not the same thing as having an empty set of permissions***. In the above table, an empty set of agent permissions means exactly that; a set of permissions containing *only* the default permission may have been granted either implicitly or explicitly.

Parent and Child Permissions

An element can also inherit permissions through the UI hierarchy: UI elements that are a child of a parent UI element inherit the permission marker of the parent, unless the child specifies a permission marker of its own.

A child element can override its parent permission marker, but it will only be effective if the agent's viewable permission set contains the parent’s permission marker as well as the child's (the agent must be able to see the container in order to interact with an element inside it). This allows the developer to make a child element interactive and the parent element not. An example use of this could be a child button within a parent container, where only the button needs to be interactive.

Permission marker set on parent element	Permission marker set on child element	Agent viewable permission set	Agent interactive permission set	Result
A		["A"]	["A"]	Agent can view and interact with both parent and child element. Child inherits permission marker A.
A	A	["A"]	["A"]	Agent can view and interact with both parent and child element.
A	B	["A"]	["A"]	Agent cannot view or interact with child element marked with B.
A	B	["A","B"]	["A"]	Agent can view child element but cannot interact with it
A	B	["A","B"]	["B"]	Agent can view and interact with the child element but cannot interact with the parent.
A	B	["B"]	["B"]	Agent cannot view or interact with child or parent element as they do not have the parent's permission marker in their viewable permission set. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.
		["default"]	["default"]	Agent can view and interact with both parent and child elements as they have the `default` permission in their viewable and interactive permission sets, and both parent and child elements implicitly have the `default` permission.

Permission marker set on parent element	Permission marker set on child element	Agent viewable permission set	Agent interactive permission set	Result
	B	["B"]	["B"]	Agent cannot view or interact with child element, because the parent has an implicit `default` permission marker, and they do not have the `default` permission in their viewable permission set. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.

Default Permission

You do not have to assign a permission marker to every UI element which you want agents to view or interact with; every element which does not have or inherit a permission automatically has the `default` permission marker.

Elements which have the `default` permission marker are viewable and interactive for any agent which has the `default` permission. Any agent which has the `default` permission includes the reserved word `default` among its set of permissions (in the`viewablePermissions` or `interactivePermissions` properties of the `agent` object).

Not every agent has the `default` permission, and an agent might have the `default` permission in its viewable permissions, but not in its interactive permissions.

Dynamic Web Element Masking

You can also mask page elements that are dynamically added and removed using AJAX. To do this, the application should call `setPermissionForElementWithId`, which allows the application to add a permission to an element which does not yet exist:

```
  AssistSDK.setPermissionForElementWithId('permission_X', 'element_id');
```

When the application calls the above method, typically when the page is loaded, **Live Assist**:

- Checks to see if the element exists on the page:
    - if it does, then the element is marked with the given permission.
    - otherwise, it stores the combination of permission and element ID.
- Listens for DOM change events, and when a new element is added:
    - if the element ID corresponds to one of the stored element IDs, **Live Assist** adds the stored permission.

Note: The list of permission markers and element IDs is cleared when the page is refreshed, so `setPermissionForElementWithId` does need to be called when the page is loaded.

The application can also call:

```
AssistSDK.setPermissionForElementInIframeWithId('permission_X', 'elementId',
iframe);
```

which does the same for an element within an iframe. The `iframe` parameter is the iframe element itself (acquired by calling `getElementById`, `createElement('iframe',...)`, or a similar function of the `Document` object).

Internationalization

The **Live Assist Web SDK** keeps its assets in `assist_assets.war`. You can edit this file to add another language:

1. Get a copy of `assist_assets.war` from `/opt/cafex/<FAS>/domain/deployment_backups` (that is, from the `/domain/deployment_backups` directory of your FAS installation). It will be named `assist_assets.war-<datetime>`, where `<datetime>` is a date and time in ISO 8601 format.
2. Unzip it and open the file `sdk/web/shared/locales/assistIi18n.en.json` (this is the English language file).
3. Edit the entries so that the values are in the target language.
4. Save the file in the same directory, under the name `assistIi18n.<lang>.json`, where `<lang>` is the 2 letter language code of the target language: `es` for Spanish, `fr` for French, and so on.
5. Re-zip the file, maintaining the original file structure, and redeploy it to the server (update `assist_assets.war` with the new file – see the ***FAS Administration Guide***).

