API Reference

Genesys Web Engagement 8.5.1

1/6/2022

Table of Contents

Genesys Web Engagement API Reference

Welcome to the *Genesys Web Engagement 8.5.1 API Reference*. This document provides you with the information you need to use the Genesys Web Engagement REST and Javascript APIs. See the summary of chapters below.

Monitoring JS API

Use this JavaScript API to submit events to the Genesys Web Engagement Server.

Monitoring JS API

Chat JS API

Use this JavaScript API to control all aspects of the chat session.

Chat Service JS API

Chat Widget JS API

Notification Service REST API

Use this REST API to send notifications to your web pages.

Notification Service REST API

Engagement REST API

Use this REST API to start or cancel an engagement attempt.

Engagement REST API

History REST API

Use this REST API to find the web history stored by the Genesys Web Engagement Server.

Pacing REST API

Use this REST API to access pacing information.

History REST API

Pacing REST API

Business Events DSL

Use this information to define business events in the DSL.

Business Events DSL

Monitoring JS API

Description

You can use the Monitoring JS API in your web pages to send events (read more about how these events are structured here) to the Genesys Web Engagement Server. This is independent from the set of events and conditions that are defined in the DSL files that are loaded by the browser's Monitoring Agents. The design model for the Monitoring JS API is highly flexible, and its use can be extended well beyond the common model of user-triggered events — the design decision is up to you. You can submit `UserInfo`, `SignIn`, `SignOut`, and your own custom business events using this API.

Important

All commands and options are **case sensitive**.

The entry point for this API is the global `_gt` (Genesys Tracker) object which implements the `push()` method. The `_gt.push()` method takes an array as a parameter, which can contain any type of information, so longs as it follows this format:

```
_gt.push(['<commandName>', <options>])
```

- `<commandName>` — name of the event command.
- `<options>` — options for the command.

_gt.push(['event', eventName, { data: options }])

Sends business events to the server.

Parameters

Parameter name	Type	Mandatory	Description
eventName	string	yes	The name of the event. This field is equivalent to the name attribute of the DSL <event> element.
options	object	no	An object of any properties you want to

Parameter name	Type	Mandatory	Description
			track with the event.

Example with mandatory parameters

```
_gt.push(['event', 'AddToCart']);
```

Example with additional parameters

```
_gt.push(['event', 'AddToCart', { data: {
    productName : 'Sony',
    productModel: 'JVB72',
    productPrice: '1000',
    productCurrency:'USD'
}}]);
```

_gt.push(['event', 'UserInfo', { data: options }])

The Tracker application relies on your website to trigger transitions between visitor states (See Visitor Identification). This method sends a **UserInfo** system event that includes customer information. You should only send this event if your website has authenticated the user. You should also send this event when an authenticated session has been ended and the user returns to the website. For more information, see Recognized Visitors.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	yes	The user's ID. For instance, the user account name or email address.
<customParameter>	boolean / number / string / object	no	An object containing additional key/value pairs for sending with the event.

Example with mandatory parameters

```
_gt.push(['event', 'UserInfo', { data: {
    userID: 'user@genesyslab.com'
}}]);
```

Example with additional parameters

```
_gt.push(['event', 'UserInfo', { data: {
    userID: 'user@genesyslab.com',
    name:   'Bob',
    sex:    'male',
    age:    30
}}]);
```

_gt.push(['event', 'SignIn', { data: options }])

Creates and sends the `SignIn` event, which allows the system to identify the user. Send this event when the user is authenticated by your website.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	yes	The user's ID. For instance, the user account name or email address.
<customParameter>	boolean / number / string / object	no	An object containing additional key/value pairs for sending with the event.

Example with mandatory parameters

```
_gt.push(['event', 'SignIn', { data: {
    userID: 'user@genesyslab.com'
}}]);
```

Example with additional parameters

```
_gt.push(['event', 'SignIn', { data: {
    userID: 'user@genesyslab.com',
    name:   'Bob',
    sex:    'male',
    age:    30
}}]);
```

_gt.push(['event', 'SignOut', { data: options }])

Creates and sends the `SignOut` event for the current user. This event should be sent at the time the user logs out from your website or as soon as possible after logout.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	yes	The user's ID. For instance, the user account name or email address. **Note:** Genesys recommends that you use this parameter, even though it is not mandatory.

Example without parameters

```
_gt.push(['event', 'SignOut']);
```

Example with additional parameters

```
_gt.push(['event', 'SignOut', { data: {
    userID: 'user@genesyslab.com'
}}]);
```

_gt.push(['event', 'PageEntered', { data: options }])

Creates and sends the PageEntered event. The Tracker application sends a PageEntered event automatically when a new page is loaded. This method should only be used with a Single Page Application.

Notes

- The `PageEntered` event updates the current `pageID`.
- Make sure that a `PageExited` event is sent before sending a `PageEntered` event.
- The `PageEntered` event does not affect the DSL event sequence defined on initial page load. It also does not lead to categorization or to reinitialization of the DSL.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents PageEntered event information.

Possible key/value pairs in "options"

Parameter name	Type	Default Value	Description
title	string	document.title	The title of the current page.

Example without parameters

```
_gt.push(['event', 'PageEntered']);
```

Example with additional parameters

```
_gt.push(['event', 'PageEntered', {
    data: {
        title: 'My Page Title'
    }
}]);
```

_gt.push(['event', 'PageExited'])

Creates and sends the PageExited event. The Tracker application sends a PageExited event automatically on page unload. This method should only be used with a Single Page Application.

Example without parameters

```
_gt.push(['event', 'PageExited']);
```

_gt.push(['event', options]) (Deprecated)

Important
Deprecated in 8.5

Sends business events to the Web Engagement Server.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represent event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
eventName	string	yes	The identification name of the business event. This field is equivalent to the name attribute of the DSL <event> element.
<customParameter>	object	no	An object of additional key/value pairs to send along with the event.

Example with mandatory parameters

```
_gt.push(['event', {
        eventName: 'AddToCart'
}])
```

Example with additional parameters

```
_gt.push(['event', { eventName: 'AddToCart',
        productName : 'Sony',
        productModel:  'JVB72',
        productPrice:  '1000$'
}])
```

_gt.push(['sendUserInfo', options]) (Deprecated)

Important
Deprecated in 8.5

Genesys Web Engagement relies on your website to trigger the transitions between visitor states (see Visitor Identification).

This event sends the system "UserInfo" event with customer information. You should send this event when a user visits your website after closing the browser window on an authenticated session. For details, see Recognized Visitors.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	yes	The identification ID for the user. For instance, the user account name or the email address.
<customParameter>	object	no	An object of additional key/value pairs to send along with event.

Example with mandatory parameters

```
_gt.push(['sendUserInfo',  {
        userID: 'user@genesyslab.com'
}])
```

Example with additional parameters

```
_gt.push(['sendUserInfo',  {
        userID: 'user@genesyslab.com',
        name:   'Bob',
        sex:    'male',
        age:    30
}])
```

_gt.push(['sendSignIn', options]) (Deprecated)

Important
Deprecated in 8.5

This event creates and sends the system "SignIn" event. Send this event when the user is authenticated by the website. This allows the system to identify the user and creates a new "session" with a `sessionID` that is unique to a visit and will last the duration of the visit. Only Authenticated visitors have an associated sessionId.

Parameters

Parameter name	Type	Mandatory	Description
options	object	yes	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	yes	The identification ID for the user. For instance, the user account name or the email address.
<customParameter>	object	no	An object of additional key/value pairs to send along with event.

Example with mandatory parameters

```
_gt.push(['sendSignIn',  {
        userID: 'user@genesyslab.com'
}])
```

Example with additional parameters

```
_gt.push(['sendSignIn',  {
        userID: 'user@genesyslab.com',
        name:   'Bob',
        sex:    'male',
        age:    30
}]);
```

_gt.push(['sendSignOut', options]) (Deprecated)

Important
Deprecated in 8.5

This event creates and sends the "SignOut" system event for the current user. This event should be sent if the user performs a logout on the website or as soon as possible after the logout action was done. **Note:** The `sessionId` lasts for the duration of the authenticated user's visit to your website. It is stored in a cookie and sent with every event that occurs between "SignIn" and "SignOut", and is changed automatically after every "SignIn" event.

Parameters

Parameter name	Type	Mandatory	Description
options	object	no	A set of key/value pairs that represents event information.

Possible key/value pairs in "options"

Parameter name	Type	Mandatory	Description
userID	string	no	The identification ID for the user. For instance, the user account name or the email address. **Note:** Genesys recommends using this parameter even though it is not mandatory.
<customParameter>	object	no	An object of additional key/value pairs to send

Parameter name	Type	Mandatory	Description
			along with event.

Example with no parameter

```
_gt.push(['sendSignOut'])
```

Example with additional parameters

```
_gt.push(['sendSignOut', {
        userID: 'user@genesyslab.com'
}])
```

_gt.push(['getIDs', callback])

Gets visit identification information from the Tracker application. The callback contains an object with the `visitID`, `globalVisitID`, `pageID`, and `alias` fields.

Important
The **alias** field is deprecated in 8.5 and will be removed in 9.0

Parameters

Parameter name	Type	Mandatory	Description
callback	function(IDs)	yes	A function that is called if the request succeeds. The function is passed one argument.

"IDs" parameter

Parameter name	Type	Mandatory	Description
IDs	object { `globalVisitID`, `visitID`, `pageID`, `alias` }	yes	An object that contains visit identification information.

Example

```
_gt.push(['getIDs', function(IDs) {
     console.log('IDs: ', IDs);
}])
```

_gt.push(['getConfig', callback])

Gets configuration information from the Tracker application. The callback contains an object with the `visitID`, `httpEndpoint`, `httpsEndpoint`, and other fields.

Parameters

Parameter name	Type	Mandatory	Description
callback	function(config)	yes	A function that is called if the request succeeds. A single argument is passed to the function.

"config" parameter

Parameter name	Type	Mandatory	Description
IDs	object { `httpEndpoint`, `httpsEndpoint`, ... }	yes	An object that contains configuration information.

Example

```
_gt.push(['getConfig', function(config) {
    console.log('config: ', config);
    console.log('httpEndpoint: ' + config.httpEndpoint);
    console.log('httpsEndpoint: ' + config.httpsEndpoint);
}]);
```

Events

You can use handler functions to register behaviors to take effect when the Tracker application is generating events, and to further manipulate those registered behaviors.

Here is how to bind a handler function to a Tracker. The handler will be invoked whenever the event is fired:

```
_gt.push(['on', eventName, handler]);
```

To remove a previously-bound handler function from an object:

```
_gt.push(['off', eventName, handler]);
```

beforeSendEvent

`_gt.push(['on', 'beforeSendEvent', handler])`

`_gt.push(['off', 'beforeSendEvent', handler])`

The `beforeSendEvent` event is triggered before sending an event to the server.

Handler Arguments

Argument name	Type	Description
event	object	A set of key/value pairs that represents event information.

Example

```
var myEventHandler = function (event) {
    if (event.eventName === 'Search') {
        // Send event to Google Analytics service
        ga('send', 'event', 'search', event.eventName, event.data);
    }
};

// subscribe
_gt.push(['on', 'beforeSendEvent', myEventHandler]);
// unsubscribe
_gt.push(['off', 'beforeSendEvent', myEventHandler]);
```

`myEventHandler` is a function that you can execute each time the event is triggered but before it is sent to the server.

notificationMessage

`_gt.push(['on', 'notificationMessage', handler]);`

`_gt.push(['off', 'notificationMessage', handler]);`

The `notificationMessage` event is triggered when a message is received from the server.

Handler Arguments

Argument name	Type	Description
message	object	A set of key/value pairs that represents one notification message.

Example

The following example shows how to subscribe to the default message for starting a chat:

```
var notificationMessageHandler = function (message) {
    if (message.channel === 'gpe.setVariable' && message.data.value.type === 'chat') {
        alert('Chat from notificationMessageHandler');
    }
};

_gt.push(['on', 'notificationMessage', notificationMessageHandler]);
```

How To — Enable a trigger after another trigger

DSL is a great tool to create business events on your website without requiring programming knowledge, but it's not a JavaScript representation in XML. There are some use cases when DSL is not enough; instead, you should use the Monitoring JS API.

Let's look at this example use case:

You have a web page with a text field and a submit button. If a user starts typing in the text field and, for example, 100 seconds pass with no "submit" — you want to make sure that is reported to Web Engagement.

You can implement the functionality that's described in the use case with the following approach:

```
...
<p><input class="comment" type="text"></p>
<p><input class="submit" type="button" value="submit"></p>

<script>
    var timeout;

    $('.comment').focus(function() {
        if (!timeout) {
            console.log('timer started');
            timeout = setTimeout(function () {
                console.log('send event');
                _gt.push(['event', {eventName: 'myEvent'}])
            }, 100 * 1000)
        }
    });

    $('.submit').click(function() {
        if (timeout) {
            console.log('clean timeout');
            window.clearTimeout(timeout);
            timeout = undefined;
        }
    });
</script>
...
```

Tracking Single Page Applications

A Single Page Application (SPA) is a web application or website that loads all of the resources required to navigate throughout an entire site when the first page on that site is loaded.

As the user clicks links and interacts in other ways with the page, any new content is loaded dynamically. The application may update the URL in the address bar to emulate traditional page navigation, but it never requests another full page.

By default, the Tracker Application works well with traditional websites, because it sends a **PageEntered** event every single time the user loads a new page. However, for an SPA where you're loading new page content dynamically rather than as full page loads, the Tracker script only sends the first **PageEntered** event—because all subsequent page entries are *virtual*. This means you have to track these subsequent (virtual) **PageEntered** events manually, as each piece of new content is loaded.

To track dynamically loaded content as a distinct page, you can use the Tracker API to send a **PageEntered** event. To do this, you need to specify the URL and title, as shown here:

```
_gt.push(['event', 'PageEntered', {
    url: 'http://example.com/my-page-url?id=1',
    data: {
        title: 'My Page Title'
    }
}]);
```

If you specify a URL value in a **PageEntered** event, the app will only send that URL value to the server—it will not update the URL value stored in the Tracker application itself.

This means that if you send other events and don't explicitly include the current URL value, the Tracker application will associate those events with whatever URL was stored at the time of the initial page load.

To avoid this issue, it's usually best to update the Tracker app configuration data with the URL and page title from a newly loaded "page" before you send any other events for that "page." This will ensure that these events are associated with the correct page data.

To update the Tracker configuration, use the **config** command:

```
_gt.push(['config', {
    page: {
        url: 'http://example.com/my-page-url?id=1',
        title: 'My Page Title'
    }
}]);
```

Once the Tracker configuration has been updated with the proper data for the new "page," you can send a **PageEntered** event without overriding page-related parameters. For example:

```
_gt.push(['config', {
    page: {
        url: 'http://example.com/my-page-url?id=1',
        title: 'My Page Title'
    }
}]);
```

```
_gt.push(['event', 'PageEntered']);
/*
    event.url - 'http://example.com/my-page-url?id=1'
    event.data.title - 'My Page Title'
 */
```

Here is a more complex use case:

```
_gt.push(['config', {
    page: {
        url: 'http://example.com/my-page-url?id=1',
        title: 'My Page Title'
    }
}]);

_gt.push(['event', 'PageEntered']);
/*
    event.url - 'http://example.com/my-page-url?id=1'
    event.data.title - 'My Page Title'
 */

_gt.push(['event', 'PageEntered', {
    url: 'http://example.com/new-url'
}]);
/*
    event.url - 'http://example.com/new-url'
    event.data.title - 'My Page Title'
 */

_gt.push(['event', 'MyCustomEvent']);
/*
    event.url - 'http://example.com/my-page-url?id=1'
 */
```

Note: The second PageEntered event and all subsequent PageEntered events in a single-page application do not reset the timer for timeout events defined in the DSL.

Chat JS API

The Genesys Chat API lets you control all aspects of the chat session, using either the Genesys Chat Widget, or creating a chat widget of your own:

- Chat Widget JS API — Use this API to implement or customize the Genesys Chat Widget.
- Chat Service JS API — Use this API to control the chat session or create your own chat widget.

Chat Widget 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.

How the API is Structured

The Chat Widget JS API is made up of the three main methods:

- startChat—Use this method as the entry point for starting a chat session.
- restoreChat—Use this method to restore the chat widget after page reload when working in "embedded" mode (the chat window is rendered on a website's page, not in a separate brower window)
- startChatInThisWindow—Use this method for chat sessions appearing in a separate dedicated HTML page. For cases where chat is started in "popup" mode (chat is rendered in a separate browser window).

Additional methods are also available since chat version 850.6.0, see Additional Methods.

Browser Support and Cookies

Browser Support

Chat widget fully supports IE8+, Firefox, Safari and Chrome desktop browsers. On mobile, Safari on iOS 7 and Google Chrome on latest Android are officially supported, though chat may properly function in other browsers / OSes. Notes on IE7 support:

- Chat widget depends on JSON parsing functionality. To enable it in IE7, you have to use a third-party library (for example, json2.js by JSON creator Douglas Crockford). Note that this kind of library might be already included if you use chat as part of another product (like Web Engagement).
- Built-in UI uses data:uri technology for images, which is not supported by IE7. This is why the minimize and close buttons and the Genesys logo are not visible in this browser. You have to customize the CSS if you need chat support in IE7.

Note on cookies

The Chat Widget uses a few cookies that allow it to function properly across multiple pages / browser windows. Also, to function properly across different subdomains, chat widget writes cookies to the **"root" domain of your site**. For example, if your site's domain is `www.example.com.au`, the cookies are written to the `.example.com.au` domain, which makes them available to all other subdomains of your site.

Getting Access to Chat Session API

Embedded Mode

Getting access to the Chat Session API in embedded mode

If you want to ensure consistent access to theChat Session API in "embedded" mode, you need to accommodate two done callbacks: one for `startChat` and `restoreChat`.

For example,

```
function handleChatSession(session) {
    // Use session API here to send messages, subscribe to events etc.
}

chat.restoreChat(options).done(handleSession)
    .fail(function() {
        chat.startChat(options).done(handleSession);
    });
// Example is artificial in that it starts new chat unconditionally, if it has not been
started yet.
```

```
// See documentation below for more elaborate examples of working with chat start/restoration.
```

Popup Mode

Getting access to the Chat Session API in popup mode

To get access to the Chat Session API in "popup" mode, you have to accommodate the done callback for the `startChatInThisWindow` method in the separate HTML page for the widget.

For example,

myChatWidget.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Genesys Chat</title>
</head>
<body>
<script src="js/chatWidget.js"></script>
<script>
  chat.startChatInThisWindow().done(function(session) {
    // Use session API here
  });
</script>
</body>
</html>
```

...and then pass the URL of this HTML page to the `startChat` method on your website's page:

```
chat.startChat({
    embedded: false,
    widgetUrl: 'http://example.com/myChatWidget.html',
    // ...
});
```

Customizing the User Interface

There are different options for customizing the chat widget UI — from style modifications via CSS to complete disabling, with JavaScript-based customization in between.

Template-based

Template-based Customization

> **Important**
>
> Make sure to read about the templates option first.

You can use a template-based customization approach, which boils down to editing plain HTML, if any of the following are true:

- You want to modify the structure of the widget.
- You want to add content or CSS classes to the widget.
- You do not want to use JavaScript to work with the DOM.

The basic algorithm for template-based customization is to do the following:

1. Get the HTML with default templates. You can get the default templates from:
    - Web Engagement Server 8.5.1 at `http(s)://<WE_HOST>[:<WE_PORT>]/server/api/resources/v1/chatTemplates.html`.
    - Web Engagement Server 8.5 at `http(s)://<WE_HOST>[:<WE_PORT>]/server/resources/chatTemplates.html`.
    - Web Engagement Frontend Server 8.1.3 at `http(s)://<FRONTEND_HOST>[:<FRONTEND_PORT>]/frontend/resources/chatTemplates.html`.
2. Modify the templates.

    > **Tip**
    >
    > All localization data is available inside your templates as a `data.nls` object.

    > **Important**
    >
    > To keep all functionality intact, Genesys recommends you do not remove elements or CSS classes when editing the chat templates.

3. Insert the modified template(s) into your page's HTML code somewhere between the <body> and </body> tags.

    Example HTML page:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Page</title>
  [GENESYS INSTRUMENTATION]
</head>
<body>
<h1>This is my page.</h1>
<p>It has some content.</p>

<script type="text/html" data-gwc-template="embeddedWindow">
  [MODIFIED TEMPLATE CONTENT]
</script>
<script type="text/html" data-gwc-template="chatMessage">
  [MODIFIED TEMPLATE CONTENT]
</script>
</body>
```

Starting with chat versions **850.6.0**, chat automatically recognizes and uses the template from the `data-gwc-template` attribute. If using an earlier version, see the next step.

4. Another option is to host the modified HTML templates somewhere accessible using HTTP. If you choose to download templates over the network, configure chat to use downloaded templates by passing the URL of your HTML file to the **templates** option of your chat configuration.

Important

If you use both in-page templates and downloaded templates, the downloaded templates take priority.

Tip

You can use the Web Engagement Server server to host static resources. This is especially useful for resources like chat templates because the server is configured with JSONP support. For details, see Hosting Static Resources - JSONP.

Chat Widget Template System

The chat widget templates are included in the **chatTemplates.html** file. This file is not a static HTML file, but a collection of small client-side templates wrapped in `<script>`tags. To learn more about this templating technique, see http://en.wikipedia.org/wiki/JavaScript_templating. The chat widget template system is based on the popular lodash/underscore templates: http://lodash.com/docs#template, http://underscorejs.org/#template.

The chat widget template uses 5 HTML-based templates:

- **chatRegistration** for the built-in registration form
- **chatTyping** for the "Agent is typing" message

- **chatMessage** for all other messages in a chat
- **chatView** for the basic structure of the chat widget
- **embeddedWindow** for the "chat window" wrapper in "embedded" mode

These templates are included in the **chatTemplates.html** file, which has the following structure:

```
<!-- embeddedWindow.html -->
<script type="text/html" data-gwc-template="embeddedWindow">
    ...
</script>

<!-- chatRegistration.html -->
<script type="text/html" data-gwc-template="chatRegistration">
    ...
</script>

<!-- chatView.html -->
<script type="text/html" data-gwc-template="chatView">
    ...
</script>

<!-- chatMessage.html -->
<script type="text/html" data-gwc-template="chatMessage">
    ...
</script>

<!-- chatTyping.html -->
<script type="text/html" data-gwc-template="chatTyping">
    ...
</script>
```

Important

To keep all functionality intact, Genesys recommends that you do not remove elements or CSS classes when editing the chat templates.

Customization Examples

- Substituting the Genesys Logo with a Custom Image
- Adding Extra Content to the Chat Widget
- Displaying a character counter in the message area
- Replace **Skip Registration** button with **Exit** button
- Automatically expand text area based on user input

CSS-based

CSS-based Customization

The chat widget JavaScript contains all of the CSS needed to render chat, which is automatically added to the <head> section of the web page when chat is initialized.

You can override any of the default styles by adding a `<link>` or `<style>` tag with CSS rules for the chat widget (since the default CSS is added to the beginning of <head>, your custom styles will always take higher priority).

Tip

To see exactly what you can override, use the developer tools native to most web browsers. Currently, we do not document our CSS and do not guarantee backwards compatibility for future versions of chat.

Important

To avoid potential CSS conflicts, all of chat widget's class names are prefixed with "gwc-".

Customization Examples

- Modifying the Styling of the Chat Messages
- Adding Extra Content to the Chat Widget

JavaScript-based

JS-based Customization

In case you want to modify more than just the style (look and feel) of the chat, but also the logic of the widget, you can use "hooks" for customizing the built-in UI. See the ui option for details.

Most of the time, you can make additions or modifications to the chat widget layout using the template-based customization. You might choose JavaScript-based customization over template-based if:

- The customization is small enough so that the DOM-related work can be easily done with JavaScript (provided you are experienced with JavaScript).
- The customization impacts not only the layout, but also the logic (for example, if you need additional event handlers).

Complete override

It is possible to disable the built-in UI and implement your own based on the session API:

```
startChat({
  //...
  ui: false,
}).done(function(session, options) {
  // Implement your own UI using session API
});
```

Customization Examples

- Using the ui.onBeforeMessage Hook to Add Desktop Modifications
- Displaying a confirmation alert when users close the chat widget
- Implementing a client-side chat session timer
- Inserting a line break with Shift+Enter
- Automatically opening a URL pushed by an agent
- Showing the number of unread messages in a minimized chat widget
- Showing an agent typing notification in the minimized chat widget
- Displaying a character counter in the message area
- Replace **Skip Registration** button with **Exit** button
- Automatically expand text area based on user input

Localization

The Chat Session API (`startChat` and `restoreChat` methods particularly) includes an optional `localization` parameter that accepts one of the following:

- JavaScript object with localization data
- Function that returns an object with localization data
- Function that accepts a callback and calls that callback with an object containing localization data
- URL of and external JSON file with the localization data

Localization data is fetched or passed before chat initialization and merged with the default localization.

