Oracle Fusion Service

How do I use Computer Telephony Integration (CTI)?

Oracle Fusion Service
How do I use Computer Telephony Integration (CTI)?

G29494-08

Author: David Yetter

Contents

Get Help

There are a number of ways to learn more about your product and interact with Oracle and other users.

Get Help in the Applications

Some application pages have help icons to give you access to contextual help. If you don't see any help icons on your page, click your user image or name in the global header and select Show Help Icons. If the page has contextual help, help icons will appear.

Get Training

Increase your knowledge of Oracle Cloud by taking courses at *Oracle University*.

Join Our Community

Use *Cloud Customer Connect* to get information from industry experts at Oracle and in the partner community. You can join forums to connect with other customers, post questions, suggest *ideas* for product enhancements, and watch events.

Share Your Feedback

We welcome your feedback about Oracle Applications user assistance. If you need clarification, find an error, or just want to tell us what you found helpful, we'd like to hear from you.

You can email your feedback to *oracle_fusion_applications_help_ww_grp@oracle.com*.

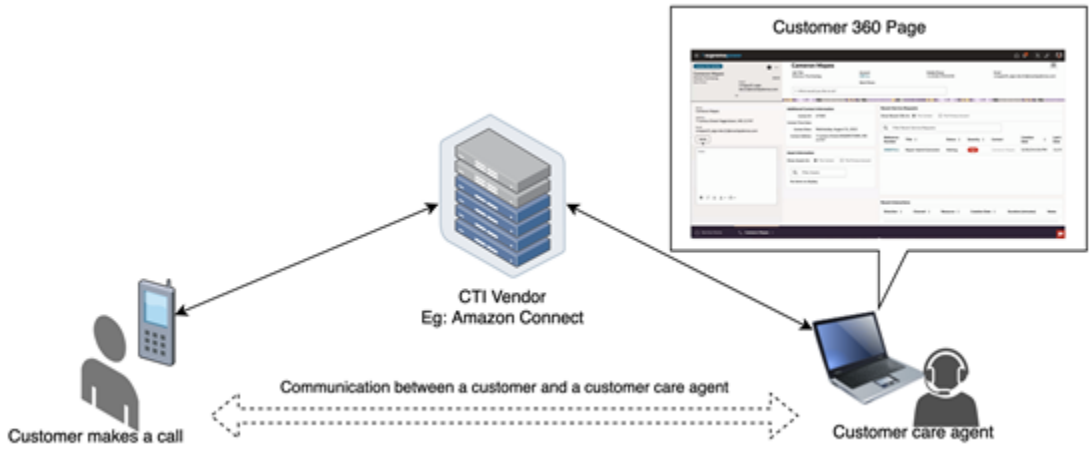
Thanks for helping us improve our user assistance!

1 How do I use Computer Telephony Integration (CTI)?

What's Computer Telephony Integration (CTI)?

Computer Telephony Integration (CTI) enables integration of third-party media toolbars with Service Center, Help Desk, Fusion Sales, and CX for utilities. You can display a media toolbar by enabling the Partner CTI Service and by assigning users the Access Partner Media Toolbar privilege which provides users access to the media toolbar.

Here's a look at a basic CTI flow. A customer calls the customer care number which is received by an agent. Customer information is displayed to the agent on a Customer 360 page. The CTI provider eases the communication between the customer and the customer care agent.



Use CTI to:

- Make an outbound call and skip manual dialing
- Receive phone calls

Integrate with CTI to:

- Receive notifications of incoming calls and either accept or reject the call
- Use automatic caller identification
- Search for a contact
- Get optional caller verification
- Display administrator-defined screen pop pages
- Record interactions
- Use optional call wrap up

Here's an overview of how all the pieces fit together:

- CTI enables the integration of third-party media toolbars with your Fusion application.
- Oracle provides the MCA (Multi Channel Architecture) framework for communication between the Fusion application and the Media Toolbar application (also known as, the Partner application) through a JavaScript library, the UI Events Framework (UEF).
- Consider the media toolbar a container provided by Fusion and the Partner application is embedded in it.
- The Partner application is a web application that you must implement. You can do this, or your Systems Implementor can do it, or you can contact one of Oracle's telephony partners listed on Oracle Cloud Marketplace.

CTI integration overview

The first step for setting up telephony integration with Fusion applications is to select a CTI supplier. The CTI supplier must support both incoming and outbound calls, and provide API support to start the calls from the web application.

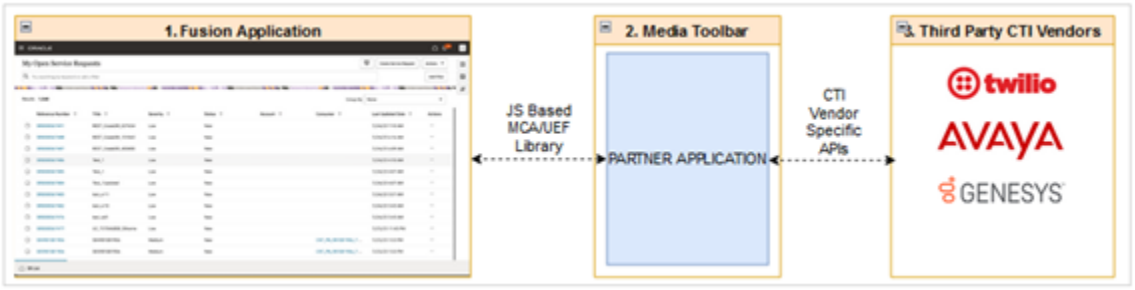
Oracle Fusion CTI integration isn't restricted to a specific CTI supplier, you can choose any CTI supplier and implement the partner application based on the supplier APIs.

Once you choose your CTI supplier and have a basic idea of how to call the supplier APIs, the next step is to implement the Partner application which is a web application.

After implementing the Partner application, you're ready to integrate the partner application with the Fusion media toolbar. The media toolbar is a dialog box from the Fusion application that embeds in your hosted web application.

To use the media toolbar within the Fusion application, you'll need to setup a media toolbar and configure users, providing them the required privileges and access to the toolbar. For configuration details, see *Configure the Media Toolbar*. The communication between the Fusion application and the partner application can be done via a JavaScript library, the UI Events Framework (UEF) which is provided by Oracle. See *How do I use the UI Events Framework?* to get an understanding of UEF. This framework supports both inbound and outbound calls and can be integrated with any third-party JavaScript based applications.

The following diagram gives an overview:



Inbound call flow

The inbound call flow handles call from your CTI provider within the Fusion application.

Outbound call flow

The outbound call from the Fusion application is directed to the partner application and from there to the CTI provider.

What's a Media Toolbar?

Now, let's understand what a Media toolbar is.

Consider the media toolbar as a container provided by Fusion in which the media toolbar application (the partner application) can be embedded as shown in the following diagram:

Use the media toolbar to create communication between the Fusion application and the partner application and do the following:

- Notify Fusion about incoming calls, agent availability, and so on.
- Notify the partner application of user interactions in the Fusion application, such as when an outbound call is started in Fusion, sending a notification that the call was either accepted or declined from the Fusion application.

What's a partner application?

So, what's a partner application? It's a web application that the customer or partner implements by consuming the CTI supplier APIs and UI Events Framework APIs to support CTI features. The customer or partner owns this partner application. The partner application have any of these characteristics:

- An application developed by you
- An application provided by your partner
- Readily available toolbars provided by some CTI vendors

If you're building the partner application, it must be hosted on your server.

> **Note:** From this point onward, whenever we talk about the media toolbar application, we're referring to the partner application built by you, your partner, or its provided by the CTI supplier. You're responsible for hosting the toolbar application and the URL must be made available over `https`.

You use the media toolbar to create a communication channel between the Fusion application and the partner application. Here are a few examples of what you can do:

- Notify Fusion about incoming calls, agent availability, and so on.
- Notify the partner application of user interactions in the Fusion application, such as when an outbound call is started in Fusion, sending a notification that the call was either accepted or declined from the Fusion application.

How can I launch the media toolbar?

You need to configure the media toolbar in Fusion before it's ready to use. Follow the instructions the Configure the media toolbar topic to get up and running.

Once its set up, here's how you launch it:

1. Sign in to the Fusion application.
2. You'll see a disabled phone icon on the homepage.
3. Click the icon to open the media toolbar in a new browser window.

   The partner application you've developed will load into the media toolbar window.

How do I communicate to the Fusion window from the media toolbar?

The media toolbar application you've configured might be a JavaScript application without any integrations with Fusion. To communicate with the Fusion window, you must inject a JavaScript library provided by Oracle called UI Events Framework (UEF) into your media toolbar application to fire the necessary APIs required for the integration. The integration to Fusion from the media toolbar application is a front end-to-front end based integration through the UEF library.

To access the media toolbar must have the **Access Partner Media Toolbar** privilege in *Configure a user for CTI* we'll assign this privilege to a user.

Here's the script:

```
<script src="https://static.oracle.com/cdn/ui-events-framework/libs/ui-events-framework-client.js"></script>
```

2 Set Up CTI Configurations

Configure a user for CTI

You must assign the Access Partner Media Toolbar (SVC_ACCESS_PARTNER_MEDIA_TOOLBAR_PRIV) privilege a user's role for access the media toolbar.

Ready to use roles

These ready-to-use roles already have the `Access Partner Media Toolbar` privilege:

- HR Help Desk Administration
- HR Help Desk Analysis
- HR Help Desk Service Request Management
- HR Service Request Administration
- HR Service Request Analysis
- HR Service Request Management
- Internal Service Request Administration
- Internal Service Request Analysis
- Internal Service Request Management
- Service Request Administrator
- Service Request Power User
- Service Request Troubleshooter

Setup for custom roles

If you're not using any of the ready-to-use roles this section will help you set up your custom roles.

1. Sign in to the Fusion application as an administrator and navigate to **Tools > Security Console**.
2. Click Roles, then search for your custom role by name.
3. Select the role, and click **Edit Role**.
4. In the Edit Custom Role page, click **Next** to go to section 2.
5. Choose the **Privileges** tab, and select **Add Function Security Policy**.
6. Search for the Privilege: **SVC_ACCESS_PARTNER_MEDIA_TOOLBAR_PRIV**.
7. Click **Add Privilege to Role**.
8. Make sure the privilege is show in the Privilege Name column.
9. Click Next until you reach the **Summary** page, then click **Save and Close**.