When calling `AssistSDK.startSupport`, provide a `locale` parameter in the configuration object. The value should be the 2 letter language code for the target language.

Integrating with FCSDK

Live Assist SDKs use facilities from **Fusion Client SDKs**, and rely on an instance of the FCSDK being available, so all the facilities of FCSDK are available for you to use if you want.

When you call the `Assist.startSupport` method and provide a `destination`, but no `correlationId` or `sessionToken`, in the `AssistConfig`, **Live Assist** automatically starts a voice and video call and a co-browse session with the agent, and automatically ends the call when the application calls `Assist.endSupport`. If you want more control over the voice and video call than this, you can access FCSDK objects from the global `UC` object:

- If you called `startSupport` with a configuration object which does *not* include a session token, it automatically requests a session token and initializes FCSDK with it. In this case, the `UC` object is automatically available as a global object for you to use.
- If you started a co-browse only session, there is no call under the control of the FCSDK, so the FCSDK objects are not available.

Having obtained a `UC` object, you can use the facilities available from its `phone` object to control the call:

```
var call = UC.phone.getCall(CALL_ID);
 ...
call.end();
```

See the ***FCSDK Developer Guide*** for details on what call control facilities are available, and how to use them.

Starting a call without Voice and Video

You can use **Live Assist** in **co-browse only** mode, if the voice or video call is provided independently of **Fusion Client SDK** and **Live Assist**, or when something like a chat session is used instead of a voice and video call.

To prevent **Live Assist** from placing a call using the **Fusion Client SDK**, the application should provide a **correlation ID** that **Live Assist** uses to correlate the consumer and agent sides of the co-browsing session. This allows an application to use the features of **Live Assist** (for example, co-browsing, document push, annotation, and remote control) without voice or video.

For example, to add a link to click for support:

```
<a title="Live Assist"
  onclick="AssistSDK.startSupport({correlationId : 'your_correlation_ID'})">
  Live Assist</a>
```

where the parameter specified is the unique ID used to correlate agent and consumer sessions. The newly created session for co-browsing is associated with the correlation ID which you supplied.

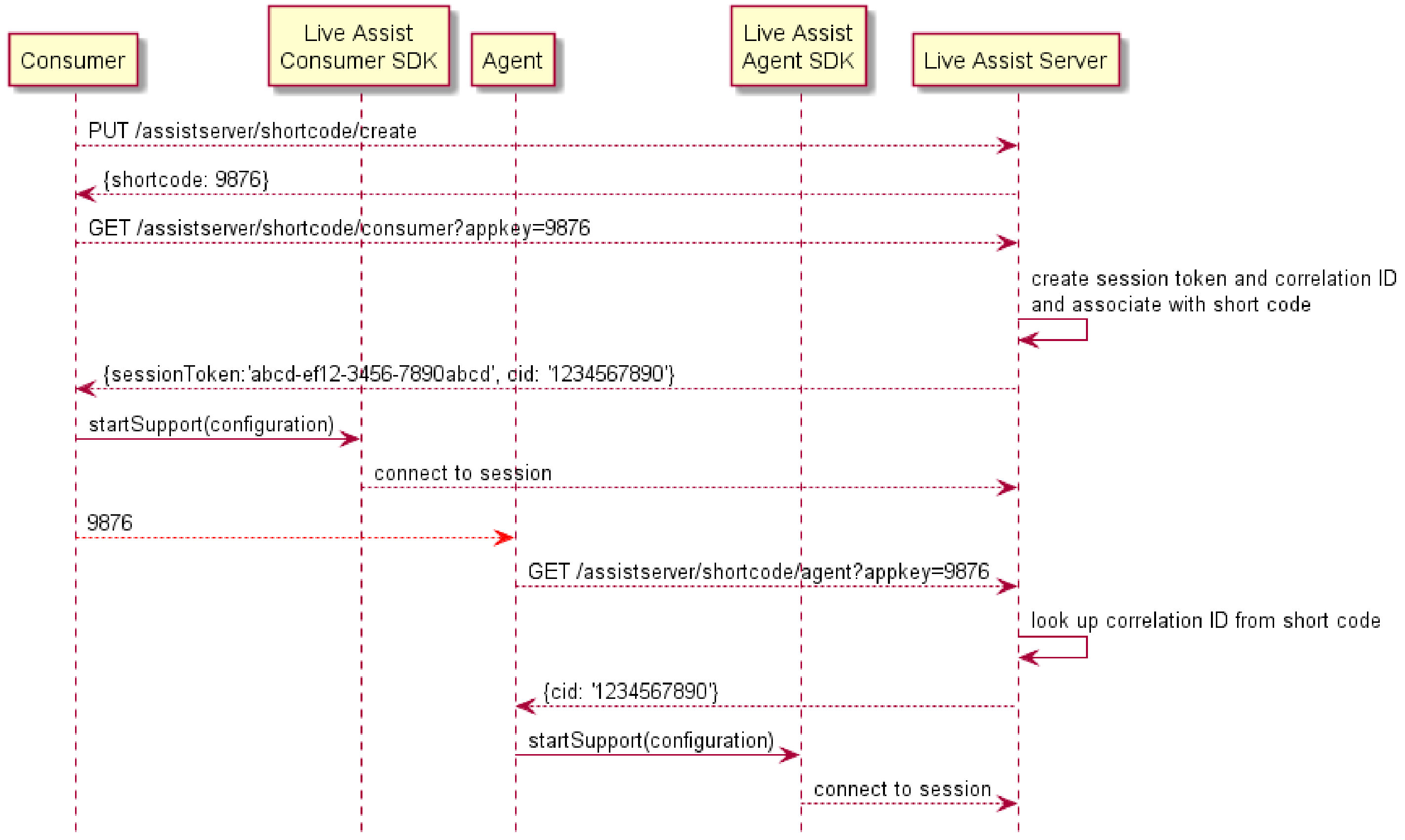
Note:

- In a co-browse only session, the application must explicitly call `endSupport` when the call ends (or when the co-browse session is no longer needed), as **Live Assist** does not present its default UI to the user.
- The correlation ID needs to be known to both parties in the call, and needs to be unique enough that the same correlation ID is not used by two support calls at the same time. The application developer must decide the mechanism by which this happens, but possible ways are for both parties to calculate a value from data about the call known to both of them, or for one side to generate it and communicate it to the other on the existing communication channel. There is also a REST service provided by **Live Assist** which will create a correlation ID and associate it with a short code; see the Informing the Agent of the Correlation ID section on the next page.

Note: The correlation ID needs to be known to both parties in the call, and needs to be unique enough that the same correlation ID is not used by two support calls at the same time. The application developer must decide the mechanism by which this happens, but possible ways are for both parties to calculate a value from data about the call known to both of them, or that one side calculates it and communicates it to the other. There is also a REST service provided by **Live Assist** which will create a correlation ID and associate it with a short code; see the Informing the Agent of the Correlation ID section below.

Informing the Agent of the Correlation ID

Live Assist gives some help to the application in informing the agent of the correlation ID; it can create a **short code** and associate it with the correlation ID when it creates the session, and the client can send the short code out-of-band to the agent:


Consumer
Live Assist Consumer SDK
Agent
Live Assist Agent SDK
Live Assist Server
PUT /assistserver/shortcode/create
{shortcode: 9876}
GET /assistserver/shortcode/consumer?appkey=9876
create session token and correlation ID and associate with short code
{sessionToken:'abcd-ef12-3456-7890abcd', cid: '1234567890'}
startSupport(configuration)
connect to session
9876
GET /assistserver/shortcode/agent?appkey=9876
look up correlation ID from short code
{cid: '1234567890'}
startSupport(configuration)
connect to session

The advantage of communicating a short code, rather than communicating the correlation ID directly, is that the short code generated by the **Live Assist** server is guaranteed to be both unique during the communication process, and short enough for the client to communicate by voice (or whatever other out-of-band communication channel is in use) without error.