Important

For security reasons, in "popup" mode (separate window) data is fetched via AJAX (not JSONP), which means that **the chat widget html file and the localization file must be on the same domain**. Otherwise (or in case of invalid URL) an error is thrown and the chat silently fails.

In "embedded" mode, the localization data is always fetched using JSONP. This lets you host the data on another domain, separate from the site itself. You can use Web Engagement server to host the content for you, as they both support JSONP out-of-the-box. See GWE Architecture—Hosting Static Resources.

```
// This will work: widget and l18n are on same domain: example.com
chat.startChat({
  widgetUrl: 'http://example.com/chatWidget.html',
  localization: 'http://example.com/chatLocalization.json',
  embedded: false
});

// This won't work: they are on different domains
chat.startChat({
  widgetUrl: 'http://example.com/chatWidget.html',
  localization: 'http://another-domain.com/chatLocalization.json',
  embedded: false
});

// This will work: files are on different domains, but chat is started in
embedded mode,
// so JSONP is used (here we use Genesys Web Engagement server for JSONP)
chat.startChat({
  widgetUrl: 'http://example.com/chatWidget.html',
  localization: 'http://<GWE_SERVER>/server/api/resources/v1/
chatLocalization.json',
  embedded: true
});
```

The passed localization object or external JSON file may contain any of the fields listed below. Fields that are not present will be taken from built-in default localization.

Built-in localization

```
{
  "chatTitle": "Genesys Chat",
  "chatWelcome": "Hello! Next available customer representative will be with you shortly.",
  "defaultUsername": "User",
  "defaultAgentName": "Representative",
  "ownUsername": "You",
  "chatEnded": "Chat session ended",
  "agentJoined": "Representative {name} has joined the session",
  "agentLeft": "Representative {name} has left the session",
  "agentTyping": "{agentName} is typing...",
  "serverStopped": "Connection to chat has been interrupted. Trying to restore...",
  "serverUnreachable": "Chat is unavailable now. Please try again later.",
  "networkInterrupted": "Connection to chat has been interrupted. Trying to restore...",
  "networkRestored": "Connection restored.",
  "unknownServerError": "Chat is unavailable now. Please try again later.",
  "chatSessionExpired": "Chat session has expired.",
  "regFirstName": "First Name:",
  "regLastName": "Last Name:",
  "regEmail": "Email:",
  "regSubmit": "Start Chat",
  "regSkip": "Skip Registration",
```

Built-in localization

```
  "regWelcomePart1": "Please enter few details about you, and press Start Chat button.",
  "regWelcomePart2": "Next available customer representative will be with you shortly.",
  "regErrorRequiredField": "\"{field}\" is a required field",
  "regErrorInvalidEmail": "Please enter a valid email address",
  "leaveSessionPrompt": "You are going to leave chat session."
}
```

Procedure: Step-by-step instruction for using an external localization file

Steps

1. Create a localization file with all / some of the fields overridden.
   For example suppose you want to have chat title and welcome message in Russian:

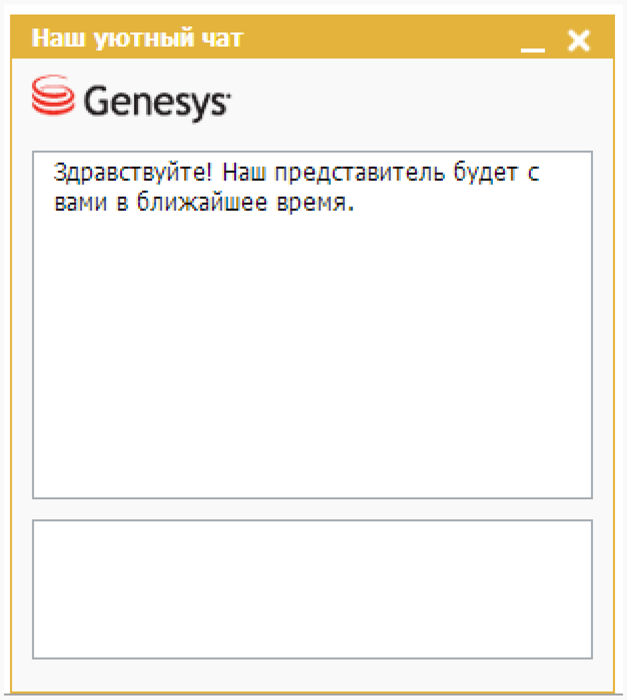
   ```
   {
     "chatTitle": "Наш уютный чат",
     "chatWelcome": "Здравствуйте! Наш представитель будет с вами в ближайшее время."
   }
   ```

   Copy and paste the code above (or craft your own) and save it as `.json` file.
2. We recommend that you check to make sure that the syntax of the JSON file is correct. You can do this online, using this service: http://jsonlint.com/
3. Host the file on any server. If you want to support localization for chat in "embedded" mode, you have to configure JSONP support on the server.

   Important

   Genesys Web Engagement supports serving JSONP. See GWE Architecture—Hosting Static Resources.
4. Use the URL for this file as the value of the `localization` option in `startChat`.

   ```
   chat.startChat({
     localization: 'http://my-server.com/my-chat-localization.json',
     // other parameters
   })
   ```
5. After starting the chat, you should see something like this (sample is in "embedded" mode):



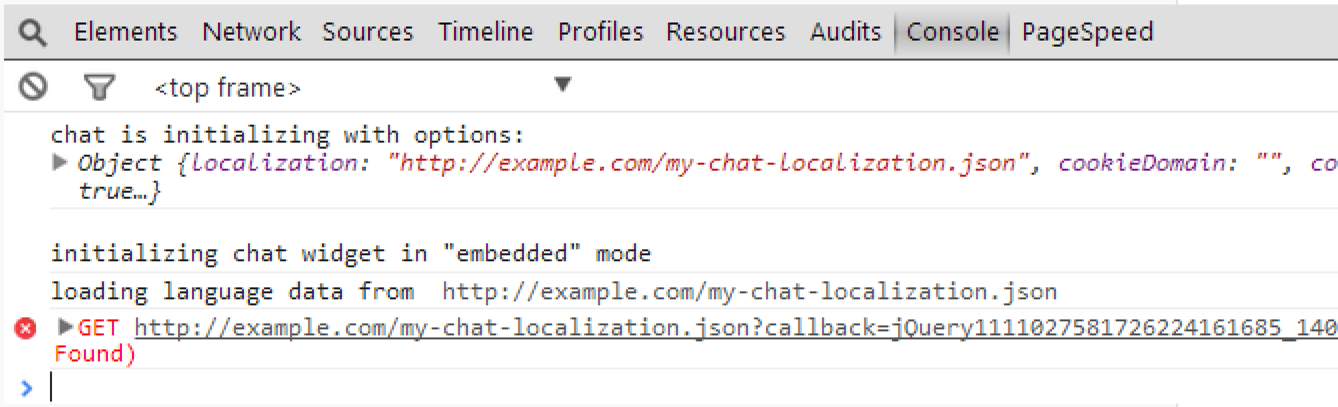
Здравствуйте! Наш представитель будет с вами в ближайшее время.

6. **Troubleshooting**
   If localization fails to load or parse, the chat will not start, signaling that something went wrong. To see what went wrong you will have to:

   - Enable "debug" mode:

   ```
   chat.startChat({
     localization: 'http://my-server.com/my-chat-localization.json',
     debug: true,
     // other parameters
   });
   ```

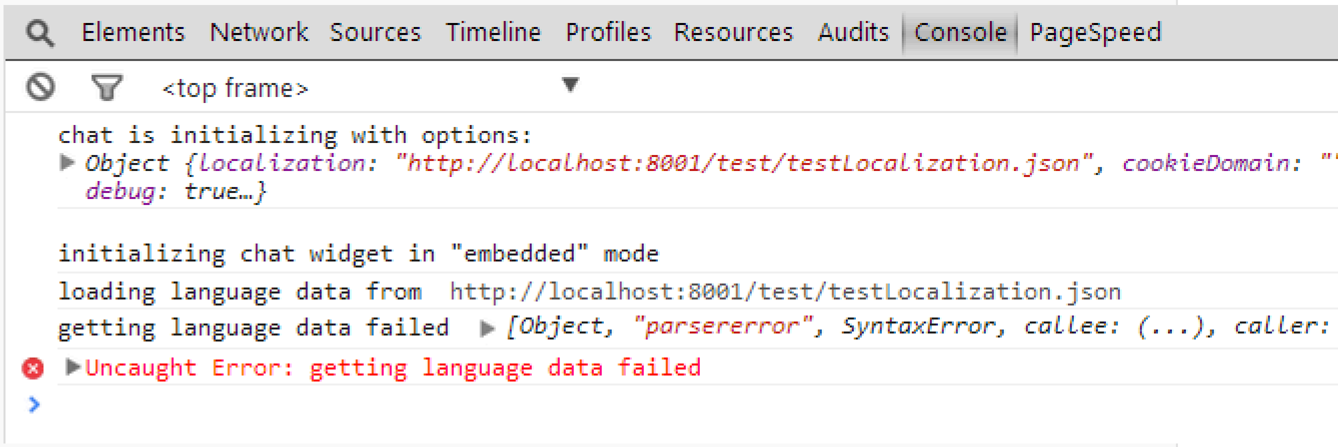
   - Enable browser developer tools (for example, in Chrome and most of other browser you can press F12 to launch).
   - You should see something like this:


Elements Network Sources Timeline Profiles Resources Audits Console PageSpeed
<top frame>

```
chat is initializing with options:
▶ Object {localization: "http://example.com/my-chat-localization.json", cookieDomain: "", co
  true…}
initializing chat widget in "embedded" mode
loading language data from  http://example.com/my-chat-localization.json
▶GET http://example.com/my-chat-localization.json?callback=jQuery1111027581726224161685_140
Found)
>
```

Here you can see that localization was not found on the server (response 404).

And here is another example of what you can see if JSON is either invalid or improperly configured:


Elements Network Sources Timeline Profiles Resources Audits Console PageSpeed
<top frame>

```
chat is initializing with options:
▶ Object {localization: "http://localhost:8001/test/testLocalization.json", cookieDomain: "
  debug: true…}
initializing chat widget in "embedded" mode
loading language data from  http://localhost:8001/test/testLocalization.json
getting language data failed  ▶ [Object, "parsererror", SyntaxError, callee: (...), caller:
▶Uncaught Error: getting language data failed
>
```

Additional Methods

Besides the main API methods, the following methods are also available since chat version 850.6.0:

- onBeforeChatOptionsApplied(callback)—Use this method to modify options before the chat starts or restores. This method may be useful if you do not control the `startChat()` call but still need to control some of the options.
- onSession(callback)—This is an alternative method of accessing the Chat Service JS API. This method is an alternative to accessing the Chat Service JS API through the done callbacks of the `startChat` and `restoreChat` methods. This method may be useful if you do no control the `startChat` call.
- close()—Close the chat widget. Added in 850.5.0
- toggle() —Minimize or restore the chat widget. Added in 850.6.0.
- onMinimized(callback) —Add a callback when the chat widget is minimized or restored. Added in 850.6.0
- isMinimized()—Check if the chat widget is minimized. Added in 850.6.0
- VERSION—Returns current chat version. Added in 850.1.0

Customization Examples

In the examples below we use Google Chrome as the web browser, but you can find similar developer features in any other modern browser.

Warning

The HTML structure and CSS classes in the chat widget are subject to change and Genesys does not guarantee backwards compatibility in future versions. This means that you might need to update your customizations when you update the chat widget.

Procedure: Modifying the Styling of the Chat Messages

Purpose: Customization Type: CSS-based

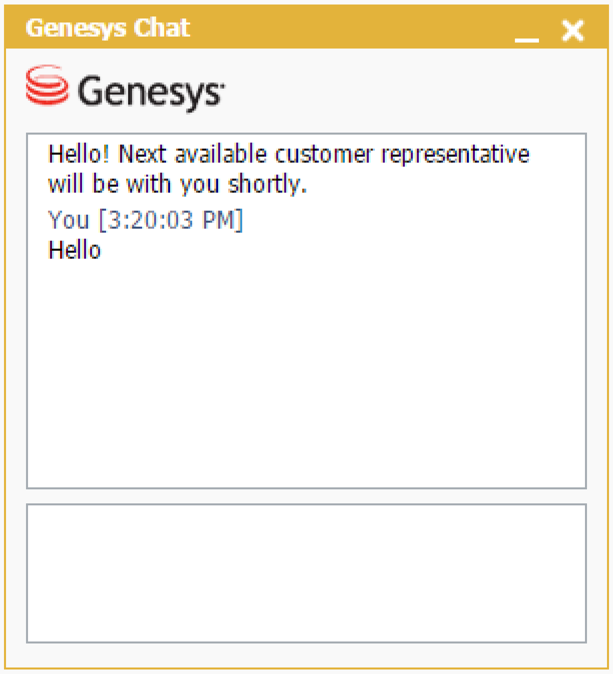
In this example we play with the styling of the messages that appear in the chat widget. This should give you a taste of what a CSS-based customization might look like.

Prerequisites

- You must have basic knowledge of CSS and HTML.

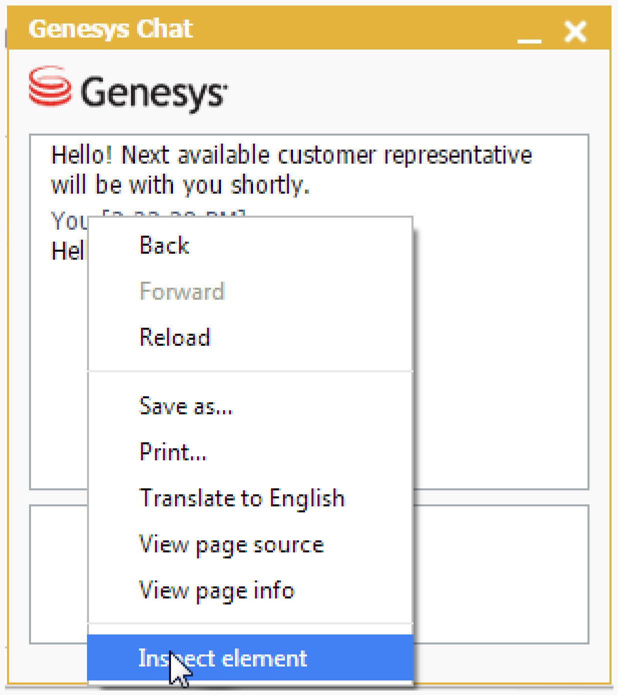
Steps

1. Launch a web page that is instrumented with the chat widget.
2. Start a chat session and send a chat message. You will see something like this:

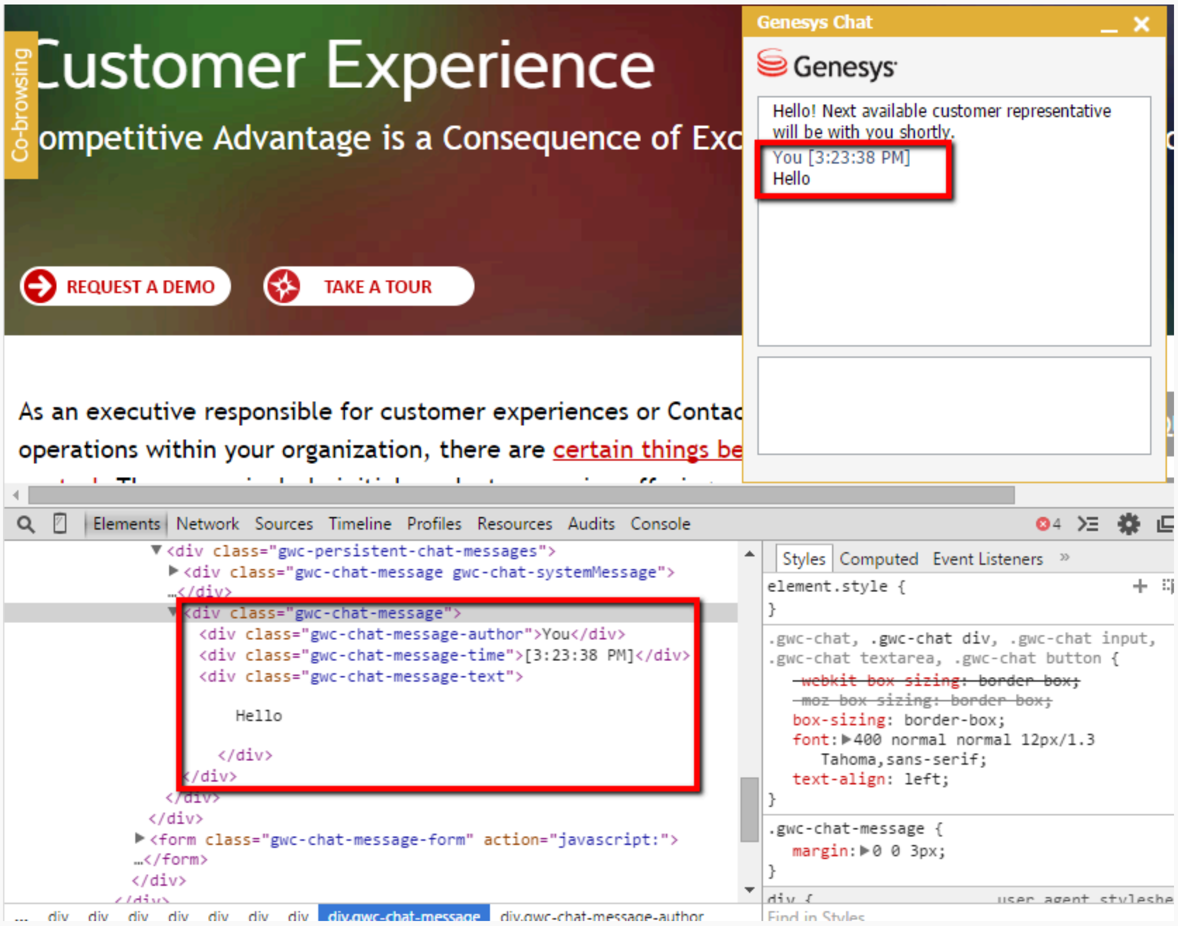


Hello! Next available customer representative will be with you shortly.
You [3:20:03 PM]
Hello

3. Right-click the message and choose "Inspect Element" to start the Chrome developer tools.



We can see that the chat message consists of three elements, each with its own dedicated CSS class. We will use this information to create new styling for these elements.



Hello! Next available customer representative will be with you shortly.
You [3:23:38 PM]
Hello

As an executive responsible for customer experiences or Contac
operations within your organization, there are certain things be

```
▼<div class="gwc-persistent-chat-messages">
  ▶<div class="gwc-chat-message gwc-chat-systemMessage">
  …</div>
  ▼<div class="gwc-chat-message">
     <div class="gwc-chat-message-author">You</div>
     <div class="gwc-chat-message-time">[3:23:38 PM]</div>
     <div class="gwc-chat-message-text">

         Hello

       </div>
   </div>
  </div>
</div>
▶<form class="gwc-chat-message-form" action="javascript:">
…</form>
</div>
</div>
```

```
element.style {
}
.gwc-chat, .gwc-chat div, .gwc-chat input,
.gwc-chat textarea, .gwc-chat button {
  ~~-webkit-box-sizing: border-box;~~
  ~~-moz-box-sizing: border-box;~~
  box-sizing: border-box;
  font:▶400 normal normal 12px/1.3
     Tahoma,sans-serif;
  text-align: left;
}
.gwc-chat-message {
  margin:▶0 0 3px;
}
div {                              user agent styleshe
```

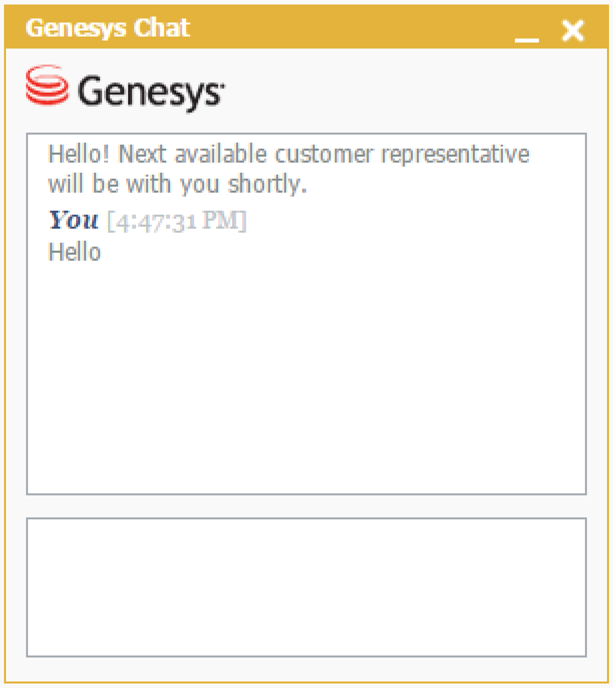
… div div div div div div div div.gwc-chat-message div.gwc-chat-message-author

Find in Styles

4. Next, we create our custom CSS to modify the colors of some of the text and the font used for the author name.

```
/* 1. Make the name stand out */
div.gwc-chat-message-author {
    font-family: Georgia;
    font-style: italic;
    font-weight: bold;
}

/* 2. Make the date more subtle */
div.gwc-chat-message-time {
    font-family: Georgia;
    color: #bdc3c7;
}

/* Make the body of a message a bit less contrast */
div.gwc-chat-message-text {
    color: #7f8c8d;
}
```

5. Add this CSS to your web page.
6. Reload the page. The chat message has the new look and feel we defined in the CSS.



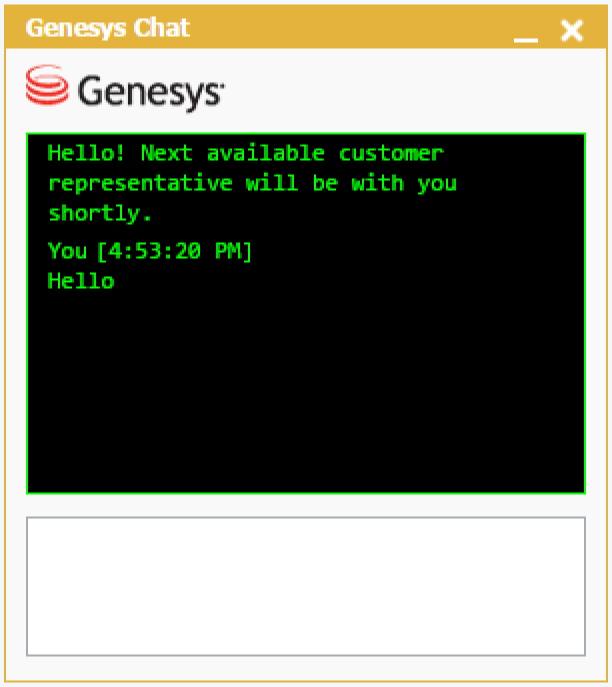
Hello! Next available customer representative will be with you shortly.
You [4:47:31 PM]
Hello

Tip

Here's a fun example of how to transform the chat message into an old-style computer terminal:

```
.gwc-chat-message-container {
    background: #000;
    border-color: #0f0;
}
.gwc-chat-message > div {
    font-family: monospace;
    color: #0f0;
}
```

And the result:



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

You [4:53:20 PM]
Hello

Procedure: Substituting the Genesys Logo with a Custom Image

Purpose: Customization Type: Template-based

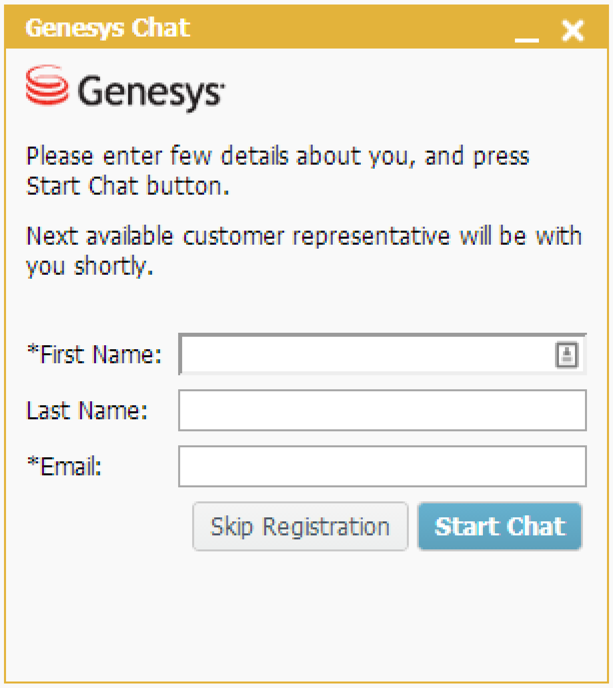
In this example we customize the Genesys logo that appears in the chat widget.

