Client Samples

Genesys Mobile Engagement 8.5.2

5/7/2025

Table of Contents

Genesys Mobile Services Client Samples

These pages provide a brief overview of the sample applications included with Genesys Mobile Services. As additional sample applications are made available, this guide will be updated to provide supporting information.

> **Important**
>
> The JavaScript Sample is now documented in the Service Management UI Help.

Change History

The following topics have been added or changed in the GMS 8.5.202.03 release.

- Android SDK was added.
- The Android Sample moved to Maven.

The following topics have been added or changed in the GMS 8.5.201.04 release.

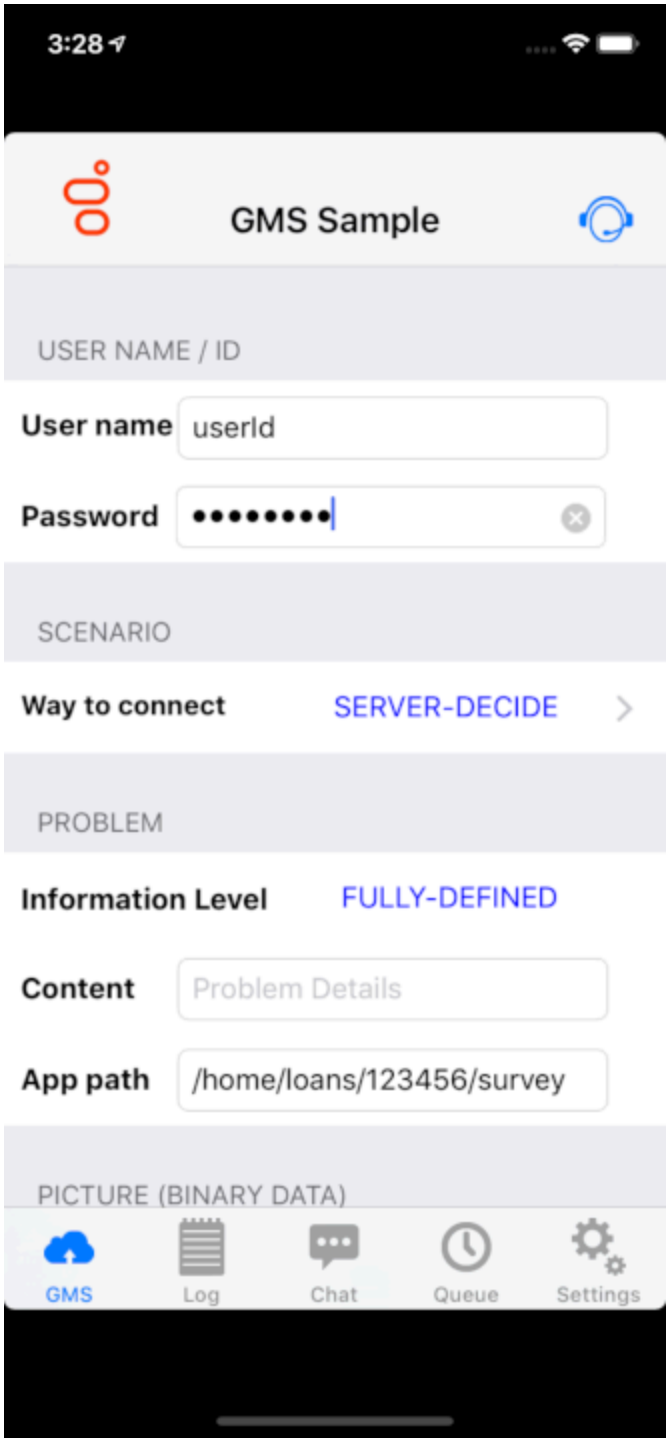
- The Composer Sample for UserOriginated – ClickToCall has been added.

The following topics have been added or changed in the GMS 8.5.200.07 release.

- The Android Sample has been updated.

iOS Sample

Updated in 8.5.2


3:28
GMS Sample
USER NAME / ID
User name
userId
Password
SCENARIO
Way to connect
SERVER-DECIDE
PROBLEM
Information Level
FULLY-DEFINED
Content
Problem Details
App path
/home/loans/123456/survey
PICTURE (BINARY DATA)
GMS
Log
Chat
Queue
Settings

> **Important**

> The source code is downloadable in BitBucket.

This sample application provides you with the following scenarios:

- Chat-Now
- Chat-Wait
- Voice-Now-UserOrig
- Voice-Now-UserTerm
- Voice-Wait-UserOrig
- Voice-Wait-UserTerm
- Voice-Scheduled-UserTerm

Notes:

- Log into the UI hub at this URL:
  `<GMS Local Host>:8080/genesys`
- Go to **Settings** and change the GMS service URL to point to the Callback service that you configured.
- The application does not trim +1 from the phone number. Make sure the phone number in the **Settings** tab is compatible with your test environment if you are using either matching by DNIS+ANI or matching by ANI only. Depending on your setup, T-Server reports ANI with or without a leading country code.

Disclaimer

Android Sample

Updated in 8.5.204, 8.5.200

Download in Bitbutcket

In 8.5.202, the Android Sample moved to BitBucket, where you can download the code sample. The Android SDK used in this sample is available in jcenter/bintray and in Maven, and includes the Javadoc JAR file that you can import in Android Studio. Once imported, the JAR file also provides automatic completion and documentation while using the Android Sample SDK.

> **Important**
>
> All installation and code details are provided in the BitBucket ReadMe where you can download the Android Sample.

In 8.5.114, the app was extended to support:

- Chat V2 features, that are now available in the `CHAT-V2` scenario:
  - File Transfer
  - Typing notifications
  - Typing preview
- Mobile Push Notifications to Android devices using Firebase Cloud Messaging (FCM)
- ApiGee integration

Disclaimer

Android SDK Sample

Introduced in 8.5.204.00

> **Important**
> The SDK described on this page is a sample, not a Genesys product. The related-code is meant to be used with the Android Sample for demonstration purposes only. See the Disclaimer section for further details.

The Android SDK used in this sample is available in jcenter/bintray and in Maven, and includes the Javadoc JAR file that you can import in Android Studio. Once imported, the JAR file also provides automatic completion and documentation while using the Android Sample SDK.

Getting Started

Use your favorite editor to edit the `app/build.gradle` file.

1. Add the GMS SDK with the correct version to the dependencies sections:

```
repositories {
        jcenter()
}
dependencies {
    ...
        implementation 'com.genesys.gms.android:sdk:<version_from_maven_repo>'
    ...
}
```

For example, if you download your SDK from the revision 241 use `8.5.201.00.rev.241` for the version.

2. Ensure that the Android `minSdkVersion` option is not less than 19 and that the Android `compileSdkVersion` option matches the latest version available (here, 27).

```
android {
    compileSdkVersion 27
    defaultConfig {
        ...
            minSdkVersion 19
        ...
    }
}
```

Structure

The Android SDK is divided into two parts:

- `Settings`: Handles all the settings that the API needs to connect with GMS and Callback.
- `Chat`: Handles all the actions required during a Chat session.