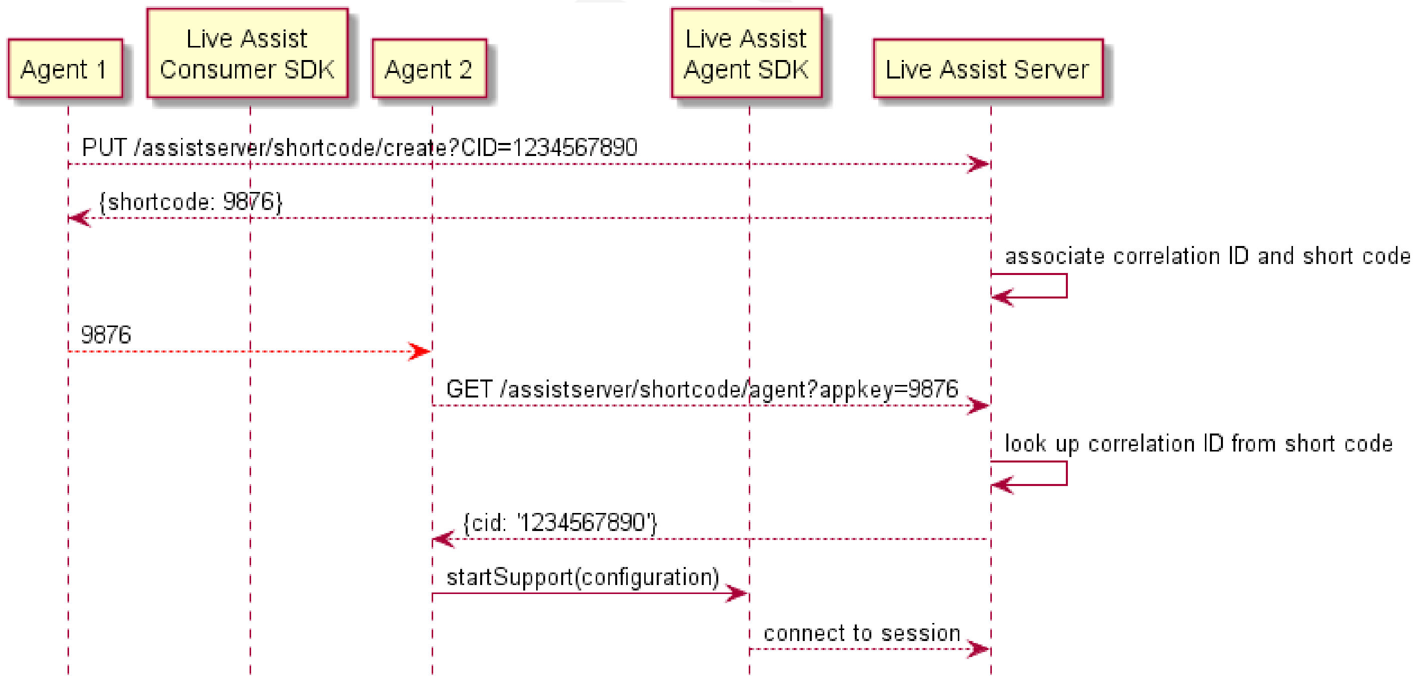
Note:

- The short code expires 5 minutes after creation; it should therefore be used as soon as possible after being created.

- Once a short code has been used by both agent and consumer to communicate a correlation ID, it is discarded, and may be used by a different agent and consumer to communicate a different correlation ID.

Including a Second Agent

There is a second scenario in which the short code REST service can be used when there is no existing call between the two parties. If an agent is in a co-browse session with a consumer and wishes to include a second agent in the same co-browse session, the agent already knows the correlation ID of the **Live Assist** session, and can use it in the initial call to the REST service, to associate that correlation ID with a newly created short code:


Agent 1
Live Assist Consumer SDK
Agent 2
Live Assist Agent SDK
Live Assist Server
PUT /assistserver/shortcode/create?CID=1234567890
{shortcode: 9876}
associate correlation ID and short code
9876
GET /assistserver/shortcode/agent?appkey=9876
look up correlation ID from short code
{cid: '1234567890'}
startSupport(configuration)
connect to session

The second agent application uses the short code in exactly the same way as before to connect to the same co-browse session as the first agent and the consumer.

WebSocket Initiation

To prevent getting a 302 error reported to you by a WebSocket during handshaking, ensure that your deployment allows direct access to the WebSocket endpoint:

```
wss://<fas address>:<port>/assistserver/topic
```

Controlling Updates to the Agent's View

During co-browsing, **Live Assist** observes changes in the Document Object Model on the consumer's web page, and updates the agent's view if an element changes. If a page makes frequent changes to the DOM, such as changing an element's `style` attribute, each of these changes causes an update to the agent's view, making the agent's console unresponsive.

Note: The code which does this may not be explicit; some JavaScript frameworks make repeated changes to the DOM as part of their normal operation.