   Now users associated with the custom role who have the Access Partner Media Toolbar privilege can use the media toolbar.

Configure the Media Toolbar

Your Fusion application will now require some more configuration in the media toolbar for use in your partner application. Here's what you do:

1. Enable SVC_PARTNER_MEDIA_TOOLBAR_ENABLED profile option.
2. Create a toolbar and then assign it to users.

Enable the SVC_PARTNER_MEDIA_TOOLBAR_ENABLED profile option

The SVC_PARTNER_MEDIA_TOOLBAR_ENABLED profile option controls the visibility of the partner media toolbar. If you don't enable the profile option, the partner media toolbar is hidden. You can set this profile option either at the site level or at the user level.

1. Sign in to your Fusion application as an administrator.
2. In the Setup and Maintenance work area, click the **Tasks** icon.
3. Search for **Manage Administrator Profile Values**, then click the task link in the results.
4. In the Profile Option Code field, search for **SVC_PARTNER_MEDIA_TOOLBAR_ENABLED**.

   You can set this profile option at the site level or the user level.
5. Click the **+** button to add a row for the user you want to enable the media toolbar for.
6. In the newly added row, choose the Profile Level as **User** and from the User Name drop-down list, click Search to find your user.

   > **Note:** If you want the partner media toolbar available for all users, set the Site Profile Level to **Yes**.
7. Choose the profile value as 'Yes' for the selected user.
8. Click Save and Close.

   Once finished, your user will see a disabled phone icon when signing in to the Fusion application.

Create a toolbar and assign it to the user

Now that you've enabled the toolbar for your user, the URL of the partner application that you've developed must be configured in the Fusion application to enabled your partner application to load when the media toolbar is opened from the Fusion application.

First we'll create the toolbar.

1. Sign in to your Fusion application as an administrator.
2. In the Setup and Maintenance work area, click the **Tasks** icon.
3. Search for **Manage Media Toolbar Configuration**, then click the task link in the results.
4. Click the **+** button to add a row for the toolbar.
5. Provide the following required information:
   a. A toolbar name.
   b. Set status as **Enabled**.
   c. Set the layout as **Redwood**.
   d. Set the **Communication Panel URL** as the URL of the Partner Application that you've deployed.

6. Click **Save and Close**.

Now we'll assign the toolbar to a user. Do this task only if you've more than one toolbar configured in the Manage Media toolbar section or to assign a toolbar that's not marked as the default toolbar to any of your users.

1. In the Setup and Maintenance work area, click the **Tasks** icon.
2. Search for **Manage Administrator Profile Values**, then click the task link in the results.
3. Enter the SVC_OVERRIDE_PARTNER_TOOLBAR_SELECTION Profile Option Code and click **Search**.
4. Click the **+** button to add a row for the user.
5. In the newly added row, choose the Profile Level as **User** and from the **User Name** drop-down list, click **Search** to find the user.
6. Click **Save and Close**.
7. Choose the toolbar you previously created from the **Profile Value** drop-down list.
8. Click **Save and Close**.

That's all there's to configuring the media toolbar in the Fusion application. For developing the partner application, it's better to understand some basic concepts of CTI integration and the scenarios to be handled as part of the integration. The following topics will give you an overview of the media toolbar integration along with some basic concepts to consider when building a partner application

3 Media Toolbar Integration

Introduction to the media toolbar integration

Now that you've gone over the Fusion application configurations needed for loading your partner application, you can now launch the media toolbar and your CTI application will load inside the media toolbar. Now you can use the UI Events Framework library to communicate with the Fusion application. So lets look at some basic concepts of how the media toolbar integration works.

Let's look at how a third-party application can be converted into a media toolbar application so that it can be integrated with your CTI provider and Fusion application.

First, we'll look at components involved in integrating your CTI provider and Fusion application.

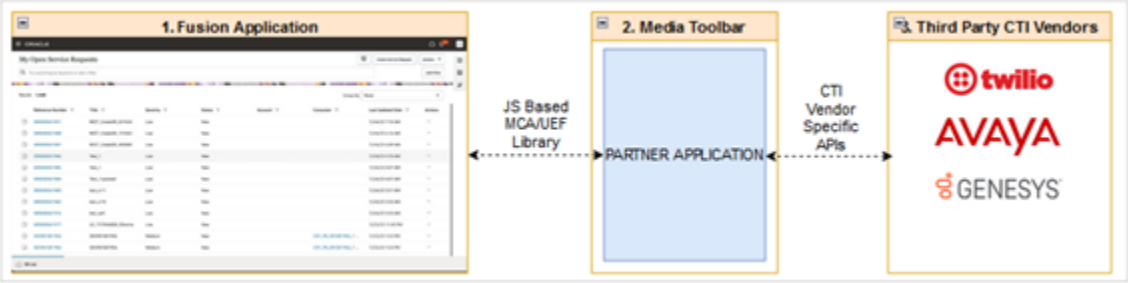
Then, we'll understand the basic concepts required for the CTI integration.

And finally, we'll see some scenarios handled from your partner application.

Once done with all that, you can build your media toolbar application from scratch.

Components

Here's a diagram that shows the components of the integration:



API documentation

How do Use the UI Events Framework playbook

Concepts

For integrating a third party supplier, Fusion CTI provides a library two types of APIs, Actions and Events.

General steps to execute the actions and events APIs are:

- Get the proper context
- Create the request object
- Call the API

Use Action APIs when a partner application wants the Fusion application to perform certain operations. For example, the partner application notifies the Fusion application to render an incoming call dialog box when an incoming call is received. For this you use the *NewCommEvent* action.

Actions

Action	Description
agentStateEvent	This action can be fired from the partner application to make the current logged-in agent ready to make or accept calls. This will make the phone icon enabled in the Fusion application.
newCommEvent	When an incoming call is received from the CTI supplier to the partner application, fire the **`newCommEvent`** action from the partner application to inform the Fusion application about the incoming call. The Fusion application will render a dialog box with the name and incoming phone number.
startCommEvent	A partner application can fire this action whenever the call is disconnected from the CTI supplier or the Fusion application (by agent).
outboundCommError	A partner application can fire this action to notify the Fusion application that an error occurred during the initiation of the outbound event.
getConfiguration	A partner application can fire this action to get configuration information that enables the toolbar to evaluate the features supported by the Fusion application.
disableFeature	Informs Fusion application that a subset of available functionality must be disabled because the toolbar hasn't implemented the functionality.
readyForOperation	Notifies the Fusion application that the toolbar is ready for operation.

Here's example code for firing any action:

```
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE')
 as IPhoneContext;

// Step 2: Create the request object
const request: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('<Action Name>');
// Set request object properties

// Step 3: Invoke the API
```

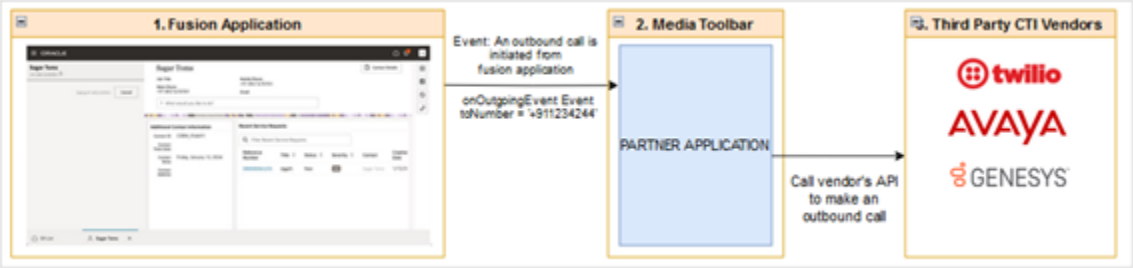
```
phoneContext.publish(request).then((res: IOperationResponse) => {
 // response after firing the action
}).catch(() => {
})
```

Events

Whenever the agent performs an action in the Fusion application, the Fusion application raises an event through the MCA library and these events can be listened to by the partner application.

Use Events APIs whenever the partner application needs to listen to the actions performed by the agent in the Fusion application.

For example, the Fusion application raises an event when an outbound call is started from the Fusion application using the *onOutgoingEvent*.



Supported Events

Event	Description
onToolbarInteractionCommand	Once a call is received in the Fusion application, the agent can accept, reject, or disconnect it. onToolbarInteractionCommand is fired from the fusion application based on the agent's action (accept, reject, or disconnect). • • **`accept`**: The **`onToolbarInteractionCommand`** is fired with the command of **`accept`** when the agent accepts the call from the Fusion application. • **`reject`**: The **`onToolbarInteractionCommand`** is fired with the command of **`reject`** when the agent declines the call from the Fusion application. • **`disconnect`**: The **`onToolbarInteractionCommand`** is fired with the command of **`disconnect`** when the agent hangs up the call from the Fusion application. • **`hold`**: The **`onToolbarInteractionCommand`** is fired with the command of **`hold`** when the agent clicks the **Hold** button from the Fusion application. • **`unhold`**: The **`onToolbarInteractionCommand`** is fired with the command of **`unhold`** when the agent clicks the **Unhold** button from the Fusion application. • **`mute`**: The **`onToolbarInteractionCommand`** is fired with the command of **`mute`** when the agent clicks the **Mute** button from the Fusion application. • **`unmute`**: The **`onToolbarInteractionCommand`** is fired with the command of **`unmute`** when the agent clicks the **Unmute** button from the Fusion application. • **`record`**: The **`onToolbarInteractionCommand`** is fired with the command of **`record`** when the agent clicks the **Record** button from the Fusion application.

