API Reference

Chat Service JS API

4/30/2025

Contents

Chat Service JS API

Deprecation notice

- **Starting with the 8.5.000.38 release of Genesys Web Engagement, Genesys is deprecating the Native Chat and Callback Widgets—and the associated APIs (the Common Component Library)—in preparation for discontinuing them.**

  This functionality is now available through a single set of consumer-facing digital channel APIs that are part of Genesys Mobile Services (GMS), and through Genesys Widgets, a set of productized widgets that are optimized for use with desktop and mobile web clients, and which are based on the GMS APIs.

  Genesys Widgets provide for an easy integration with Web Engagement, allowing you to proactively serve these widgets to your web-based customers.

  > ### Important
  > Although the deprecated APIs and widgets will be supported for the life of the 8.5 release of Web Engagement, Genesys recommends that you move as soon as you can to the new APIs and to Genesys Widgets to ensure that your functionality is not affected when you migrate to the 9.0 release.
  >
  > - Note that this support for the Native Chat and Callback Widgets and the associated APIs will not include the addition of new features and that bug fixes will be limited to those that affect critical functionality.

Use the Genesys Chat Service JS API to create your own chat widget and modify all the parameters used to control chat sessions.

> ### Tip
> You can use Genesys Chat Widget if you are not building your own widget user interface from scratch. This widget is highly customizable and provides access to this API as well. See Chat Widget JS API.

How The API Is Structured

The Chat Service JS API is divided into two parts:

- Launching the chat session — Describes a set of commands and callbacks for creating a chat session.
- Controlling the chat session — Describes a set of commands and callbacks for manipulating an ongoing chat session (sending messages, receiving updates, and so on).

Example of how to start a chat session

Take a look at how you can start a chat session in this example:

```
// Example assumes API is publicly exported to chat global variable

function handleSession(session) {
    // Add callbacks, for example:
    // session.onAgentConnected(function(event) {...});
    // session.onMessageReceived(function(event) {...});
    // ...
    // Use commands, for example:
    // session.sendTyping();
    // session.sendMessage('Hi there');
}

chat.restoreSession()
    .done(handleSession)
    .fail(function (event) {
        if (event.error) {
            // Session has been started on previous page, but failed to restore.
            //  event.error.code and event.error.description may contain some information.
        } else {
            // If there is no session to continue, start a new chat session.
            chat.startChat({
                serverUrl: 'http://www.example.com/server/cometd/'
            })
            .done(handleSession)
            .fail(function (event) {
                // See event.error.code and event.error.description for details of the failure
            });
        }
    });
```

The **session** object (initiated by `startSession()` or `restoreSession()`) provides a set of **commands** and **callbacks**. The commands (which can be used to specify callback functions for successful or unsuccessful flows) are used to send messages to the server side. The set of callbacks handle the messages sent back from the server.

Using a third-party mechanism for chat session state persistence

If for whatever reason you don't want to use the default chat state persistence mechanism, you can provide your own.

Here is an example of how that might work:

```
// For example (abstract):
myStateStorage = {
    write: function(state) {
                // save state here
        setCookie('chatSessionState', JSON.stringify(state));
```

```
    },
    read: function() {
                // return state here
        return JSON.parse($.cookie('chatSessionState'));
    }
}
chat.startSession({
    serverUrl: '...',
    stateStorage: myStateStorage
});
```

Extended API to support integrated solutions

When using the chat session to send extra information needed for integrated solutions, it is helpful to have a separate mechanism for exchanging notifications, outside of the chat session transcript. The Extended API provides this mechanism, as well as a way of accessing interaction User Data.

> ### Tip
> Only data that was set by this application (or was set specially for this application) will be accessible.

Use Cases

1. You need to pass `CoBrowseSessionId` parameter from the chat widget (browser side) to the agent's party (IWS plugin) in order to automatically start co-browse session.
2. You need to pass information from the routing strategy URL for an "after-chat" survey.

Here is a description of the extended API:

```
// Extended API is available through session object
session.setUserData(userData)
    .done(function (event) { /* */ })
    .fail(function (event) { /* event.error.code, event.error.description */ });
session.getUserData(userDataKey)
    .done(function (event) { /* event.userData */ })
    .fail(function (event) { /* event.error.code, event.error.description */ });
session.deleteUserData(userDataKey)
    .done(function (event) { /* event.timestamp */ })
    .fail(function (event) { /* event.error.code, event.error.description */ });
```

Let's look at the details of this extended API.

session.setUserData

Provides a way to specify a list of key-value pairs that you can use as UserData in chat interaction.

Use-case

1. During chat session, you need to transmit data to the agent's party. For example, the ID of a

conversation running in a parallel media (Co-browse, WebRTC and so on) This method accepts a single argument: an object with key-value pairs for setting the UserData. This method returns a "promise" object similar to all other session methods.

For example,

```
session.setUserData({
    FirstName: 'John',
    LastName: 'Doe',
    MySpecialProperty: 'foobar'
}).done(function() {
    // Data correctly set
}).fail(function(event) {
    // Examine event to find out information on what went wrong
});
```

session.getUserData

Provides a way to read specified keys from User Data.

> **Important**
>
> The client's code can only obtain data that was specified by this code previously, or data that was added to the interaction *specifically* for this code (for example, this data can be attached by a related routing strategy).

Accepts a single argument: the key as a string.

Returns a "promise" object similar to all other session methods.

"done" callback

For this method, the callback provides a way to access the required data: it is resolved with an event object that contains the UserData property:

```
session.getUserData('FirstName').done(function(event) {
    // event.userData.FirstName is the required value
}).fail(function(event) {
    // See event.error.code, event.error.description
})
```

session.deleteUserData

Provides possibility to remove specified keys from User Data.

Accepts a single argument: the key as a string.

```
session.deleteUserData('FirstName').done(function(event) {
    // data successfully deleted
}).fail(function(event) {
    // Something went wrong. See event.error.code, event.error.description
});
```