If you find that one or more of the elements on a page changes frequently, you can prevent it from causing **Live Assist** to update the agent's view, by including the element in the `mutationBlacklist` property of the configuration object when you call `startSupport`:

```
var config;
config.destination = 'customer-support';
config.mutationBlacklist = {elements:['mutating-element-id']};
 ...
AssistSDK.startSupport(config);
```

The `mutationBlacklist` object contains three lists:

Property	Contents
`elements`	List of element `id` attributes; these elements are added to the `mutationBlacklist`.
`classes`	List of `class` attributes; all elements with one of these `class` attributes are added to the `mutationBlacklist`.
`animations`	List of `animation-name` attributes; all elements with one of these `animation-name` attributes are added to the `mutationBlacklist`.

DOM changes (such as changing the `style` attribute) for elements which have been added to the `mutationBlacklist` do not make the agent's view update; other changes, such as scrolling the screen (*including* entering data into a form element which is in the `mutationBlacklist`), do make the agent's view update. When the agent's view does update, it includes the view of any blacklisted elements in their state at the time of the update.

A common reason to blacklist an element is if a web page has an animated logo. The agent does not need to see the animation, and updating the agent's view for each change makes the agent console unresponsive. Add the animated element to the blacklist:

```
config.mutationBlacklist = {animations: ['logo-animation'], classes:
['animated'] };
```

to prevent animation in any element which has a `class` attribute of `animated`, or an `animation-name` attribute of `logo-animation`, from making the agent's view update. The agent sees an instantaneous snapshot of the animated element.

Consumer Session Creation

A client application needs an **FCSDK Web Gateway session token** and a **correlation ID** to establish a co-browsing session. When the application calls `startSupport`, **Live Assist** uses a built-in mechanism to create a session token for the voice and video call, and associates it with a correlation ID for the co-browse. The built-in mechanism provides a standalone, secure mechanism for creating a session token and a correlation ID, but the process is not integrated with any pre-existing authentication and authorization system, and assumes that if a client can invoke `startSupport`, it is permitted to do so.

If you wish to integrate your **Live Assist** application with an existing authentication and authorization system, you can disable the built-in mechanism (by setting the **Anonymous Consumer Access** setting to `disabled` using the Web Administration service; see the ***Live Assist Overview and Installation Guide*** for how to do this), and replace it with a bespoke implementation which uses the existing system to authorize and authenticate the client.

Once you have authenticated and authorized the application using the pre-existing system, the application needs to create a session token (see the ***Fusion Client SDK documentation*** for details of how to create the session token) and associate it with a correlation ID.

Session Token Creation

A bespoke implementation needs the following general steps:

1. Create a Web Application that can invoke the **Session Token API** REST Service, exposed by the **FCSDK Web Gateway**.
2. Provide the appropriate **Fusion Client SDK** (if in use) configuration in a JSON object (the **session description**).
3. Add **Live Assist**-specific data to the session description:
   - `AED2.metadata.role`

     This should be set to `consumer`

- `AED2.metadata.auditName`

  Optional name to use to identify the consumer in event log entries (see the ***Live Assist Overview and Installation Guide*** for details about event logging.

- `AED2.allowedTopic`

  A regular expression which limits the correlation IDs which the session token can be used to connect to. A value of `.*` allows the session token to be used to connect to any support session with any correlation ID. For security reasons, we recommend that this should be set to the value of the correlation ID which will actually be used:

```
{
  ...
  "voice": {
    ...
  },
  "aed": {
    "accessibleSessionIdRegex": "customer-ABCDE",
    ...
  },
  ...
  "additionalAttributes": {
  {
    "AED2": {
      "metadata": {
        "role": "consumer",
        "auditName":"audit name"
      },
      "allowedTopic": "customer-ABCDE"
    }
  },
  ...
}
```

4. Request a session token by sending an HTTP `POST` request to the **Session Token API**, providing the session description in the body of the `POST`.

For steps 1, 2, and 4, see the ***FCSDK Developer Guide, Creating the Web Application***.

Note: The ***FCSDK Developer Guide*** documents both `voice` and `aed` sections - at least one of these must be present to create the session token. However, if the session description includes a `voice` section (which it must if voice and video functionality is required), then only the `AED2` entries are needed for **Live Assist**

functionality. If voice and video functionality is not needed, and there is no `voice` section, then there must be an `aed` section as well as the `AED2` section entries.