Event	Description
	• **`stopRecord`**: The **`onToolbarInteractionCommand`** is fired with the command as **`stopRecord`** when the agent clicks the **Stop Record** button from the Fusion application. • **`transfer`**: The **`onToolbarInteractionCommand`** is fired with the command as **`transfer`** when the agent clicks the **Transfer** button from the Fusion application. • **`setActive`**: The **`onToolbarInteractionCommand`** is fired with the command of **`setActive`** when the agent clicks the **Set Active** button from the Fusion application.
onDataUpdated	This event is used to listen to a **`DataUpdate`** event in the Fusion application. This event is also fired when a **`wrapup`** is saved along with the wrap up details as payload.
onOutgoingEvent	Whenever an outbound call is started from the Fusion application, this event is fired and the partner application can listen to this event and inform the CTI supplier to make the outbound call.
onToolbarAgentCommand	This event is used to pass interaction commands to the application from CTI.

Here's example code for subscribing to any event:

```
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE')
 as IPhoneContext;

// Step 2: Create the request object
const request: IMcaEventRequest = uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('<Event
 Name>');
// Set request object properties

// Step 3: Invoke the API
phoneContext.subscribe(request, (response: IEventResponse) => {
 // subscription response
});
```

Here's sample code for listening to the `onDataUpdated` event:

```
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE')
 as IPhoneContext;

// Step 2: Create the request object
const request: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onDataUpdated');
// Set request object properties
request.setAppClassification('ORA_SERVICE');

// Step 3: Invoke the API
phoneContext.subscribe(request, (response: IEventResponse) => {
 // subscription response
 const data = response.getResponseData();
 console.log("onDataUpdated Response: ", dataUpdateResponse);
});
```

Scenarios

Here are the basic scenarios for the CTI application:

- Make agent available to make or accept calls
- Handle incoming calls
- Handle the outbound calls

How to initialize the partner application as an MCA application

1. Inject the UEF (ui-events-framework) JavaScript library into the partner application. Here's the code to inject the library into the application:

```
// Step 1: Include the javascript library in partner application
<script src="https://static.oracle.com/cdn/ui-events-framework/libs/ui-events-framework-client.js"></
script>
```

2. Initialize the UEF library using the following line:

```
// Step 2: Initialize the library from partner application
const uiEventsFrameworkInstance: IUiEventsFrameworkProvider = await
 CX_SVC_UI_EVENTS_FRAMEWORK.uiEventsFramework.initialize('MCA_APP', 'v1');
```

   And here's the JavaScript code for initializing UEF:

```
// Step 2: Initialize the library from partner application
const uiEventsFrameworkInstance = await
 CX_SVC_UI_EVENTS_FRAMEWORK.uiEventsFramework.initialize('MCA_APP', 'v1');
```

Scenario 1: Make agent available to make or accept calls

After signing in to the Fusion application agents can make themselves available to accept or make calls. For this you use the *agentStateEvent* action. You can execute the following code from the partner application to make the agent available:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE')
 as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaAgentStateEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('agentStateEventOperation') as
 IMcaAgentStateEventActionRequest;
// Set request object properties
requestObject.setEventId('1');
requestObject.setIsAvailable(true);
requestObject.setIsLoggedIn(true);
requestObject.setState('AVAILABLE');
requestObject.setStateDisplayString('Idle');

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaAgentStateEventActionResponse) => {
 console.log('AGENT IS READY', operationResponse);
```

```
}).catch(() => {
})
```

Here's the JavaScript code:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('agentStateEventOperation');
// Set request object properties
requestObject.setEventId('1');
requestObject.setIsAvailable(true);
requestObject.setIsLoggedIn(true);
requestObject.setState('AVAILABLE');
requestObject.setStateDisplayString('Idle');
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('AGENT IS READY', operationResponse);
}).catch(() => {
});
```

After executing the event the **Phone** icon shows the agent is available.

Scenario 2: Handle incoming calls

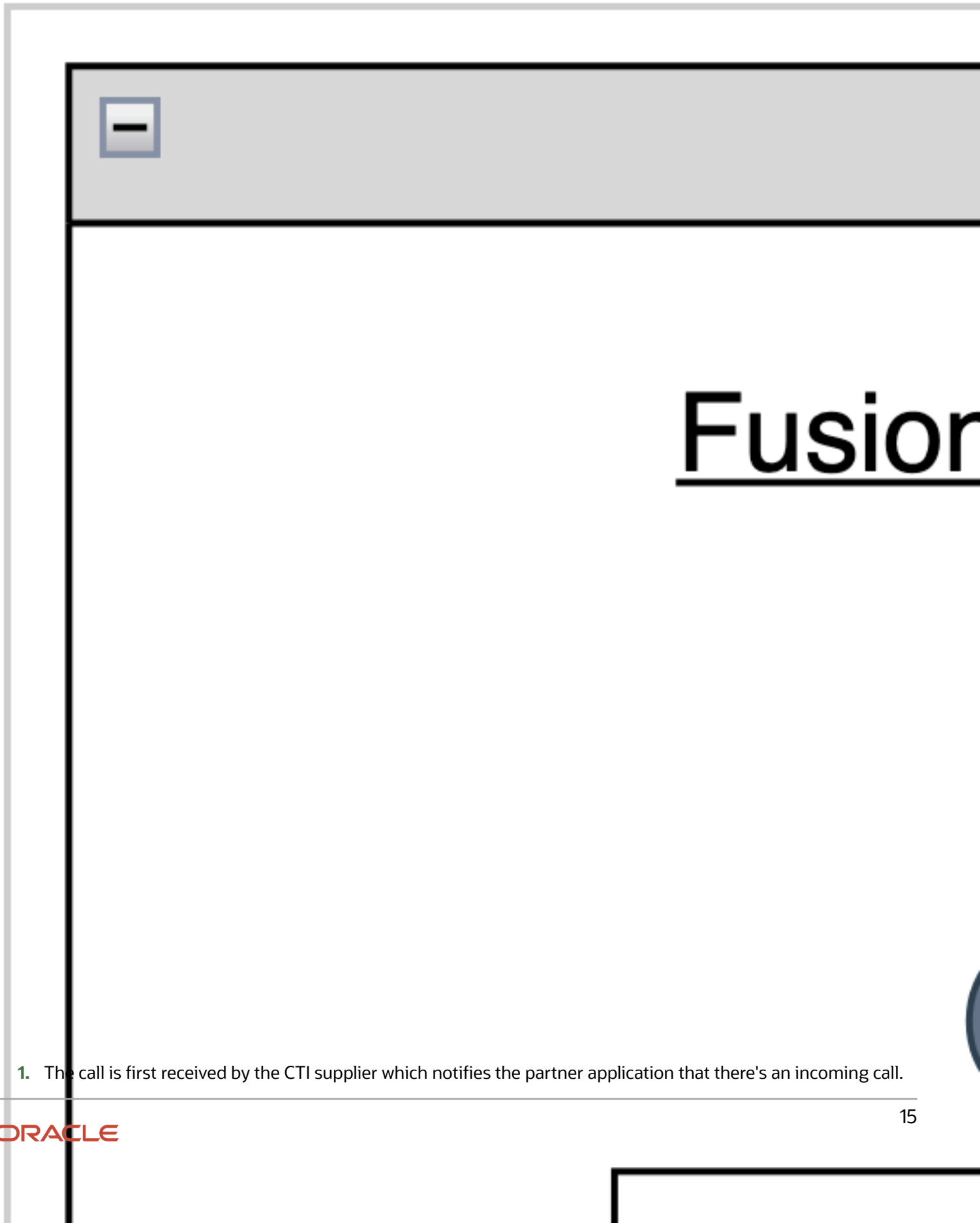
You can divide this scenario into there states:

1. Ring received.
2. Call accepted in the Fusion application.
3. Call disconnected in the Fusion application.

Ring Received

Sa customer is calling the service center number. The Fusion application checks whether a contact record exists in the Fusion application and it also displays an incoming call offer notification in the UI.

The following diagram shows the sequence of actions that are executed during this scenario:



1. The call is first received by the CTI supplier which notifies the partner application that there's an incoming call.

2. The partner application can notify the incoming call to the Fusion application by firing the *newCommEvent* action.

   You can execute the following code from the partner application to fire the `newCommEvent` action:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent') as
 IMcaNewCommEventActionRequest;
// Set request object properties
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');
requestObject.getInData().setInDataValueByAttribute('SVCMCA_ANI', '62738490');
requestObject.getInData().setInDataValueByAttribute("SVCMCA_COMMUNICATION_DIRECTION",
 "ORA_SVC_INBOUND");
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaNewComActionResponse) => {
 console.log('NEWCOMM FIRED', operationResponse);
 // Custom logic for handling the partner application UI
}).catch(() => {
})
```

   Here's the JavaScript:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent');
// Set request object properties
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');
requestObject.getInData().setInDataValueByAttribute('SVCMCA_ANI', '62738490');
requestObject.getInData().setInDataValueByAttribute("SVCMCA_COMMUNICATION_DIRECTION",
 "ORA_SVC_INBOUND");
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('NEWCOMM FIRED', operationResponse);
 // Custom logic for handling the partner application UI
}).catch(() => {
});
```

   > **Note:** You can set the standard list of attributes listed in *System Tokens* in the `setInDataValueByAttribute` function. You can also pass any custom tokens. Here's an example:
   > `requestObject.getInData().setInDataValueByAttribute("IVR_NUM_1", "TokenValue1");`

3. After the Fusion application identifies the action, it performs the contact number lookup and returns the contact details in the action response.

   To configure the default reverse lookup, see *Create Lookup Filters*.
4. The Fusion application notifies the agent about the incoming call offer notification through a dialog box.
5. The agent can either answer or decline the call.

Call Accepted

The agent clicks the **Answer** button in the Fusion application to accept the call.

The following diagram shows the sequence of actions performed when the agent accepts the call from the Fusion application.

1. When the agent clicks the **Answer** button in the Fusion application, the *onToolbarInteractionCommand* event is fired with the command of `accept`.

   The following code can be executed from the partner application to subscribe to `onToolbarInteractionCommand` event:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand') as
 IMcaEventRequest;

// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse: IMcaOnToolbarInteractionCommandEventResponse) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse =
 eventResponse.getResponseData();
 const command: string = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
 }
});
```

   Here's the JavaScript code:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand');
// Step 3: Invoke the API
```

```
phoneContext.subscribe(requestObject, (eventResponse) => {
 const eventResponseDetails = eventResponse.getResponseData();
 const command = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
 }
});
```

2. The partner application receives this event and if the event is to `accept` a call, the partner application notifies the CTI supplier to accept the call.
3. After the CTI supplier notifies the partner application that the call is accepted, the partner application can fire the *startCommEvent* action.