Prerequisites

- You must have basic knowledge of CSS and HTML.

Steps

1. Get the default chat templates HTML (see templates) and save it in a place where it is convenient for you to edit.
2. Start a chat. You will see something like this:

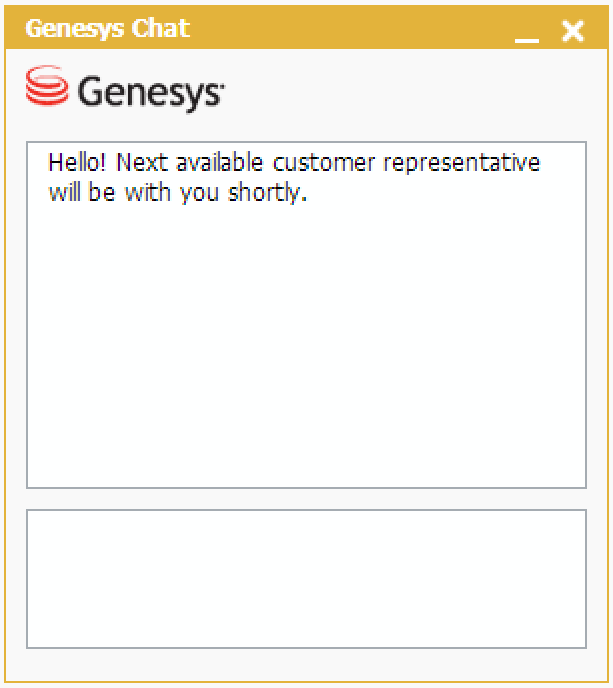


Please enter few details about you, and press Start Chat button.

Next available customer representative will be with you shortly.

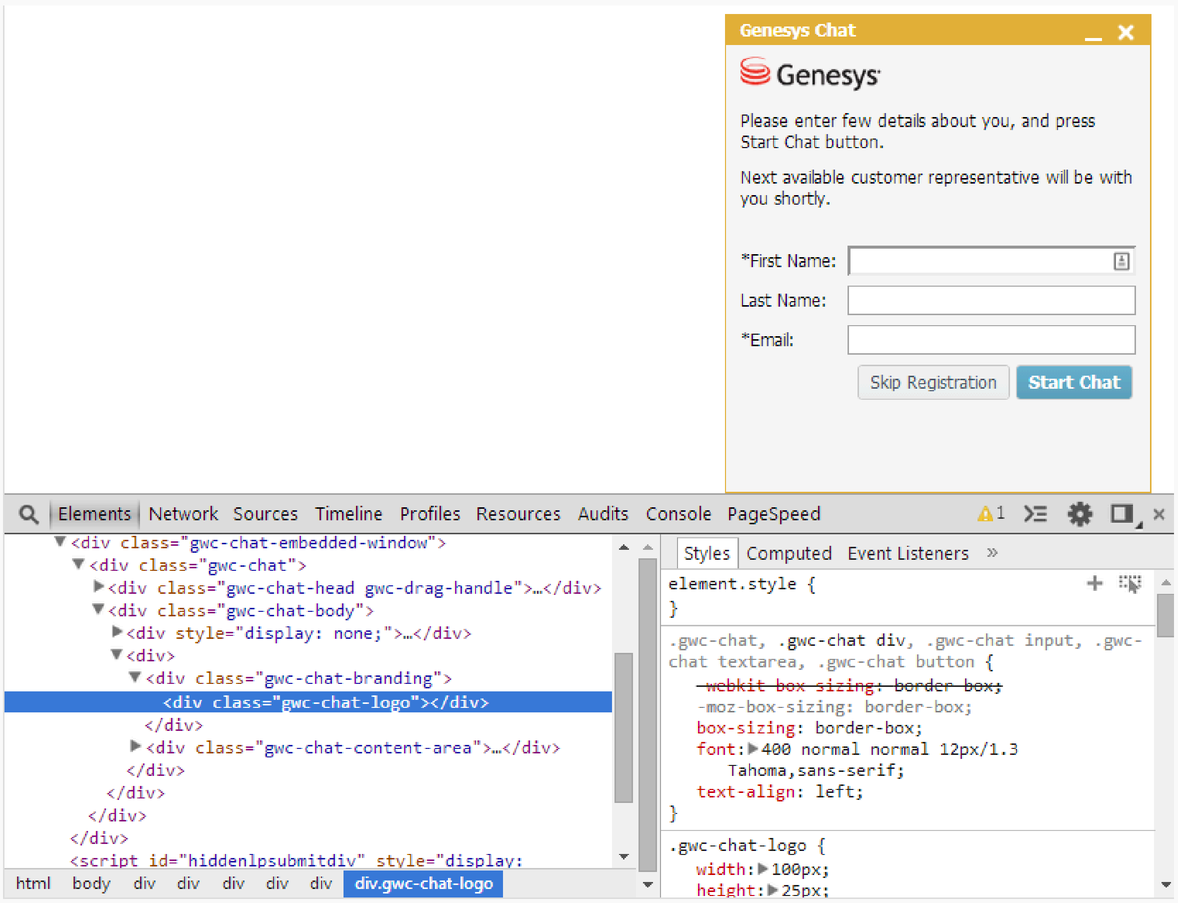
*First Name:	
Last Name:	
*Email:	

Or this, if registration is disabled:



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

3. Right-click the Genesys logo and choose "Inspect Element". The Chrome developer tools open and highlight the corresponding DOM element.


Genesys Chat
Genesys
Skip Registration
Start Chat
Elements Network Sources Timeline Profiles Resources Audits Console PageSpeed
Styles Computed Event Listeners

Please enter few details about you, and press Start Chat button.

Next available customer representative will be with you shortly.

*First Name:
Last Name:
*Email:

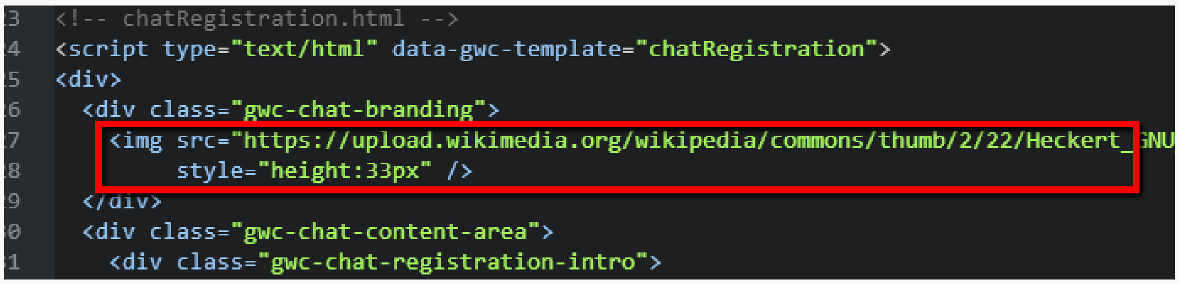
```
▼<div class="gwc-chat-embedded-window">
  ▼<div class="gwc-chat">
    ▶<div class="gwc-chat-head gwc-drag-handle">…</div>
    ▼<div class="gwc-chat-body">
      ▶<div style="display: none;">…</div>
      ▼<div>
        ▼<div class="gwc-chat-branding">
            <div class="gwc-chat-logo"></div>
          </div>
        ▶<div class="gwc-chat-content-area">…</div>
        </div>
      </div>
    </div>
  </div>
  <script id="hiddenlpsubmitdiv" style="display:
```

html body div div div div div div.gwc-chat-logo

```
element.style {
}
.gwc-chat, .gwc-chat div, .gwc-chat input, .gwc-
chat textarea, .gwc-chat button {
   ~~-webkit-box-sizing: border-box;~~
   -moz-box-sizing: border-box;
   box-sizing: border-box;
   font:▶400 normal normal 12px/1.3
      Tahoma,sans-serif;
   text-align: left;
}
.gwc-chat-logo {
   width:▶100px;
   height:▶25px;
```

Now that we know the CSS class of the element, we can look for it in the templates.

4. Open the templates HTML file in your favorite text editor and replace the <div> that has the **gwc-chat-logo** class with an image element. In this example, we use the publicly available logo of the GNU project:



```
<!-- chatRegistration.html -->
<script type="text/html" data-gwc-template="chatRegistration">
<div>
  <div class="gwc-chat-branding">
    <img src="https://upload.wikimedia.org/wikipedia/commons/thumb/2/22/Heckert_GNU
         style="height:33px" />
  </div>
  <div class="gwc-chat-content-area">
    <div class="gwc-chat-registration-intro">
```

Important

There are two logo elements in the templates: one in the **chatRegistration** template and another in the **chatView** template.

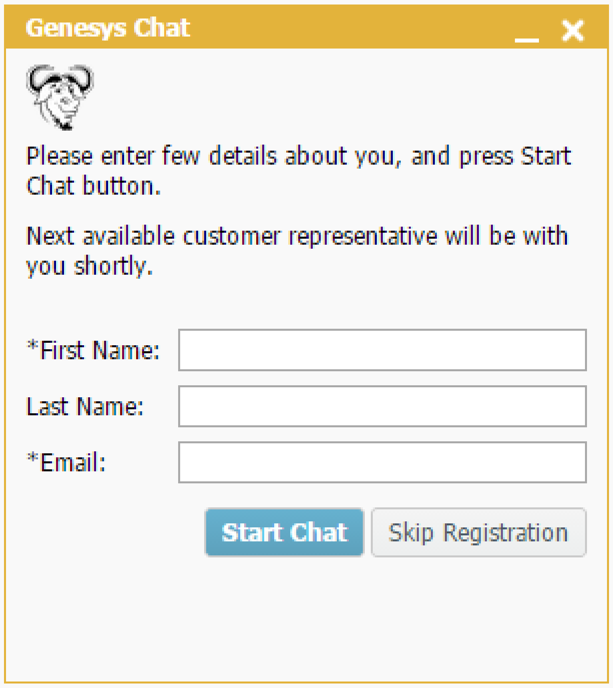
5. Save the modified templates file and "host" it somewhere that is accessible via HTTP.
6. Configure the chat to use the modified template file by providing the URL of the file to the **templates** option:

```
<script>
 var _genesys = {
     chat: {
         templates: 'http(s)://example.com/chatTemplates.html'
     }
 };
 </script>
 <INSTRUMENTATION_SNIPPET>
```

Important

This example uses the Integrated JavaScript Application to add chat to the page. See Configuring Chat for instructions on configuring the chat widget using the Integrated Application.

7. Start a new chat. You will see something like this:


Genesys Chat

Please enter few details about you, and press Start Chat button.

Next available customer representative will be with you shortly.

*First Name:	
Last Name:	
*Email:	

Tip

For "popup" mode implementations, you can use the same algorithm except instead of adding the CSS to your site, add it to the chat widget page.

Procedure: Setting the Ground to Create Your Own Chat Widget (popup mode)

Purpose: To customize the chat UI in popup mode, you can create an HTML page for the chat widget and then use the Chat Widget JS API to add functionality.

Prerequisites

- You must have basic knowledge of CSS and HTML.

Steps

1. Make the chat open the window using this calling code on your web page:

   ```
   chat.startChat({
     widgetUrl: <URL_OF_YOUR_HTML_PAGE_HERE>,
     ui: false
    });
   ```

2. Inside the widget HTML, start the chat and hook your UI to the API:

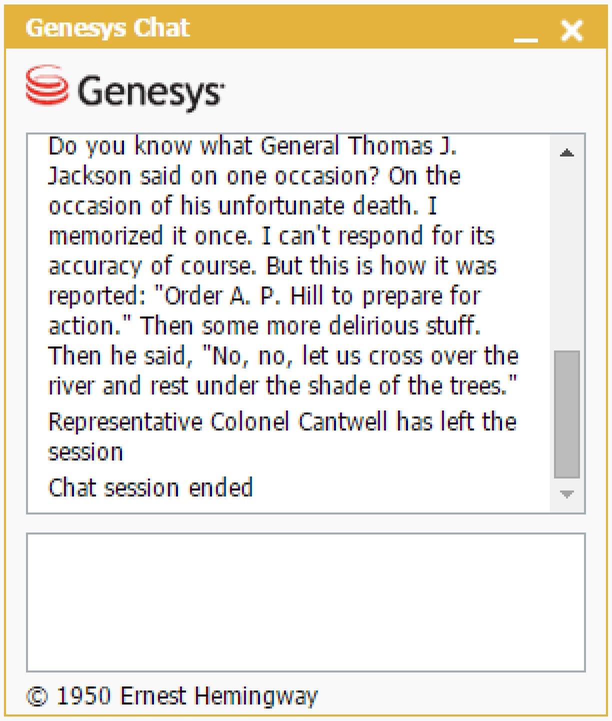
   ```
   <!DOCTYPE html>
    <html>
    <head>
      <title>My Custom Chat Widget</title>
    </head>
    <body>
    <script src="<PATH_TO_chatWidget.js>"></script>
    <script>
      chat.startChatInThisWindow().done(function(session) {
        // Implement your own UI using session API
      });
    </script>
    </body>
    </html>
   ```

Procedure: Adding Extra Content to the Chat Widget

Purpose: Customization Type: Template-based and CSS-based

In this example we add extra content to the chat widget: a "Copyright" notice at the bottom of the widget. We use the templates for this because they are a great fit for adding anything extra to the widget. We will also need to to adjust the CSS to make our changes look good.

Here is a look at the end result we are trying to achieve:



Do you know what General Thomas J. Jackson said on one occasion? On the occasion of his unfortunate death. I memorized it once. I can't respond for its accuracy of course. But this is how it was reported: "Order A. P. Hill to prepare for action." Then some more delirious stuff. Then he said, "No, no, let us cross over the river and rest under the shade of the trees."

Representative Colonel Cantwell has left the session

Chat session ended

© 1950 Ernest Hemingway

Tip

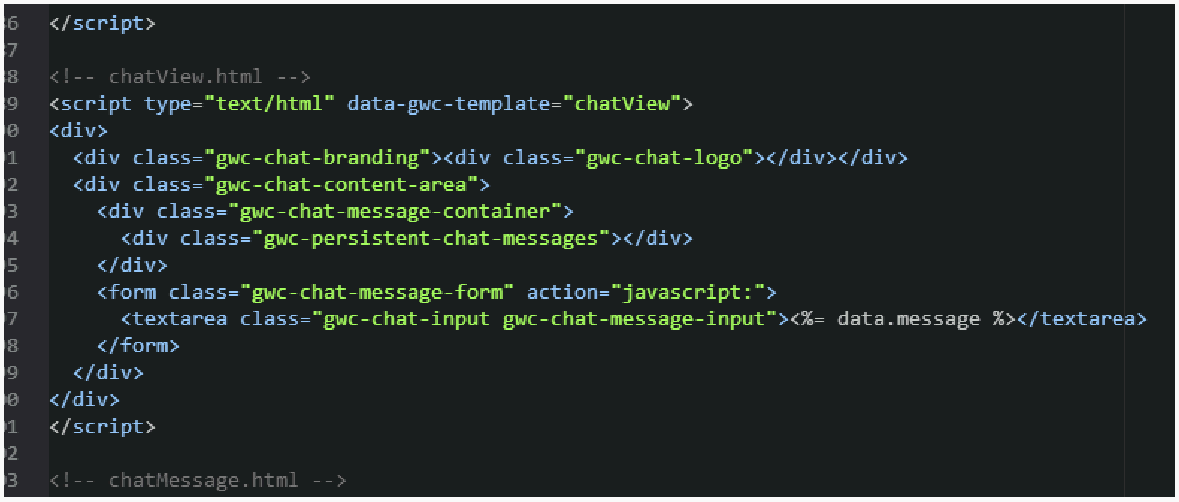
You can use the algorithm in this example to add, remove, or modify any part of the widget.

Prerequisites

- You must have basic knowledge of CSS and HTML.
- Some experience with web browser developer tools would be helpful.
- You have read the templates section.

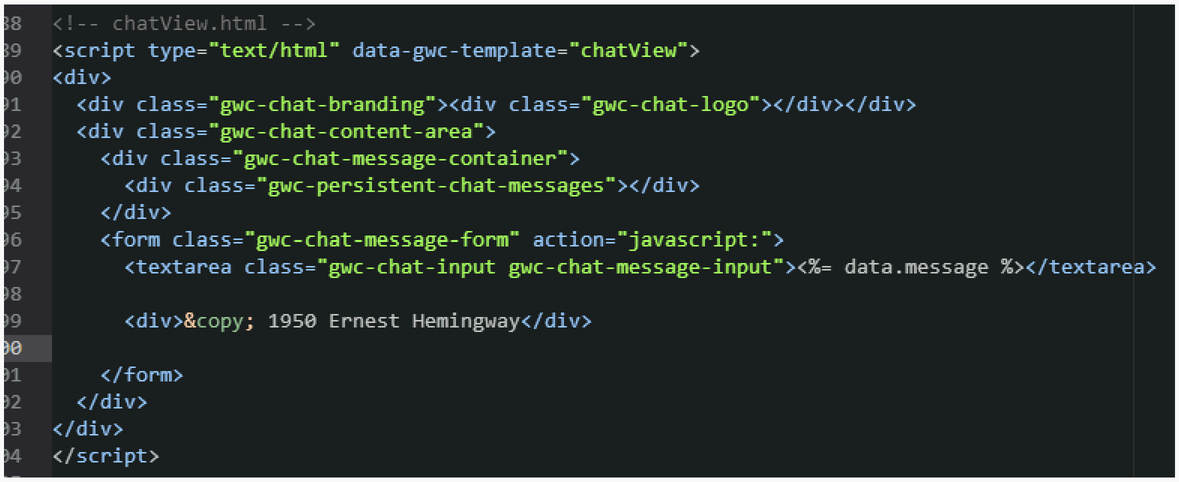
Steps

1. Get the default chat templates HTML (see templates) and save it in a place where it is convenient for you to edit.
2. Open the file in your favorite text editor and find the **chatView** template. This template is responsible for the general structure of the widget (the area that displays the messages and the input area) and this is where we will add our new content.



```
</script>

<!-- chatView.html -->
<script type="text/html" data-gwc-template="chatView">
<div>
  <div class="gwc-chat-branding"><div class="gwc-chat-logo"></div></div>
  <div class="gwc-chat-content-area">
    <div class="gwc-chat-message-container">
      <div class="gwc-persistent-chat-messages"></div>
    </div>
    <form class="gwc-chat-message-form" action="javascript:">
      <textarea class="gwc-chat-input gwc-chat-message-input"><%= data.message %></textarea>
    </form>
  </div>
</div>
</script>

<!-- chatMessage.html -->
```

3. Add the new content to the template. In this example, we add it to the `<form>` below the input area. This is not semantic, but makes further CSS-related work a bit easier.



```
<!-- chatView.html -->
<script type="text/html" data-gwc-template="chatView">
<div>
  <div class="gwc-chat-branding"><div class="gwc-chat-logo"></div></div>
  <div class="gwc-chat-content-area">
    <div class="gwc-chat-message-container">
      <div class="gwc-persistent-chat-messages"></div>
    </div>
    <form class="gwc-chat-message-form" action="javascript:">
      <textarea class="gwc-chat-input gwc-chat-message-input"><%= data.message %></textarea>

      <div>&copy; 1950 Ernest Hemingway</div>

    </form>
  </div>
</div>
</script>
```

4. Save the modified templates file and "host" it somewhere that is accessible via HTTP.

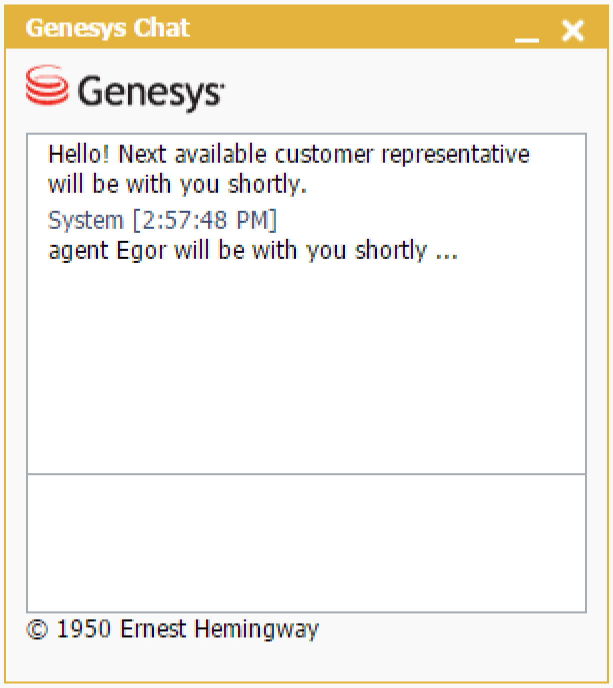
5. Configure the chat to use the modified template file by providing the URL of the file to the **templates** option:

```
<script>
var _genesys = {
    chat: {
        templates: 'http(s)://example.com/chatTemplates.html'
    }
};
</script>
<INSTRUMENTATION_SNIPPET>
```

Important

This example uses the Integrated JavaScript Application to add chat to the page. See Configuring Chat for instructions on configuring the chat widget using the Integrated Application.

6. Start a new chat. You will see something like this:



Hello! Next available customer representative will be with you shortly.
System [2:57:48 PM]
agent Egor will be with you shortly ...

© 1950 Ernest Hemingway

The content is there, but the layout looks a bit broken. To fix it, we're going to use some CSS.

7. Follow the same algorithm that is used for every CSS customization:
   a. Inspect an element with your browser's developer tools.
   b. See if it has the styling you need.
   c. If no, try its parents or children until you find the right element.
   d. Modify the CSS rules for the element until you're satisfied with the results.
   e. Save all your modifications as separate CSS.
   f. Add this CSS to your page; it will override the initial CSS of the widget.

   For this particular example, we can use the following simple CSS rule:

```
.gwc-chat-message-form {
    bottom: 3px;
}
```

Essentially, we are moving the form a bit closer to the bottom edge because it now contains an additional `<div>` that occupies the space that was previously occupied by the margin.

8. Add the CSS to your web page.

Procedure: Using the ui.onBeforeMessage Hook to Add Desktop Notifications

Purpose: Customization Type: JavaScript-based

In this example we use the browser Notification API and Page Visibility API to show a notification when a user receives a chat message while on another browser tab.

The purpose of the example is to show how you can use the UI hooks to add the functionality to the chat widget.

> Important
>
> **The browser APIs used in this example are an experimental technology.** This technology's specification has not stabilized, so you should check the linked pages above for details about usage in various browsers. Also note that the syntax and behavior of an experimental technology is subject to change in future versions of browsers as the spec changes. At the time this example was created (October 2014), it worked successfully on the desktop version of Chrome.

Prerequisites

- You must have basic knowledge of JavaScript.

Steps

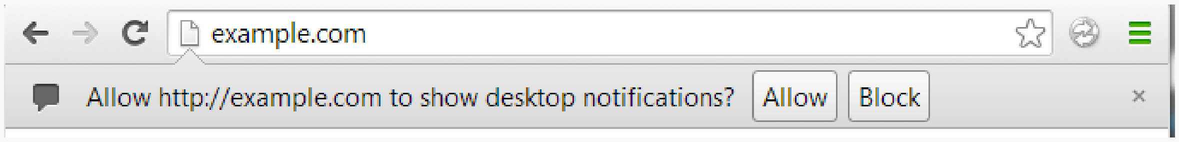
1. Create a function that shows a notification in the browser (if the browser supports the APIs) and add it to our instrumentation:

```
var _genesys = {
    chat: {
        ui: {
            onBeforeMessage: function(messageEl, messageText) {

                // If page is in focus or page visibility API is not supported,
quit.
                if (!document.hidden) {
                    return;
                }

                // If notification API is not supported, quit.
                if (!Notification) {
                    return;
                }


                if (Notification.permission === 'granted') {
                    var notification = new Notification(document.location.host + '
representative says:', {
                        icon: 'http://placekitten.com/51/50', // add an avatar
                        body: messageText // include text entered by agent
                    });
                    // When notification is clicked, bring the tab with chat into
focus.
                    notification.onclick = function() {
                        window.focus();
                    }
                }
            }
        }
    }
}

// When page is loaded, ask the user permission to show notifications
if (Notification) {
    Notification.requestPermission();
}
```

Important

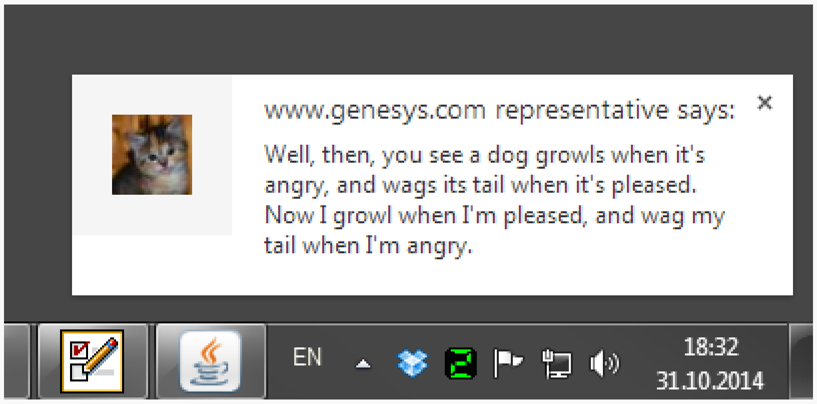
This example uses the Integrated JavaScript Application to add chat to the page. The Integrated Application makes sure that the option is passed to the `startChat()` and `restoreChat()` methods. This means the modification is applied whether the chat is started on the current page or restored after the navigation/page reload See Configuring Chat for instructions on configuring the chat widget using the Integrated Application.

