API Reference

Genesys Co-browse 8.5.0

2/26/2022

Table of Contents

Genesys Co-browse API Reference	**3**
History REST API	**4**
Realtime API (CometD and REST)	**6**
Client-Initiated CometD Channels	7
Server-Initiated CometD Channels (Notifications)	9
RESTful Realtime Functions	11
JavaScript API	**13**
Configuration API	14
Co-browse API	23
Chat API	32
External Media Adapter API	35

Genesys Co-browse API Reference

Welcome to the *Genesys Co-browse 8.5 API Reference*. This document provides you with the information you need to use the Genesys Co-browse APIs. See the summary of chapters below.

History REST API

Use this API to find information about past Co-browse sessions.

History REST API

Realtime API

Use this CometD and REST API to manage current Co-browse sessions.

Realtime API

JavaScript API

The JavaScript API allows you to customize the Co-browse JavaScript application.

Overview

Configuration API

Co-browse API

Chat API

External Media API

History REST API

Information about every past Co-browse session is available through the REST API, sitting on top of the Cassandra database. The REST API (REST resources) is hosted by the Co-browse Server.

Each session history record (one per session) is identified by a session ID (UUID), which is unique across the Co-browse cluster within any given period of time. This ID must not be confused with the session token (a random 9-digit sequence) that is used to connect the Co-browse session.

The session history ID is attached to the primary interaction, voice or chat, with the "CoBrowseSessionId" key. For development purposes, it can also be found in the logs with the logging level `info`: "Session created. Token: `{}` Id: `{}`". Full session history is available after a session is deactivated. You can set how long session history is kept in the database and available through the REST API with the retention policy configuration options.

In this initial release of Genesys Co-browse, the REST API is simple but it will be extended in future releases.

Get session history

Request

HTTP method: GET
Resource: /history/sessions/{sessionHistoryId}
Parameters:

- sessionHistoryId - session history identifer (UUID)

Example request: http://192.168.73.77:8700/cobrowse/rest/history/sessions/83d03970-c959-11e2-857d-082e5f12b9a1

Response

Headers

```
HTTP/1.1 200 OK
Date: Thu, 19 Sep 2013 12:21:19 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Server: Jetty(8.1.8.v20121106)
```

Body

```
{
   "id":"83d03970-c959-11e2-857d-082e5f12b9a1",
   "sessionToken":"519333886",
   "creationTime":1369939707527,
   "activationTime":1369939712516,
   "deactivationTime":1369939743522,
   "pages":[
      {
         "url":"http://www.genesyslab.com/general-pages/about-us.aspx",
         "enteredTimestamp":1369939712594,
         "duration":11
      },
      {
         "url":"http://www.genesyslab.com/news-and-events/index.aspx",
         "enteredTimestamp":1369939723764,
         "duration":7
      },
      {
         "url":"http://www.genesyslab.com/general-pages/genesys-history.aspx",
         "enteredTimestamp":1369939731685,
         "duration":11
      }
   ]
}
```

Timestamps are Unix timestamps in milliseconds.

Duration is in seconds. For the first page in a Co-browse session it starts counting when Co-browse session starts and for the last page it ends counting when the Co-browse session ends.

Realtime API (CometD and REST)

Most of the server API supporting live sessions is CometD based, but a few functions require REST / plain HTTP transport. For details, see:

- Client-Initiated CometD Channels
- Server-Initiated CometD Channels (Notifications)
- RESTful Realtime Functions

Client-Initiated CometD Channels

Create Session

CometD Channel	/service/session/create
Description	Creates a new co-browse session.

Request:

```
{}
```

Response:

```
{
  "sessionToken": "123"
}
```

Join Session

CometD Channel	/service/session/join
Description	Allows the user to join the session.

Request:

```
{
  "sessionToken": "123",
  "role": 2, // 1 - customer, 2 - agent, 3 - controller
  "name": "Bob" //optional (for controller it does not make sense at all)
}
```

Response:

```
{
  "userToken": "abc789",
  "userId": 2,
  "users": [
    {
      "userId": "1",
      "role": 1,
      "name": "John"
    }
  ],
  "sessionHistoryId": "abc789"
}
```

Every connected user receives a `userToken` providing personalized session access.

Exit Session

CometD Channel	/service/session/exit
Description	Allows the user to exit the session explicitly. The CometD disconnection should be handled via session deactivated notification.

Request:

```
{
    userToken: "abc123"
}
```

Response:

```
{}
```

Stop Session

CometD Channel	/service/session/stop
Description	Stops the co-browse session. This is available for the Controller only. The session is deactivated and all users receive the session deactivated notification.