See the sections below for further details on using these APIs.

> **Important**
>
> The Javadoc is included in your JAR file.

Using Settings

First of all, to use the Settings part of the SDK, you need to implement the `SettingsActivity` class and create a `SettingsManager` instance. This instance contains and handles all the settings that you need for each subsequent actions.

For example:

```
public class SettingsActivity extends Activity implements SettingsHandler {

    [...]
    SettingsManager settingsManager;

    public void aMethod() {

        //Initialize settingsManager
        //SettingsManager(Context, SettingsActivity)
        settingsManager = SettingsManager.createInstance(this, this);
    }

    @Override
    public void onError(Exception exception) {

    }

    @Override
    public void dial(DialogResponse dialog) {

    }

    @Override
    void availableTimeslots(Map<String, String> var1){

    }

}
```

To retrieve the `SettingsManager` instance after the initialization, use the `getInstance` method.

```
SettingsManager settingsMgr = SettingsManager.getInstance();
```

In this handler, you can implement the `dial` method to process the dialog response. The example below provides an example of implementation:

```
@Override
public void dial(DialogResponse dialog) {
    switch (dialog.getAction()) {
        case DIAL:
            Log.d("Label", dialog.getLabel());
            callbackApi.makeCall(Uri.parse(dialog.getTelUrl()));
            break;
        case MENU:
            if (dialog.getContent() == null || dialog.getContent().size() == 0) {
                // It's empty! No menu to show...
                break;
            }
            DialogResponse.DialogGroup group = dialog.getContent().get(0);
            String groupName = group.getGroupName();
            final List<DialogResponse.GroupContent> groupContents = group.getGroupContent();
            if (groupContents == null || groupContents.size() == 0) {
                // It's empty! There are no options...
                break;
            }
            String[] menuItems = new String[groupContents.size()];
            for (int i = 0; i < groupContents.size(); i++) {
                menuItems[i] = groupContents.get(i).getLabel();
            }
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle(dialog.getLabel() + "\n" + groupName)
                    .setItems(menuItems, new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int which) {
                            String url = groupContents.get(which).getUserActionUrl();
                            settingsManager.startDialog(url);
                        }
                    });
            builder.create().show();
            break;
        case CONFIRM:
            String text = dialog.getText();
            Toast.makeText(this, text, Toast.LENGTH_LONG).show();
            break;
        default:
            break;
    }
}
```

Ensure to save all the settings by using the `saveDatas` method:

```
settingsManager.saveDatas(settings);
```

The `settings` object stores all the useful settings, like the name and telephone number of the user and all server settings such as the address, the port, and so on. The `saveDatas` methods allows to save all the settings and get them later or after closing the application. For example, after an `onResume` event in an activity, retrieve the saved data by calling `SettingsManager.getDatas()`:

```
settings = SettingsManager.getDatas();
```

When you start a call or a chat session as detailed in other sections, ensure to configure all the settings and then use the `connect` method like in this snippet:

```
public void sendData () {
    Map<String, String> params;

    settingsManager.saveDatas(getInformations());
```

```
    ScenarioType scenario =
ScenarioType.fromString(scenarioSpinner.getSelectedItem().toString());

    params = callbackApi.getScenarioParams(getInformations(), scenario);

    if (scenario == ScenarioType.CHAT_V2) {
        Intent intent = new Intent(this, ChatActivity.class);
        startActivity(intent);
    } else {
        settingsManager.connect(serviceNameTextView.getText().toString(), params);
    }

}
```

Important

Remember that if you want to start a scheduled call, you must set the `dateTime` attribute in the `settings` instance as an ISO 8601 date.

ChatHandler

To start a chat session, create the UI and open your own activity. To create your `ChatActivity` class, extend the `Activity` class and implement the `ChatHandler` interface.

```
public class ChatActivity extends Activity implements ChatHandler {
    [...]
    ChatClient chatClient;

    public void aMethod() {
        //ChatClient(Context, ChatHandler)
        chatClient = new ChatClient(context, this);
    }

    @Override
    public void onMessage(ChatV2Response response){
        [...]
    }

    //The url of the downloaded file
    @Override
    public void onFileDownloaded(String url){
        [...]
    }


    @Override
    public void onFileUploaded(Uri uri){
        [...]
    }


    @Override
    public void onError(Exception e){
        [...]
    }

}
```

First create an instance of `ChatClient`, like in the `aMethod` example above. After initializing the API,

you can start a Chat session by calling the `connect` method.

```
chatClient.connect();
```

The `connect` method automatically starts the chat session and call the “onMessage” or “onError” when it is needed. You can send chat start and stop typing events by calling `startTyping` and `stopTyping` methods, as shown below:

```
chatClient.startTyping()
chatClient.stopTyping()
```

To send a message, simply use the `sendMessage` method.

```
chatClient.sendMessage(message)
```

You can also download files through the `downloadFile` method.

```
chatClient.downloadFile(messageId)
```

To download a file from the chat session, use the messageID that is provided in the response of the “onMessage” event. The `downloadFile` method returns the URI of the given file where the user can start the download.

To end the chat session, call `disconnect`:

```
chatClient.disconnect()
```

Firebase Cloud Messaging

Refer to the instructions detailed in the Android Sample’s Readme to implement FCM. Once all the firebase instructions are completed (Connect app, add FCM, access the registration token, handle message), you can receive messages in the `onMessageReceived()` method. Process the message in your handler, for example, as follows:

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    // Check if message contains a data payload.
    if (remoteMessage.getData().size() > 0) {
        SettingsManager settingsManager = SettingsManager.getInstance();
        settingsManager.handleMessage(remoteMessage.getData().get("message"));
    }
}
```

Disclaimer

GENESYS AND ITS SUPPLIERS BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, ECONOMIC, INCIDENTAL, OR SPECIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, ANY LOST REVENUES OR PROFITS).

Classic Callback Sample

You can download the Classic Callback (VoiceXML) sample application via a zip file. The zip file also includes Composer callflow diagrams.

Genesys Mobile Services Classic Callback Sample

> **Important**
> This example's strategy was created with Composer 8.1.300. If you are using Composer 8.1.400 or newer, you must upgrade the Composer project.

Overview

The Genesys Mobile Services (GMS) Classic Callback sample illustrates how to implement an IVR (Genesys Voice Platform VoiceXML) application that communicates with GMS and performs classic Callback scenarios. It is primarily meant to be used by developers as a reference to build a Composer callflow application that interacts with GMS.

Scenarios

The following scenarios are implemented as one Composer project, which consists of one driver callflow and a corresponding subcallflow for each scenario.

IVR Controlled Callback Using GMS Stat Server API

1. Caller is in Genesys Voice Platform (GVP) and requests an Agent.
2. GVP gets the estimated wait time (EWT) using GMS Stat Server API.
3. GVP logic determines Callback should be offered (EWT threshold).
4. GVP offers Caller the choice of a Callback ASAP or to stay on the line and wait for an Agent.
    - Caller chooses Callback ASAP and hangs up. GVP creates a new service request to GMS using Callback API. Caller is called back when an Agent about to become available.
    - Caller chooses to stay on the line and wait for an Agent, and the call is connected when an Agent is available.

IVR Controlled Callback Using GMS Callback API

1. Caller is in GVP and requests an Agent.

2. GVP start a new GMS service (HTTP API) and a virtual interaction is placed in the queue to hold the position of the caller.
3. GVP gets the estimated wait time (EWT) and Position for the virtual interaction using the GMS Callback API.
4. If the EWT-Position Threshold is not met, implying that the wait time is not long.
    1. Wait for an Agent by continuing polling GMS Callback Service API until Agent becomes available or the EWT threshold is reached.
5. If the EWT-Position Threshold is met, implying that the wait time is long.
    1. GVP offers Caller the choice of a Callback as soon as an Agent becomes available and without losing position in the queue, or to stay on the line and wait for an Agent. Optionally informs the caller about the EWT and Position in the queue.
    2. Caller chooses Callback and hangs up. The caller is called back when an Agent is about to become available.
    3. Caller chooses to stay on the line and wait for an Agent, and the call is connected when an Agent is available.

Prerequisites

In order to use this sample application, the following prerequisites are required:

- GMS installed and running, with JDK 1.7.
- The services that you want to use must be deployed.
- The sample can be deployed on all web servers supported by Composer 8.5.
- (Optional to play VoiceXML) MRCP enabled for GVP using the tts option. GVP uses MRCP speech synthesis technology to incorporate text-to-speech for use in voice applications.

Running the Sample

1. Download the zip file.
2. Extract the zip into the following directory: `<gms install folder>/webapps`.
3. Modify the `<gmsinstalldir>/webapps/ClassicCallbackSample/src/AppRoot.vxml` file to set values for the following variables, which are dependent on the environment:
    - `gms_uri`
    - `gms_service_name`
    - `stat_api_params` – this value should be same as the Virtual Queue specified in the GMS service configuration for the preceding service (gms_service_name)
    - `gms_inbound_service_rp` – any routepoint where a strategy/workflow can route the call to the agent.

- `gms_transfer_routepoint` - routepoint where the GMS classic call inbound service is configured.

4. Edit the start.ini file to make sure that it contains:

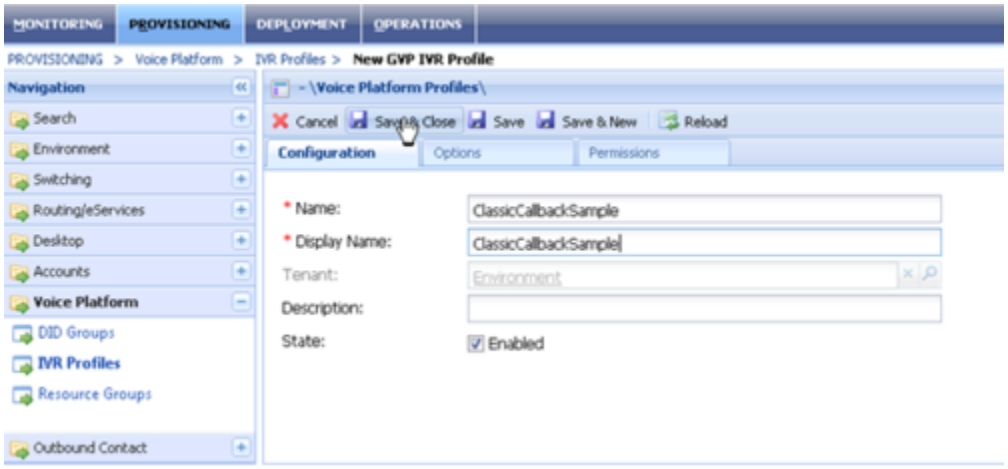
```
--
module=server,jsp,jmx,resources,websocket,ext,plus,annotations,deploy,security,servlets,continuation
etc/jetty.xml
etc/jetty-ssl.xml
etc/jetty-deploy.xml
etc/jetty-http.xml
etc/jetty-https.xml
jetty.send.server.version=false
```

> **Important**
>
> Comment any **rewrite** line. You should not run the sample in a Production server.

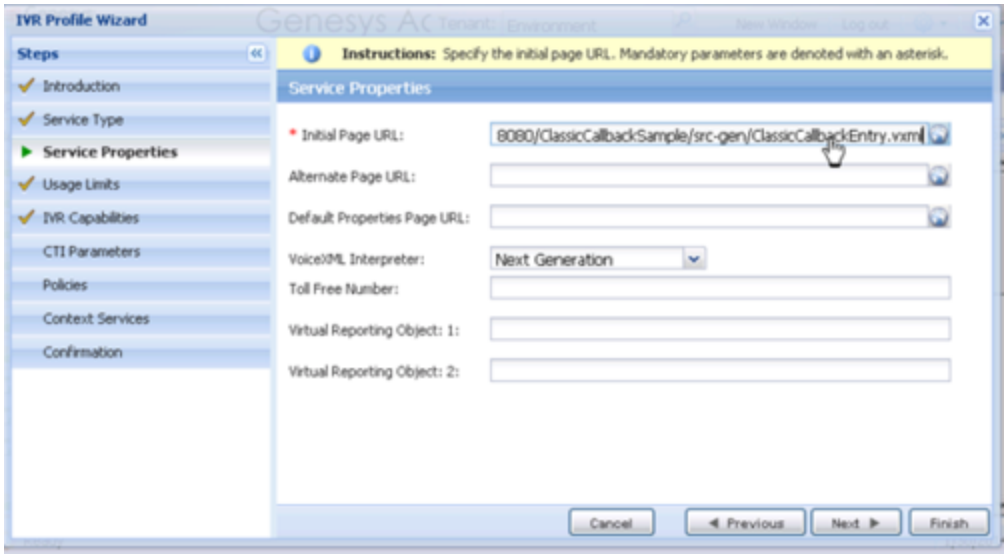
Verify that the following URL is accessible: http://<gmshost>:<gmsport>/ClassicCallbackSample/src-gen/ClassicCallbackEntry.vxml

Configure IVR Objects

1. In Genesys Administrator > Provisioning, create a new IVR Profile named `ClassicCallbackSample` with a Service Type of `VoiceXML`.



2. Provision the IVR Profile to point to the URL shown in Step 4.



3. Setup a DID group (a trunk group to the IVR Profile mapping).

4. Configure the DID as a `TrunkGroup` that points to Resource Manager.

5. Make a call to the DID and follow the voice prompts.

Brief Introduction to the Code (Composer Callflow Diagrams)

ClassicCallbackEntry.callflow

1. Common entry into the IVR samples for all of the supported scenarios.
2. Acts as a driver application for the sample.
3. Plays welcome message.
4. Collects user data (account number).
   - 1111 - for Scenario 1.
   - 2222 – for Scenario 2.
   - Other – play error prompt and repeat Step 4.
5. Based on the selected scenario, invokes the following subroutines:
   - SubGMSStatsCallback.callflow for Scenario 1.
   - SubIVRConrolledGMSCallback.callflow for Scenario 2

SubGMSStatsCallback.callflow

1. Get EWT using GMS Statistic API .../genesys/1/statistics (EstimatedWaitTime).
2. Get current interactions waiting in queue .../genesys/1/statistics (current_In_Queue).
3. Check Callback offer threshold.

- Offer Callback.
  - Invoke SubCallbackOfferDialog.
  - Callback declined.
    - Yes, default route the call (continue with no Callback).
    - No, exit.
- Do not offer Callback.
  - Default route the call (continue with no Callback).

SubIVRControlledGMSCallback.callflow

1. Start Callback service (USERORIGINATED).
2. Get EWT and Position using GMS Callback API .../genesys/1/callback (check-queue-position).
3. Check agent available.
   - Agent available.
     - Get access number.
     - Blind transfer to access number.
     - Exit.
   - Agent not available, so check Call

Custom Callback Plugin Sample

You can download the GMS Custom Callback Plugin sample application via the following zip file, which includes Composer call flow and workflow diagrams.

- Genesys Mobile Services Custom Callback Plugin Sample, version 8.5.114

For a description of the On-Dial plugin interface and configuration, see the Playing Treatments page in the Callback Solution Guide.

Plugin Description

The Genesys Mobile Services (GMS) Custom Callback Plugin sample implements an On-Dial plugin to interface with the GMS Callback service. Developers should use this sample as a reference to build a Composer application that is invoked as a plugin from GMS Callback. Though the actual functionality of the plugin can be extended to execute custom logic like sending push notifications, SMS, or emails, it is beyond the scope of this sample to demonstrate these. For implementing custom logic, refer to GMS, ORS, and Composer documentation.

This example covers:

- Parsing the JSON content of the `_user_data` parameter into an object.
- Checking the `_interaction_id` parameter to determine if the call was answered.
- Checking the `_call_state` parameter for the human answer and machine answer use cases.
- Playing a music file message for scenarios involving a machine answer.
- Invoking a VXML application for scenarios involving a human answer.
- Setting user data in the interaction.
- Implementing a reusable sub-workflow that detaches the interaction and returns the plugin reply.

Note that the Callback Service template provides default treatments when an outbound call reaches a human or not. Customization of the template is required to provide custom treatment. In this release, the custom treatment is supported through SCXML or VoiceXML plugins as configurable options for the Callback Service template. This reduces the customization efforts and gives you more control over how the outbound call is treated.

Prerequisites

In order to use this sample application, you need the following:

- GMS installed and running, with JDK 1.7 or JDK 1.8.

- The services that you want to use must be deployed.
- The sample can be deployed on all web servers supported by Composer 8.5.

Running the Sample

1. Download the zip file here
2. Unzip to C:\temp to create a `GMSExamplePlugin` folder.
3. Open Composer.
4. Browse `File > Import > General` and select the `GMSExamplePlugin` folder. Click **OK** to import.
5. Generate the code and deploy the project.
6. Make sure that the scxml application is accessible via `http`.
7. Configure _plugin_on_dial_url = <scxml http path>
8. Execute the `VOICE-USERTERM-NOW` scenario.

Disclaimer

Interaction Workspace Plugin Sample

You can download the Interaction Workspace Plugin sample application binary and sources using the following links. The Sources zip file contains a Visual Studio project for customization.

> **Important**
> Interaction Workspace (IW) is also known as Workspace Desktop Edition (WDE).

Modified in 8.5.200

Sources

- Genesys Mobile Services IWS Plugin Sample – Sources version 2.3

Overview

The Interaction Workspace Plugin (IWS) sample illustrates how to implement an IWS plugin application that communicates with GMS and allows an agent to do the following:

- Preview a Callback, and then decide to accept or reject the request.
- Set disposition on completed/cancelled Callbacks, and reschedule or retry a Callback when the outbound call is unsuccessful.

The sample is primarily meant to be used by developers as a reference to build an IWS plugin application that integrates with Interaction Workspace and interacts with GMS to provide Preview/Retry Callback.

Scenarios

- For the Preview feature, see the User Terminated Preview scenario.
- For the Retry feature, all User Terminated scenarios can be used with:
  - _enable_disposition_dialog = true
  - _business_hours_service set with the name of your Office Hours service.

See Preview and Disposition Scenarios page for further details.

Prerequisites

In order to use this sample application, the following prerequisites are required:

- GMS 8.5.005.xx and higher for Preview Callback
- GMS 8.5.006.xx and higher for Preview and Retry Callback
- Interaction Workspace 8.1.401.20 or Workspace Desktop Edition 8.5.106.21 or higher
- (Optional for customization) Visual Studio Development Environment (use Visual Studio Professional 2013) for Interaction Workspace 8.1.401.xx version or for Workspace Desktop Edition 8.5.106.21+ versions
  - Install .NET framework 3.5 (for IWS 8.1.401.xx) or .NET framework 4.5 (for IWD 8.5.106.21+); then, select the correct .NET framework to build the project.

Compiling the Sample

Download the Visual Studio project and extract the files to create a folder `GMS_IWSPlugin`. Open the `GMS_IWSPlugin.sln` file in Visual Studio.

Procedure

1. This project is primarily dependent on the IWS libraries. To ensure that the dependency libraries are available to the project build process, copy the contents of `<Interaction Workspace Install Folder>/InteractionWorkspace` (folder containing `InteractionWorkspace.exe`) in the build target folder. You can do this by setting `Project Properties > Build Events > Open file > Pre-build` event command line. For example: `xcopy "C:\Program Files (x86)\GCTI\Interaction Workspace\InteractionWorkspace" "$(TargetDir)" /y /i /s`
2. For debugging, make sure that the `Project Properties > Debug` settings are correct.
3. After customization is complete and a successful build is done, copy the contents of `$TargetDir` back to the folder containing `InteractionWorkspace.exe` in the IWS installation.