   The following Typescript code can be executed from the partner application to fire a `startCommEvent` action:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const request: IMcaStartCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('startCommEvent') as
 IMcaStartCommEventActionRequest;
request.setAppClassification('ORA_SERVICE');
// Set event ID as a unique identifier of the current call event
request.setEventId('1');

// Step 3: Invoke the API
phoneContext.publish(request).then((operationResponse: IMcaStartComActionResponse) => {
 // Extract the required data from response and use for updating the partner app
 const contactName: string = operationResponse.getResponseData().getData()['SVCMCA_CONTACT_NAME']; //
 sample for getting the contact name
}).catch(() => {
});
```

   And here's the JavaScript code:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const request = uiEventsFrameworkInstance.requestHelper.createPublishRequest('startCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
request.setAppClassification('ORA_SERVICE');
// Set event ID as a unique identifier of the current call event
request.setEventId('1');
// Step 3: Invoke the API
phoneContext.publish(request).then((operationResponse) => {
 // Extract the required data from response and use for updating the partner app
```

```
 const contactName = operationResponse.getResponseData().getData()['SVCMCA_CONTACT_NAME']; // sample for
 getting the contact name
}).catch(() => {
});
```

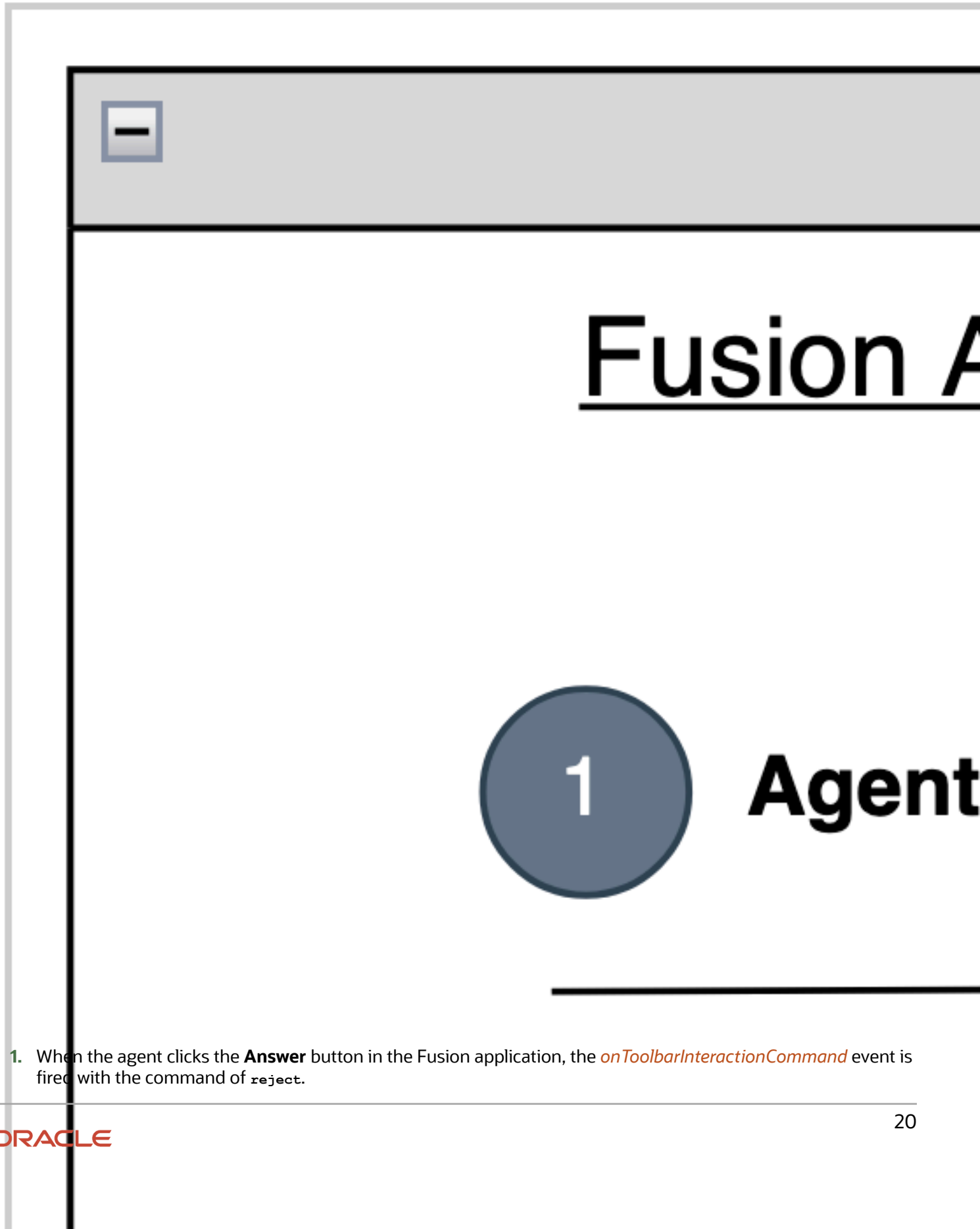
4. When the Fusion application identifies this action, the matched contact and an engagement panel will be opened based on the rules configured in the Fusion configuration management for the screen pop.

   To configure screen pop rules, see *Configure screen pop rules for Service Center*.

Call Rejected in the Fusion application

The agent clicks the **Decline** button in the Fusion application to reject the call.

The following diagram shows the sequence of actions to be performed when the agent rejects the call from the Fusion application:



1. When the agent clicks the **Answer** button in the Fusion application, the *onToolbarInteractionCommand* event is fired with the command of `reject`.

The following code can be executed from the partner application to subscribe to `onToolbarInteractionCommand` event:

> **Note:** If you've already added the subscription in the previous steps, you can skip the following code and add your logic in the `reject` case in the existing subscription you've added.

Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand') as
 IMcaEventRequest;

// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse: IMcaOnToolbarInteractionCommandEventResponse) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse =
 eventResponse.getResponseData();
 const command: string = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
 }
});
```

Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand');
// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse) => {
 const eventResponseDetails = eventResponse.getResponseData();
 const command = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
```

```
 }
});
```

2. The partner application receives this event and if the event is to reject the call, the partner application notifies the CTI supplier to reject the call.
3. Once the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the *closeCommEvent* action with the reason of `reject`.

   The following code can be executed from the partner application to fire a closeCommEvent action:

   Here's the Typescript example:

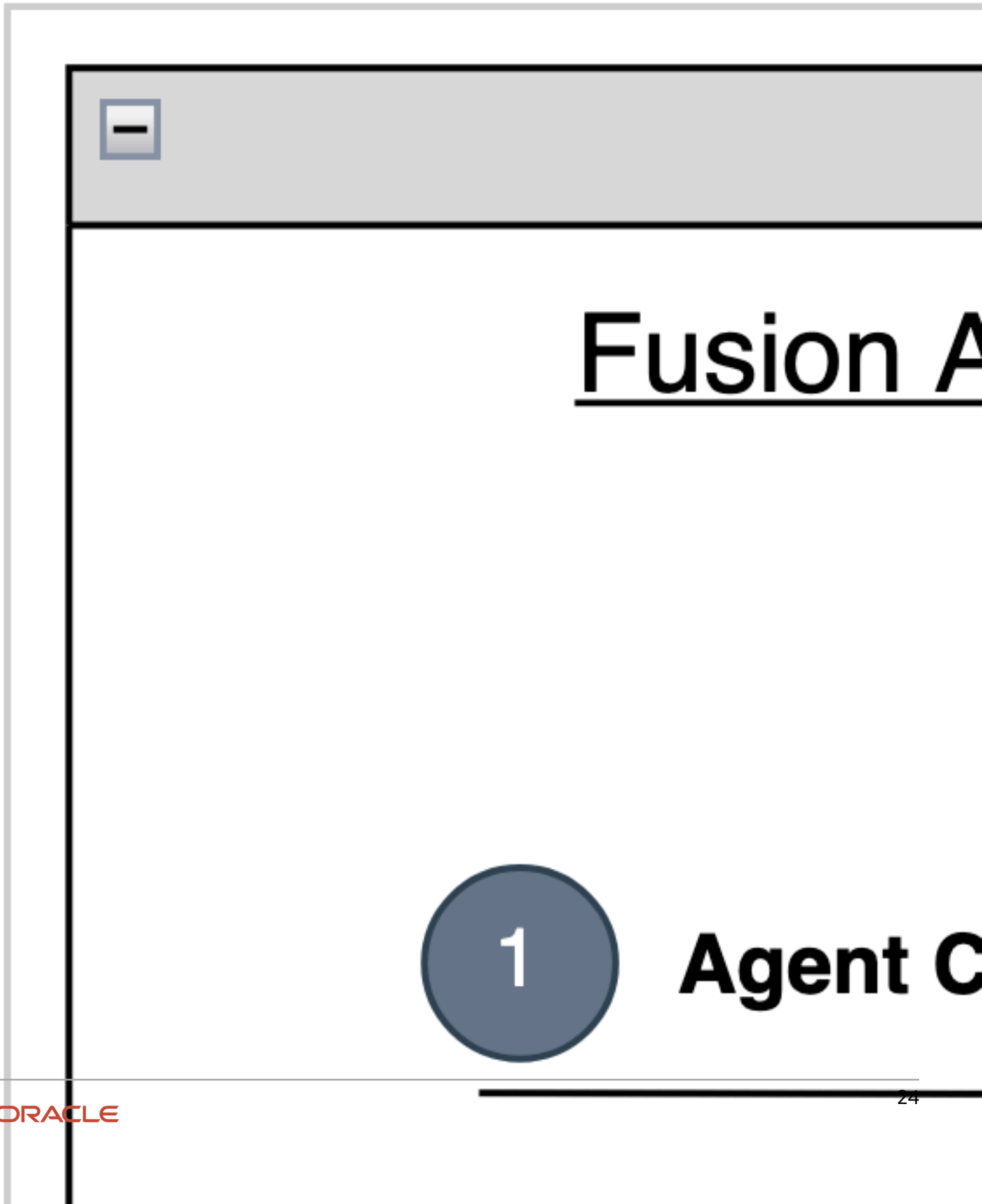
```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent') as
 IMcaCloseCommEventActionRequest;

// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as reject
requestObject.setReason("REJECT");
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaCloseComActionResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
})
```