Request:

```
{
    userToken: "abc123"
}
```

Response:

```
{}
```

Server-Initiated CometD Channels (Notifications)

Joined Session

CometD Channel	/service/session/joined
Description	Notification to all users who are already in the session about a new joined user.

Notification:

```
{
  "userId": "2",
  "role": 2,
  "name": "Bob"
}
```

Activated Session

CometD Channel	/service/session/activated
Description	Notification to all clients about session activation.

Notification:

```
{
  "activationTime": 1368722791040 // UTC time in ms
}
```

Deactivated Session

CometD Channel	/service/session/deactivated
Description	Notification to all clients about session deactivation. When the client receives a session deactivated notification, it should disconnect from CometD. Sending the exit session command is not needed.

Notification:

```
{
  "activationTime": 1368722791040 // UTC time in ms
  "deactivationTime": 1368722820929 // UTC time in ms
}
```

RESTful Realtime Functions

The RESTful resources that manage live co-browse sessions are sub-resources of `<cobrowse-app>/rest/live`. For example, `http://127.0.0.1:8700/cobrowse/rest/live`

Create Session

Request URL	/sessions
HTTP Method	POST
Description	Creates a new Co-browse session.

Response:

```
{
  "sessionToken": "845800826",
  "sessionServerName": "Co-browse_Server"
}
```

The HTTP response has a cookie, `gcbSessionServer`, which should stick further HTTP requests to the server hosting the created session.

Get Session

Request URL	/sessions/{id}
HTTP Method	GET
Description	Returns live session public data. The main purpose is to determine which server the session is hosted on (this is needed to integrate the agent Co-browse plug-in with the Co-browse cluster). The id is the live session ID.

Response:

```
{
  "sessionToken": "845800826",
  "sessionServerName": "Co-browse_Server",
  "sessionServerUrl": "https://cobrowse-node/cobrowse" // serverUrl option value, may be
absent
}
```

Important

`sessionServerUrl` is returned only if the **serverUrl** option is set for the node. This is used for URL-based stickiness.

Stop Session

Request URL	/users/{userToken}/session/stop
HTTP Method	GET
Description	Initiates Co-browse session deactivation for the Controller's session. `userToken` is the Controller's `userToken`.

Response

```
{
  "activationTime": 1368722791040, // UTC time stamp in ms
  "deactivationTime": 1368722820929 // UTC time stamp in ms
}
```

Health Check

Request URL	/health
HTTP Method	GET
Description	Checks if the the Co-browse Server node is alive and ready to handle requests.

This resource is useful for Load Balancing. You may also use this resource to check if any server is a live before showing the Co-browsing button in the user's browser.

Important

This resource is not a subresource of `<cobrowse-app>/rest/live`. The full URL may have a format like http://127.0.0.1:8700/cobrowse/health.

Response:

This resource replies with an empty response and a `200 OK` HTTP status if the node is alive and ready.

JavaScript API

The JavaScript API allows you to customize the Co-browse JavaScript application. The JavaScript API is split into the following parts:

- Configuration API—used to configure Co-browse and its integration with other media. Also used to subscribe to the main Co-browse JavaScript API and the Chat API.
- Co-browse API—the main Co-browse API. It provides methods and callbacks to work with Co-browse and can be used to implement a custom UI for co-browsing.
- Chat API—API of the built-in Chat widget. Can be used to customize the widget and to access the lower level Chat Service API.
- External Media API—allows you to integrate Co-browse with a custom chat, WebRTC or any other JavaScript based media.

Configuration API

This API configures Co-browse and its integration with other media. It is also used to subscribe to the main Co-browse JavaScript API and the Chat API.

Co-browse is configured via a global `_genesys` variable. To configure Co-browse (and/or Chat), create a <script> such as the following example and add it to your instrumentation:

```
<script>
var _genesys = {
    cobrowse: {
        /* Co-browse configuration options */
    },
    chat: {
        /* Chat configuration options */
    }
};
</script>
<INSTRUMENTATION_SNIPPET>
```

Important

- Co-browse is designed to make configuration optional. If any configuration options are not present, Co-browse will use the pre-defined default values.
- For reference on Chat configuration options, see startChat Options. All options specified in `_genesys.chat` are internally passed to the `startChat()` method call.

Warning

For backward compatibility with previous versions of Co-browse, the name of the global configuration variable can also be _gcb. This is deprecated and may be discontinued in later versions, so it is recommended that you switch to _genesys **immediately** if you are currently using _gcb.

Accessing the Co-browse and Chat APIs

Since the main Co-browse JavaScript file is added to the page asynchronously, you cannot instantly access the Co-browse and Chat APIs. Instead, you must create a function that will accept the APIs as an argument. There are two approaches to creating this function.