2. Load a page on your site. You should see a request to allow the notifications (see the last line of the code snippet in Step 1). For example, in Chrome the request might look like this:



Allow http://example.com to show desktop notifications?

3. Click **Allow**.
4. Start a chat session and join as an agent (or wait for the agent to join if you do not control the environment).
5. Switch to another tab in the browser
6. Send a message from the agent (or wait for the agent to send the message).
7. A notification appears on your desktop:



www.genesys.com representative says:

Well, then, you see a dog growls when it's angry, and wags its tail when it's pleased. Now I growl when I'm pleased, and wag my tail when I'm angry.

8. Click the message. The page with the chat widget gets the focus.

Procedure: Displaying a confirmation alert when users close the chat widget

Purpose: In this example we make users confirm they want to close the chat widget. The example applies to embedded mode as popup mode does this out-of-the-box.

Customization Type: JavaScript-based

Prerequisites

You must have basic knowledge of JavaScript.

Steps

1. Write a javascript function that asks user to confirm they want to close the chat. This function should return `true` if users answer "Yes" and `false` otherwise. For example:

```
function() { return confirm('Do you really want to close the chat?'); }
```

2. Add your function as a `click` handler for the element with class `gwc-chat-control-close`. This element is the close button of the chat widget. You must add your function only after the element exists on the page.

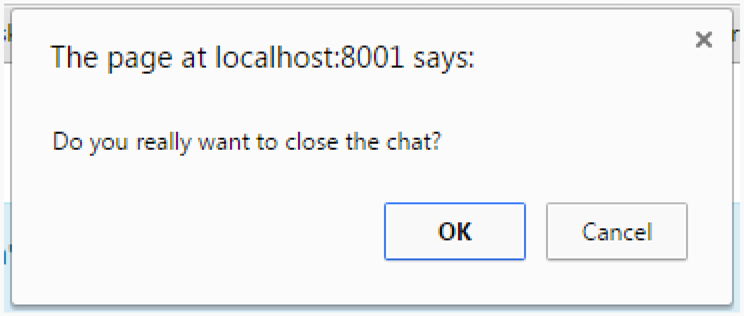
   If registration is enabled, add your functions to the `ui.onBeforeRegistration` handler. You must also wrap your functions in `setTimeout` so your function executes *after* the chat renders. Your instrumentation should look like this:

```
var _genesys = {
    debug: true,
    chat: {
        ui: {
            onBeforeRegistration: function () {
                setTimeout(function() {
                    document.querySelectorAll('.gwc-chat-control-close')
                        .addEventListener('click', function() {
                            return confirm('Do you really want to close the
chat?');
                        }, false);
                }, 0);
            }
        }
};
```

3. If registration is not enabled, add your function to `ui.onBeforeChat` instead of `ui.onBeforeRegistration`:

```
var _genesys = {
    debug: true,
    chat: {
        ui: {
            onBeforeChat: function () {
                setTimeout(function() {
                    document.querySelectorAll('.gwc-chat-control-close')[0]
                        .addEventListener('click', function() {
                            return confirm('Do you really want to close the
chat?');
                        }, false);
                }, 0);
            }
        }
};
```

Now, when users try to close chat, they see a standard notification:


The page at localhost:8001 says:

Do you really want to close the chat?

If they click **OK** the chat closes but persists otherwise.

Procedure: Implementing a client-side chat session timer

Purpose: In this example we implement a simple code snippet that ends the chat if the user has not sent or received any messages for a period of time.

Customization Type: JavaScript-based

Prerequisites

You must have basic knowledge of JavaScript.

Steps

1. Add a hook to your instrumentation to access the Chat APIs:

```
var _genesys = {
     chat: {
         onReady: function(chat) {

         }
     }
 };
```

> **Important**
> This example uses the integrated javascript application to add chat to the page.

2. After you access the Chat Widget API, add an onSession handler to access the Chat Service API:

```
var _genesys = {
     chat: {
         onReady: function(chat) {
             chat.onSession(function(session) {

             });
         }
     }
 }
```

3. Implement a function that triggers the timer. For example:

```
chat.onSession(function(session) {
    var timeout = 30000,  // 30 seconds
        sessionTimeout;
        function startCountdown() {
            sessionTimeout = setTimeout(function() {
            // session timed out
            }, timeout);
        }
});
```

4. Now, connect the timing functionality to the **onMessageReceived** chat event. We restart the countdown as soon as the user receives a message. When the countdown ends, we leave the session. A user's own messages also trigger **onMessageReceived**. The timer expires when neither the user or agent send any messages in the time period.

```
chat.onSession(function(session) {
    var timeout = 30000,  // 30 seconds
        sessionTimeout;
    function startCountdown() {
        sessionTimeout = setTimeout(function() {
            session.leave();
        }, timeout);
    }

    session.onMessageReceived(function(event) {
        clearTimeout(sessionTimeout);
        startCountdown();
    });
});
```

The whole example now looks like this:

```
var _genesys = {
    chat: {
        onReady: function(chat) {

            chat.onSession(function(session) {
                var timeout = 30000,  // 30 seconds
                    sessionTimeout;

                function startCountdown() {
                    sessionTimeout = setTimeout(function() {
                        session.leave();
                    }, timeout);
                }

                session.onMessageReceived(function(event) {
                    clearTimeout(sessionTimeout);
                    startCountdown();
                });
            });

        }
    }
};
```

Procedure: Inserting a line break with Shift+Enter

Purpose: By default, the chat widget sends a message with **Enter** and **Ctrl+Enter** inserts a line break. In this example, we make **Shift+Enter** also insert a line break.

Customization Type: JavaScript-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This examples uses the jQuery library.

Steps

1. Use the ui.onBeforeChat hook to access the `textarea` element used to enter messages.

```
var _genesys = {
    chat: {
        ui: {
            onBeforeChat: function(chatElement) {
                var textarea = jQuery(chatElement).find('.gwc-chat-message-input');
            }
        }
    }

};
```

2. Bind a handler to the `keypress` event of the `textarea`:

```
var textarea = jQuery(chatElement).find('.gwc-chat-message-input');
textarea.keypress(function (e) {

});
```

3. In the event handler, insert a line break at the current `textarea` value when the user presses the `Enter` and `Shift` keys. Return `false` to prevent default behavior of sending the message. If the user does not press the `Shift` and `Enter` keys, the handler passes through and triggers default behavior.

```
var textarea = jQuery(chatElement).find('.gwc-chat-message-input');
textarea.keypress(function (e) {
    // Enter key was pressed
    if (e.which === 13 || e.which === 10) {


        // If Shift was pressed, break line.
        if (e.shiftKey) {
           textarea.val(textarea.val() + '\\n');
```

```
                return false;
            }
        }
    });
```

The complete configuration looks like this:

```
var _genesys = {
    chat: {
        ui: {
            onBeforeChat: function(chatElement) {
                var textarea = jQuery(chatElement).find('.gwc-chat-message-input');
                textarea.keypress(function (e) {

                    // Enter key pressed
                    if (e.which === 13 || e.which === 10) {

                        // If Shift key pressed, insert break line and prevent
default behavior
                        if (e.shiftKey) {
                           textarea.val(textarea.val() + '\\n');
                           return false;
                        }
                    }
                });
            }
        }
    }

};
```

Procedure: Automatically opening a URL pushed by an agent

Purpose: In some desktops, the agent can *push* a URL instead of sending plain text. By default, the chat widget renders a pushed URL using the a tag, making it a link. In this example, we make links pushed by the agent open automatically.

Customization Type: JavaScript-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This example requires Integrated JavaScript Application version 850 and above.

Steps

1. In your instrumentation, configure *anytime* access to the Chat Widget API:

   ```
   <script>
    var _genesys = {
      chat: {
        onReady: []
      }
    };
   ```

   Learn more about his snippet here, Integrated JavaScript Application—Obtaining Chat and Co-browse APIs

2. Somewhere in your code, access the Chat Widget API:

   ```
   _genesys.chat.onReady.push(function(chatWidgetApi) {

    });
   ```

3. Use the Chat Widget API to subscribe to the chat session. The chat session object implements the Chat Service API:

   ```
   _genesys.chat.onReady.push(function(chatWidgetApi) {

       chatWidgetApi.onSession(function(chatSession) {

       });

    });
   ```

4. Use the chat session to subscribe to the onMessageReceived event:

   ```
   _genesys.chat.onReady.push(function(chatWidgetApi) {

       chatWidgetApi.onSession(function(chatSession) {

           chatSession.onMessageReceived(function(event) {

           });

       });

    });
   ```

5. If the incoming event is of type URL, open the URL in the current window:

   ```
   _genesys.chat.onReady.push(function(chatWidgetApi) {

       chatWidgetApi.onSession(function(chatSession) {

           chatSession.onMessageReceived(function(event) {
              if (event.content.type === 'url') {
                 window.location = event.content.text;
              }
           });

       });
   ```

```
    });
```

6. If you use chat in *pop-up* mode where the chat widget opens in a separate browser window, you must modify the code to open the URL in the *parent* window instead of the chat widget window. You must also place the code in the `chatWidget.html` file linked in the widgetUrl option.

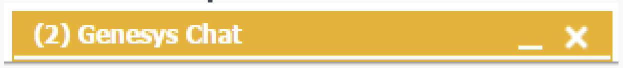
   chatWidget.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Genesys Chat</title>
</head>
<body>
<script src="js/chatWidget.js"></script>
<script>
  chat.startChatInThisWindow().done(function(session) {
    session.onMessageReceived(function(event) {
      if (event.content.type === 'url') {
        window.opener.location = event.content.text;
      }
    })
  });
</script>
</body>
</html>
```

Procedure: Showing the number of unread messages in a minimized chat widget

Purpose: In this example, we add a simple counter that notifies the user of any unread messages. This example applies to *embedded* mode where the widget is rendered on the web page directly.

Counter example:



Customization Type: JavaScript-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This example assumes you are using chat as part of the Integrated JavaScript Application.
- The example uses the jQuery library.

Steps

1. Add an HTML element to the ebeddedWindow template. to use as our counter. For example, below we add `<span class="gwc-chat-counter"></span>`. See Template-based Customization for more details.

```
<script type="text/html" data-gwc-template="embeddedWindow">
  <div class="gwc-chat-embedded-window">
    <div class="gwc-chat">
      <div class="gwc-chat-head gwc-drag-handle">
        <div class="gwc-chat-window-controls">
          <div class="gwc-chat-control gwc-chat-control-minimize">
            <div class="gwc-chat-icon gwc-chat-icon-minimize"></div>
          </div>
          <div class="gwc-chat-control gwc-chat-control-close">
            <div class="gwc-chat-icon gwc-chat-icon-close"></div>
          </div>
        </div>
        <div class="gwc-chat-title"><span class="gwc-chat-counter"></span> <%=
data.chatTitle %></div>
      </div>
      <div class="gwc-chat-body"></div>
    </div>
  </div>
</script>
```

2. Configure you instrumentation for *anytime* access to the Chat Widget API:

```
<script>
var _genesys = {
  chat: {
    onReady: []
  }
};
```

Learn more about his snippet here, Integrated JavaScript Application—Obtaining Chat and Co-browse APIs

3. Somewhere in your code, access the Chat Widget API:

```
_genesys.chat.onReady.push(function(chatWidget) {

});
```

4. Create a variable to hold the number of unread messages and set it to `0`. Use the Chat Widget API to subscribe to the chat session. The chat session object implements the Chat Service API.

```
_genesys.chat.onReady.push(function(chatWidgetApi) {
    var messageCount = 0;

    chatWidgetApi.onSession(function(chatSession) {

    });

});
```

5. In the session API, use the onMessageReceived method to subscribe to new messages. When a message comes in, increment the counter and update your counter. Use the isMinimized() method from the Chat Widget API to check if the widget is minimized.

```
_genesys.chat.onReady.push(function(chatWidgetApi) {
    var messageCount = 0;

    chatWidgetApi.onSession(function(chatSession) {
        session.onMessageReceived(function(event) {
            if (chat.isMinimized()) {
                messageCount++;
                $('.gwc-chat-counter').text('(' + messageCount + ')');
            }
        });
    });

});
```

6. Use the onMiminized method from the Chat Widget API to reset the counter when the user restores the chat widget.

```
_genesys.chat.onReady.push(function(chatWidgetApi) {
    var messageCount = 0;

    chatWidgetApi.onSession(function(chatSession) {
        session.onMessageReceived(function(event) {
            if (chat.isMinimized()) {
                messageCount++;
                $('.gwc-chat-counter').text('(' + messageCount + ')');
            }
        });
    });
    chatWidgetApi.onMinimized(function(isMinimized) {
        if (!isMinimized) {
            messageCount = 0;
            $('.gwc-chat-counter').text('');
        }
    });
});
```

Tip

This example does not properly support page reloads. To add support for page reloads:

- Use the `restored` property in the onMessageReceived callback to determine if the message is *new* or replayed.
- Use browser storage (for example, sessionStorage) to save and restore the counter value.

Procedure: Showing an *agent typing* notification in the minimized chat widget

Purpose: Building on the previous example, in this example we show a notification in the minimized widget when the agent is typing. This example only applies to *embedded* mode.

Customization Type: JavaScript-based

Prerequisites

- You must have basic knowledge of JavaScript.
- You completed the previous example, Showing the number of unread messages in a minimized chat widget.

Steps

1. Use the onAgentTyping method in the session API to subscribe to the agent typing event.

   ```
   session.onAgentTyping(function(event) {

   });
   ```

2. Create new content for your counter element. When the agent starts typing, append `...` to the

unread messages counter. If there are no unread messages, display `....` When the agents stops typing, just show the number of unread messages, if any.

```
session.onAgentTyping(function(event) {
    var text;
    if (event.isTyping) {
        text = messageCount ?  '(' + messageCount + '...)' : '(...)';
    } else {
        text = messageCount ? '(' + messageCount + ')' : '';
    }
});
```

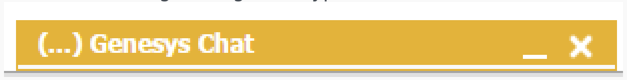
3. Add the content to the counter only when the chat is minimized:

```
session.onAgentTyping(function(event) {
    var text;
    if (!chat.isMinimized()) {
        return;
    }
    if (event.isTyping) {
        text = messageCount ?  '(' + messageCount + '...)' : '(...)';
    } else {
        text = messageCount ? '(' + messageCount + ')' : '';
    }
    $('.gwc-chat-counter').text(text);
});
```

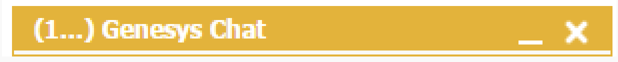
The whole snippet, including code from the previous example:

```
_genesys.chat.onReady.push(function(chatWidgetApi) {
    var messageCount = 0;

    chatWidgetApi.onSession(function(chatSession) {
        session.onMessageReceived(function(event) {
            if (chat.isMinimized()) {
                messageCount++;
                $('.gwc-chat-counter').text('(' + messageCount + ')');
            }
        });
        session.onAgentTyping(function(event) {
            var text;
            if (!chat.isMinimized()) {
                return;
            }
            if (event.isTyping) {
                text = messageCount ?  '(' + messageCount + '...)' : '(...)';
            } else {
                text = messageCount ? '(' + messageCount + ')' : '';
            }
            $('.gwc-chat-counter').text(text);
        });
    });
    chatWidgetApi.onMinimized(function(isMinimized) {
        if (!isMinimized) {
            messageCount = 0;
            $('.gwc-chat-counter').text('');
        }
        });
    });
```

Now, when the agent begins to type the user sees:



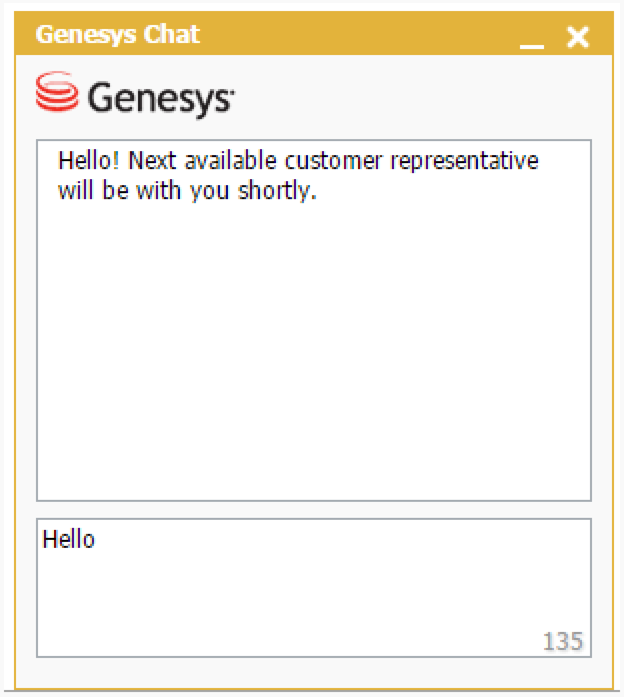
or



Procedure: Displaying a character counter in the message area

Purpose: This example adds a 140 character message limit and displays the number of remaining characters.

Character Counter:


Genesys Chat
Genesys
Hello! Next available customer representative will be with you shortly.
Hello
135

Customization Type: JavaScript-based, template-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This example assumes you are using chat as part of the Integrated JavaScript Application.
- The example uses the jQuery library.

Steps

1. Access the default **chatView** template. See Template-based Customization
2. Modify the template to set a limit on the text area and add a counter element. We use the class name `gwc-input-counter`.

```
<script type="text/html" data-gwc-template="embeddedWindow">
<div>
  <div class="gwc-chat-branding"><div class="gwc-chat-logo"></div></div>
  <div class="gwc-chat-content-area">
    <div class="gwc-chat-message-container">
      <div class="gwc-persistent-chat-messages"></div>
    </div>
    <form class="gwc-chat-message-form" action="javascript:">
      <textarea class="gwc-chat-input gwc-chat-message-input" maxlength="140"><%=
data.message %></textarea>
    </form>
    <div class="gwc-input-counter">140</div>
  </div>
</div>
</script>
```

> **Important**
>
> We use the `maxlength` property to limit the input on the `<textarea>`. The `maxlength` property is part of the HTML5 spec and may be unavailable in older browsers such as IE9 and below. For older browsers, you must use a JavaScript solution like the one described here, http://stackoverflow.com/a/12131507/697388.

3. At this point, the character counter is not visible in the chat widget. To make the counter visible, add we the following CSS:

```
<style>
.gwc-input-counter {
    position: absolute;
    bottom: 16px;
    right: 14px;
    color: #999;
    font-weight: bold;
    text-shadow: 1px 1px 1px lightgrey;
}
</style>
```

4. Now we add an event listener to the text area. Use the ui.onBeforeChat hook to access the text area element within chat:

```
var _genesys = {
    chat: {
        ui: {
            onBeforeChat: function(chatHtml) {
                $(html).find('textarea'); // our textarea
            }
        }
    }
};
```

Use the event listener to calculate the number of characters remaining and update the counter element:

```
var _genesys = {
    chat: {
        ui: {
            onBeforeChat: function(chatHtml) {
                $(html).find('textarea').on('input', function () {
                    var charsLeft = this.maxLength - this.value.length;
                    $('.gwc-input-counter').text(charsLeft);
                });
            }
        }
    }
};
```

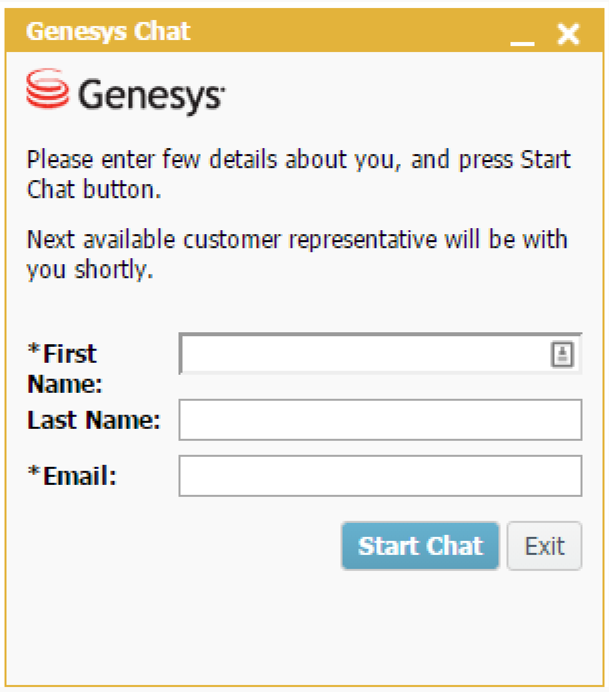
Important

This example uses the `input` event which may be unavailable in older browsers. You may want to use keyup or another event instead.

Procedure: Replace **Skip Registration** button with **Exit** button

Purpose: In this example we replace the **Skip Registration** button with an **Exit** button. Clicking **Exit** closes the chat widget.

Exit button:



Please enter few details about you, and press Start Chat button.

Next available customer representative will be with you shortly.

*First Name:	
Last Name:	
*Email:	

Customization Type: JavaScript-based, template-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This example assumes you are using chat as part of the Integrated JavaScript Application.
- The example uses the jQuery library.

Steps

1. Add the `Exit` string to the localization bundle. You can do this in different ways, see Chat Widget API—Localization. In this example, we create a `regExit` key with value `Exit` and pass the key-value pair to the `localization` option. By doing so, we extend the built-in localization with our new key:

```
<script>
var _genesys = {
    chat: {
        localization: {
            'regExit': 'Exit'
        }
    }
};
<script>
```

2. Modify the template to render our new text instead of **Skip Registration**. Copy and paste the default `chatRegistration` template into your page, see more about templates in Chat Widget API—Template-based Cusomization. Substitute `<%= data.nls.regSkip %>` with `<%= data.nls.regExit %>`. The updated template looks like this:

```
<script type="text/html" data-gwc-template="chatRegistration">
  <div>
    <div class="gwc-chat-branding"><div class="gwc-chat-logo"></div></div>
    <div class="gwc-chat-content-area">
      <div class="gwc-chat-registration-intro">
        <p class="gwc-chat-registration-intro-p">
          <%= data.nls.regWelcomePart1 %></p>
        <p class="gwc-chat-registration-intro-p">
          <%= data.nls.regWelcomePart2 %></p>
      </div>
      <form>
        <div class="gwc-chat-controls-container">
          <div class="gwc-chat-control-group gwc-chat-control-group-required">
            <label for="gcbChatFirstName"
                   class="gwc-chat-label">
              *<%= data.nls.regFirstName %>
            </label>
            <div class="gwc-chat-controls">
              <input id="gcbChatFirstName" name="FirstName"
                     class="gwc-chat-registration-input"
                     type="text"/>
              <div class="gwc-chat-validation-error"></div>
            </div>
          </div>
          <div class="gwc-chat-control-group">
            <label for="gcbChatLastName" class="gwc-chat-label">
              <%= data.nls.regLastName %>
            </label>
            <div class="gwc-chat-controls">
              <input id="gcbChatLastName" name="LastName"
                     class="gwc-chat-registration-input"
                     type="text"/>
              <div class="gwc-chat-validation-error"></div>
            </div>
          </div>
          <div class="gwc-chat-control-group gwc-chat-control-group-required">
            <label for="gcbChatEmail" class="gwc-chat-label">
              *<%= data.nls.regEmail %>
            </label>
            <div class="gwc-chat-controls">
              <input id="gcbChatEmail" name="EmailAddress"
                     class="gwc-chat-registration-input"
                     type="email"/>
              <div class="gwc-chat-validation-error"></div>
            </div>
          </div>
        </div>
        <div class="gwc-chat-registration-buttons">
          <div class="gwc-chat-registration-skip">
            <button class="gwc-chat-button gwc-chat-button-light"
 id="gcbChatSkipRegistration" type="button">
              <%= data.nls.regExit %>
```