```
<Dictionary EnglishName="English" CultureName="English" Culture="en-US">
<Value Id="InvitationDialog.FormCaption" Title="Callback Invitation" />
<Value Id="InvitationDialog.RemainingTime" Title="seconds remain" />
<Value Id="InvitationDialog.Message" Text="A personal callback is offered to you. Please
refer to the information below and use one of the response buttons." />
<Value Id="DisplayData_1" Translation="Field 1" />
<Value Id="DisplayData_2" Translation="Field 2" />
<Value Id="DisplayData_3" Translation="Field 3" />
<Value Id="DisplayData_4" Translation="Field 4" />
<Value Id="DisplayData_5" Translation="Field 5" />
<Value Id="DisplayData_6" Translation="My New Preview Field" />
<Value Id="ResponseButton_accept" Translation="Accept" />
<Value Id="ResponseButton_reject" Translation="Reject" />
<Value Id="ResponseButton_reschedule" Translation="Reschedule" />
<Value Id="ResponseButton_xcancel" Translation="Cancel" />
</Dictionary>
```

Running the Sample

Download the binary files and extract them to the Interaction Workspace or Workspace Desktop installation directory. Make sure that the binary files are extracted into the same folder as the `InteractionWorkspace.exe`.

Setting Preview Field Labels and Adding/Removing Fields

1. Open the file `<plugin dir>/Languages/Genesyslab.Desktop.Modules.GMS.CallbackInvitation.en-US.xml`.
2. Update the **bolded** text to intended values, and then restart Interaction Workspace. These new values will be displayed in the IWS plugin dialog on GMS preview.