You can assign the function to the special property of a global configuration variable:

```
<script>
var _genesys = {
    onReady: function(APIs) {
        APIs.cobrowse // Co-browse API
        APIs.chat     // Chat widget API
    }
};
</script>
<INSTRUMENTATION_SNIPPET>
// or
<script>
var _genesys = {
    cobrowse: {
        onReady: function(cobrowseApi) { ... }
    }
};
</script>
<INSTRUMENTATION_SNIPPET>
```

Alternatively, you can modify configuration to make the APIs accessible at any point in your application through a _genesys global variable.

To do this, you must first assign an array to the onReady property:

```
<script>
var _genesys = {
    onReady: []
};
</script>
<INSTRUMENTATION_SNIPPET>
// or
<script>
var _genesys = {
    cobrowse: {
        onReady: []
    }
}
</script>
<INSTRUMENTATION_SNIPPET>
```

You can then obtain the APIs at any point in your application using the following code snippet:

```
_genesys.onReady.push(function(APIs) {
    APIs.cobrowse // Co-browse API
    APIs.chat     // Chat widget API
});
// or
_genesys.cobrowse.onReady.push(function(cobrowseApi) { ... });
```

Tip

For more information on the <INSTRUMENTAITON_SNIPPET>, see Web Site Instrumentation#Basicinstrumentation.

Disabling Chat or Co-browse

You can disable the built-in Chat, or disable Co-browse (in order to use only Chat). To do that, pass the value `false` to the respective configuration subsection:

```
<script>
var _genesys = {
    chat: false
};
</script>
```

```
<script>
var _genesys = {
    cobrowse: false
};
</script>
```

Configuring the Co-browse and Chat Buttons

Configuration API enables you to configure the built-in reactive buttons using `_genesys.buttons`. For example:

```
<script>
var _genesys = {
    buttons: {
        chat: false,
        position: 'right'
    }
};
</script>
```

The `_genesys.buttons` section enables some basic configuration of the **Live chat** and **Co-browsing** buttons. It has three optional properties:

- position: Can be either `left` (default) or `right`
- cobrowse: defaults to `true`
- chat: defaults to `true`

Note that you can override only the properties that you want to be changed. Other properties will be used with their default values. For example this configuration:

```
var _genesys = {
    buttons: {
        chat: false
    }
};
```

actually means this:

```
var _genesys = {
    buttons: {
        chat: false,
        cobrowse: true, // inherited default
        position: 'left' // inherited default
    }
};
```

Disabling buttons

As seen in snippet above, you may pass the value `false` to disable the **Co-browsing** and/or the **Live chat** buttons. This might be useful if you want to start chat or co-browsing from your own custom button (or from any other element or event), using the Co-browse API or Chat API.

Providing Custom HTML for Buttons

You can also pass functions that return HTML Element to `buttons.cobrowse` or `buttons.chat`. In this case the output of the function will be used to render the button instead of using default image.

Tip

In this case your custom button(s) will inherit the positioning of the default button(s).

Here's a simple example that makes use of jQuery library to generate HTML Elements:

```
function createCustomButton() {
    return jQuery('<div class="myButtonWrapper"><button
class="myButton">Chat!</button></div>')[0];
}

var _genesys = {
    buttons: {
        chat: createCustomButton
    }
};
```

Important

Note that is NOT mandatory to use jQuery in order to provide a custom HTML element. The example above does return an HTML element out of a jQuery object by retrieving the first element from jQuery collection via `[0]`.

Localizing the Live Chat and Co-Browsing Buttons

By default the buttons are images and therefore they cannot be localized in the same way as the rest of the interface. To localize these buttons, you can use one of the two following methods:

- Provide custom localized buttons instead of the default ones, as explained in Providing Custom HTML for Buttons.
- Override the appearance of the buttons using CSS.

For more information about localizing Co-browse and Chat, see Localization.

Co-browse Configuration Options

Important

For reference on Chat configuration options, see startChat Options. All options specified in `_genesys.chat` are internally passed to `startChat()` method call.

Tip

For backward compatability with previous versions of Co-browse, the name of the global configuration variable can also be `_gcb`. The use of `_gcb` is deprecated and may be discontinued in later versions. If you are using `_gcb`, we recommend that you switch to `_genesys`.

The following options are configurable as properties of an object passed to `_genesys.cobrowse`:

debug

Default: `false`

Set to `true` to enable debugging console logs. You can enable debug logs for Co-browse only, Chat only, or for both.

Example:

```
<script>
// Enable debugging logs for both Co-browse and Chat:
var _genesys = {
    debug: true;
};
</script>

<script>
// Enable debugging logs only for Co-browse:
var _genesys = {
    cobrowse: {
        debug: true;
    }
};
</script>

<script>
// Enable debugging logs only for chat:
var _genesys = {
    chat: {
        debug: true;
    }
};
</script>
```

disableBuiltInUI

Default: `false`

Set to `true` to use a custom Co-browse UI. Use the Co-browse API to implement a custom UI.