```
                </button>
              </div>
              <div class="gwc-chat-registration-submit">
                <button class="gwc-chat-button" id="gcbChatRegister" type="submit">
                  <%= data.nls.regSubmit %>
                </button>
              </div>
            </div>
          </form>
        </div>
      </div>
    </script>
```

3. Use the ui.onBeforeRegistration hook along with jQuery and the close() method to add the button's behavior:

```
<script>
var _genesys = {
    chat: {
        localization: {
            'regExit': 'Exit'
        },
        ui: {
            onBeforeRegistration: function(regForm) {
                var $skipBtn = jQuery(regForm).find('#gcbChatSkipRegistration');
                $skipBtn.on('click', function() {
                    chat.close();
                    return false; // prevent default behavior
                });
            }
        }
    }
};
<script>
```

Now, when the user clicks on the **Exit** button, the chat widget closes.

Procedure: Automatically expand text area based on user input

Purpose: In this example we automatically expand the text input area in the chat widget as the user enters text.

Customization Type: JavaScript-based, template-based

Prerequisites

- You must have basic knowledge of JavaScript.
- This example assumes you are using chat as part of the Integrated JavaScript Application.
- The example uses the jQuery library.

Steps

1. Access the `textarea` element using the ui.onBeforeChat hook:

```
<script>
var _genesys = {
    chat: {
        ui: {
            onBeforeChat: function(html) {
                var $textarea = jQuery(html).find('textarea');
            }
        }
    }
};
</script>
```

2. Add a listener to the `textarea`. When the `textarea` receives input, the listener checks the `textarea` size and increases the size as needed:

```
//...
var $textarea = jQuery(html).find('textarea');
$textarea.on('input', function () {
    if (this.clientHeight < this.scrollHeight) {
        this.style.height = this.scrollHeight + 'px';
    }
});
```

Important

This example uses the `input` event which may be unavailable in older browsers. You may want to use `keyup` or another event instead.

3. At this point, when the `textarea` grows it covers the message area. We fix this by shrinking the messages as we expand the `textarea`:

```
//...
if (this.clientHeight < this.scrollHeight) {
    this.style.height = this.scrollHeight + 'px';
    $('.gwc-chat-message-container').css({
        bottom: this.scrollHeight + 25
    });
}
```

4. As a final adustment, we limit how large the `textarea` can grow. We set a height limit of 150 px:

   ```
   if (this.clientHeight < this.scrollHeight && this.scrollHeight < 150) {
     //...
   ```

   The full code snippet now looks like this:

   ```
   <script>
    var _genesys = {
        chat: {
            ui: {
                onBeforeChat: function(html) {
                    var $textarea = jQuery(html).find('textarea');
                    $textarea.on('input', function () {
                        if (this.clientHeight < this.scrollHeight && this.scrollHeight
    < 150) {
                            this.style.height = this.scrollHeight + 'px';
                            $('.gwc-chat-message-container').css({
                                bottom: this.scrollHeight + 25
                            });
                        }
                    });
                }
            }
        }
    };
    </script>
   ```

startChat(options)

Method	Options	Options (cont'd)
• Description • Returned Promise • done • fail	• serverURL • embedded • transport • localization • templates • widgetUrl • windowSize • windowName • windowOptions • debug	• logger • registration • Custom registration • userData • createContact • ui • hooks • maxOfflineDuration • disableWebSockets

Description

This is the main entry point for configuring and starting a chat session.

Returned Promise

`startChat` returns a "promise" object with two chainable methods: done and `fail`.

> **Important**
> Currently, the promise is resolved or rejected only if the chat is started in "embedded" mode. If it is started in separate window ("popup" mode), you have to use promise returned by `startChatInThisWindow` method to get access to the Chat Session API.

done

Use this method to get access to the chat session service API as resolved with an instance of the session object.

```
chat.startChat(options).done(function(session) {
  // session.sendMessage, session.onAgentConnected and all other method are at your disposal.
});
```

Tip

See Chat session commands for Chat Session API documentation . Note that if you need to access the Chat Session API, you will probably want to get access not only in cases when the session is started, but also when it is restored. You can use `restoreChat`'s "done" callback for that. See Getting Access to Chat Session API for more info.

Important

The promise is never resolved before the chat session is created. This means that if registration is enabled, the **done** callback will not fire until the registration is complete and processed by the server.

fail

Resolved with an event containing an error describing what went wrong.

Event structure

Parameter	Meaning
event.error.code	Code of error
event.error.descpription	Description of the error (English is default language).

Tip

For a list of possible error codes, see Error Codes.

Chat widget error codes

In addition to the regular list of possible errors, the Chat Widget adds some of its own errors:

Error code	Error description
Chat widget-specific error codes (range 200 -249)	
200	Chat is already running on this page.
201	Chat is already running on another page.

For example,

```
chat.startChat(options).fail(function(event) {
    if (event.error === 201) {
        alert('Chat is already running on another page');
    }
});
```

Options

serverUrl

Type	Default Value	Mandatory	Description
string	undefined	Yes (except when transport options is provided)	URL of the CometD chat server for default (built-in) CometD transport.

embedded

Type	Default Value	Mandatory	Description
boolean	false	No	Sets chat mode of operation: "embedded" (chat widget is rendered directly on a page) or "popup" (chat opens in a separate browser window). Default is "popup". Pass the value `true` to switch to "embedded" mode.

Important

If chat is configured for embedded mode, the chat widget will disappear as soon as the user leaves the website or navigates to any non-instrumented web page. If the user returns to the page before the timeout has expired, chat will automatically be restored. To configure the timeout, use the maxOfflineDuration option. The `maxOfflineDuration` options may be pre-configured when you use chat as part of a particular solution such as the Integrated JavaScript Applicaiton

transport

Type	Default Value	Mandatory	Description
Object	undefined	No (except when serverUrl is omitted)	Custom transport instance (for example, REST-based).

localization

Type	Default Value	Mandatory	Description
Object, string, or function(function?)	undefined	No	Provider of customer localization. This value can be one of the following: • JavaScript object with localization data. Added in **850.0.0**. • Function that returns an object with localization data. Added in **850.4.0**. • Function that accepts a callback and calls that callback with an object containing localization data. Added in **850.4.0**. • URL of and external JSON file with the localization datav If omitted, default English localization will be used. See Localization for more on how to localize the chat widget. Important If you use Template-based Customization, all localization data is available inside your templates as a `data.nls` object.

templates

Type	Default Value	Mandatory	Description
string	none	No	The URL of the HTML file containing templates used to render the chat widget. The request is made using either JSONP or AJAX, following the same logic for localization files (see Localization). Default templates are included into the JavaScript source so by default, there are no requests made to load them. For more information, see Template-based Customization. Important Since chat version **850.5.0** you do not necessarily have to download the custom templates over the network. See Template-based Customization for more.

widgetUrl

Type	Default Value	Mandatory	Description
string	undefined	No (except when embedded is set to `false` - "popup" mode)	URL of chat widget html that will be open in external window when operating in "popup" mode.

windowSize

Type	Default Value	Mandatory	Description
Object {width: number, height: number}	{ width: 400, height: 500 }	No	Size of external chat window when operating in "popup" mode. Important Note that `windowOptions` can override `windowSize`.

windowName

Type	Default Value	Mandatory	Description

string	`genesysChatWindow`	No	A string representing the new window that is passed to the `window.open` call when opening the chat widget window.
			
			Important
			`windowName` does **not** specify the title of the new window.
			
			Tip
			This option only works in "popup" mode (embedded is either absent or set to `false`)
			
			For example,
			
			`chat.startChat({`
			`  windowName: 'myWindowName'`
			`  //...`
			`});`
			`// => window.open(<widgetUrl>, 'myWindowName', ...);`

windowOptions

Type	Default Value	Mandatory	Description

Object	value of `windowSize` option	No	An object containing the options that will be passed `window.open` when opening a chat widget window.
			
			Important
			This option only works in "popup" mode (embedded is either absent or set to `false`)
			
			Use this object to pass any window options, such as position (`top, left`), whether to show browswer buttons (`toolbar`), location bar (`location`), and so on. See Window.open for the full list. All options are converted to a string that is passed to `window.open` call. For example,
			
			`chat.startChat({`
			`    // open chat widget in top left corner of the screen`
			`    windowOptions: {`
			`        left: 0,`
			`        top: 0`
			`    },`
			`    // ...`
			`});`
			`// => window.open(<widgetUrl>, <windowName>, 'left=0,top=0,...')`
			
			Note that `windowOptions` is merged with `windowSize`, but has higher priority. For example,
			
			`chat.startChat({`
			`        windowSize: {`
			`                width: 200,`
			`                height: 400`
			`        },`
			`        windowOptions: {`

Type	Default Value	Mandatory	Description
			```
			                left: 0,
			                top: 0,
			                width: 300 //
			this value will be used, and
			height will be taken from
			windowSize
			        },
			        // ...
			});
			// => window.open(<widgetUrl>,
			<windowName>,
			'left=0,top=0,width=300,height=400')
			```

debug

Type	Default Value	Mandatory	Description
boolean	false	No	Pass the value `true` to enable chat debugging logs (by default standard `console.log` is used, see the logger option if you want to override that).

logger

Pass a function that will be used for chat logging (if debug is set to `true`) instead of the default `console.log`. The function has to support the interface of the `console.log` — it must accept an arbitrary number of arguments and argument types.

> **Important**
> To use the custom logging function in a separate window, you have to pass it directly on the widget page to the `startChatInThisWindow` method.

registration

Type	Default Value	Mandatory	Description
(boolean|function)	true	No	By default chat starts with a built-in registration form (that you can customize using `ui.onBeforeRegistration`). Pass the value `false` to disable this default built-in registration form.

Custom registration

Pass a function to customize registration workflow.

The function accepts one argument: a done function that must be called with an object containing the data collected during registration.

This may sound complex but actually it is pretty straightforward:

```
chat.startChat({
  registration: function(done) {
    done({
      EmailAddress: 'john.doe@example.com'
    });
  }
  //...
});
```

In the example above, it is assumed that all data is known beforehand and so registration may be completed synchronously and in a way that hides the operation from the end user. However, in reality you may want to send an additional request to obtain the necessary data:

```
chat.startChat({
  registration: function(done) {
    // Suppose you have a special URL that returns current user's credentials that you want
to pass to chat session:
    jQuery.get('/account/credentials', function(data) {
      done(data);
    });
  }
  //...
});
```

You may have noticed that both of these examples are artificial, in the sense that they do not provide any UI but simply silently register the user with already available data. To provide a registration UI, you will have to return the DOM object representing your UI from your custom registration function. For example, like this:

```
chat.startChat({
  // Simple jQuery-based example
  registration: function(done) {
    var $form = $('<form />'),
        $email = $('<input type="email" name="email" placeholder="Enter your Email" />');

    // bind done function to be called when the form is submitted
    $form.on('submit', function() {
      done({
        EmailAddress: $email.val()
      }};
    });

    $email.appendTo($form);

    // return form DOM representation: it will be displayed in the chat widget
    return $form.get(0);
  }
  //...
});
```

Although the example above may seem complicated, this approach is very powerful as it allows you to reuse any JavaScript stack you use on your site, be it jQuery, client-side templating, more full-featured frameworks like AngularJS or any other JS-based technology or their combinations.

userData

Can be used to directly attach necessary UserData to a chat session. See Custom Registration and Extended API to support integrated solutions for other ways of working with UserData.

```
chatAPI.startChat({
    userData: {
        visitID: currentVisitID
    }
});
```

createContact

Type	Default Value	Mandatory	Description

boolean	true	No	Determines whether new contact should be created from registration data if it doesn't match any existing contact. Only effective if registration data is present (collected either by built-in or custom registration workflow).
			
			`// createContact is not effective. Contact will not be created.`
			`chat.startChat({`
			`  registration: false`
			`  // ...`
			`});`
			
			`// Contact will be identified and if doesn't exist, it will be created.`
			`chat.startChat({`
			`  createContact: true,`
			`  // ...`
			`});`
			
			`// Contact will be identified; if doesn't exist, it won't be created.`
			`chat.startChat({`
			`  createContact: false,`
			`  // ...`
			`});`
			
			Tip
			Technically, this option controls the lookup (and possibly the creation) of client's record in the UCS database. By default,

Type	Default Value	Mandatory	Description
			• If registration is disabled (or provides no data, which can be the case with custom registration function), the contact **will not be looked up** in UCS; • If registration is enabled and provides some data (in fact any data, since we do not validate on the browser side), the contact **will be looked up** in UCS and if not found, it **will be created**. • However, if `createContact` is set to `false`, the contact **will be looked up** in UCS and if not found **it will not be created**.

ui

Type	Default Value	Mandatory	Description

(boolean|Object)	true	No	Pass the value `false` to disable the chat widget UI completely. Or pass an object with "hook" functions that can modify the built-in UI.
			
			All "hooks" receive an object: a DOM representation of a particular UI part, which you can then modify. The structure of these objects can be determined using browser developer tools (F12 in Chrome) and may be modified in future versions of chat widget.
			
			Important
			Backward compatibility of the DOM structure provided in the UI hooks is NOT guaranteed.
			
			`function makeEverythingRed(htmlElement) {`
			
			`jQuery(htmlElement).find('*').css({ color: 'red' });`
			`}`
			`function makeEverythingBold(htmlElement) {`
			
			`jQuery(htmlElement).find('*').css({ 'font-weight': 'bold' });`
			`}`
			
			`chat.startChat({`
			`  ui: {`
			`    onBeforeChat: makeEverythingRed,`
			`    onBeforeRegistration: makeEverythingRed,`
			`    onBeforeMessage: makeEverythingBold`

Type	Default Value	Mandatory	Description

			```
			  },
			  // ...
			});
			```
			 Tip
			See more examples in the Customizing the User Interface section.

Available "hooks" are:

Hooks	Description
onBeforeChat	Sent before the "main" chat UI is rendered (messages area and message input field). Can be used to modify the layout / functionality of the chat widget.
onBeforeRegistration	Sent before the registration form is rendered (and only if it is enabled).
onBeforeMessage	Sent before every message that gets appended to the chat. For example, messages from user, agent, system, typing, and so on are all included. This hook has a special ability of "filtering out" certain messages in the chat. If the function attached to it explicitly returns `false`, that particular message is not added to the UI. This is useful for passing internal or system data using the the chat channel. To ease message analysis, the hook function receives a second argument: the message text. Tip For example usage, see: • Using the ui.onBeforeMessage Hook to Add Desktop Modifications

maxOfflineDuration

Type	Default Value	Mandatory	Description
number	5	No	Time (in seconds) during which state cookies are stored after page reload/navigation. If cookies expire, the chat is not restored. This option specifies how long the chat session will live after the user leaves the website. A value of five seconds is usually long enough for chat to survive page reloads, but the chat interaction is likely to be lost if the user navigates to another site during the chat interaction (in *embedded* mode) and comes back to your site. To support this kind of workflow, consider increasing the value of this option or use the Integrated JavaScript Application, where the option value is increased to 600 seconds by default.

disableWebSockets

Type	Default Value	Mandatory	Description
boolean	false	no	You may disable WebSockets for chat by passing `true` to this option. By default, chat attempts to use WebSockets for connections to the server. When WebSocket connections are unavailable ,such as when the load balancer does not support WebSockets, chat switches to other HTTP-based means of

Type	Default Value	Mandatory	Description
			communication. Disabling WebSockets can speed up the time it takes for chat to switch to another means of communication. Important If you choose to disable WebSockets, you should also pass this option to restoreChat(options). This option is only effective with the default (built-in) transport.

restoreChat(options)

Description

Use this method to restore the chat widget after page reload/navigation.

> **Important**
> Currently, this method only works for "embedded" mode.

If the previous page was unloaded during the registration phase, `restoreChat` will try to restore the chat widget with the registration form.

Returned promise

`restoreChat` returns a "promise" object with two chainable methods: done and `fail`.

done

This method can be used to get access to chat session service API.

```
chat.restoreChat(options).done(function(session) {
  // session.sendMessage, session.onAgentConnected and all other methods are at your disposal.
});
```

See Chat Service JS API for documentation about using the session API. If you need to access Chat Session API, you will probably want to get the access not only in cases when session is restored, but also when it is started fresh. You can use `startChat`'s "done" callback for this. See Getting access to Chat Session API for more info.

> **Important**
> The promise is not resolved before the chat session is created. This means that if the chat widget is restored during the registration phase (the registration form is displayed to the user), the **done** callback is not sent until the registration is complete and processed by server.

fail

If chat restoration fails because of an error (and not because the chat session does not exist), the **fail** callback receives an event argument with an error property, similar to the `startChat().fail` callback.

```
chat.restoreChat(options)
  .fail(function(event) {
    // If there was chat session, but restoration fails, signal failure.
    if (event.error) {
        alert('chat restoration failed');
        return;
    }
    // If there was no chat session, bind start chat to "start chat" button
    jQuery('#myChatButton').on('click', function() {
        chat.startChat(startChatOptions);
    }
  })
  .done(function(session) {
    // session.sendMessage, session.onAgentConnected and all other method are at your
disposal.
  });
```

Tip

For a list of possible error codes, see Error Codes.

Options

Some chat states are restored automatically after page reload/navigation. However, most options must be passed to `restoreChat` directly. Supported options are:

- embedded — Must be explictly passed as `true`. Otherwise an error occurs.
- transport — Include if providing a custom transport.
- registration — If you are using a custom function for registration and you want this registration to be restored, you must pass this option to both `startChat` and `restoreChat`.
- ui — If you want to customize/disable the chat UI, pass this option to both `startChat` and `restoreChat`.
- localization — For localization, pass the custom localization URL to both `startChat` and `restoreChat`.
- debug — If you want logs enabled, you must pass this option to `restoreChat` explicitly.
- logger
- maxOfflineDuration
- disableWebSockets

startChatInThisWindow(options)

Description

Use this method in a separate window ("popup" mode) to render a chat widget that occupies the whole page. Normally, if you use default provided `chatWidget.html`, you never have to use this method.

Returned promise

This method returns a promise similar to the one returned by startChat.

Options

Most of the options are extracted from the URL, which is formed by the `startChat` method, if called to create a chat in "popup" mode. However, some options cannot be passed in the URL but must be passed directly. These include:

- ui — Use to pass an object with "hooks" to customize the built-in UI.
- registration — Use to pass a function with custom registration workflow.
- transport — Use in case you provide a custom transport.
- logger

onBeforeChatOptionsApplied(callback)

Description

Important

This method may not be available in older versions of chat shipped with Genesys Web Engagement 8.1.

Use this method to modify options before the chat starts or restores. This method may be useful if you do not control the `startChat()` call but still need to control some of the options. For example, you can use this method when you use chat as part of the Integrated JavaScript Application.

Example

```
_genesys.chat.onReady.push(function(chat) {
  chat.onBeforeChatOptionsApplied(function(options) {
    // you can modify options object here, for example:
    if (userIsAuthorized) {
      options.registration = false;
    }
  });
});
```

onSession(callback)

Description

Important

This method may not be available in older versions of chat shipped with Genesys Co-browse 8.1 and Genesys Web Engagement 8.1.

This is an alternative method of accessing the Chat Service JS API. This method is an alternative to accessing the Chat Service JS API through the done callbacks of the `startChat` and `restoreChat` methods. This method may be useful if you do not control the `startChat` call, for example, when you use chat as part of the Integrated JavaScript Application.

Example

```
_genesys.chat.onReady.push(function(chat) {
  chat.onSession(function(session) {
    session.sendMessage('Automatically sent chat message');
  });

});
```

close()

This method is called if you programmatically close the chat widget. Only works for "embedded" mode.

Important
Added in **850.5.0.**

toggle()

Use this method to minimize/restore the chat widget. Only works for "embedded" mode.

Important

Added in **850.6.0**.

onMinimized(callback)

Use this method to track when chat widget is minimized or restored. The callback is called with `true` when chat is minimized, and with `false` when chat is restored. Only works for "embedded" mode.

```
chat.onMinimized(function(isMinimized) {
  if (isMinimized) {
    // chat was minimized
  } else {
    // chat was restored
  }
});
```

Important

Added in **850.6.0**.

isMinimized()

Use this method to check if the chat widget is minimized. Returns `true` if the chat widget is rendered and minimized, `false` otherwise. For example, `false` is returned if the chat widget is not rendered at all or if the chat widget is rendered but not minimized. Only works for embedded mode.

```
if (chat.isMinimized()) {
  // do something
}
```

Important

Added in **850.6.0**.

VERSION

This method returns the current chat version, for example, `850.1.0`.

Version consists of three parts:

1. The first part, `850` in the example, matches the general Genesys version. For example, chat shipped with Web Engagement 8.5.0 will always have version 850.X.X.
2. The second part, `1` in the example, is incremented after each major API change such as when a method is added.
3. The third part, `0` in the example, is incremented after minor changes and bug fixes.

Important
Added in **850.1.0**.

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
});
```

Launching the Chat Session

There are two methods you can use to launch chat session:

- startSession — Use this entry-point method to start a new chat session, with various options for controlling the shape of that session.
- restoreSession — Use this entry-point method to launch an existing chat session. For example, in cases where a browser page with a chat window is reloaded.

startSession

Description

Entry-point method for creating new chat session.

Returned Promise

`startSession` (as well as all session commands) returns a "promise" object with two chainable methods: done and `fail`.

done	This method should be used to get access to the chat session object.
	
	`chat.startSession(options).done(function(session)`
	`{`
	`  // session.sendMessage,`
	`session.onAgentConnected and all other`
	`method are at your disposal.`
	`});`

fail	Resolved with an event containing an error message describing what went wrong. Event structure: **Parameter** | **Meaning** event.error.code | Code specifying the particular error event.error.description | Description of error (English is default language). Tip For a list of possible error codes, see Error Codes.

Parameter	Meaning
event.error.code	Code specifying the particular error
event.error.description	Description of error (English is default language).

Options

Options are expected as a single object with named properties. For example:

```
chat.startSession({
    serverUrl: '...',
    username: 'Anonymous'
});
```

Option	Type	Default Value	Mandatory	Description
`serverUrl`	string	undefined	Yes (except when transport option is provided)	URL of the CometD chat backend if built-in CometD transport is to be used.
`username`	string	'User'	No	Optional parameter. Name (nickname) of the visitor who initiated the chat session. If absent, and chat user is anonymous, default value is used.
`subject`	string	undefined	No	Subject of the chat session. If option is absent, parameter is left empty.
`userData`	Object	undefined	No	Represents a set of parameters that can\should be specified when creating the chat session. Typical use-case: providing data obtained from a registration form. `chat.startChat({ // Passed data will be used to identify and/or create user contact. userData: { FirstName: 'John', LastName: 'Doe', EmailAddress: 'johndoe@example.com' } });`

Option	Type	Default Value	Mandatory	Description
				Important Data you pass to `userData` is always merged with the default values of `userData`. `userData` contains the following keys: • `source` with value `web` • `pageTitle` with value set to current page's title, `document.title` • `url` with value set to the current page's URL, `document.location` • `referrer` with value set to the previous page's URL, `document.referrer` If you need to override and of these values, pass the values along with other `userData`. For example, the following sets `source` to `null` and leaves `startPage` as is: `chat.startChat({ userData: { FirstName: 'John', LastName: 'Doe', EmailAddress: 'johndoe@example.com', source: null } });`
`transport`	Object	[built-in CometD transport]	Only if `serverURL` is absent	Pass custom transport instance here, if needed (for example, REST-based). By default, built-in CometD chat transport is used.
`stateStorage`	{{read: function(): Object, write:	[internal built-in stateStorage]	No	Provides a method for dealing with

Option	Type	Default Value	Mandatory	Description
	function(Object|null)}}			session state persistence across page reloads. By default, built-in cookie-based state storage is used. Note that in certain cases, the `stateStorage.write` method is called with a special argument, `null`, which means that all existing state has to be cleared. For details, see Using a third-party mechanism for chat session state persistence.
`maxOfflineDuration`	number	5	No	Time (in seconds) during which session state cookies are stored after page reload/ navigation. Default is 5 seconds. If cookies expire, the chat session will not be restored via `restoreSession()`. This option only takes effect on default (built-in_ state storage. If custom state storage is passed, this option does not take effect.
`disableWebSockets`	boolean	`false`	No	Disable WebSockets for connections to the server. Only effective if the default transport is used. If you disable WebSockets, you should also pass this option to restoreSession().
`logger`	function	[internal console.log-based logger]	No	A function responsible for logging. If you pass a custom logging function to this option, the

Option	Type	Default Value	Mandatory	Description
				function should accept an arbitrary number of different argument types (similar to console.log).

restoreSession

Description

Entry-point method for launching an already existing chat session (for example, when a browser page with a chat window was reloaded).