```
<Dictionary EnglishName="English" CultureName="English" Culture="en-US">
  <Value Id="InvitationDialog.FormCaption" Title="Callback Invitation" />
  <Value Id="InvitationDialog.RemainingTime" Title="seconds remain" />
  <Value Id="InvitationDialog.Message" Text="A personal callback is offered to you. Please
refer to the information below and use one of the response buttons." />
  <Value Id="DisplayData_1" Translation="'''Field 1'''" />
  <Value Id="DisplayData_2" Translation="'''Field 2'''" />
  <Value Id="DisplayData_3" Translation="'''Field 3'''" />
  <Value Id="DisplayData_4" Translation="'''Field 4'''" />
  <Value Id="DisplayData_5" Translation="'''Field 5'''" />
  <Value Id="DisplayData_6" Translation="'''Field 6'''" />
  <Value Id="ResponseButton_accept" Translation="Accept" />
  <Value Id="ResponseButton_reject" Translation="Reject" />
  <Value Id="ResponseButton_reschedule" Translation="Reschedule" />
  <Value Id="ResponseButton_xcancel" Translation="Cancel" />
  <Value Id="ResponseButton_done" Translation="Done" />
  <Value Id="DispositionDialog.FormCaption" Title="Callback Disposition" />
  <Value Id="DispositionDialog.RemainingTime" Title="seconds remain" />
  <Value Id="DispositionDialog.Message" Text="A disposition is offered to you. Please refer
to the information below and use one of the response buttons." />
</Dictionary>
```

3. Add or remove the `<Value Id=”DisplayData_...` items as necessary while ensuring that the `DisplayData_1, DisplayData_2...` order is maintained.

About the Code

UserEventListener.cs

- `EventHandlerTServerReceived` – Listens to all user events and processes only the Callback preview user events from GMS Callback.

InvitationDialog.xaml.cs

- `PlaceDisplayData` – Loads the preview data from `userevent` into the preview dialog.
- `PlaceResponseButtons` – Creates clickable buttons to allow agents to take action after previewing the displayed Callback information.
- `ResponseButtonClick` – Returns an agent response to GMS based on the button clicked.

DispositionDialog.xaml.cs

- `PlaceDisplayData` – Loads the disposition data from `userevent` into the disposition dialog.
- `PlaceResponseButtons` – Creates clickable buttons to allow agents to send disposition after taking actions according to the displayed Callback disposition information.
- `ResponseButtonClick` – Returns an agent response to GMS based on the button clicked.

Disclaimer

Statistic Custom Template Sample

You can download the sample custom template for statistics using the following link:

Statistic Custom Template Sample

Overview

GMS enables you to easily develop your own service by using services composition and custom business logic. The sample template provided here addresses a statistic use case on how to get information about `maxInteractions`, `currentMargin`, and `currentInteraction` from a group of approximately 500 agents. The goal is to use this statistic on the customer-side (browser) to show how quickly a request has been answered.

The response from the GMS statistic will be similar to the Pulse statistic. For example:

```
{
    "maxInteractions": 2,
    "currentInteractions": 0,
    "currentMargin": 2
}
```

While the GMS Stat Service APIs allow you to query statistics against Stat Server, the APIs return the statistic response as a JSON object. For the use case described here, the response from GMS for this type of statistic may not meet business needs with the response size, CPU consumption, and security.

However, the benefits of using the Statistic custom template are:

- Network Load - Size of transmitted information is significantly reduced.
- CPU Consumption - Reduced on client-side; all parsing/formatting process is completed on the GMS/ ORS-side.
- Security - Sensitive information is not exposed.

Prerequisite

GMS 8.5.005.xx and higher.

Running the Sample

1. Download the zip file. You can save the zip file to the same directory as the templates that are included with the GMS installation: `<GMS installation directory>/service_templates`, or you can create a

new directory to store your custom templates.

2. Go to the GMS Service Management UI > Tools > Service Templates.
3. Load the template:
    1. Click `Add Service Template`.
    2. Select the `get-gastat.zip` file.
4. Create the service:
    1. Go to the GMS Service Management UI > Services > Configured Services.
    2. Click `Add Service`.
    3. Select the `get-gastat` service, and enter a name for your service.
5. You can now configure the custom statistic service.
6. When you have finished with the configuration, you can call your new service using a REST client.

About the Code

The template is a zip file that contains two files:

- `get-gastat.json` - Contains the configuration options associated with the service (service type, statistic details, and so on).
- `get-gastat.scxml` - Contains the logic of the service. In this sample, it contains a call to the GMS Stat Service API as well as the parsing and formatting logic.

Disclaimer

Composer Sample for UserOriginated – ClickToCall

This sample enables you to customize the Click-to-Call scenario, also known as User-Originated or User-Orig scenario in the Genesys Mobile Services Deployment Guide. You can download this sample using this link:

Genesys Mobile Services Composer Sample

Customize and import your Scenario

1. Download and unzip the above zip file.
2. Modify the sample source code and generate new SCXML files using Composer.
3. Navigate to the `<GMS Installation Directory>/service_templates` folder.
    1. Unzip the callback.zip file.
    2. In the callback subdirectory, replace the required SCXML files with the SCXML files that you generated.
    3. Zip the `callback` folder to replace the current callback.zip file.
4. Start the Service Management UI and upload the `<GMS Installation Directory>/service_templates/callback.zip` file.
5. Make sure that you successfully complete the steps of the Testing your deployment section of the Genesys Mobile Services Deployment Guide.

Scenario Example

The data used in the following step-by-step are examples that will not match your environment.

1. Create a callback.

2. The Callback session goes to `ReserveImmediate`, and submits a "request-access" query to GMS. GMS returns the `_access_number` and `_access_code` (if needed) in response to the Callback session.

3. When the "request-access" is done, the callback session response is sent to GMS. For example, you could see:

```
'_access_code' [str] = "n/a"
'_expiration_time' [str] = "29"
```

```
'_tel_url' [str] = "[tel:9050020]"
'_action' [str] = "DialNumber"
'_access_number' [str] = "9050020"
'_id' [str] = "118-025ff6e8-8f88-43dd-8174-1d72808c2e7e"
'_label' [str] = "Connecting ..."
'_dialog_id' [str] = "0"
```

The 9050020 access number provides the access to `GMSCallbackInboundGMSMatch`, which has the following `_url` option, for instance: `http://gms:8080/genesys/1/document/service_template/callback/src-gen/IPD_Voice_GMSMatch.scxml`

4. The customer calls the provided `_access_number`, then GMS submits the request to invoke `IPD_Voice_GMSMatch.scxml` (inbound session).

5. In VoiceRoute.scxml, if `_access_code` is required, the system plays the prompt and waits for a DTMF.

6. Then, the script goes to GMSMatch directly, and the inbound session submits a "match-interaction" request with access-number=9050020 (or the access_code if needed) to GMS.

7. If the request succeeds, GMS returns the `_id` and `_data_id` data, for example `_data_id=118-262d1a29-d905-4cfc-bbf7-58129d95a0d4, _id=002IIV4PVKDE1BV41K017B5AES000001`

> **Important**
>
> Here, `_id` is the `ors_session` id and is being passed as `_gms_service_id`.

8. If the service is not a `builtin` service, the script associates the inbound session using the `_gms_service_id` (_id) to the callback session; if the service is a `builtin` service, the script stays in the inbound session.

9. The Callback session goes on with reservation and connects to the agent.

Disclaimer

INCIDENTAL, OR SPECIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, ANY LOST REVENUES OR PROFITS).

Chat Version 2 CometD Sample

This code sample shows how to implement a WebSocket client application for GMS Chat Version 2 using CometD. In this example, the client application opens a CometD connection, submits a chat request, and starts receiving notifications. Once the chat session is over, the application closes the CometD channel.

The client application uses CometD to connect the Chat Server via GMS.

- GMS initiates the Chat session by sending a request to the Chat Server.
- The Chat Server connects the agent and publishes notifications to the client application through GMS CometD channels.
- The client application submits chat messages and receives notifications through GMS CometD channels.

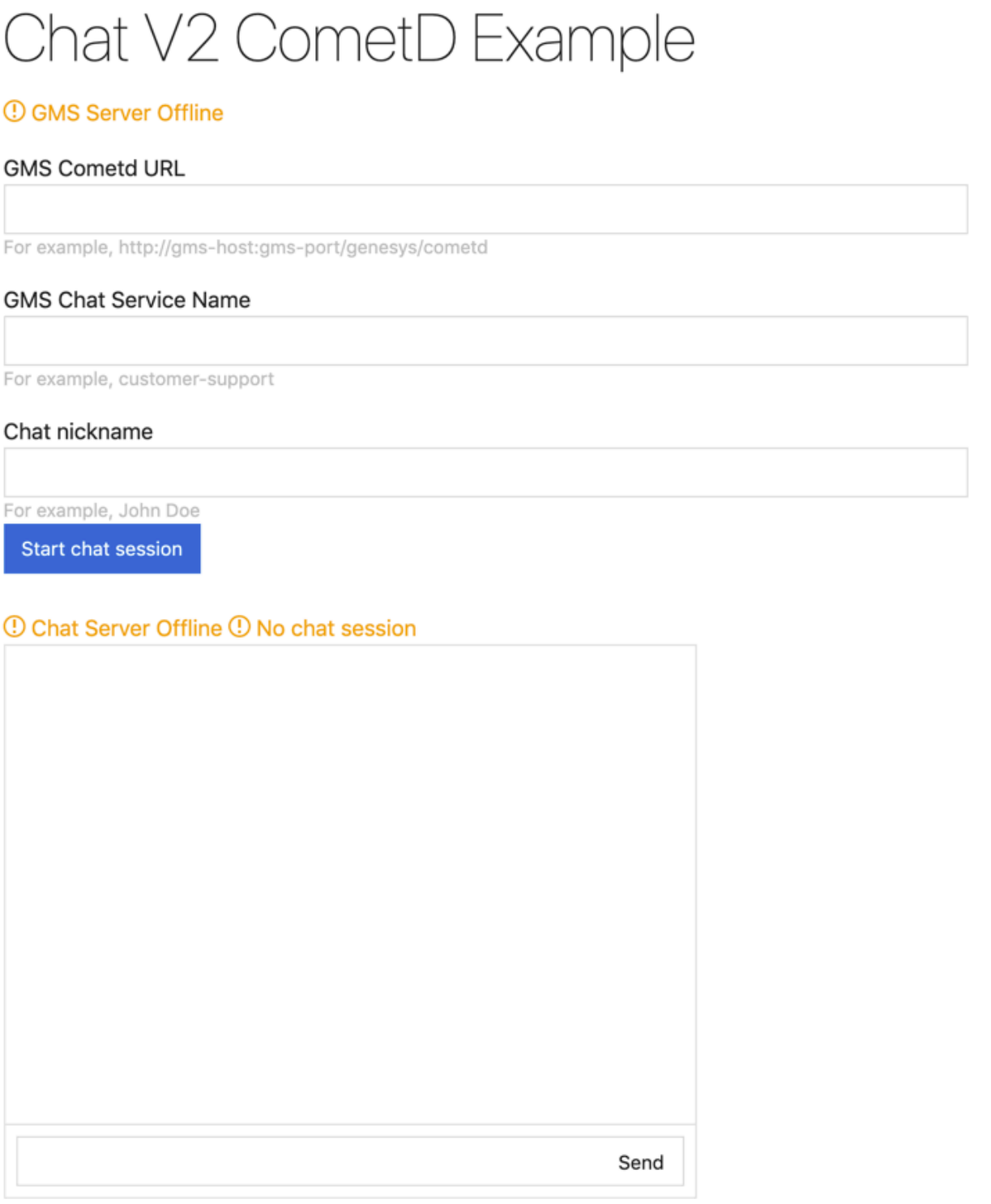
This code sample uses the CometD Project for Javascript, in addition to Vue.js and Metro4, for UI components.

Link to video

Prerequisites

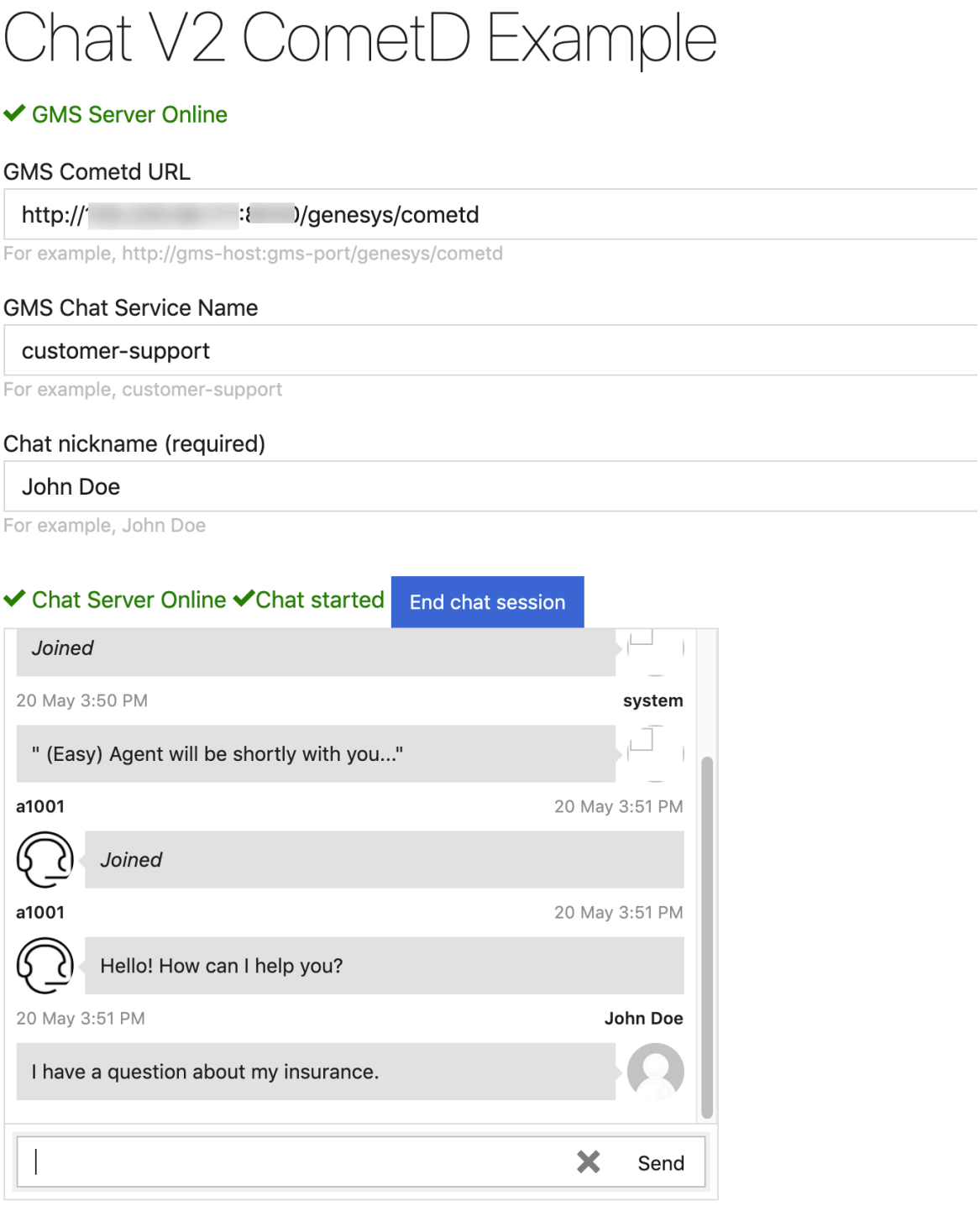
- Genesys Mobile Services 8.5.2+
- Refer to the Prerequisites for the CometD API.
- Configure Digital Channels.
- Create a chat service channel, for example, `customer-support`.
- Download the Chat V2 CometD Sample.

Run the GMS Chat CometD Sample



- Unzip the chat sample zip file.

- In the `chatv2-cometD-sample` directory, run the `npm install` command.
- Open the `index.html` file with a browser.



- Fill in the form, and then click **Start chat session**. You can start submitting chat messages.

Implementation Details

Step 1. Get started with CometD

When you click the **Start Chat** button, the VueJs `chatApp` component (or `chatUi`) calls the `chatV2Start` function which performs the following actions:

1. Initialize and configure a CometD instance. See CometD documentation for further details.
2. Register callback listeners for each channel that submits notifications.
   - CometD Meta Channels
   - GMS CometD Chat service channels which are responsible for publishing events and notifications. In this sample, the service used is "customer-support". As a result, the channel for this service is: `/service/chatV2/customer-support`.
3. Handshake.