Example:

```
var _genesys = {
    cobrowse: {
        disableBuiltInUI: true
    }
};
```

You can still start the Co-browse session with the configuration above but the main components of the UI such as the toolbar and notifications will be absent.

primaryMedia

Default: Built-in chat

Used to pass an object implementing an external media adapter interface. By default, the built-in chat is used.

Example:

```
<script>
var myPrimaryMedia = {
    initializeAsync: function(done) { /* initialize your media here and then call done() */ },
    isAgentConnected: function() { /* return true or false depending on whether an agent is
connected */ },
    sendCbSessionToken: function(token) { /* send the Co-browse session token to agent */ }
};
</script>

<script>
var _genesys = {
    cobrowse: {
        primaryMedia: myPrimaryMedia
    }
};
</script>
<INSTRUMENTATION SNIPPET>
```

See External Media Adapter API for more details.

Warning

If Co-browse does not detect any primary media or detects that the agent is not connected with the primary media, Co-browse will still ask the user, "Are you on the phone with representative?" before starting the Co-browse session.

CSS

Default: Server synchronization strategy, `{server: true}`

This option manages the CSS synchronization strategy. Additional CSS synchronization on top of DOM synchronization allows you to **replay** style changes that occur when the user moves his or her mouse over an element with a `:hover` style rule.

[+] Additional details

For example, if you have the following CSS, Co-browse CSS synchronization makes the underlining visible to the agent when the consumer moves her mouse over a link, and vice versa, the underlining will be visible to the user when the agent moves the mouse over a link:

```
a:hover {
    text-decoration: underline;
}
```

Server strategy is the default and preferred setting. The server strategy setting allows the Co-browse server to proxy every CSS resource, including inline CSS. This strategy synchronizes CSS hover effects regardless of the domain the CSS resource is loaded from.

Example:

```
<script>
var _genesys = {
    cobrowse: {
        css: {
          server: true
        }
    }
};
</script>
```

Important

If the `css` option is not specified, the Co-browse JavaScript application behavior is equivalent to the configuration snippet above.

Warning

There are limitations on handling invalid CSS. This may lead to partial or complete loss of hover synchronization. It may also cause partial failure of general style synchronization. See Troubleshooting CSS Synchronization for details.

maxOfflineDuration

Default: `600` (seconds)

This option specifies the time in seconds that a reference to a Co-browse session is stored after page load. The default value is 600 seconds (10 minutes). If this period expires, the Co-browse session will end by time out.

> **Important**
>
> If you modify this option, it must match the same option on the server, maxInterval Option.

You can apply this option to both Chat and Co-browse, as in this example:

```
<script>
var _genesys = {
        maxOfflineDuration: 300 // applied to both Chat and Co-browse
};
</script>
```

disableWebSockets

Default: `false`

Use this option if you need to disable WebSocket communication such as when your load balancer does not support WebSockets and you do not want to wait for Co-browse to automatically switch to another transport.

> **Important**
>
> Due to the highly interactive nature of Co-browse, we highly recommended you do **not** disable WebSockets. We recommend that you configure your load balancers/ proxies infrastructure to support WebSockets. Disabling WebSockets may have a huge impact on Co-browse performance.

You can apply this option to both Chat and Co-browse, as in this example:

```
<script>
var _genesys = {
        disableWebSockets: true // applied to both Chat and Co-browse
};
</script>
```

localization

Default: undefined

Use this option to localize Genesys Co-browse and/or built-in Chat. For a detailed description, see Localization.

setDocumentDomain

Default: `true`

Determines if Co-browse sets the `document.domain` property. If set to `true`, Co-browse modifies the `document.domain` property. If set to `false`, Co-browse does not modify `document.domain`.

Available since Co-browse JavaScript version `8.5.002.02`. For your Co-browse JavaScript version, see the VERSION property.

Important

Co-browse modifies `document.domain` to support cross-subdomain communication between iframes and the topmost context. Setting `setDocumentDomain` to `false` stops synchronization of subdomain iframes from working.

Example:

```
<script>
// Turn off setting document.domain:
var _genesys = {
    cobrowse: {
        setDocumentDomain: false
    }
};
</script>
```

Co-browse API

This API provides methods and callbacks to work with Co-browse and can be used to implement a custom UI for co-browsing.

Important

See Accessing the Co-browse and Chat APIs for information on accessing this API.