Returned Promise

`restoreSession` returns a "promise" object with two chainable methods: done and `fail`.

done	This method should be used to get access to chat session object.
	
	`chat.restoreSession(options).done(function(session)`
	`{`
	`  // session.sendMessage,`
	`session.onAgentConnected and all other`
	`methods are at your disposal.`
	`});`

fail	If the restore chat operation fails because of an error (and not because the chat doesn't exist), the fail callback receives an event argument with an error property similar to `startSession().fail callback`.
	
	`chat.restoreSession(options)`
	`  .fail(function(event) {`
	`    // If there was a chat session, but`
	`restoration fails, signal failure.`
	`    if (event.error) {`
	`        alert('chat restoration failed');`
	`        return;`
	`    }`
	`    // If there was no chat session, bind`
	`start chat to "start chat" button`
	`    jQuery('#myChatButton').on('click',`
	`function() {`
	`        chat.startChat(startChatOptions);`
	`    }`
	`  })`
	`  .done(function(session) {`
	`    // session.sendMessage,`
	`session.onAgentConnected and all other`
	`method are at your disposal.`

	`});` Tip For a list of possible error codes, see Error Codes.

Options

Options are expected as a single object with named properties. For example:

```
chat.restoreSession({
    maxOfflineDuration: 60
});
```

Option	Type	Default Value	Mandatory	Description
`transport`	Object	[built-in CometD transport]	Yes, if custom transport was passed in `startSession`	Pass custom transport here if needed (for example, REST-based). By default built-in CometD chat transport will be used.
`stateStorage`	{{read: function(): Object, write: function(Object|null)}}	[internal built-in stateStorage]	No	Provides a method for dealing with session state persistence across page reloads. By default, built-in cookie-based state storage is used. Note that in certain cases, the `stateStorage.write` method is called with a special argument, `null`, which means that all existing state has to be cleared. For details, see Using a third-party mechanism for chat session state persistence.
`maxOfflineDuration`	number	5	No	Time (in seconds) during which session state cookies are stored after page reload/ navigation. Default is 5 seconds. If cookies expire, the chat session will not be restored via `restoreSession()`. This option has no

Option	Type	Default Value	Mandatory	Description
				effect on whether custom `stateStorage` is passed.
`disableWebSockets`	boolean	`false`	No	Disable WebSockets for connections to the server. Only effective if the default transport is used. If you disable WebSockets, you should also pass this option to startSession().
`logger`	function	[internal console.log-based logger]	No	A function responsible for logging. You can pass an arbitrary number of different argument types (similar to console.log). For example: console.log.bind(console).

Tip

To find out how to control the chat session once it is started, see Controlling the chat session.

Controlling the Chat Session

Chat session commands

Session commands are used for client-to-server communication: sending commands to chat server. All commands receive their arguments as a single "options" object, similar to startSession and restoreSession. For example,

```
session.getTranscript({fromIndex: 0});

// instead of
session.getTranscript(0);
```

All exceptions to this rule are documented here.

Returned Promises

All of the commands return a "promise" object with two properties: done and `fail`.

Some of the "done" callbacks can be used to obtain specific information provided as the result of command execution. If that is the case, the information is specifically documented for particular commands. Most of the time however these callbacks serve as a way to simply acknowledge that the command was successfully executed. In this case, there is no specific documentation for the corresponding command callback.

"fail" callbacks can be used to catch errors that happen during command execution and all receive an object with exact same structure:

event.error.code	Code specifying the particular error
event.error.description	Description of the error (English is default language).

There is no specific documentation for each command's "fail" callback.

Here is an example of general command usage:

```
chat.<ABSTRACT_CHAT_COMMAND>(<COMMAND_OPTIONS>).done(function(<POSSIBLE_RESULTS>) {
  // Command executed successfully.
}).fail(function(event) {
  // Something went wrong. See event.error.code and event.error.description.
});
```

session.getTranscript

Description

A low-level method used to obtain the transcript for the current session.

Important
No playback is assumed.

Options

Parameter	Type	Default value	Mandatory	Description
`fromIndex`	number	0	No	0-based index to retrieve chat transcript from a certain position. `0` means obtaining the entire transcript.

"done" callback

Receives the transcript in serialized JS form.

Parameter	Type	Description

`event.transcript`	Array	Array of JS objects representing transcipt events.
		Sample:
		`[`
		`    {`

Parameter	Type	Description

		type: 'AgentConnected',
		party: {id: 2 /*ID of this party unique for THIS chat session*/, type: 'Agent', name: <name of an agent>},
		index: <index of THIS message in chat transcript>,
		timestamp: <UTC Timestampt>
		},
		{
		type: 'MessageReceived',
		party: {id: 1, type: <type of party which sent message: 'Client' or 'Agent' or 'External'>, name: <name of party>},
		content: {text: <text which was sent>, type: <type of text: 'text' or 'url'>},
		index: <index of THIS message in chat transcript>,
		timestamp: <UTC Timestampt>
		},
		{
		type: 'AgentDisconnected',
		party: {id: 2 /*ID of this party unique for THIS chat session*/, type: 'Agent', name: <name of an agent>},
		index: <index of THIS message in chat transcript>,
		timestamp: <UTC Timestampt>
		},
		{
		type: 'SessionEnded',
		reason: {code: <code of reason: 1: by leave request, 2 - by an agent, 3 - by error>, description: <default description>},
		timestamp: <UTC Timestampt>
		}
		]

```
session.getTranscript().done(function(event) {
  console.log('Full transcript of current chat session: ', event.transcript);
});
```

session.sendMessage

Description

Send a message to Chat Server.

Options

Parameter	Type	Default value	Mandatory	Description
`type`	string	'text'	No	Type of message: "text" or "url" If absent, "text" will be used.
message	string	undefined	Yes	Message to be sent

Since sending a "text" message is a much more frequent operation than sending a "url", a shortcut is available; you can pass a string with message contents directly to the method instead of the options object.

```
// This
session.sendMessage({ type: 'text', message: 'foobar' });

// is equivalent to this:
session.sendMessage('foobar');
```

session.sendTyping

Description

Notify Chat Server that client started or stopped typing in chat session.

Options

Parameter	Type	Default value	Mandatory	Description
isTyping	boolean	true	No	Boolean (true or false) which specifies exact meaning of this command • **true** — visitor started typing (or continues typing) in chat session. • **false** — visitor stopped typing

Parameter	Type	Default value	Mandatory	Description
				in chat session (typically a stop or pause in typing for a certain duration, for example — 5 seconds)

session.leave

Description

This command is used to complete chat session in Chat Server by request from the visitor side.

Options

This command does not take parameters.

Chat session events (callbacks)

You can pass callback functions into a chat session that will be called each time a chat context is updated, or whenever other changes take place within the session (for example, agent joins/leaves, Chat Server stops responding, and so on). Another use for session events is in a "playback" scenario, when a chat session is restored in a new browser context (for example, after page reload/navigation).

To add a callback, pass the callback function directly to the corresponding event method. Most of the callbacks receive an event object with properties containing event details. For example,

```
session.onError(function(event) {
    // event.error.code
    // event.error.description
});
```

session.onError

This event will be sent in reaction to an unexpected error occurring during the flow of the chat session.

Additionally, this event is sent when the chat component needs to notify clients about unusual cases. For example:

- Chat Server stops responding and the component tries to restore the session on another server.
- A chat session is restored on another instance of Chat Server.
- A channel to the Chat Server is opened, but the server is not yet ready to send/receive operations.

Another important scenario for this callback — it is triggered if an incorrect set of parameters (or invalid parameter value) is detecing in an incoming "raw" event. For example, if the `content` of a message in a `received message` event is not a string.

Event structure

Parameter	Description
event.error.code	Code specifying the particular error
event.error.description	Default description of error

Tip

For a list of possible error codes, see Error Codes.

session.onAgentConnected

Executed when an agent joins a chat session.

Event structure

Parameter	Description
`event.party.name`	String that represents name of the agent joined to the chat session
`event.timestamp`	Number
`event.party.id`	Theoretically, the agent name and visitor name could be the same (especially since the visitor might be filling out the the visitor name). To handle this scenario, this id is used to distinguish between agent and visitor names.
`event.index`	Index of this message in chat transcript
`event.restored`	Optional. If present and true, it means that the event was restored during session restoration (in other words, event was already reported to the consumer previously).

session.onAgentDisconnected

Executed when agent leaves chat session, for any of these possible reasons:

- Session closes because of a logout request from the Chat Widget side.
- Agent leaves the conference.
- Agent transfers the session to another agent.
- agent's desktop stops responding.
- Chat server stops responding and session is restored on the new chat session.

Event structure

Parameter	Description
`event.timestamp`	Number
`event.party.name`	String that represents the name of the agent leaving chat session.
`event.party.id`	String with ID of party in chat session.
`event.index`	Index of this message in chat transcript
`event.restored`	Optional. If present and true, it means that the event was restored during session restoration (in other words, was already reported to the consumer previously).

session.onMessageReceived

Executed when a new message appears in the chat transcript (or in the context of the "playback" process during chat session restoration).

Important

Messages sent by `session.sendMessage` are "returned back" via this event as well.

Event structure

Parameter	Description
`event.index`	0-based index of this message in chat session transcript
`event.timestamp`	Number
`event.content.text`	String with message added to the chat context

Parameter	Description

`event.content.type.url`
OR `event.content.type.text`
	One of these is true depending on message type. Usage example:
	```
	session.onMessageReceived(function(event) {
	    if (event.content.type.url) {
	        // this is "url" message
	    } else if (event.content.type.text) {
	        // this is "text" message
	    }
	});
	```

`event.party.id`	Theoretically, the agent name and visitor name could be the same (especially since the visitor may be the one filling out the visitor name). To handle this scenario, this ID is used to distinguish between agent and visitor.

`event.party.type.agent`
OR
`event.party.type.client`
OR
`event.party.type.external` OR
`event.party.type.supervisor`
	One of these is set to true, depending on who sends the message.
	• **agent** — message is sent by the agent;
	• **client** — message is sent by the visitor (in other words, the actual user of this API via `session.sendMessage()`)
	• **external** — message is sent by the system (for example, as configured in the routing strategy)
	• **supervisor** — message is sent by the supervisor

`event.party.name`	String with name of party.
`event.restored`	Optional. If present and true, it means that the event was restored during session restoration (in other words, event was already reported to the consumer).

session.onAgentTyping

Executed when agent starts, continues, or stops typing.

Event structure

Parameter	Description
`event.party.id`	ID of party in chat session.
`event.party.name`	Name of agent.
`event.isTyping`	• **true** — when agent starts (or continues) typing;

Parameter	Description
	• **false** — when agent stops typing.

session.onInterrupted

Executed when a connection to the server is interrupted (for example, a network interruption or server is down). If default transport is used, chat tries to automatically reconnect and fires `session.onContinued` when the connection is restored.

Callbacks receive no event object.

Important
This event duplicates the `session.onError` event with the "150" (network interruption) Error code. Genesys recommends that you use `session.onInterrupted/onContinued` if you want to track the connection status. The "150" error code is deprecated and might be removed in future versions.

session.onContinued

Executed only after `session.onInterrupted` when the connection to the server is restored.

Callbacks receive no event object.

session.onSessionEnded

Executed when the client drops out of a chat session due to one of the following reasons:

- Logout request.
- Chat session is finished from the agent's side.
- Chat Server stops responding, resulting in a timeout during which the session is not restored.

Parameters

Parameter	Description
`event.reason.error` OR `event.reason.agent`	One of these is set to true depending on the reason • **error** — a major error occurs (for example, Chat

Parameter	Description
OR `event.reason.leaveRequest`	Server stops responding) • **agent** — agent ends the session; • **leaveRequest** — visitor sends a session.leave command and it was successful

Miscellaneous methods

session.isAgentConnected

Provides a convenient way to synchronously determine if at least one agent is present in the session.

Returns boolean (true or false).

Error Codes

The Chat API provides stable error codes, which can come from either the server or browser side. These error codes are used in EventError events.

This table lists the possible error codes and their descriptions

Error code	Error Description
Chat command syntactical error codes (range 1 - 49)	
1	Unknown chat command
2	One or more mandatory parameters are missed in chat command
3	Value of chat command parameter is incorrect
Chat session run-time error codes (range 50-99)	
50	Chat session ID is unknown
51	Chat command is stopped by decorator
52	Error during chat command execution
Chat server-related error codes (100-149)	
100	Chat Server is not available
101	Login to Chat Server was failed
Chat transport error codes (150-199) (generated on browser side)	
Chat widget error codes (200-249) (generated on browser side)	

Chat Transport Error Messages

This event will be sent in reaction to an unexpected error occurring during the flow of the chat session.

Additionally, this event is sent when the chat component needs to notify clients about unusual cases. For example:

- Chat Server stops responding and the component tries to restore the session on another server.
- A chat session is restored on another instance of Chat Server.
- A channel to the Chat Server is opened, but the server is not yet ready to send/receive operations.

The server generates most of these errors, which the JS transport then passes to the calling code as is. However, there are cases when the JavaScript is responsible for generating an error event — for example, when the connection with server is completely lost.

Here is a list of this kind of JS-generated error:

Error code	Error Description	Comments
Chat transport errors (range 150-199)		
150	Network connection interrupted	Important This error is deprecated and might be removed in future versions. Use `session.onInterrupted/ onContinued` events instead. The transport should report this error when the connection is lost for a period of time. For the default CometD transport, this is a configurable interval with a default value of 3 seconds. Other transports might want to keep to this default.
151	Invalid server response.	This error is not reported by the transport itself, but by upper-level code that validates server responses. You do not need need to reimplement this if you are developing custom transport.
152	Session restoration failed.	This error happens when a chat session is restored from client state (cookie or another), but the session has already expired on the server.

Notification Service REST API

Description

The Notification Service REST API is enables you to keep users of your website engaged and informed. You can reach your entire user base quickly and effectively with notifications that are delivered to your web pages. The Notification Service is responsible for delivering the push messages (events) to the Browser Tier. On the Browser Tier, each message is handled by the Notification Agent (a module of the Tracker Application). The Notification Agent provides a list of predefined messages that can be used in engagement scenarios. Each message uses the "gpe." prefix in the channel name.

The Notification Service REST API has two methods:

- Notify by Visit ID — delivers notifications from the GWE Server to the browser tier based on the visit ID (through CometD).
- Send Disposition Code for Invitation from Browser to GWE Server — delivers notification disposition (result) from the browser tier to the GWE Server (through HTTP GET).

Notify by Visit ID

Description

This method performs an engagement notification through the Web Engagement Server by visit ID. If the request is performed correctly, then the Web Engagement Server creates and sends a notification message to the CometD `/notification/{visitID}` channel.

Request

Method	POST		
Body	JSON array of Notification Messages		
URL	http://<gwe_server_host:gwe_server_port>/server/data/notification/notify/{visitID} The HTTPS schema is also allowed, if configured.		
Parameters			
Name	**Value**	**Mandatory**	**Description**
visitID	string	yes	The visit ID for the engagement notification.

Response

Response	Standard HTTP Responses
Response Type	none

Example

Request

```
curl -X POST \
  -H "Content-Type: application/json" \
  -d '[{"page": "654E0122924F42EA82B5C581A394555D", "channel":
"gpe.appendContent", "data": { "url": "http://www.genesys.com/invite.html",
"method": "GET", "container": "body" }}]' \
  http://server:port/server/data/notification/notify/654E0122924F42EA82B5C581A394555D
```

Notification Messages

The notification request body is represented by an array of JSON objects, one object for each notification message:

```
[
    {
        // message 1
    },
    {
        // message 2
    },
    ...
]
```

Message Structure

Notification messages should follow this format:

```
{
   "page":"yyy",
   "channel":"any-string",
   "data":{
      "someKey":"someValue"
      ...
   }
}
```

Option	Type	Mandatory	Default Value	Description
page	string	yes	undefined	The page attribute can have the following values:

Option	Type	Mandatory	Default Value	Description
				• **<pageID>** — The notification message is only received by the page with the unique page ID. For example, **654E0122924F42EA82B5C5** • **all** — The notification message is received by all pages. • **active** — The notification message is received only by the page with focus in the browser.
channel	string	yes	undefined	The channel name. See Standard Messages for details.
data	object	no	undefined	Channel-specific data. See Standard Messages for details.

Standard Messages

The following messages are standard in Genesys Web Engagement.

gpe.appendContent

With this message, you can append content retrieved by **url** to the **container** element.

Options

Option	Type	Mandatory	Default Value	Description
url	string	yes	undefined	The path of the resource to load. This option can have either an absolute

Option	Type	Mandatory	Default Value	Description
				(http://www.genesys.com/invite.html) or relative value (/invite.html). Note: The relative value is relative to the Web Engagement Server.
method	string	no	"JSONP"	Specifies the type of request to make: • `GET` — You should only use the GET method if the resource is located on the same server, in order to avoid issues with cross-origin resource sharing. • `JSONP` — For the JSONP method, the server you point to in the `url` option must support the JSONP protocol.
container	string	no	"body"	The jQuery selector for the loaded content container. The loaded content is appended to the current element.

Example

The following message uses the AJAX GET request to load data from the server.

```
{
    "page": "654E0122924F42EA82B5C581A394555D",
    "channel": "gpe.appendContent",
    "data": {
        "url": "http://www.genesys.com/invite.html",
        "method": "GET",
        "container": "body"
    }
}
```

gpe.setVariable

You can use this message to set the variable on the page in the window context.

Options

Option	Type	Mandatory	Default Value	Description
variable	string	yes	undefined	The global variable name. The variable name could contain a namespace, for example: `com.service.param` — this object will be available by `window.com.service.param`.
value	string	no	undefined	The variable value.

Example

This example sets the variable `window.serviceData` to the object `{ 'name': 'Pat', 'message': 'Hello world!' }`. If the variable is already defined, then it is extended.

```
{
    "page": "654E0122924F42EA82B5C581A394555D",
    "channel": "gpe.setVariable",
    "data": {
        "variable": "serviceData",
        "value": {
            "name": "Pat",
            "message": "Hello world!"
        }
    }
}
```

gpe.callFunction

This message calls the corresponding function on the page.

Options

Option	Type	Mandatory	Default Value	Description
function	string	yes	undefined	The name of the function. The name can contain the namespace of the function. For example, `com.service.alert` — this means that

Option	Type	Mandatory	Default Value	Description
				`window.com.service.alert()` will be called.
arguments	array	no	undefined	The arguments to pass to the function. This should always be an array.

Example

This example calls the `window.reactToServerMessage('Hello world!', 1)` function.

```
{
  "page": "654E0122924F42EA82B5C581A394555D",
  "channel": "gpe.callFunction",
  "data": {
    "function": "reactToServerMessage",
    "arguments": ["Hello world!", 1]
  }
}
```

Using the API to Customize Widgets

Deprecation notice

- **Starting with the 8.5.000.38 release of Genesys Web Engagement, Genesys is deprecating the Native 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 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.

You can also use the Notification Service REST API to customize the out-of-the-box Genesys Web Engagement invitation widgets. See the following sections for details:

- Chat Invitation Message
- Ads Message

Chat Invitation Message

The chat invitation message is a JSON object that is sent by Orchestration Server to the browser. The chat invitation is represented by two serial messages: `gpe.setVariable` and `gpe.appendContent`.

For example:

```
[
   {
      "page":"654E0122924F42EA82B5C581A394555D",
      "channel":"gpe.setVariable",
      "data":{
         "variable":"com.genesyslab.gpe.invite.data",
         "value": {
            "type":"chat",
            "engagementID":"21def142-5ba9-4d64-8cd4-83d3c9d78ba9",
            "subject":"Chat",
            "message":"Good evening! Would you like some help with the selection? Our
technical experts are available to answer questions.",
            "acceptBtnCaption":"Chat",
            "cancelBtnCaption":"No Thanks",
            "inviteTimeout":30,
            "modal":false,
            "chatOptions": {
               "serverUrl": "http://demosrv.genesyslab.com:9081/server/cometd",
               "widgetUrl": "/server/api/resources/v1/chatWidget.html",
               "registration":true,
               "embedded": false,
               "userData": {
                  "visitID":"3ea3efab-ca33-4383-acbd-90720db72288"
               }
            }
         }
      }
   },
   {
      "page":"654E0122924F42EA82B5C581A394555D",
      "channel": "gpe.appendContent",
      "data": {
         "url": "/server/api/resources/v1/invite.html",
      }
   }
]
```

You can use the Notification Service REST API to modify the chat invitation, which uses `gpe.setVariable` to set the `com.genesyslab.gpe.invite.data` variable to a list of options. You can update these options to customize the chat invitation. See the table below for details.

Field	Type	Mandatory	Default Value	Description
type	string	yes	undefined	The type of engagement (typically, "chat").
engagementID	string	no	undefined	The unique identifier of the proactive engagement offer (part of the notification message).
subject	string	no	"Chat", if the `type` is set to "chat".	The invitation widget title.
message	string	no	"Hello! Would you like some help with the selection? Our technical experts are available to answer questions."	The main message in the invitation widget.
acceptBtnCaption	string	no	"Chat", if the `type` is set to "chat".	The title of the accept invitation button.
cancelBtnCaption	string	no	"No Thanks"	The title of the cancel invitation button.
inviteTimeout	Number	no	30	The timeout (in seconds) for the invitation widget. After this time interval, the invitation widget is automatically closed.
modal	Boolean	no	false	Specifies whether to show the invitation widget as a modal dialog.
chatOptions	object	yes	undefined	This object is passed to the chat application in the `startChat` method. See Chat JS Application API for details.

Ads Message

You can use the `gpe.appendContent` message to load the **ads.html** file.

```
[
   {
      "page": "654E0122924F42EA82B5C581A394555D",
      "channel": "gpe.appendContent",
      "data": {
         "url": "/server/api/resources/v1/ads.html"
      }
   }
]
```

Send Disposition Code for Invitation from Browser to GWE Server

Description

The disposition code lets Web Engagement know the status of the invitation: whether it was accepted, rejected, or ignored. Based on this status, GWE decides what it should do with the invitation. The browser side (invitation widget) sends the disposition code to the Web Engagement Server with a GET request.

Request

Method	GET		
URL	http://<gwe_server.host:gwe_server.port>/server/data/invites/?result=<result code>ŋagementID=<ID of proactive engagement offer>&pageID=<ID of page where invite appeared>&visitID=<current visit ID>[&media=<selected media>] The HTTPS schema is also allowed.		
Parameters			
Name	**Value**	**Mandatory**	**Description**
result	string	yes	A string that describes the result type. The following values are allowed: • accept • timeout • pageExit • cancel
engagementID	string	no	The unique identifier of the proactive engagement offer (part of the notification message). If this parameter is not specified, Web Engagement will only be able to process one engagement request for

Method	GET		
			the current page instance.
pageID	string	yes	The ID of the page where the invitation widget appeared.
visitID	string	yes	The current visit ID.
media	string	no	The media for which the invitation was applied.

Response

Response	Standard HTTP Responses 200 - OK 400 - FAULT
Response Type	none

Example

The code below shows an example of how to send the disposition code from the browser to GWE.

```
_gt.push(['getIDs', function(IDs) {
    $.ajax({
      type: "GET",
      url: "http://server:port/server/data/invites",
      data: {   result: "accept",
                engagementID: "21def142-5ba9-4d64-8cd4-83d3c9d78ba9",
                media: "chat",
                visitID: IDs.visitID,
                pageID: IDs.pageID
            }
    })
      .done(function( msg ) {
        // Data has been saved
      });
}]);
```

Tip

You can use the Monitoring JS API to access the visitID and pageID parameters.

Engagement REST API

Description

The Engagement REST API describes the rules for communication between an external source (typically the SCXML strategy) and the Web Engagement Server in order to start or cancel an engagement.

Start Engagement Attempt

Description

This request notifies the Web Engagement Server that is should start the engagement attempt. The notification is transferred to the Web Engagement Server and then to the visitor's browser.

Request

Method	POST		
URL	http://<gwe_server_host:gwe_server_port>/server/data/gateway/engage The HTTPS schema is also allowed, if configured.		
Parameters			
Name	**Value**	**Mandatory**	**Description**
ixnProfile	JSON object	yes	This object describes the notification parameters. For details about the structure of this JSON object, see Start Engagement as a Result of the Engagement Logic Strategy

Response

Response	Standard HTTP Responses 200 - OK 400 - FAULT
Response Type	JSON

Cancel Engagement Attempt

Description

This request notifies the Web Engagement Server that the current engagement attempt should be cancelled and the invitation should not be shown to the visitor.

Request

Method	POST
URL	http://<gwe_server_host:gwe_server_port>/server/data/gateway/noengage The HTTPS schema is also allowed, if configured.