   Here's the JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as reject
requestObject.setReason("REJECT");
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
});
```

4. After the Fusion application identifies this action, the call dialog box is discarded from the UI.

Call disconnected from the Fusion application


Fusion A
1
Agent C

The agent clicks on the **End Call** button in Fusion when the conversation is complete.

1. When the agent clicks on the **End Call** button in the Fusion application, and the *onToolbarInteractionCommand* event is fired with the command of `disconnect`.

   The following code can be executed from the partner application to subscribe to the `onToolbarInteractionCommand` event:

   > **Note:** If you've already added the subscription in the previous steps, you can skip the following code and add your logic in the `disconnect` case in the existing subscription you've added.

   Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand') as
 IMcaEventRequest;

// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse: IMcaOnToolbarInteractionCommandEventResponse) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse =
 eventResponse.getResponseData();
 const command: string = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
 }
});
```

   Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand');
// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse) => {
 const eventResponseDetails = eventResponse.getResponseData();
 const command = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
```

```
 // Notify CTI Vendor to disconnect the call
 break;
 case "reject":
 // Notify CTI Vendor to disconnect the call
 break;
 }
});
```

2. The partner application receives this event and if the event is to disconnect a call, the partner application notifies the CTI supplier to disconnect the call.
3. Once the CTI supplier notifies the partner application that the call is disconnected, the partner application can fire the `closeCommEvent` action with the reason as `WRAPUP`.

   Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent') as
 IMcaCloseCommEventActionRequest;
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties
requestObject.setReason("WRAPUP");
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaCloseComActionResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
})
```

   And here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties
requestObject.setReason("WRAPUP");
// Set event ID as a unique identifier of the current call event
requestObject.setEventId('1');
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
});
```

4. When the Fusion application identifies this action, it renders the wrap-up window in the UI.

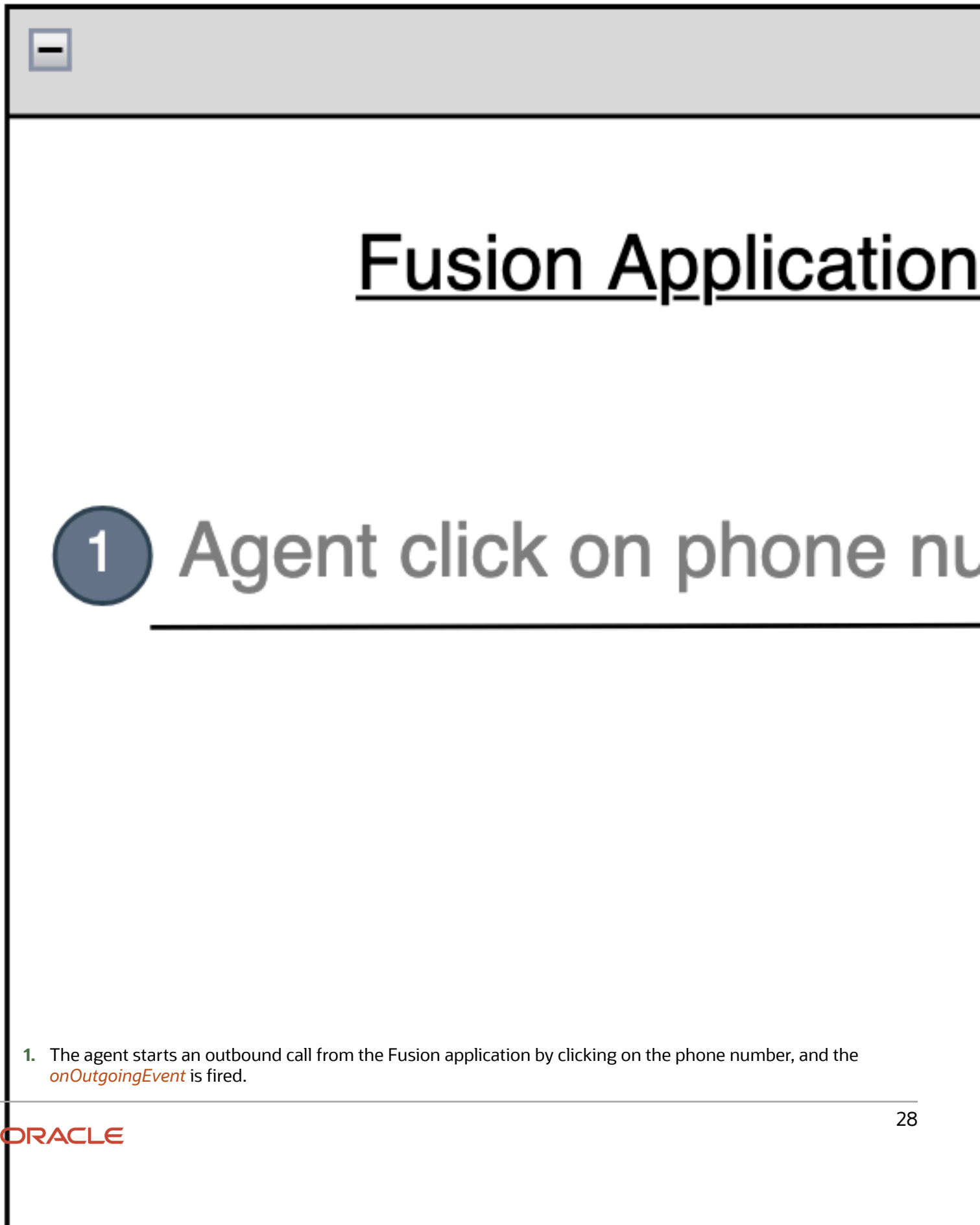
Scenario 3: Outbound calls

The outbound scenario can be divided into four states:

1. Start the outbound call.
2. The call is accepted by the customer.
3. The call is disconnected by agent or customer.
4. The call is declined by the customer.

Start outbound call

The agent clicks on the phone number from the Fusion application and the call state is ringing.



1. The agent starts an outbound call from the Fusion application by clicking on the phone number, and the *onOutgoingEvent* is fired.

The following code can be executed from the partner application to subscribe to the `onOutgoingEvent` event.

Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onOutgoingEvent') as
 IMcaEventRequest;
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');

let outboundSubscriptionResponse: IMcaonOutgoingEventResponse;
// Step 3: Invoke the API
phoneContext.subscribe(requestObject, async (eventResponse: IMcaonOutgoingEventResponse) => {
 // Use the event response to get the phone number and invoke the initiate call api of CTI vendor
 // Keep the outbound subscription response in a global variable. The outData needs to be set while
 firing newCommEvent and startCommEvent actions
 outboundSubscriptionResponse = eventResponse;
});
```

Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onOutgoingEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
let outboundSubscriptionResponse
// Step 3: Invoke the API
phoneContext.subscribe(requestObject, async (eventResponse) => {
 // Use the event response to get the phone number and invoke the initiate call api of CTI vendor
 // Keep the outbound subscription response in a global variable. The outData needs to be set while
 firing newCommEvent and startCommEvent actions
 outboundSubscriptionResponse = eventResponse;
});
```

2. The Partner application listening for this event notifies the CTI supplier to start the call.
3. The partner application notifies the call initiation to the Fusion application by publishing the *newCommEvent*action.

The following code can be executed from the partner application to fire a `newCommEvent` action.

Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
```

```
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent') as
 IMcaNewCommEventActionRequest;
// Set request object properties
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId('1');

// outboundSubscriptionResponse is the variable having the subscription response of onOutGoing event
const customerInData = {
 ...outboundSubscriptionResponse.getResponseData().getOutData(),
 'callData': {
 "phoneLineId": "1",
 "eventId": outboundSubscriptionResponse.getResponseData().getOutData().SVCMCA_CALL_ID,
 },
 'SVCMCA_COMMUNICATION_DIRECTION': 'ORA_SVC_OUTBOUND',
 'SVCMCA_WRAPUP_TIMEOUT': '',
 'appClassification': ''
};
for (let key in customerInData) {
 requestObject.getInData().setInDataValueByAttribute(key, customerInData[key]);
}

// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaNewComActionResponse) => {
 // Handle the custom logic for UI updates here
}).catch(() => {
})
```