Co-browse in iframes

Some Co-browse UI elements such as the the co-browsing button and toolbar should not appear when Co-browse is in an iframe. Common Co-browse UI elements such as notifications that an element is masked should appear whether or no Co-browse is in an iframe. As such, there are two contexts for the Co-browse JavaScript API:

- Top context, available when Co-browse is not rendered in an iframe.
- Child context, used when a page is rendered in an iframe. For the child context, a subset of the top context API is available.

isTopContext

The `isTopContext` variable can be used determine which context Co-browse is rendered in. `isTopContext` is passed to the onReady method and equals `true` if Co-browse is rendered in the top context and `false` otherwise.

Example:

```
var _genesys = {
    cobrowse: {
        onReady: function(api, isTopContext) {
            // common functionality
            api.onMaskedElement.add(function() {/* deal with masked elements here*/});
            if (!isTopContext) {
                return;
            }
            // top context functionality goes below
    }
};
```

Tip

See Accessing the Co-browse and Chat APIs if you are unfamiliar with the onReady syntax above.

Signals and Callbacks

The Co-browse API exposes a number of **signals** in both the top and child contexts. Each signal is object with the three following methods:

- `add(function)`—adds a callback
- `addOnce(function)`—adds a callback that will be executed only once
- `remove(function)`—removes a callback

The naming convention for signal names begins with "on" and follows the format **onSomethingHappened**.

Important
Signals act similar to **deferred** objects. If you add a callback to an event that has already happened, the callback will be called immediately. For example, if you add a callback to the `onAgentJoined` signal when the event has already happened, the callback will be called immediately.

Session Object

Many callbacks receive a `session` object as an argument. This object has the following properties:

- `token`—String containing the session token shared with the agent and possibly shown in the UI. The token is a 9 digit string such as "535176834".
- `agents`—Array of connected agents. Each element in the array is an object with no properties.

Common API

The following elements and properties are available from both the top and child Co-browse contexts:

VERSION

String containing current JS version. For example, `8.5.000.90`.

```
console.log(_genesys.cobrowse.VERSION);
```

Tip

- Available since Genesys Co-browse 8.5.
- The JavaScript version does not necessarily match the product or server version.

markServiceElement(element)

Service elements do not show up in the agent's view. This function is used to mark service elements in a custom Co-browse UI.

Arguments:

- `element`—HTML element that will be masked.

Important

Elements must be marked as **service** elements **before** the Co-browse session begins. If the Co-browse session has already started, **service** elements should be marked before they are added to the DOM. It is also possible to mark elements as **service** without using this function. Doing so is useful for static HTML content. To do so, add an attribute `data-gcb-service-node` with value `true`. This available since version 8.5.001.20. Use `_genesys.cobrowse.VERSION` to check the version.

Important

The `markServiceElement()` method should not be used to hide sensitive information. Business functions like DOM Control or Data Masking should be used for sensitive content such as private user data.

Plain DOM Example:

```
function createCustomCobrowseUI(cobrowseApi) {
    var toolbar = document.createElement('div');
    toolbar.className = 'cobrowseToolbar';
    toolbar.textContent = 'Co-browse is going on';
    cobrowseApi.markServiceElement(toolbar); // don't show the toolbar to agents
    cobrowseApi.onConnected.add(function() {
        document.body.appendChild(toolbar);
    })
}
```

jQuery Example:

```
// Create a simple jQuery plugin
$.fn.cbMarkNode = function() {
    return this.each(function() {
        cobrowseApi.markServiceElement(this);
    });
};

// And then:
$('<div class="cobrowseToolbar">Co-browse is going on</div>').cbMarkNode().appendTo('body');
```

Static content example, without JS API usage:

```
<div id="myChatWidget" data-gcb-service-node="true">...</div>
```

onMaskedElement

This signal is dispatched when Co-browse encounters an element that is subject to data masking.

Arguments:

- `element`—HTML Element

This signal is dispatched multiple times when Co-browse initiates and can be dispatched again if a masked element is added to the page dynamically.

Example:

```
cobrowseApi.onMaskedElement.add(function(element) {
    $(element).tooltip({
        content: 'Content of this elements is masked for representatives.'
    });
});
```

Consider a scenario where you have the following HTML elements on your **example.html** page:

```
<div class="vcard">
    <p class="fn"><a class="url" href="#">Dr. John Doe</a><p>
    <p class="adr">
        <span class="street-address">Imaginary Hospital</span><br>
        <span class="region">Doctorville</span><br>
        <span class="postal-code">742617</span><br>
        <span class="country-name">Great Britain</span>
    </p>
    <p class="tel">+44 (0)1234 567890</p>
</div>
```

And you also have the following data masking configuration:

```
<?xml version="1.0" encoding='UTF-8' ?>
<domRestrictions>
    <restrictionsSet>
        <uriTemplate type="regexp" pattern="example\.html"/>
        <dataMasking>
            <element selector=".adr"/>
            <element selector=".tel"/>
        </dataMasking>
    </restrictionsSet>
</domRestrictions>
```

In this scenario, the callback is called two times when Co-browse is initiated, and then each time you dynamically add an `.addr` or `.tel` element to the page.