```
function chatV2Start(url, channel) {

    chatUi.runToast("Connecting", 'info');
    // Instantiate CometD
    // For each chat session, open a CometD connection
    objCometD = new org.cometd.CometD();
    // Optionally, install an extension.
    objCometD.registerExtension('timestamp', new org.cometd.TimeStampExtension());

    // Enable WebSockets
    objCometD.websocketEnabled = objCometDSettings.websocketEnabled;

    // Provide your GMS CometD Endpoint
    objCometDSettings.cometURL = url;
    objCometD.configure({
        url: url,
        logLevel: "info"
    });

    /// Add listeners for cometD meta endpoints
    objCometD.addListener('/meta/handshake', fHandshake);
    objCometD.addListener('/meta/connect', fConnect);
    objCometD.addListener('/meta/disconnect', fDisconnect);
    objCometD.addListener('/meta/publish', fPublish);
    // Listen to chat notifications
    objCometDSettings.channel = channel;
    objCometD.addListener(objCometDSettings.channel, refresh);

    // Now, handshake
    objCometD.handshake();

}
```

> **Important**
>
> The above CometD code is a one-time setup only during the lifecycle of an active CometD connection.

Step 2. Handle ChatV2 API Queries

To simplify the code, this sample calls the `request` method for CometD requests once the chat session is started. The supported operations match the API queries listed in the Chat V2 CometD API reference page, for example:

- `sendMessage`
- `disconnect`
- `requestNotifications`

The `request` function always provides the `secureKey` parameter which is available in the message responses as soon as the chat session is started.

```
function request (operation, params) {
    if (objCometD && boolCometDConnected) {
        var data = {
            operation: operation,
            secureKey: objSessionData.secureKey,
        };
        var finalData = $.extend(data, params);
        objCometD.publish(objCometDSettings.channel, finalData);
    } else {
        chatUi.runToast("CometD is not connected!", "alert");
    }
}
```

Step 3. Implement CometD Meta Callbacks

fHandshake

The CometD library calls this function automatically to submit the result of the Handshake operation.

- If successful, the GMS communication channel is ready and the sample submits a `requestChat` request.
  - If the sample detects a reconnection, it sends a `requestNotifications` request to make sure that you get all chat notifications. Note that the handshake callback function gets automatically called by CometD during any failure trying to re-establish the connection to GMS.
- If not, the Handshake is failed and the sample needs to disconnect CometD.

> **Important**
>
> In this sample, the callback function counts the number of handshake exceptions and disconnects from CometD only when reaching the `intHandshakeExceptionLimit` limit. By default, CometD makes several attempts if you don't close the connection. This ensures to display a connection error only if the handshake keeps failing.

```
function fHandshake(response) {

    console.log('fHandshake', response);
```

```
    if (response.successful === false) {
        if (++intHandshakeExceptionCount === intHandshakeExceptionLimit) {
            /// Too many exceptions, close CometD connection
            disconnectFromGMS();
            intHandshakeExceptionCount = 0;
            // Update UI
            chatUi.stopWaiting();
            chatUi.runToast("Handshake failed", 'alert');
        }
    } else if (response.successful === true) {
        chatUi.runToast("Handshake success", 'info');

        boolCometDConnected = true;
        chatUi.boolGMSConnected = true;
        intHandshakeExceptionCount = 0;

        if (response.secureKey && boolRestoring) {
            chatUi.runToast("Restoring notifications", 'info');
            // If the handshake is successful, submit a request notifications request
            // otherwise client messages are not exchanged properly.
            console.log("request notifications handshake");
            request("requestNotifications", {
                secureKey: response.secureKey,
                transcriptPosition: intLastTranscriptPosition,
                message: "" // get current text in text input
            });
            boolRestoring = false;
        } else {
            chatUi.stopWaiting();
            chatUi.runToast("Start chat session", 'info');
            // Start a chat session
            var requestChatData = {
                operation: "requestChat",
                nickname: chatUi.strNickname,
                subject: "Test CometD Chat v2",
                text: "",
                "userData": {
                    "interests": "javascript"
                }
            }
            objCometD.publish(objCometDSettings.channel, requestChatData);

        }
    }
}
```

fConnect

When implementing Chat V2 using CometD, you can get disconnected from GMS and from the Chat Server, which interrupts the chat session.

- CometD connection status is tracked with `boolCometDConnected` in the **fConnect()** function.
- Chat Server connection status is tracked with `boolChatServerOffline` in the **fDisconnect()** function.

In the **fConnect()** callback, the sample submits a `requestNotifications` request if the Chat Server is disconnected. Otherwise, if the Chat Server was previously connected and if the response is not successful (`response.successful === false`), it means that the `/meta/connect` request failed. In

this scenario, the sample checks for the error received and, if the error code is 402, it processes it as if the Chat Server is disconnected.

Based on the advice received in the response, the CometD library submits the handshake request again, if needed, and does not require that you take care of it.