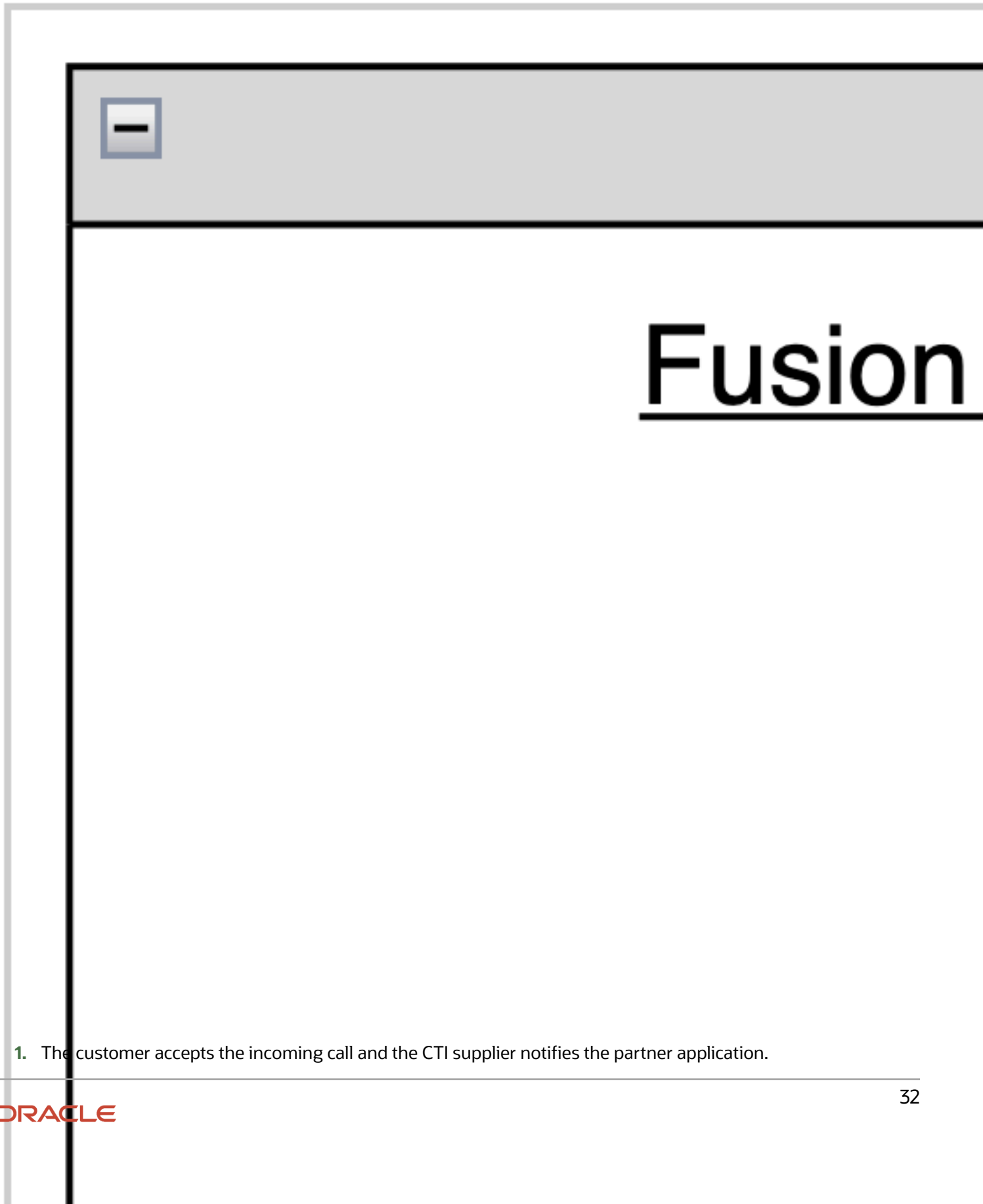
Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent');
// Set request object properties
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId('1');

// outboundSubscriptionResponse is the variable having the subscription response of onOutGoing event
const customerInData = {
 ...outboundSubscriptionResponse.getResponseData().getOutData(),
 'callData': {
 "phoneLineId": "1",
 "eventId": outboundSubscriptionResponse.getResponseData().getOutData().SVCMCA_CALL_ID,
 },
 'SVCMCA_COMMUNICATION_DIRECTION': 'ORA_SVC_OUTBOUND',
 'SVCMCA_WRAPUP_TIMEOUT': '',
 'appClassification': ''
};
for (let key in customerInData) {
 requestObject.getInData().setInDataValueByAttribute(key, customerInData[key]);
}
```

```
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 // Handle the custom logic for UI updates here
}).catch(() => {
});
```

4. The action response from the Fusion application has the contact details that can be used by the Partner application for rendering the UI.

Call accepted by the customer



1. The customer accepts the incoming call and the CTI supplier notifies the partner application.

2. The partner application publishes the *startCommEvent* action to notify the Fusion application.

   The following code can be executed from the partner application to fire the `startCommEvent` action.

   Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const request: IMcaStartCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('startCommEvent') as
 IMcaStartCommEventActionRequest;
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
request.setAppClassification('ORA_SERVICE');
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
request.setEventId('1');

// Set values from outData of outboundSubscriptionResponse variable to startCommEvent request object
for (const property in outboundSubscriptionResponse.getResponseData().getOutData()) {
 request.getInData().setInDataValueByAttribute(property,
 outboundSubscriptionResponse.getResponseData().getOutData()[property]);
}

// Step 3: Invoke the API
phoneContext.publish(request).then((operationResponse: IMcaStartComActionResponse) => {
 // Extract the required data from response and use for updating the partner app
 const contactName: string = operationResponse.getResponseData().getData()['SVCMCA_CONTACT_NAME']; //
 sample for getting the contact name
}).catch(() => {
});
```

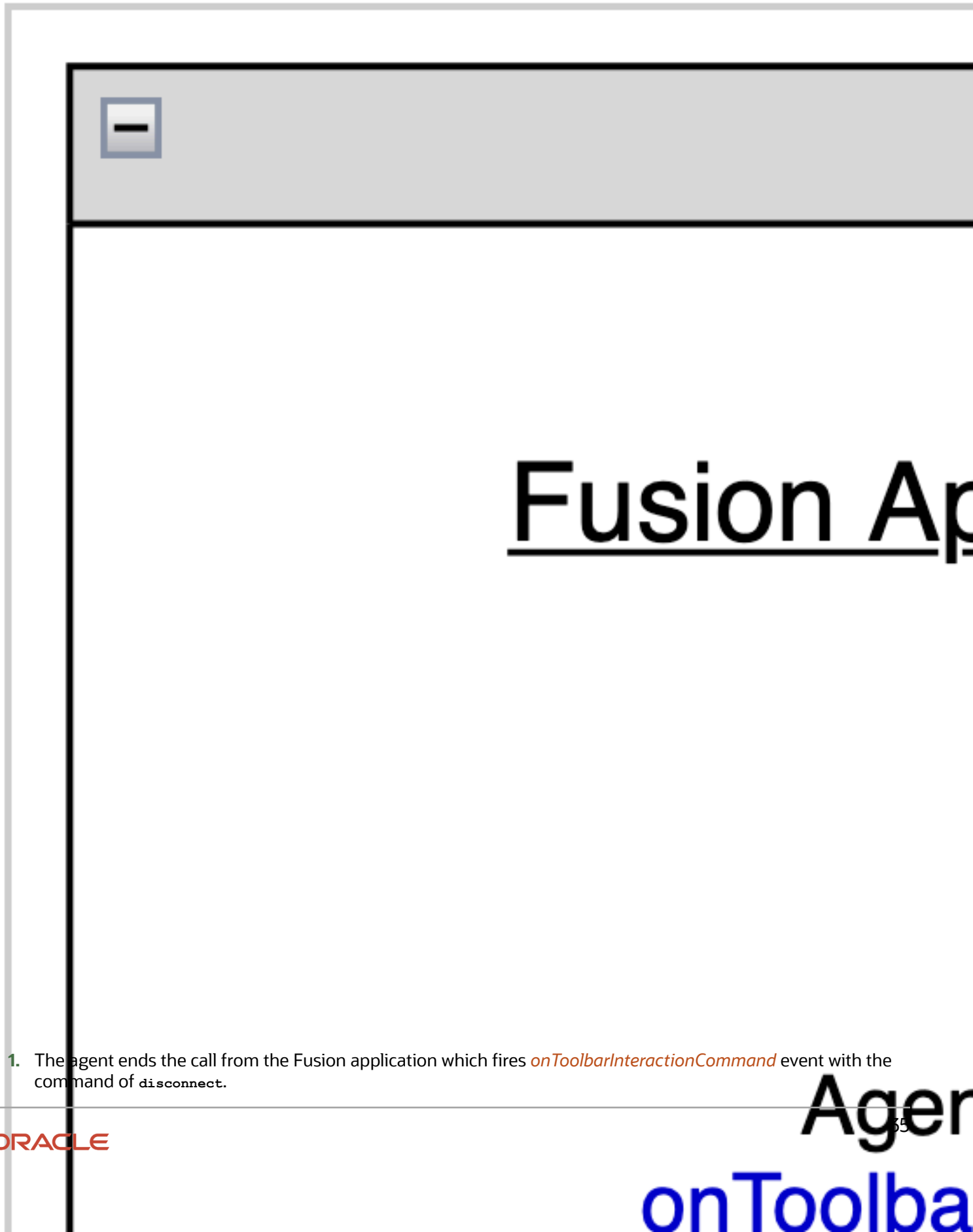
   Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const request = uiEventsFrameworkInstance.requestHelper.createPublishRequest('startCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
request.setAppClassification('ORA_SERVICE');
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
request.setEventId('1');

// Set values from outData of outboundSubscriptionResponse variable to startCommEvent request object
for (const property in outboundSubscriptionResponse.getResponseData().getOutData()) {
 request.getInData().setInDataValueByAttribute(property,
 outboundSubscriptionResponse.getResponseData().getOutData()[property]);
}

// Step 3: Invoke the API
phoneContext.publish(request).then((operationResponse) => {
 // Extract the required data from response and use for updating the partner app
 const contactName = operationResponse.getResponseData().getData()['SVCMCA_CONTACT_NAME']; // sample for
 getting the contact name
```

```
}).catch(() => {
});
```

3. On completing the `startCommEvent` the partner application can update the UI as required to show the in-progress call.

Call disconnected by the agent or customer



1. The agent ends the call from the Fusion application which fires *onToolbarInteractionCommand* event with the command of `disconnect`.

The following code can be executed from the partner application to subscribe to `onToolbarInteractionCommand` event:

> **Note:** If you've already added the subscription in the previous steps, you can skip the following code and add your logic in the `disconnect` case in the existing subscription you've added.

Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaEventRequest =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand') as
 IMcaEventRequest;

// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse: IMcaOnToolbarInteractionCommandEventResponse) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse =
 eventResponse.getResponseData();
 const command: string = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call (In this use-case your logic should be here)
 break;
 case "reject":
 // Notify CTI Vendor to reject the call
 break;
 }
});
```

Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject =
 uiEventsFrameworkInstance.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand');
// Step 3: Invoke the API
phoneContext.subscribe(requestObject, (eventResponse) => {
 const eventResponseDetails = eventResponse.getResponseData();
 const command = eventResponseDetails.getCommand();
 switch (command) {
 case "accept":
 // Notify CTI Vendor to accept the call
 break;
 case "disconnect":
 // Notify CTI Vendor to disconnect the call (In this use-case your logic should be here)
 break;
 case "reject":
 // Notify CTI Vendor to reject the call
 break;
```