Top Context API

The following methods and properties are available only when Co-browse is rendered in the **top** context.

isBrowserSupported()

> **Important**
> For a list of officially supported browsers see Browser Support. This method checks for the presence of required browser APIs and may return `true` for browsers not officially supported.

This method checks for the presence of MutationObserver and a few other required APIs, not for browser type and version. It returns a boolean with the value of `true` when the browser supports required APIs and `false` otherwise. The built-in integration module uses this function to show a message if a user tries to start Co-browse in an unsupported browser. You may use it, for example, to hide the Co-browse button completely.

startSession()

This method instantiates a new Co-browse session. It will throw an error if the browser is not supported.

exitSession()

This method exits and ends an ongoing Co-browse session.

downgradeMode()

This method immediately switches the current session from Write Mode to Pointer Mode. The built-in Co-browse UI executes this method when an end user clicks "Revoke Control" while in Write Mode.

See related signals: onModeUpgradeRequested and onModeChanged.

onInitialized

This signal is dispatched after the page is loaded and the Co-browse business logic is initialized.

Arguments:

- `session`— Session `object` representing the ongoing session or `null` if there is no ongoing session.

Example:

```
cobrowseApi.onInitialized.add(function(session) {
    if (!session) {
        showCobrowseButton();
    } else {
        showCobrowseToolbar(session);
    }
})
```

onSessionStarted

This signal is dispatched when a Co-browse session is successfully started and joined by the customer such as when the `Co-browse` button is pressed or when `startSession()` is called.

Arguments:

- `session`—Session `object` representing the ongoing session.

Example:

```
function notifyCobrowseStarted(session) {
    alert('Co-browse has started. Spell this session token to our representative: ' +
session.token);
}
cobrowseApi.onSessionStarted.add(notifyCobrowseStarted);
```

onAgentJoined

This signal is dispatched when an agent successfully joins a session.

Arguments:

- `agent`—Object representing the new agent. This object has no properties.
- `session`—Session `object` representing the ongoing session.

Example:

```
cobrowseApi.onAgentJoined.add(function(agent, session) {
    alert('Representative has joined the session');
});
```

onAgentNavigated

This signal is dispatched when the **agent** user initiates navigation such as refresh, back, forward, or when the agent enters a URL into the agent Co-browse UI. Signal is dispatched a few seconds before the navigation happens. This can be used to, for example, send a warning to the user or disable the `Exit session` button before navigation.

Arguments:

- `details`—Object containing the following navigation detail fields:
    - `command`—String with the value of `back`, `refresh`, `forward`, or `url`.

- `url`—Optional string that is present only if the `command` field has the value of `url`

Example:

```
// Let's assume we have a "growl" function available to show growl-like notifications
cobrowseApi.onAgentNavigated.add(function(details) {
    if (details.url) {
        growl('Representative navigated to the page: ' + details.url);
    } else {
        growl('Representative has pressed the "' + details.command + '" button. Page will be
refreshed');
    }
});
```

onNavigationFailed

This signal is dispatched when the navigation request from the agent fails to execute such as when the agent navigates forward when there is no forward history. You can use this signal to to re-enable the `Exit` button and/or show a notification.

The callback receives no arguments.

Example:

```
// Let's assume we have a "growl" function available to show growl-like notifications
cobrowseApi.onNavigationFailed.add(function() {
    growl('Navigation request by representative has failed');
});
```

onModeUpgradeRequested

This signal is dispatched when an agent requests upgrading the Co-browse session to Write Mode.

Arguments:

- done—The function passed by the Co-browse code. Call it with `true` to allow the transition to Write Mode, or with `false` to prohibit.

Example:

```
cobrowseApi.onModeUpgradeRequested.add(function(done) {
  if (confirm('Representative requests control over the web page. Allow?') {
    done(true); // allow upgrading to Write Mode
  } else {
    done(false); // disallow and stay in Pointer Mode
  }
});
```

Note: If you're going to implement something similar to the example above, don't forget to disable the built-in UI.

onModeChanged

This signal is dispatched when the Co-browse session Mode changes, either to Pointer or Write.

Arguments:

- `mode`—An object with two boolean properties:
    - `pointer`—This is `true` if the session has switched from Write to Pointer Mode. Otherwise, it's `false`.
    - `write`—This is `true` when the session has switched from Pointer to Write Mode.

Example:

```
cobrowseApi.onModeChanged.add(function(mode) {
    if (mode.write) {
        alert("Representative has now control over the page");
    } else if (mode.pointer) {
        alert("Representative can no longer control the page").
    }
});
```

onSessionEnded

This signal is dispatched when a Co-browse session ends.