Parameters

Name	**Value**	**Mandatory**	**Description**
cancelData	JSON object	yes	This object describes the parameters for the cancelled engagement attempt. For details about the structure of this JSON object, see Cancelling Engagement as a Result of the Engagement Logic Strategy

Response

Response	Standard HTTP Responses 200 - OK 400 - FAULT
Response Type	JSON

History REST API

The Web Engagement History REST API reference lists all the RESTful elements that you can use to access the web history stored by the Genesys Web Engagement Server.

Resource Access Template

To access resources, you must respect the following syntax:

```
Method <schema>://<server>:<port>/server/data/<resourceName>[/<resourceId>[/<sub-
resourceName>]]
```

- `<method>` is `GET` or `POST`.
- `<schema>` is `http` or `https`.
- `<server>` is the hostname or the IP address of the Web Engagement Server.
- `<port>` is the port of the Web Engagement Server.
- `<resourceName>` is the name of the requested resource.
- `<resourceId>` is the identifier of the requested resource.
- `<sub-resourceName>` is the name of a sub-resource.

For instance, to retrieve the list of identities on MyHostname, you can use the following request:

```
GET http://MyHostname:9081/server/data/identities
```

To retrieve a given identity, you need its identifier:

```
GET http://MyHostname:9081/server/data/identities/pat.thompsom@genesys.com
```

To retrieve the pages of user pat.thompsom@genesys.com, you can use the following request:

```
GET http://MyHostname:9081/server/data/identities/pat.thompsom@genesys.com/pages
```

Contents

The History REST API contains the following resources:

- Event Resource
- Identity Resource
- Page Resource
- Visit Resource

Note that each resource chapter lists all the operations available with the resource's path, not all the operations related to the resource. For instance, the Event Resource chapter lists all the methods associated with the **server/data/events/** path. Additional methods to retrieve events for a given identity are available in the Identity Resource chapter. You can also use the Operations Index to get the complete list of available operations.

HTTP Response Codes and Errors

The Genesys Web Engagement Server returns HTTP status codes and messages for every operation, in the requested format. Status codes match standard HTTP codes, but messages can differ and provide additional details included in the header of the response.

Important
Additional results and error codes may be returned due to external web servers and layers involved in your operations.

Successful Result

A successful response to a request is marked by HTTP Status Code 200 (OK). In that case, your application may get additional information in the header and the body of the response. Refer to the Response section of your operation's page to get the detailed list of returned information. The following table lists the standard HTTP codes used by Genesys Web Engagement Server for a successful response.

Code	Title	Description
200	OK	Success!
204	No Content	For "filtered collection read" requests only. The request was correct and successful but the server has no appropriate entities to return.

Errors

For responses with HTTP status code 4xx or 5xx, the response body contains an application-specific description of the error instead of a representation of the requested resource. The following table lists the specific errors that operations can encounter. This list is not restrictive; additional error codes could be returned due to external web servers and layers involved:

Code	Title	Description
400	Bad Request	General error which could be caused by: • Missing required parameter. • Parameter value of

Code	Title	Description
		unexpected type.
401	Not Authorized	Credentials are missing or incorrect, or the given user is not allowed to execute a given service (such as an administrative service method that changes the profile schema). See Basic Access Authentication.
403	Forbidden	The operation is forbidden and the reason is specified in the error message.
404	Not Found	For "identified entity read" request only. The specified URI is invalid, or the requested resource (such as identity, visit, event, and so on) does not exist.
500	Internal Server Error	An unexpected error occurred in the Web Engagement Server (for instance, a runtime exception). The error message suggests you forward logs to Genesys Customer Care.
502	Bad Gateway	Returned when one or more of the systems required to fulfill the response (the Cassandra database or the Genesys environment, for example) are either unavailable or returned an error.
503	Service Unavailable	Web Engagement Server is unable to process the given request. Example situations include: • Too many requests.

Authentication

The Web Engagement History REST API supports the Basic HTTP authentication (see http://www.ietf.org/rfc/rfc2617.txt) scheme.

Important
HTTP authentication should be used with Secured HTTP communication (HTTPS).

Configuration

The REST API security is configured in the security section of the Web Engagement Server application. The following configuration options are mandatory to enable authentication:

- auth-scheme
- user-id
- password

Note: If authentication is used, every REST API client must support that authentication type and the clients must know the authentication credentials. You must configure authentication for Interaction Workspace and if your Engagement strategies use the REST interface, you must also add your authentication credentials. See Configuring Authentication in the Deployment Guide for details.

Basic Authentication

This authentication scheme passes unencrypted credentials, so it is unsafe unless you use a secured connection (HTTPS).

Operations and Resources Index

The History REST API contains the following resources and operations:

Event Resource

Description

The event resource contains information related to a Business or System event that occurred on a specific web page at a given time. Two types of event can be issued:

- **System** events are constant and cannot be customized. There are two groups of System events:
    - Visit related (VisitStarted, PageEntered, PageExited);
    - Identity related (SignIn, SignOut, UserInfo).
- **Business** events are custom events that you can define within the DSL. See Managing Business Events for details.

The possible event names are listed in the following table:

Name	Type	Description
VisitStarted	System	Generated by the Browser Tier when the visitor starts visiting the website and enters the first page. It creates a new visit resource and then all the pages visited by the visitor are associated with this visit resource.
PageEntered	System	Generated by the Browser Tier when the visitor enters a page. The Web Engagement Server creates a page resource which is associated with the visit. The page resource can be associated with an identity according to the identification scope of the visitor.
PageExited	System	Generated by the Browser Tier when the visitor exits a page. The Web Engagement Server updates page resource accordingly. If the visitor comes back to the page, a new page resource is created.
SignIn	System	Generated by the Browser Tier when the visitor signs in (or is authenticated with the company web portal). It captures the identification information processed to authenticate the visitor, for instance, the email

Name	Type	Description
		address used to login. When the Web Engagement Server receives this event, it updates or creates the identity resource associated with the identifier.
SignOut	System	Generated by the Browser Tier when the visitor signs out.
UserInfo	System	Generated when the Browser Tier submits information about the visitor. This event occurs when the visitor is recognized or updates his or her profile information.
Timeout or InactivityTimeout	Business	Default business event with Timeout set to 10 seconds. Generated when the visitor's mouse is no longer moving. This Business event needs customization through the DSL rules. See Managing Business Events for details.
Search	Business	Default business event. Generated when the web visitor is searching on the website. This Business event needs further DSL customization. See Managing Business Events for details.
Custom name	Business	Custom business event created through DSL customization. See Managing Business Events for details.

Resource Details

Field	Type	Mandatory	Description
eventID	string	yes	The unique ID of the event resource.
eventName	string	yes	Name of the event. See the Description section for further information.
eventType	System, Business	yes	Event type.
category	string	no	Category related to the generated event.
serverTimestamp	long	yes	Server timestamp.
browserPageID	string	yes	The browser page ID.

Field	Type	Mandatory	Description
globalVisitID	string	yes	Global Visit ID.
url	string	yes	The url of the page.
timestamp	long	yes	Timestamp.
pageID	string	yes	ID of the associated page resource.
data	string[]	no	Additional JSON data, specific to the event.

Related Requests

You can retrieve event resources by using the following operations:

- Query event
- Query events by identity
- Query events by page
- Create new event for visit
- Query events by visit

Examples

Retrieving a UserInfo Event

Request

```
GET http://127.0.0.1:9081/server/data/events/5cdca781-3fa3-11e2-aee5-00505625a04f
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:48:08 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:48:08 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 587
{"eventID":"5cdca781-3fa3-11e2-aee5-00505625a04f",
"category":"",
"eventType":"SYSTEM","eventName":"UserInfo",
"serverTimestamp":1354798164472,"browserPageID":"2D869014426A4CAA8FA5C0D7B8668D0A",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"url":"http://www.genesyslab.com/
afu_FLS_intermediary.page?returnUrl=/?","timestamp":1354798163915,
"visitID":"f24c60f6-0728-4f3d-
b8b4-1e7bad2dc8a3","pageID":"5c911f90-3fa3-11e2-aee5-00505625a04f",
"data":{"userID":"user@genesyslab.com","sex":"male","name":"user1","age":30}}
```

Retrieving a PageEntered Event

Request

```
GET http://127.0.0.1:9081/server/data/events/c4203381-3fa3-11e2-aee5-00505625a04f
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:48:08 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:48:08 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 587
{"eventID":"c4203381-3fa3-11e2-aee5-00505625a04f","category":"",
"eventType":"SYSTEM","eventName":"PageEntered",
"serverTimestamp":1354798337720,"browserPageID":"AA8C3E0B8C9543D58D9CAB122A714124",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"url":"http://www.genesyslab.com/%3f","timestamp":1354798337628,
"visitID":"f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3",
"pageID":"c489cac0-3fa3-11e2-aee5-00505625a04f",
"data":{"urlReferrer":"http://www.genesyslab.com/afu_FLS_intermediary.page?returnUrl=/%3f",
"localTime":"2012-12-06T12:52:17.628Z",
"title":"404 - Not found"}}
```

Query event

Description

The query event method retrieves a given event.

Request

Method	GET		
URL	/events/**${eventId}**		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${eventId}	string	yes	Event identifier.

<references />

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	Event

Example

The following operation retrieves a UserInfo event by its ID.

Request

```
http://127.0.0.1:9081/server/data/events/5cdca781-3fa3-11e2-aee5-00505625a04f
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:48:08 GMT
```

```
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:48:08 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 587
{"eventID":"5cdca781-3fa3-11e2-aee5-00505625a04f",
"category":"",
"eventType":"SYSTEM","eventName":"UserInfo",
"serverTimestamp":1354798164472,"browserPageID":"2D869014426A4CAA8FA5C0D7B8668D0A",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"url":"http://www.genesyslab.com/
afu_FLS_intermediary.page?returnUrl=/?","timestamp":1354798163915,
"visitID":"f24c60f6-0728-4f3d-
b8b4-1e7bad2dc8a3","pageID":"5c911f90-3fa3-11e2-aee5-00505625a04f",
"data":{"userID":"user@genesyslab.com","sex":"male","name":"user1","age":30}}
```

Global Visit Resource

Description

The global visit resource contains information about the relationship between a visitor's browser instance and their identity. Currently, the resource has one operation Query identities by globalVisitID, which retrieves a list of identities for a given globalVisitID (the identifier for the browser).

Query identities by globalVisitID

Description

This method retrieves a list of identities for the given global visit ID.

Request

Method	GET		
URL	/global_visits/**${globalVisitId}**/identities		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${globalVisitId}	string	yes	Global visit identifier.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON array of Identity

Identity Resource

The identity resource contains visitor information and is created when the Web Engagement Server receives the SignIn event. The Browser Tier generates this event when the visitor signs in (or is authenticated with the company web portal). The identification information submitted by the visitor is used to create the identity ID (for instance, the email address or an account name).

At this point, the identification scope of the visitor is `Authenticated`. If the visitor signs out, the Web Engagement Server receives the SignOut event and the identification scope becomes `Recognized`. For further information on events, see Event Resource.

Resource Details

Field	Type	Mandatory	Description
identityId	string	yes	The unique ID of the given identity. The visitor provides the identityId information when registering on the website; for instance, it can be the email address.
name	string	no	User name.
location	string	no	User location.
entityInCS	string	no	Entity in the Contact Server.
visitScope	Authenticated, Recognized	yes	Specify the visit's identification scope in relation to the visitor. If the visitor is authenticated, you can retrieve session information. If the visitor signs out, the visit scope changes to recognized.
eventIds	string[]	no	Array of event IDs associated with this identity.
events	event[]	no	Array of the event resources associated with this identity.
pageIds	string[]	no	Array of page IDs associated with this identity.
pages	page[]	no	Array of the page

Field	Type	Mandatory	Description
			resources associated with this identity.
visitIds	string[]	no	Array of visit IDs associated with this identity.
visits	visit[]	no	Array of the visit resources associated with this identity.

Related Requests

- Query events by identity
- Query identity
- Query identities
- Query pages by identity
- Query visits by identity
- Query identities by visit
- Query user agents by identity

Examples

Retrieving Identities

The following request retrieves all the identities available.

Request

```
GET http://127.0.0.1:9081/server/data/identities
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:17:09 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:17:09 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 477
[{"eventIds":null,"events":null,"pageIds":null,"pages":null,
"sessionIds":null,"sessions":null,"identityId":"pat.thompsom@genesyslab.com",
"name":null,"location":null,"entityInCS":null,"visitScope":"Authenticated",
"visitIds":null,"visits":null},
{"eventIds":null,"events":null,"pageIds":null,"pages":null,
"sessionIds":null,"sessions":null,"identityId":"user@genesyslab.com",
"name":null,"location":null,"entityInCS":null,"visitScope":"Authenticated",
"visitIds":null,"visits":null}]
```

Retrieving Identities with page IDs

The following request retrieves all the identities available, including the associated page IDs.

Request

```
GET /server/data/identities?include_pages=true
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:28:13 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:28:13 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 627
[{"eventIds":null,"events":null,
"pageIds":["11ff6de0-446e-11e2-bfd2-00505625a04f"],
"pages":null,"sessionIds":null,"sessions":null,"
identityId":"pat.thompsom@genesyslab.com",
"name":null,"location":null,"entityInCS":null,
"visitScope":"Authenticated","visitIds":null,
"visits":null},
{"eventIds":null,"events":null,
"pageIds":["45f0eda1-3fa4-11e2-aee5-00505625a04f","c489cac0-3fa3-11e2-aee5-00505625a04f",
"06afd7f0-3fa4-11e2-aee50505625a04f"],"pages":null,"sessionIds":null,
"sessions":null,"identityId":"user@genesyslab.com",
"name":null,"location":null,"entityInCS":null,"visitScope":"Authenticated",
"visitIds":null,"visits":null}]
```

Query events by identity

Description

This method retrieves the events associated with a given identity.

Request

Method	GET		
URL	/identities/**${identityId}**/events		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${identityId}	string	yes	Identity identifier.
age	integer	no	The maximum age for the event, in seconds. Older events are not be returned.
eventName	string	no	Event name.
eventType	string	no	Event type.
category	string or "all categories"	no	Category name or the "all categories" key, which means that the results include any event associated with a category or a combination of categories.
url	string	no	Event URL.
globalVisitID	string	no	Associated global Visit ID.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Event resources.

Examples

Activating Paging

The following example retrieves the first three events for the 'user@genesyslab.com' Identity.

Request

```
GET http://127.0.0.1:9081/server/data/identities/user@genesyslab.com/events?page_size=3
```

Response

```
HTTP/1.1 200 OK
Date: Thu, 13 Dec 2012 14:26:47 GMT
Content-Type: application/json; charset=UTF-8
Date: Thu, 13 Dec 2012 14:26:47 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Paging-Next: user@genesyslab.com#1354798334842#c2994560-3fa3-11e2-aee5-00505625a04f
Content-Length: 1508
[{"eventID":"5cdca781-3fa3-11e2-aee5-00505625a04f",
"category":"","eventType":"SYSTEM","eventName":"UserInfo",
"serverTimestamp":1354798164472,
"browserPageID":"2D869014426A4CAA8FA5C0D7B8668D0A",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"url":"http://www.genesyslab.com/afu_FLS_intermediary.page?returnUrl=/?",
"timestamp":1354798163915,"visitID":"f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3",
"pageID":"5c911f90-3fa3-11e2-aee5-00505625a04f",
"data":{"userID":"user@genesyslab.com","sex":"male","name":"user1","age":30}},
{"eventID":"62a62150-3fa3-11e2-aee5-00505625a04f",
"category":"","eventType":"BUSINESS","eventName":"Timeout-10","serverTimestamp":1354798174181,
"browserPageID":"2D869014426A4CAA8FA5C0D7B8668D0A","globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20","url"
afu_FLS_intermediary.page?returnUrl=/%3f",
"timestamp":1354798174090,"visitID":"f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3",
"pageID":"5c911f90-3fa3-11e2-aee5-00505625a04f","data":{}},
{"eventID":"6e920a60-3fa3-11e2-aee5-00505625a04f","category":"",
"eventType":"BUSINESS","eventName":"Timeout-30","serverTimestamp":1354798194182,
"browserPageID":"2D869014426A4CAA8FA5C0D7B8668D0A",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"url":"http://www.genesyslab.com/afu_FLS_intermediary.page?returnUrl=/%3f",
"timestamp":1354798194128,"visitID":"f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3",
"pageID":"5c911f90-3fa3-11e2-aee5-00505625a04f","data":{}}]
```

Retrieving the next page of results

The following request uses the Paging-Next header parameter of the previous response (`user@genesyslab.com#1354798334842#c2994560-3fa3-11e2-aee5-00505625a04f`) to retrieve the next three events.
Request

```
GET http://127.0.0.1:9081/server/data/identities/user@genesyslab.com/events?page_size=3
&page_value="user@genesyslab.com#1354798334842#c2994560-3fa3-11e2-aee5-00505625a04f"≠xt=true
```

Response

```
HTTP/1.1 200 OK
Date: Thu, 13 Dec 2012 14:35:11 GMT
Content-Type: application/json; charset=UTF-8
Date: Thu, 13 Dec 2012 14:35:11 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Paging-Next: pat.thompsom@genesyslab.com#1355325030949#11ed4570-446e-11e2-bfd2-00505625a04f
Paging-Prev: "user@genesyslab.com10135479833484210c2994560-3fa3-11e2-aee5-00505625a04f"
Content-Length: 1507
[{"eventID":"03956571-446e-11e2-bfd2-00505625a04f","category":"Internet","eventType":"SYSTEM","eventName":"Sign
"serverTimestamp":1355325007175,"browserPageID":"7FA47396264C4F6A8FED4F4E9710B0C4",
"globalVisitID":"66c0976d-cbdd-4e94-accb-26093803cf54",
"url":"http://www.genesyslab.com/catalogue-topic/t_new_internet_options.page?",
"timestamp":1355325006715,
"visitID":"42788a69-785d-4860-be39-45048d32441c",
"pageID":"f57b03a0-446d-11e2-bfd2-00505625a04f",
"data":{"userID":"pat.thompsom@genesyslab.com"}},
{"eventID":"072fa330-446e-11e2-bfd2-00505625a04f",
"category":"Internet",
"eventType":"BUSINESS","eventName":"Timeout-30",
"serverTimestamp":1355325013219,
"browserPageID":"7FA47396264C4F6A8FED4F4E9710B0C4",
"globalVisitID":"66c0976d-cbdd-4e94-accb-26093803cf54",
"url":"http://www.genesyslab.com/catalogue-topic/t_new_internet_options.page?",
"timestamp":1355325013148,
"visitID":"42788a69-785d-4860-be39-45048d32441c",
"pageID":"f57b03a0-446d-11e2-bfd2-00505625a04f",
"data":{}},
{"eventID":"0fbe5780-446e-11e2-bfd2-00505625a04f",
"category":"Internet","eventType":"SYSTEM",
"eventName":"PageExited","serverTimestamp":1355325027576,
"browserPageID":"7FA47396264C4F6A8FED4F4E9710B0C4",
"globalVisitID":"66c0976d-cbdd-4e94-accb-26093803cf54",
"url":"http://www.genesyslab.com/catalogue-topic/t_new_internet_options.page?",
"timestamp":1355325027495,
"visitID":"42788a69-785d-4860-be39-45048d32441c",
"pageID":"f57b03a0-446d-11e2-bfd2-00505625a04f",
"data":{}}]
```

Query identity

Description

This method retrieves a given identity.

Request

Method	GET		
URL	/identities/**${identityId}**		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${identityId}	string	yes	Identity identifier.
include_visits	• `true` • `false`	no	If `true`, the returned identity contains the reference list of associated visits.
include_visits_detail	• `true` (if `include_visits=true`) • `false`	no	If `true`, the returned identity contains the associated visit resources. You can only use this parameter if `include_visits` is set to `true`.
include_pages	• `true` • `false`	no	If `true`, the returned identity contains the reference list of the associated pages.
include_pages_detail	• `true` (if `include_pages=true`) • `false`	no	If `true`, the returned identity contains the associated pages. You can only use this parameter if `include_pages` is set to `true`.
include_events	• `true` • `false`	no	If `true`, the returned identity contains the reference list of associated events.

Method	GET		
include_events_detail	• true (if include_events=true) • false	no	If true, the returned identity contains the associated events. You can only use this parameter if include_events is set to true.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Identity.

Example

The following request retrieves the identity resource with the pat.thompsom@genesyslab.com ID.

Request

```
http://127.0.0.1:9081/server/data/identities/pat.thompsom@genesyslab.com
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:17:09 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:17:09 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 477
[{"eventIds":null,"events":null,"pageIds":null,"pages":null,
"sessionIds":null,"sessions":null,"identityId":"pat.thompsom@genesyslab.com",
"name":null,"location":null,"entityInCS":null,"visitScope":"Authenticated",
"visitIds":null,"visits":null}]
```

Query identities

Description

This method retrieves a list of identities. By default, only the identities are returned: the associated visits, events, and pages are not included in the results.

Request

Method	GET		
URL	/identities		
Parameters			
Name	**Value**	**Mandatory**	**Description**
location	string	no	Filters the identity results based on the given geolocation parameter.
userAgent	string	no	Returns identities with an exact match for the UserAgent attribute.
include_visits	• `true` • `false`	no	If `true`, each returned identity contains the reference list of associated visits.
include_visits_detail	• `true` (if `include_visits=true`) • `false`	no	If `true`, each returned identity contains the associated visits resources. You can only use this parameter if `include_visits` is set to `true`.
include_pages	• `true` • `false`	no	If `true`, each returned identity contains the reference list of the associated pages.
include_pages_detail	• `true` (if `include_pages=true`) • `false`	no	If `true`, each returned identity contains the associated pages. You can only use this parameter if `include_pages` is set to

Method	GET		
			`true`.
include_events	• `true` • `false`	no	If `true`, each returned identity contains the reference list of associated events.
include_events_detail	• `true` (if `include_events=true`) • `false`	no	If `true`, each returned identity contains the associated events.You can only use this parameter if `include_events` is set to `true`.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON array of Identity.

Example

Retrieving Identities with page IDs

The following request retrieves all the identities available, including the associated page IDs.