```
function fConnect(response) {
    if (!objCometD || objCometD.isDisconnected()) {
        boolCometDConnected = false;
        chatUi.runToast("CometD is no longer connected!", "alert");
        return;
    }

    boolCometDConnected = response.successful == true;

    // If chat was disconnected but CometD has reconnected
    if (boolDisconnected && boolCometDConnected) {
        boolDisconnected = false;
        if (!chatUi.boolChatServerOnline) {
            /// Requesting chat messages$
            request("requestNotifications", {
                transcriptPosition: intLastTranscriptPosition
            });
            /// Updating the chat server status
            chatUi.boolChatServerOnline = true;
            chatUi.runToast("CometD reconnected", "info");
        }
    } else if (!boolDisconnected && !boolCometDConnected) {
        if (response.error !== "402:: Unknown client") {
            boolDisconnected = true;
            chatUi.boolGMSConnected = false;
            /// Notify disconnection status
            chatUi.runToast("CometD is disconnected! " + response.error, "alert");

        }
    }
}
```

fDisconnect

When implementing Chat V2 using CometD, the sample or the agent can submit a GMS disconnect request, which interrupts the chat session and also notifies the '/meta/disconnect' channel.

- The CometD connection status is tracked with `boolCometDConnected` in the **fConnect()** function.
- The Chat Server connection status is tracked with `boolChatServerOffline` in the **fDisconnect()** function.

The **fDisconnect** function is called when CometD is disconnected. First, it checks if it needs to re-establish the connection, and if so, it submits a `/handshake` request.

> **Important**
>
> To get notifications after the reconnection, the sample needs to request notifications again. This action is performed in the handshake handler.

```
function fDisconnect(response) {
    console.log('disconnect', response);
    if (response.successful) {
        boolDisconnected = true;
        chatUi.boolChatServerOnline = false;
        chatUi.boolGMSConnected = false;
        chatUi.boolSessionActive = false;
        /// if CometD is not set to false,
        /// trying to reconnect
        if (objCometD) {
            chatUi.runToast("CometD - disconnection", "alert");
            boolRestoring = true;
            objCometD.handshake();
        } else {
            objSessionData.secureKey = "";
            chatUi.runToast("CometD - disconnection", "info");
        }

    } else {
        chatUi.runToast("CometD - disconnection " + response.error, "error");
    }
}
```

Step 4. Process Notifications

Refresh

The `refresh` handler receives chat notifications either received after a successful `requestChat` query or requested with the `requestNotifications` query. In addition to the status code, it receives all of the transcripts and events related to chat messages. In the refresh handler, the sample checks the response status code and updates accordingly.

- An API **statusCode** value of `0` indicates that the operation was successful and all fields in the response have valid values.
    - In this scenario, the sample retrieves the `secureKey` parameter required to submit further Chat V2 requests during the chat session.
- An API **statusCode** value of 1 or 2 indicates that there was an exception and Chat Server is offline.

Further processing of the chat messages is performed in the **parseTranscript()** function detailed in a separate section.

```
function refresh(response) {
    console.log('refresh', response, boolCometDConnected);
    chatUi.runToast("Received notification", "info");
    if (boolCometDConnected) {
        response = response.data;

        /// Make sure to retrieve session parameters
        // only secureKey is required
        if (response.secureKey) objSessionData.secureKey = response.secureKey;

        if (response.statusCode === 2 && response.errors && response.errors.length > 0) {
            /// Process the received errors
            chatUi.boolSessionActive = false;
            var errorToDisplay = "Chat API error: <br/>";
            response.errors.forEach(element => { errorToDisplay += element.advice + "<br/>"
});
```

```
            chatUi.runDialog(errorToDisplay, "alert");
        }


        // Prevents situation where you can't end an expired chat session
        if (response.statusCode === 0) {

            if (!chatUi.boolChatServerOnline) {
                /// Chat Server is back online
                chatUi.boolChatServerOnline = true;
                chatUi.runToast("Chat server is back online", "info");
            }

            if (response.messages.length > 0) {
                // Parse the chat transcript
                parseTranscript(response);
            }

        } else if (response.statusCode === 1) {
            chatUi.boolChatServerOnline = false;
            chatUi.runToast("Chat server is offline", "warn");
        }
    }
}
```

Parse Transcript

In this sample, the **ParseTranscript** function checks all the messages for the current chat session. To avoid publishing multiple times the same messages, it checks the index of each message and publishes only new ones. Additionally, if the chat session has ended, the function also calls the **disconnectFromGMS()** function to close the GMS CometD connection.

```
function parseTranscript(transcript) {
    // Update chat display if needed
    $.each(transcript.messages || [], function () {
        /// TIP: Use the index to check if you previously processed this message
        if (this.index > intLastTranscriptPosition) {
            switch (this.type) {
                case 'ParticipantJoined':
                    chatUi.displayMessage(this.from.nickname, '<i> Joined </i>');
                    break;
                case 'Message':
                    if (this.from.type !== 'Client') {
                        // In this sample, the chat widget already displayed
                        // the client message
                        chatUi.displayMessage(this.from.nickname, this.text);
                    }
                    break;
                case 'ParticipantLeft': chatUi.displayMessage(this.from.nickname, '<i> Left
</i>');
                    break;
                default: console.log(this);
            }
            intLastTranscriptPosition = this.index;
        }

    });
    // Check if the session has ended
    if (transcript.chatEnded === true) {
        chatUi.boolSessionActive = false;
        disconnectFromGMS();
```

```
    }
}

function disconnectFromGMS() {
    console.log("disconnect from GMS");

    /// Clean up
    objSessionData.secureKey = "";
    intLastTranscriptPosition = -1;

    /// Stop cometD
    if (boolCometDConnected && objCometD) {
        objCometD.disconnect();
        objCometD = false;
        boolCometDConnected = false;
    }
}
```

Step 5. Terminate the Chat session

To terminate the Chat session, if the user clicks the **End Chat Session** button, the sample sends a Chat V2 `disconnect` request.

As a result, the sample disconnects from CometD after receiving a response notification with `chatEnded = true`. Note that it also receives a meta disconnect event processed in the `fDisconnect` handler.

```
/// Terminate the chat session before disconnecting CometD
function terminateChatSession() {
    chatUi.runToast("Terminate Chat session...", "warn");
    intLastTranscriptPosition = -1;
    request('disconnect', {});

}
```

Disclaimer