Arguments:

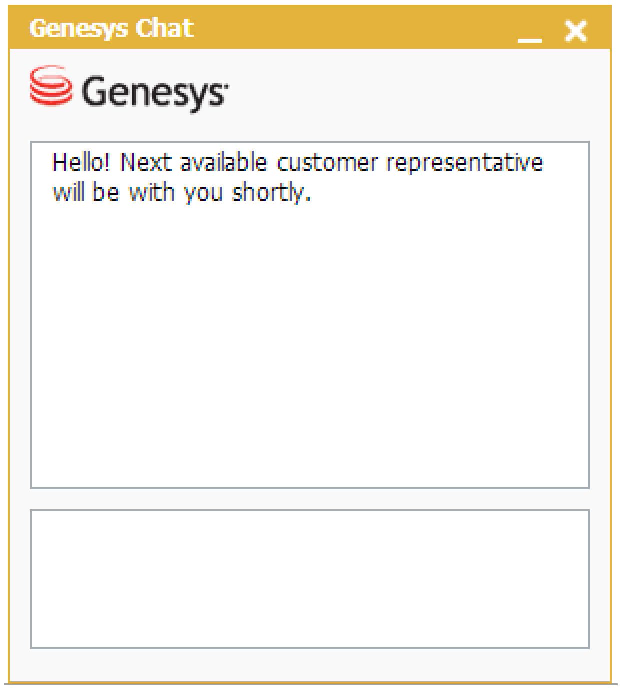
- `details`—Object with the follwing field:
    - `reason`—Field with value of a string or undefined. Possible string values:
        - `self`—The user has exited the session by clicking the `Exit` button or calling the `exitSession()` API method.
        - `external`—The agent has closed the session. Some server errors may also result in this value.
        - `timeout`—The session has timed out such as when a user reopens a page with an expired Co-browse cookie.
        - `intactivityTimeout`—The agent did not join a session in the configured amount of time.
        - `serverUnavailable`—The Co-browse server was unreachable. Added in Genesys Co-browse release 8.5.001.xx.
        - `sessionsOverLimit`—Agent is busy with another co-browse session and is prohibited from starting another session at the same time, see One-Session Agent Limitation. Added in Genesys Co-browse release 8.5.003.04.
        - `error`—There is an error such as a critical misconfiguration.

Example:

```
var cbEndedMessages = {
    self: 'You exited Co-browse session. Co-browse terminated',
    external: 'Co-browse session ended',
    timeout: 'Co-browse session timed out',
    inactivityTimeout: 'Agent did not join. Closing Co-browse session.',
    serverUnavailable: 'Could not reach Co-browse server',
    sessionsOverLimit: 'Agent is busy in another Co-browse session'
}
cobrowseApi.onSessionEnded.add(function(details) {
    alert(cbEndedMessages[details.reason] || 'Something went wrong. Co-browse terminated.');
    showCobrowseButton();
});
```

Chat API

Co-browse is shipped with a built-in chat widget. Out-of-the-box, the chat widget looks like this:



Hello! Next available customer representative will be with you shortly.

To configure the chat widget, see Configuration API.

To get access to the Chat Widget API, see Accessing the Co-browse and Chat APIs. You generally will not need to access the Chat Widget API as configuration can be done in instrumentation. The Chat Widget API can be used to get access to the lower lever Chat API. See Advanced Usage below for more details.

For a full Chat Widget API reference, see Chat Widget JS API.

Advanced Usage of the Chat API

The Chat Widget API is built on top of the Chat Service JS API. The Chat Service API can be used to send chat commands to the server, for example, *send a message* or *leave session*. The Chat Service API also lets you subscribe to session events such as `agentConnected` and `messageReceived`.

Getting Access to the Chat Service API

There are two ways to get access to the Chat Service API:

- Accessing the Chat Service API of the Built-In Chat Widget
- Use the JavaScript Bundle

Accessing the Chat Service API of the Built-In Chat Widget

The following code example shows how you can access the Chat Service API. Note that this example is a bit simplistic in that it starts chat unconditionally on every page load and does not handle errors.

```
var _genesys = {
    chat: {
        // 1. Tell Co-browse JS not to call restoreChat(),
        //    because you will call it manually.
        autoRestore: false,
        // 2. Subscribe to chat widget's "ready" event
        //    to get access to widget API.
        onReady: function(chat) {
            // 3. Use chat widget API to get access to service API.
            chat.onSession(function(session) {
                // Use chat service API here. For example,
                // session.sendMessage('Hello World!');
            })
        }
    }
}
```

Accessing the Chat Service API using the JavaScript Bundle

You can use the JavaScript bundle shipped with Co-browse to access the Chat Service JS API. This file is available at the following URL:

`http(s)://<COBROWSE_SERVER>/cobrowse/js/chatAPI.min.js`

When loaded in a browser, this file exports the Chat Service JS API as a global `chat` variable. The size of this file is 113 KB (~35 KB gzipped) and it does not require any dependencies.