Request

`http://127.0.0.1:9081/server/data/identities?include_pages=true`

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 15:28:13 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 15:28:13 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 627
[{"eventIds":null,"events":null,
"pageIds":["11ff6de0-446e-11e2-bfd2-00505625a04f"],
"pages":null,"sessionIds":null,"sessions":null,"
identityId":"pat.thompsom@genesyslab.com",
"name":null,"location":null,"entityInCS":null,
"visitScope":"Authenticated","visitIds":null,
"visits":null},
{"eventIds":null,"events":null,
"pageIds":["45f0eda1-3fa4-11e2-aee5-00505625a04f","c489cac0-3fa3-11e2-aee5-00505625a04f",
"06afd7f0-3fa4-11e2-aee50505625a04f"],"pages":null,"sessionIds":null,
"sessions":null,"identityId":"user@genesyslab.com",
"name":null,"location":null,"entityInCS":null,"visitScope":"Authenticated",
"visitIds":null,"visits":null}]
```

Query pages by identity

Description

This method retrieves the pages for a given identity.

Request

Method	GET		
URL	/identities/**${identityId}**/pages		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${identityId}	string	yes	Identity identifier.
age	integer	no	Pages' maximum age in seconds. Older pages will not be returned.
include_events	• `true` • `false`	no	If `true`, the returned pages contain the reference list of associated events.
include_events_detail	• `true` (if `include_events=true`) • `false`	no	If `true`, the returned pages contain the associated events. You can only use this parameter if `include_events` is set to `true`.
url	string	no	Page URL used to filter the results.
title	string	no	Page title used to filter the results.
category	string or "all categories"	no	A specific category name or the "all categories" key, which means that the results include any page associated with a category or a combination of categories.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Page resources.

Query user agents by identity

Description

This method retrieves the user agents for a given identity.

Request

Method	GET		
URL	/identities/**${identityId}**/userAgents		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${identityId}	string	yes	Identity identifier.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of User Agents.

Query visits by identity

Description

This method retrieves the visits for a given identity.

Request

Method	GET		
URL	/identities/**${identityId}**/visits		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${identityId}	string	yes	Identity identifier.
age	integer	no	Visits' maximum age in seconds. Older visits will not be returned.
globalVisitID	string	no	Filters the visit results based on the given global Visit ID.
userAgent	string	no	Returns the visits matching the UserAgent field.
include_pages	• `true` • `false`	no	If `true`, the returned visits contain the reference list of the associated pages.
include_pages_detail	• `true`(if `include_pages=true`) • `false`	no	If `true`, the returned visits contain the associated pages. You can only use this parameter if `include_pages` is set to `true`.
include_events	• `true` • `false`	no	If `true`, the returned visits contain the reference list of associated events.
include_events_detail	• `true` (if `include_events=`	no	If `true`, the returned visits contain the

Method	GET		
	`true`) • `false`		associated events. You can only use this parameter if `include_events` is set to `true`.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Visit resources.

Page Resource

Description

The page resource is created when the visitor enters a page (upon the `PageEntered` event). If the visitor leaves the page, and later visits the page again, a new page resource is created. For further information about events, see Event Resource.

Resource Details

Field	Type	Mandatory	Description
pageID	string	yes	The unique ID of the page resource.
url	string	yes	The page URL.
browserPageID	string	yes	The browser page ID; another page identifier unique across the visit.
pageExitedDate	long	no	Date at which the user left the page. 0 means that the user did not leave the page already.
pageEnteredDate	long	no	Date at which the user entered the page.
category	string	no	Associated category, if any.
title	string	no	Page title.
first	• `true` • `false`	no	`true` if this is the first page entered for the current visit.
eventIds	string[]	no	IDs of the associated events.
events	event[]	no	Array of the associated events.

Related Requests

- Query events by page

- Query page
- Query pages by identity
- Query pages by visit

Example

Retrieving a page

Request

```
GET http://127.0.0.1:9081/server/data/pages/c489cac0-3fa3-11e2-aee5-00505625a04f
```

Response

```
{"eventIds":null,"events":null,"pageId":"c489cac0-3fa3-11e2-aee5-00505625a04f",
"url":"http://www.genesyslab.com/%3f",
"browserPageID":"AA8C3E0B8C9543D58D9CAB122A714124",
"pageExitedDate":1354798339992,
"pageEnteredDate":1354798338412,
"category":"","title":"404 - Not found","first":false}
```

Query events by page

Description

This method retrieves the events for a given page.

Request

Method	GET		
URL	/pages/**${pageId}**/events		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${pageId}	string	yes	Page identifier.
age	integer	no	Events' maximum age in seconds. Older events will not be returned.
eventName	string	no	Event name.
eventType	string	no	Event type.
category	string or "all categories"	no	Category name or the "all categories" key, which means that the results include any event associated with a category or a combination of categories.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Event resources

Query page

Description

This method retrieves a given page.

Request

Method	GET		
URL	/pages/**${pageId}**		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${pageId}	string	yes	Page identifier.
include_events	• `true` • `false`	no	If `true`, the returned page contains the reference list of associated events.
include_events_detail	• `true` (if `include_events=true`) • `false`	no	If `true`, the returned page contains the associated events. You can only use this parameter if `include_events` is set to `true`.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	Page

Visit Resource

Description

The visit resource contains the information related to a visitor's visit, which is the browser session opened by the visitor to visit the website. The visit starts with the `VisitStarted` event (submitted by the website) and then, all the browsing activity is recorded in page and event resources associated with the visit. The visitor remains `Anonymous` until he or she authenticates on the website. When authentication occurs, the visit is associated with an identity resource.

Important
Several identities are associated with the same visitId if several visitors authenticate on the website. See Identity Resource.

Resource Details

Field	Type	Mandatory	Description
visitId	string	yes	ID of the given visit.
startDate	long	yes	Date in milliseconds at which the visit started.
endDate	long	no	Date in milliseconds at which the visit ended; 0 means that the visit is not terminated.
globalVisitID	string	no	ID of the global visit.
userAgentID	string	no	ID of the associated user agent if any; otherwise, null.
eventIds	string[]	no	Array of event IDs associated with this visit.
events	event[]	no	Array of the event resources associated with this visit.
pageIds	string[]	no	Array of page IDs associated with this visit.
pages	page[]	no	Array of the page resources associated

Field	Type	Mandatory	Description
			with this visit.

Related Requests

- Create new event for visit
- Create new visit
- Query events by visit
- Query identities by visit
- Query pages by visit
- Query visit
- Query visits by identity

Example

Retrieve a Given Visit

Request

```
GET http://127.0.0.1:9081/server/data/visits/f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3
```

Response

```
HTTP/1.1 200 OK
Date: Wed, 12 Dec 2012 16:03:03 GMT
Content-Type: application/json; charset=UTF-8
Date: Wed, 12 Dec 2012 16:03:03 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.1.0
Content-Length: 345
{"eventIds":null,"events":null,"pageIds":null,"pages":null,"
sessionIds":null,"sessions":null,
"visitId":"f24c60f6-0728-4f3d-b8b4-1e7bad2dc8a3",
"startDate":1354797882714,"endDate":0,
"activeSessionId":"00000000-0000-1000-8000-000000000000",
"globalVisitID":"c93a19a1-45db-4d59-9c85-b637daea4a20",
"userAgentId":"b4ec8ef0-3fa2-11e2-aee5-00505625a04f"}
```

Create new event for visit

Description

This method creates a new event for a given visit resource.

Request

Method	POST		
URL	/visits/**${visitId}**/events		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${visitId}	string	yes	Visit identifier.
BODY	Event JSON object which MUST contain the pageId field, to ensure that the parent page is defined for this new event.		

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK

Create new visit

Description

This method creates a new visit resource.

Request

Method	POST		
URL	/visits		
Parameters			
Name	**Value**	**Mandatory**	**Description**
None			

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	Visit JSON object

Query events by visit

Description

This method retrieves the events for a given visit.

Request

Method	GET		
URL	/visits/**${visitId}**/events		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${visitId}	string	yes	Visit identifier.
age	integer	no	Events' maximum age in seconds. Older events will not be returned.
eventName	string	no	Event name.
eventType	string	no	Event type.
category	string or "all categories"	no	Category name or the "all categories" key, which means that the results include any event associated with a category or a combination of categories.
url	string	no	Event URL.
browserPageID	string	no	Browser page ID of the event, which is another page identifier unique across the visit.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Event resources.

Query identities by visit

Description

This method retrieves a list of identities for the given visit ID.

Request

Method	GET		
URL	/visits/${visitId}/identities		
Parameters			
Name	**Value**	**Mandatory**	**Description**
include_visits	• `true` • `false`	no	If `true`, each returned identity contains the reference list of associated visits.
include_visits_detail	• `true` (if `include_visits=true`) • `false`	no	If `true`, each returned identity contains the associated visits resources. You can only use this parameter if `include_visits` is set to `true`.
include_pages	• `true` • `false`	no	If `true`, each returned identity contains the reference list of the associated pages.
include_pages_detail	• `true` (if `include_pages=true`) • `false`	no	If `true`, each returned identity contains the associated pages. You can only use this parameter if `include_pages` is set to `true`.
include_events	• `true` • `false`	no	If `true`, each returned identity contains the reference list of associated events.
include_events_detail	• `true`	no	If `true`, each returned identity contains the associated events. You

Method	GET		
	(if `include_events=true`) • `false`		can only use this parameter if `include_events` is set to `true`.
association	• "Authenticated" • "Recognized"	no	Defines the type of association between the current visit and the returned identity.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON array of Identity

Query pages by visit

Description

This method retrieves the pages for a given visit.

Request

Method	GET		
URL	/visits/**${visitId}**/pages		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${visitId}	string	yes	Visit identifier.
age	integer	no	Pages' maximum age in seconds. Older pages will not be returned.
include_events	• `true` • `false`	no	If `true`, the returned pages contain the reference list of associated events.
include_events_detail	• `true` (if `include_events=true`) • `false`	no	If `true`, the returned pages contain the associated events. You can only use this parameter if `include_events` is set to `true`.
url	string	no	Page URL used to filter the results.
title	string	no	Page title used to filter the results.
category	string or "all categories"	no	A specific category name or the "all categories" key, which means that the results include any page associated with a category or a combination of categories.
browserPageID	string	no	Browser page id, which is another page

Method	GET		
			identifier unique across the visit. A collection with maximum one page will be returned.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	JSON Array of Page resources.

Query visit

Description

This method retrieves a given visit.

Request

Method	GET		
URL	/visits/**${visitId}**		
Parameters			
Name	**Value**	**Mandatory**	**Description**
${visitId}	string	yes	Visit identifier.
include_pages	• `true` • `false`	no	If `true`, the returned visit contains the reference list of the associated pages.
include_pages_detail	• `true` (if `include_pages=true`) • `false`	no	If `true`, the returned visit contains the associated pages. You can only use this parameter if `include_pages` is set to `true`.
include_events	• `true` • `false`	no	If `true`, the returned visit contains the reference list of associated events.
include_events_detail	• `true` (if `include_events=true`) • `false`	no	If `true`, the returned visit contains the associated events. You can only use this parameter if `include_events` is set to `true`.

Response

The History REST API answers with HTTP codes for every request. The following table shows the correct response for a successful request. See HTTP Response Codes and Errors for further details on the possible codes that this request can return.

HTTP code	200
HTTP Title	OK
Body	Visit.

Pacing REST API

Overview

The Pacing API gives external components access to your pacing information, using two different methods:

- The Reactive State method returns a number that indicates the probability of whether additional reactive traffic will displace proactive traffic. **It does not return the number of agents who are ready to accept chat interactions.**
- You can, however, use the Channel Capacity method to figure out how many agents are ready to process incoming requests.

> **Important**
> Please read the following information carefully before attempting to use the Pacing API!

Reactive State

Use this method to determine whether reactive traffic is displacing proactive traffic. If so, you may want to take action, such as limiting the number of chat interactions that are initiated in response to the reactive requests. To use this method, you must configure the Pacing Algorithm to use a type that predicts reactive engagements, that is, either `SUPER_PROGRESSIVE_DUAL` or `PREDICTIVE_B_DUAL`.

> **Important**
> As noted below, a request using this method returns the probability that a new reactive engagement should be allowed for a visitor. **It does not contain information about the number of agents who can accept an incoming interaction.**

Request

Method	GET
URL	http://<gwe_server_host:gwe_server_port>/server/data/pacing/reactiveState?channel=<channelName>&groups=[<names>]

Method	GET
	The HTTPS schema is also allowed, if configured.

Parameters

Name	**Value**	**Mandatory**	**Description**
channel	string	yes	The name of a media channel, which determines if a reactive engagement is possible. Valid value is chat.
groups	string	no	The list of agent group names. If this parameter is not defined, then the reactive pacing result is consolidated over all groups. **Note:** If you want to specify more than one group, you must use the following syntax: **&groups=*Group1_name*&groups=*Group2_na***

Response

Response	`{reactiveState : <value>}` The request returns the value of the pacing reactive state. This value is the probability that a new reactive engagement should be allowed for a visitor.
Response Type	JSON

Example

```
<script>
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/reactiveState?channel=chat'})
    .done(function( result ) {
        console.log('result: ' + JSON.stringify(result.reactiveState));
                var rnd = Math.random();
        if(rnd <= result.reactiveState) {
            // Do something
        }
    });
</script>
```

Important

jQuery is required for the example above.

Channel Capacity

Important

The channel capacity functionality was introduced in Genesys Web Engagement 8.1.200.41

Use this method to figure out how many agents are ready to process incoming requests.

Note: This method does not take into account the extent to which reactive traffic is displacing proactive traffic. Because of this, if you use its results without taking other factors into account, you may reduce the effectiveness of your proactive campaigns.

Request

Method	GET
URL	http://<gwe_server.host:gwe_server.port>/server/data/pacing/channelCapacity?channel=<channelName>&groups=[<names>] The HTTPS schema is also allowed, if configured.

Parameters

Name	Value	Mandatory	Description
channel	string	yes	The name of a media channel for which you want to determine the count of ready agents. Valid value is `chat`.
groups	string	no	The list of agent group names. If this parameter is not defined, then the resulting agent count is consolidated over all groups. **Note:** If you want to specify more than one group, you must use the following syntax: **&groups=*Group1_name*&groups=*Group2_na***

Response

Response	`{capacity : <value>}` The request returns the count of ready agents in the specified group or groups—or for the entire channel, if no group is specified—according to statistics provided by Stat Server.
Response	JSON

Response	`{capacity : <value>}` The request returns the count of ready agents in the specified group or groups—or for the entire channel, if no group is specified—according to statistics provided by Stat Server.
Type	

Example

```
<script>
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/channelCapacity?channel=chat'})
    .done(function( result ) {
        console.log('Ready agents capacity is: ' + JSON.stringify(result.capacity));
    });
</script>
```

Important

jQuery is required for the example above.

Business Events DSL

Description

The monitoring rules for each Genesys Web Engagement application you create are defined in a domain specific language (DSL). The DSL specifies the document elements to monitor, the events to send to the Web Engagement Server, and the data to include with those events. For details about how these events are structured, see Events Structure.

```
<?xmlversion="1.0"encoding="utf-8"?>
<properties>
    <events>
        <event name="AddToCart">
            <trigger name="AddToCartTrigger" element="img.bdt-addToCart" action="click"
url="http://www.MySite.com/" count="1">
                <val name="productName"
value="$(event.target).parents('div.hproduct').find('h3.name a').text()"/>
                <val name="productModel"
value="$(event.target).parents('div.hproduct').find('span.model')"/>
                <val name="productSKU"
value="$(event.target).parents('div.hproduct').find('span.sku').text()"/>
                <val name="productPrice"
value="$(event.target).parents('div.hproduct').find('h4.price').text()"/>
           </trigger>
    </event>

    <event name="Search">
        <trigger name="GoSearchClick" element="td#gobutton input" action="click"
url="http://www.MySite.com" count="1"/>
        <val name="searchString" value="$('input.searchfield:text').val();"/>
    </event>
    </events>
</properties>
```

The Business Events API includes all the DSL elements that you can use to define business events. For example and details about how to implement these events, see Managing Business Events. When you modify the DSL, Genesys recommends that you use InTools, an application included with Genesys Web Engagement that helps you create, validate, and test your changes to the DSL.

<properties> (mandatory)

The `<properties>` element is the main root element of the DSL file. It has an optional debug attribute and a mandatory `<events>` child.

debug (optional) - Deprecated in release 8.1.2

The debug attribute enables debugging in the browser by setting its value to the JavaScript Boolean `true`. The debugging information opens a pop-up window and shows the JSON serialized event data for the business events before they are sent to the Web Engagement Server.

Note: In some browsers, using the debug attribute can affect the performance of the Web Engagement Server by delaying the event dispatch.

<events>

The `<events>` element contains a list of all the business events that can be generated during monitoring. These business events are captured in the `<event>` child element.

<event>

The `<event>` element contains mandatory `id` and `name` attributes, and an optional `condition` attribute. An `<event>` must also have one or more `<trigger>` children, which define the conditions that must be matched to generate an event.

Note: If the <trigger> child is omitted, the event will never be generated.

name (mandatory)

The `name` is sent to the Web Engagement Server. A DSL file may contain several `<event>` elements with identical values for `name`, but with different values for `id`. For example, if your website includes a search form, you can submit this form by clicking on the 'search' button or by pressing the 'enter' key. Inside the browser, the click and key press events are clearly distinct, but are not relevant for the Web Engagement Server.

The following example shows how to create two business events which return the same event name to the Web Engagement Server:

```
<?xmlversion="1.0"encoding="utf-8"?>
<properties>
    <events>
        <event name="Search">
           …
        </event>
        <event name="Search">
           …
        </event>
    </events>
</properties>
```

condition (optional)

The `condition` attribute is a JavaScript Boolean expression. If it is present, the event’s triggers will be installed in the page if the `condition` evaluates to `true`.

The following example creates a business event with a timer which can be triggered only if the text inside the <h1> tag on the page is "Compare":

```
<event name="InactivityTimeout4CompareProducts" condition="$('h1').text() == 'Compare'">
    <trigger name="InactivityTimeout4CompareProductsTrigger" element="" action="timer:10000"
```

```
type="timeout"
       url="http://www.MySite.com/site/olspage.jsp" count="1"/>
      …
</event>
```

Since the event (in this case 'InactivityTimeout4CompareProductsEvent') will never be generated if its triggers are not installed, the `condition` attribute allows you to place conditions on any feature of the environment that can be tested by a JavaScript Boolean expression, in order to monitor and generate events.

postcondition (optional)

A `postcondition` attribute is similar to a `condition` except it is evaluated after the business event is already generated. If it is present, the event will be sent to the Web Engagement Server if the `postcondition` evaluates to `true`.

```
<event name="InactivityTimeout4CompareProducts" postcondition="$('h1').text() == 'Compare'">
      <trigger name="InactivityTimeout4CompareProductsTrigger" element=""
action="timer:10000" type="timeout"
                                     url="http://www.MySite.com/site/olspage.jsp" count="1"/>
      …
</event>
```

<trigger> (mandatory child element)

The `<trigger>` element defines the conditions that must be matched to generate business events, as well as the data to be included with the event. If several triggers are part of the event definition, they must all match to raise the business event. If each trigger matches a different DOM event in the browser, then the set of triggers specifies a series of web events that must occur before the parent business event is submitted to the Web Engagement Server.

The `<trigger>` element has mandatory `name`, `element`, and `action` attributes, and optional `url` and `count` attributes. It can have and 0 or more `<val>` children.

name (mandatory)

This attribute specifies the name of the trigger. It must be unique in the parent `<event>` element. If an `<event>` element has multiple triggers, they must all have different names.

element (mandatory)

The `element` attribute specifies the document's DOM element to which the trigger should be attached. The value of `element` should be a jQuery selector. For details on jQuery selectors, see http://api.jquery.com/category/selectors/. The `element` can have an empty value.

action (mandatory)

The `action` specifies the DOM event to track. The trigger is matched if the DOM event specified by the `action` is targeted at the DOM element specified by the `element` attribute. The value of `action` can be set to any JavaScript event type, such as focus, mouseover, or resize. In addition to the standard DOM events, the DSL supports the following two values: `timer` and `enterpress`.

timer

If you set `action` to `timer`, this allows triggers to be based on elapsed time. The amount of time is specified by appending the number of milliseconds to `timer`, separated by a colon (":"). For example, `action=timer:10000"`, specifies a 10-second timer.

When `action="timer:nnn"`, you must provide an additional attribute, `type`, to specify how the timer works. You can set `type` to either `timeout`, `notyping`, or `nomove`. If `type="timeout"`, the timer interval begins after the page is loaded.

In the following example, the "InactivityTimeout" event is generated once the user has been inactive for 10 seconds:

```
<event name="InactivityTimeout">
      <trigger name="InactivityTimeout" element="" action="timer:10000" type="nomove"
     url="http://www.genesys.com" count="1"/>
      <val name="products" value="…” />
</event>
```

If `type="timeout"` were specified instead, the event would be generated 10 seconds after the page was loaded.

enterpress

If you set `action` to `enterpress`, this event signals that the user has pressed the "enter" key. This action is more specific than the standard DOM `keypress` event, which is raised when any key is pressed. In the following example, the user enters text in a search box and presses the "enter" key (as opposed to clicking the "search" button).

```
<event name="Search">
      <trigger name="SearchKeyDown" element="input.searchfield:text" action="enterpress"
url="http://www.MySite.com" count="1"/>
      <val name="searchString"    value="$('input.searchfield:text').val();"/>
</event>
```

type (mandatory when action="timer:nnn")

The `type` attribute is mandatory when `action="timer:nnn"`. The `type` can have a value of either `timeout`, `notyping`, or `nomove`, which specifies how the timer `action` works.

If `type="timeout"`, the timer interval begins after the page is loaded. If `type="nomove"`, the timer resets each time the user moves the mouse. If `type="notyping"`, the timer resets each time the browser registers keyboard input for the element specified in the `element` property or for any element on the page, if this property is not specified. (Note that for this type, moving the mouse will not reset the timer.)

In the following example, the "InactivityTimeout" event is generated after the user has been inactive for 10 seconds.

```
<event name="InactivityTimeout" condition="$('h1').text() == 'Compare'">
    <trigger name="InactivityTimeout" element="" action="timer:10000" type="nomove"
       url="http://www.MySite.com/site/olspage.jsp" count="1"/>
    <val name="products" value="…" />
</event>
```

If `type="timeout"` was specified instead, the event would be generated 10 seconds after the page was loaded.

url (optional)

The `url` attribute defines the URL of the specific page that raises the business event. The business event is not submitted if the current document's URL does not match the URL parameter. This attribute can contain a JavaScript regular expression for complex use cases, as shown in the following example:

```
<event name="ExampleEvent">
    <trigger name="SimpleUrlTrigger" element="" action="timer:10000" type="timeout"
url="http://www.genesys.com/customer-experience" count="1"/>
    <trigger name="RegexpUrlTrigger" element="" action="timer:10000" type="timeout"
url="solutions|platform-services" count="1"/>
</event>
```

Note: When the `url` attribute contains one or more ? characters, you must escape them by preceding them with a backslash. For example, http://www.genesys.com/?page=customer-experience would be escaped as http://www.genesys.com/\?page=customer-experience.

count (optional)

The `count` attribute specifies how many times the trigger needs to be matched before the event is generated and sent to the Web Engagement Server.

after (optional)

You can use this attribute to specify the trigger sequence. Note that you can only use the name of the trigger from the current event with this attribute. Trigger names from other events cannot be used.

```
<event name="MySequenceEvent">
  <trigger name="buttonTrigger1" element="#button1" action="click" url="" count="2" />
  <trigger name="buttonTrigger2" element="#button2" action="click" url="" count="1"
after="buttonTrigger1" />
  <trigger name="buttonTrigger3" element="#button2" action="click" url="" count="1"
after="buttonTrigger1" />
</event>
```

In the current example, `buttonTrigger2` and `buttonTrigger3` are initiated only when a button with an ID of `button1` has been clicked twice (count=2).

`MySequenceEvent` will be generated only when all three triggers are executed.

The `after` attribute can be used with a timer trigger, as shown below.

```
<event name="MySequenceWithTimerEvent">
    <trigger name="SelectPlan" element="#button1" action="click" url="" />
    <trigger name="TimeoutTrigger" element="" action="timer:5000" type="timeout" url=""
after="SelectPlan" />
</event>
```

<val>

The <val> element can be used to add data to the business event. You can have 0 or more <val> elements; each instance adds a field to the business event. If <val> is a child of <trigger>, it can also have access to the DOM event matched by the trigger.

name (mandatory)

The name attribute is the name of the value in the generated business event. The name of each val must be unique inside a parent event. The name is added to the generated business event's data, along with the corresponding value attribute.

The following example adds a value named "searchString" when the "Search" event is generated and sent to the Web Engagement Server.

```
<event name="Search">
    <trigger name="GoSearchClick" element="td#gobutton input" action="click" url=" "
count="1"/>
    <val name="searchString" value="$('input.searchfield:text').val();"/>
</event>
```

The following output is an example of an event (in JSON format) submitted to the Web Engagement Server when the visitor enters "my search string" in the search box and clicks the search button. The "eventName" parameter is taken from the name attribute of the <event> element, and the <param> element causes the "searchString" parameter to be added to the event's "data" field (this examples assumes that the visitor entered "my search string" as the search text). The additional fields are generated automatically by the DSL code:

```
{
   "data":{
      "searchString":"my search string"
   },
   "eventType":"BUSINESS",
   "eventName":"Search",
   "eventID":"D88B2FF5A9C24095837CF105FB6D5CF9",
   "pageID":"A9D1E9265D444351876C13D6C5FA5FAD",
   "timestamp":1309962580226,
   "globalVisitID":"7E67BA9701124F738CAC80DDFEA1D705",
   "visitID":"4608DD210B034AC18C65C2C2275CD8B6",
   "userID":"",
   "url":"http://www.bestbuy.com/site/",
   "category":""
}
```

value (optional)

The value attribute specifies the value to associate with the name attribute in the field of the generated event. Its value can be any JavaScript code which returns a serializable object.

The following example tracks search events and includes the search string in the event when it is sent to the Web Engagement Server. In this example, since there is only one search input box on the page, the following <param> definition captures the search text and includes it in the generated event:

```
<event name="Search">
    <trigger name="GoSearchClick" element="td#gobutton input" action="click"
```

```
url="http://www.MySite.com" count="1" />
    <val name="searchString" value="$('input.searchfield:text').val();"/>
</event>
```

In the following example, the "AddToCart" event is tracked, including information about the product that was added: name, model, SKU, and price. Tracking by clicking on the "add to cart" button does not provide information about which button was clicked and which product was added to the cart. To get this information, you need to use the DOM event object: "event.target" identifies the clicked button, which can provide information related to the product.

```
<?xml version="1.0" encoding="UTF-8"?>
<event name="AddToCart">
    <trigger name="AddToCartTrigger" element="div.info-side img.bdt-addToCart" action="click"
url="http://www.MySite.com"  count="1">
        <val name="productName" value="$(event.target).parents('div.hproduct').find('h3.name
a').text()"/>
        <val name="productModel"
value="$(event.target).parents('div.hproduct').find('span.model').text()"/>
        <val name="productSKU"
value="$(event.target).parents('div.hproduct').find('span.sku').text()"/>
        <val name="productPrice"
value="$(event.target).parents('div.hproduct').find('h4.price').text().replace('Sale:', )"/>
    </trigger>
</event>
```