Another version of this file is available at `http(s)://<COBROWSE_SERVER>/cobrowse/js/chatAPI-noDeps.min.js`. The size of this file is 23 KB (~8 KB gzipped), but it requires that you have the following libraries globally available:

- $ for jQuery (v. 1.8.1 or higher)
- _ for underscore (v. 1.5.0 or higher) or lodash (v. 2.0.0 or higher)
- `org.cometd` for Cometd (v. 2.8.0)

Important

If you choose to implement your own chat widget using the Chat Service JS API in the form of a seperate JS file, your chat widget will *not* be automatically integrated with

Co-browse. Integration consists of two features:

- Co-browse automatically determines if the user is on chat when the user starts a Co-browse session.
- The Co-browse session token is automatically passed to an agent.

To support these integration features, you will also have to implement the External Media Adapter API for your chat widget and pass the implementation object to the Configuration API `primaryMedia` option.

External Media Adapter API

The External Media Adapter API can be used to substitute the built-in Co-browse chat with another external media.

Important

If you're not using the built-in Chat, you probably will want to disable it. You can do this using the Configuration API, as shown in the following example:

```
<script>
var _genesys = {
        chat: false
};
</script>
```

This will disable the Chat widget, as well as the **Live Chat** button.

An external media can be connected to Co-browse via an adapter. An adapter is a JavaScript object that is assigned to the `_genesys.cobrowse.primaryMedia` option and implements the specified interface. An external media adapter may implement the following methods:

initializeAsync(done)

Implement the `initializeAsync` method in your external media adapter when the external media initializes asynchronously and you cannot be sure the external media is ready as it is passed to the instrumentation. This method may start the (asynchronous) external media's initialization or it may subscribe to the initialization if the external media is started elsewhere.

If the `initializeAsync` method is implemented, the Co-browse JavaScript will call the method and pass it a done callback. You must call the done callback when your media finishes initialization.

Important

Note the following about the `initializeAsync` method:

- This method is called by the Co-browse JavaScript Application every time it initializes such as after every page load.
- This method is called **before** the `Live Chat` and `Co-browsing` buttons are shown. The buttons will be shown only after you call the passed done callback in your code.

The following is an example of an external chat adapter named `MyChatAdapter`:

```
function MyChatAdapter() {
    // initialize chat
    this.initializeAsync = function(done) {
        $.get('/chat/configuration', function(config) {
            var chat = new MyChat(config);
            // tell cobrowse chat is ready
            done();
        });
    };
};

// or if you have a chat with event-based API that is initiated elsewhere
function MyChatAdapter() {
    this.initializeAsync = function(done) {
        myChat.on('initialized', done);
    };
};
```

Tip

You can use the `initializeAsync` method to restore your external media after a page reload. For example, if you have a chat with a `restoreChat` function that needs to be called after page reload, you can call this `restoreChat` method in the `initializeAsync` method of the external media adapter passed to Co-browse.

Example:

```
// in the adapter:
myChatAdapter.initializeAsync = function(done) {
    myChat.restoreChat().then(done);
};

// ...
// and then in Co-browse instrumentation
var _genesys = {
    cobrowse: {
        primaryMedia: myChatAdapter
    }
};

// Now after every page reload Co-browse will
//  automatically restore your chat.
```

sendCbSessionToken(token)

Implement this method to configure the external media channel to pass the Co-browse session token to the agent. The Co-browse session token is a string consisting of 9 digits.

Example:

```
myChatAdapter = {
    sendCbSessionToken: function(sessionToken) {
        myChat.sendMessage('User has started Co-browse session: ' + sessionToken);
    }
};
```

Tip

You may customize how the Co-browse token is passed to the agent. If you use a Genesys agent desktop, such Interaction Workspace with the Co-browse plugin or Workspace Web Edition, you may want the agent to join a Co-browse session as soon as he or she receives the token. To do so, wrap the Co-browse token in a `{start:<TOKEN>}` message.

Example:

```
// For example:
myChatAdapter.sendCbSessionToken = function(token) {
  myChat.sendMessage('{start:' + token + '}');
};
```

isAgentConnected()

Important

This method must return a boolean.

The integration module checks the return value of this method before calling the `sendCbSessionToken` method. If `isAgentConnected` returns `false`, the user will be asked to connect with an agent via phone before starting Co-browse. If the `isAgentConnected` is absent, the user will be asked to connect with an agent via phone or chat before starting Co-browse.