```
 }
});
```

2. The Partner application listening for this event will handle the event by notifying the CTI supplier to disconnect the call.
3. When the call is disconnected the partner application publishes *closeCommEvent* to the Fusion application with a reason of `wrapup`.

   Here's a Typescript example:

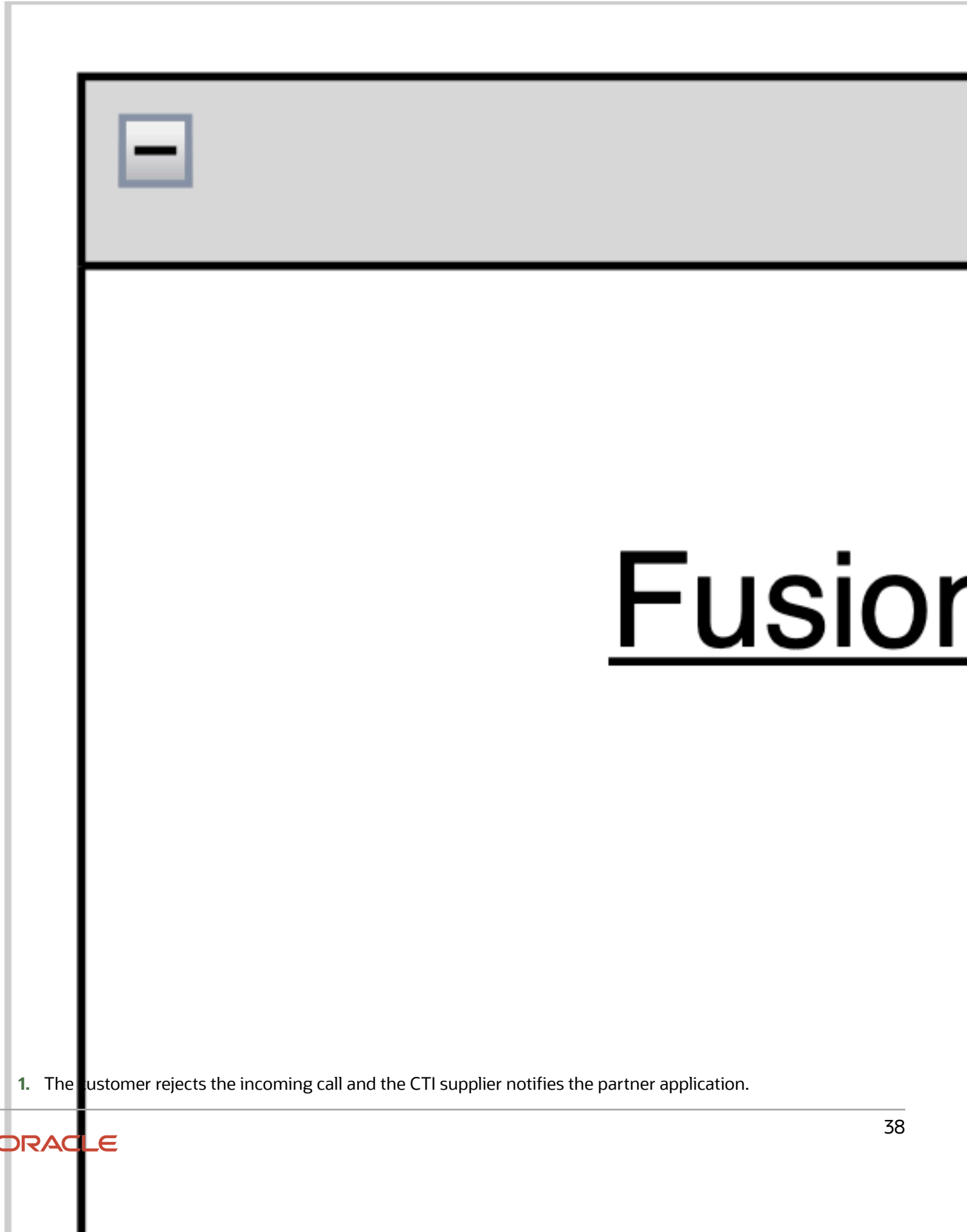
```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent') as
 IMcaCloseCommEventActionRequest;
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as Wrapup
requestObject.setReason("WRAPUP");
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId("1");

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaCloseComActionResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
})
```

   Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as Wrapup
requestObject.setReason("WRAPUP");
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId("1");
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
});
```

4. The Fusion application does the wrap up flow and displays notes.
5. On the `closeCommEvent` response the partner application updates the UI as required.

Call declined by customer



1. The customer rejects the incoming call and the CTI supplier notifies the partner application.

2. The partner application updates the UI and publishes the `closeCommEvent` to the Fusion application.

   The following code can be executed from the partner application to fire a `closeCommEvent` action.

   Here's a Typescript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext: IMultiChannelAdaptorContext = await
 uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext: IPhoneContext = await
 multiChannelAdaptorContext.getCommunicationChannelContext('PHONE') as IPhoneContext;

// Step 2: Create the request object
const requestObject: IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent') as
 IMcaCloseCommEventActionRequest;
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as reject
requestObject.setReason("REJECT");
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId("1");

// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse: IMcaCloseComActionResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
})
```

   Here's a JavaScript example:

```
// Please make sure uiEventsFrameworkInstance object is initialized successfully and it is not null
// Step 1: Get the proper context
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();
const phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');
// Step 2: Create the request object
const requestObject = uiEventsFrameworkInstance.requestHelper.createPublishRequest('closeCommEvent');
// Refer https://docs.oracle.com/en/cloud/saas/fusion-service/fairs/application-classification-
code.html#s20059918 for the list of supported app classifications
requestObject.setAppClassification('ORA_SERVICE');
// Set request object properties, in this scenario we need to set the reason as reject
requestObject.setReason("REJECT");
// Set event ID as a unique identifier of the current call event. You can also use SVCMCA_CALL_ID from
 subscription response of onOutgoingEvent
requestObject.setEventId("1");
// Step 3: Invoke the API
phoneContext.publish(requestObject).then((operationResponse) => {
 console.log('closeCommEvent fired', operationResponse);
}).catch(() => {
});
```

3. The partner application updates the UI to show that the agent is available.

Reasons for the closeCommEvent

The following table shows the supported reasons for the `closeCommEvent` action.

closeCommEvent reason	Description
WRAPUP	An established call was terminated but, Wrap Up is required. The agent should be prompted to enter post call disposition details.
ENDCOMMUNICATION	An established call is terminated but, no Wrap Up is required.
ERROR	There was an error establishing the call. Used for Outbound Dialing failures.
REJECT	The call was rejected.
TIMEDOUT	The call resulted in a ring timeout.
ABANDONED	The incoming call was hung up before the agent answered.
MISSED	The call was missed. This is a catch-all in case there isn'ta more specific reason for the call failing to be established.
CANCELED	The outbound dial attempt was canceled.
TRANSFERRED	The call was terminated because it was transferred.
CONFERENCE	A conference call was terminated.
LOST	The call was dropped.

4 Build Your Own CTI Application using OJET

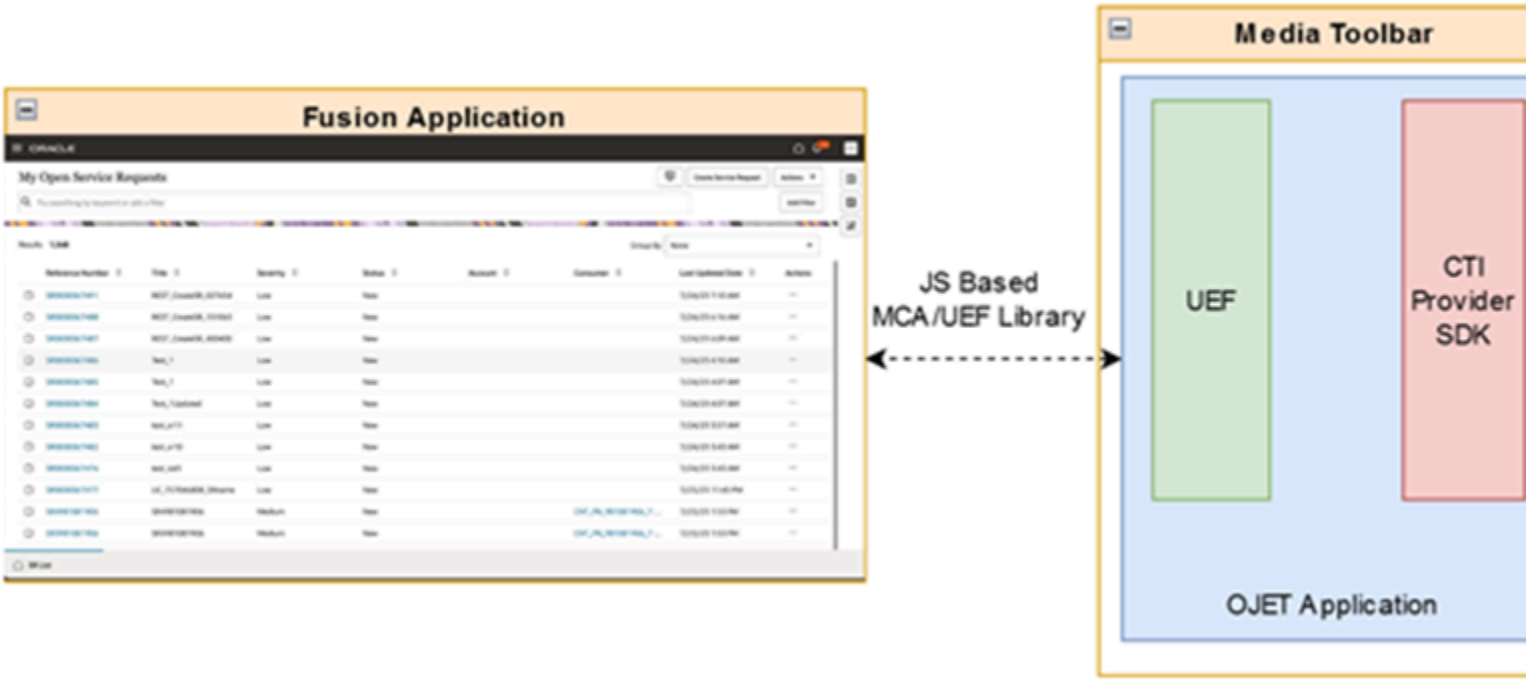
Overview

Now, lets use the concepts from the previous topics and build a media toolbar application from scratch, which you can embed in your Fusion media toolbar.

After your develop the partner application expect the following:

- From the partner application, the agent can make themselves show as available or not available for calls in the Fusion application.
- Inbound call notifications from the CTI provider can be shown to the agent as call offer notifications in the media toolbar application and in the Fusion application
- The agent can accept an inbound call from the CTI provider from the media toolbar application and from the Fusion application.
- The agent can disconnect a call from the CTI provider from the media toolbar application and from the Fusion application.
- The agent can make an outbound call from the Fusion application.

Based on these requirements, the architecture of the application can be described as:



- The Fusion application
- The Media Toolbar
- Oracle JET Framework (OJET).

- Oracle UI Events Framework
- The CTI provider
- An SDK
- Created using JavaScript
- MCA/UEF Library

We'll be developing this media toolbar application using the OJET, but there's no restriction on using any front-end-based technology for building your media toolbar application.

You can learn more about Oracle JET by checking out *Oracle JavaScript Extension Toolkit (Oracle JET)*.

Create a JET application

The first step toward building your media toolbar application is to create an OJET application with a basic template. Once you finish this step, you'll have a basic OJET application deployed in your Fusion media toolbar.

You can either work through the steps that follow to create the implementation or you can contact Oracle Cloud Customer Connect to access a compressed file which contains the source code for the basic JET application described in this topic. See *How to start your OJET application from a compressed file*.

Prerequisites

Before you get started, set up the following prerequisites:

- Contact Oracle Cloud Customer Connect for a link to download Node.js (v16.17.1).
- Install *JET Tooling*. Make sure you have the latest version. You can check the version by running the following command:

  ```
  ojet --version
  ```

  For more information, see *Oracle JavaScript Extension Toolkit (Oracle JET)*

Build the CTI Accelerator application

1. Run the following command:

   ```
   ojet create <PROJECT_NAME> --template=basic --typescript
   ```

   > **Note:** In place of <PROJECT_NAME> use a name of your choice, such as cti-accelerator.

   See *Create an Oracle JET web app by using a starter template* for more information.

2. Once the command is executed a new folder with the project name will appear. Open the folder in your favorite editor.

   As part of the basic template, you'll see some extra UI elements such as footer, and app name label in your application once it's started. We don't need these elements for this overview, so you can remove them by doing the following:

   a. Remove the Menu button components and footer tag by opening the `src/index.html` file and removing the following content from the `<oj-toolbar>` tag. Leave the `<oj-toolbar></oj-toolbar>` tag empty. See the following example:

```
<oj-toolbar>
 <!------ START REMOVE FROM HERE ------>
 <oj-menu-button id="userMenu" display="[[smScreen() ? 'icons' : 'all']]" chroming="borderless">
 <span><oj-bind-text value="[[userLogin]]"></oj-bind-text></span>
 <span slot="endIcon" :class="[[{'oj-icon demo-appheader-avatar': smScreen(), 'oj-component-icon
 oj-button-menu-dropdown-icon': !smScreen()}]]"></span>
 <oj-menu id="menu1" slot="menu">
 <oj-option id="pref" value="pref">Preferences</oj-option>
 <oj-option id="help" value="help">Help</oj-option>
 <oj-option id="about" value="about">About</oj-option>
 <oj-option id="out" value="out">Sign Out</oj-option>
 </oj-menu>
 </oj-menu-button>
 <!------ REMOVE UNTIL HERE ------>
</oj-toolbar>
```

   b. In the `src/index.html` file remove `<h1></h1>` tag also as shown in the following example:

```
<!------ REMOVE THIS LINE FROM <header> ------>
<h1 class="oj-sm-only-hide oj-web-applayout-header-title" title="Application Name"><oj-bind-text
 value="[[appName]]"></oj-bind-text></h1>
```

   c. In `src/index.html` file remove the `<footer></footer>` tag also as shown in the following example:

```
<!------ START REMOVE FROM HERE ------>
 <footer class="oj-web-applayout-footer" role="contentinfo">
 <div class="oj-web-applayout-footer-item oj-web-applayout-max-width">
 <ul>
 <oj-bind-for-each data="[[footerLinks]]">
 <template>
 <li>
 <a :id="[[$current.data.linkId]]" :href="[[$current.data.linkTarget]]">
 <oj-bind-text value="[[$current.data.name]]"></oj-bind-text>
 </a>
 </li>
 </template>
 </oj-bind-for-each>
 </ul>
 </div>
 <div class="oj-web-applayout-footer-item oj-web-applayout-max-width oj-text-color-secondary oj-
typography-body-sm">
 Copyright © 2014, 2024 Oracle and/or its affiliates All rights reserved.
 </div>
 </footer>
 <!------ REMOVE UNTIL HERE ------>
```

   d. Remove the unused variables from `src/ts/appController.ts` file as shown in the following example:

```
class RootViewModel {
 // REMOVE BELOW VARIABLES
 smScreen: ko.Observable<boolean>|undefined;
 appName: ko.Observable<string>;
 userLogin: ko.Observable<string>;
 footerLinks: Array<object>;
 // REMOVE UNTIL HERE
```

```
 constructor() {
 //.....
 // REMOVE BELOW VARIABLES
 let smQuery: string | null = ResponsiveUtils.getFrameworkQuery("sm-only");
 if (smQuery) {
 this.smScreen = ResponsiveKnockoutUtils.createMediaQueryObservable(smQuery);
 }

 // header

 // application Name used in Branding Area
 this.appName = ko.observable("App Name");

 // user Info used in Global Navigation area
 this.userLogin = ko.observable("john.hancock@oracle.com");

 // footer
 this.footerLinks = [
 { name: 'About Oracle', linkId: 'aboutOracle', linkTarget:'http://www.oracle.com/us/corporate/
index.html#menu-about'},
 { name: "Contact Us", id: "contactUs", linkTarget: "http://www.oracle.com/us/corporate/contact/
index.html" },
 { name: "Legal Notices", id: "legalNotices", linkTarget: "http://www.oracle.com/us/legal/
index.html" },
 { name: "Terms Of Use", id: "termsOfUse", linkTarget: "http://www.oracle.com/us/legal/terms/
index.html" },
 { name: "Your Privacy Rights", id: "yourPrivacyRights", linkTarget: "http://www.oracle.com/us/
legal/privacy/index.html" },
 ];
 // REMOVE UNTIL HERE
 }
}
```

Bootstrap the OJET application

1. From the Project root directory run the following command: `ojet serve`

   The command opens a new window in your browser and loads the default page of the OJET application.

   By default, the application starts in port 8000, and you can access the application at `http://localhost:8000/`. To change the port, you can pass the following argument `--server-port <port>`.

Start the application with a self signed certificate

To configure this app URL on the media toolbar we need to start the application in https. For this we need to serve the Web Application to an HTTPS Server Using a Self-Signed Certificate.

First you'll need to install a certificate in the web application directory. Then you'll configure the before_serve.js hook to do the following:

1. Create an instance of Express to host the served web application.
2. Set up HTTPS on the Express instance that you've created. To specify the HTTPS protocol, identify the location of the self-signed certificate that you placed in the application directory, and specify a password.
3. Pass the changed Express instance and the SSL-enabled server to the JET tooling so that OJET server uses your middleware configuration rather than the ready-to-use middleware configuration provided by the Oracle JET tooling.
4. To ensure that active reloads work when your web application is served to the HTTPS server, you'll also create an instance of the active reload server and configure it to use SSL.

If you can't use a certificate issued by a certificate authority, you can create your certificate (a self-signed certificate). You can use tools such as OpenSSL, Keychain Access on Mac, and the Java Development Kit's key tool utility do this this task. For example, using the Git Bash shell that comes with Git for Windows, you can run the following command to create a self-signed certificate with the OpenSSL tool:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

This creates the `cert.pem` and `key.pem` files and copies the files to the project root directory.

Once you've installed the self-signed certificates in your app, you configure the script for the before_serve hook point.

Do the following:

1. Open the `AppRootDir/scripts/hooks/before_serve.js` and make the following changes

```
'use strict';

module.exports = function (configObj) {
 return new Promise((resolve, reject) => {
 console.log("Running before_serve hook.");

 // Create an instance of Express, the Node.js web app framework that Oracle
 // JET tooling uses to host the web apps that you serve using ojet serve
 const express = require("express");

 // Set up HTTPS
 const fs = require("fs");
 const https = require("https");

 // Specify the self-signed certificates. In our example, these files exist
 // in the root directory of our project.
 const key = fs.readFileSync("./key.pem");
 const cert = fs.readFileSync("./cert.pem");
 // If the self-signed certificate that you created or use requires a
 // password, specify it here:
 const passphrase = "1234";

 const app = express();

 // Pass the modified Express instance and the SSL-enabled server to the Oracle JET tooling
 configObj['express'] = app;
 configObj['urlPrefix'] = 'https';
 configObj['server'] = https.createServer({
 key: key,
 cert: cert,
 passphrase: passphrase
 }, app);

 // Enable the Oracle JET live reload option using its default port number so that
 // any changes you make to app code are immediately reflected in the browser after you
 // serve it
 const tinylr = require("tiny-lr");
 const lrPort = "35729";

 // Configure the live reload server to also use SSL
 configObj["liveReloadServer"] = tinylr({ lrPort, key, cert, passphrase });

 resolve(configObj);
 });
};
```

2. Once you've finished these configuration steps, run the series of commands (`ojet build` and `ojet serve`, for example) that you typically run to build and serve your web app.

3. As the certificate that you're using is a self-signed certificate rather than a certificate issued by a certificate authority, the browser that you use to access the web app displays a warning the first time that you access the web app. Acknowledge the warning and click the options that allow you to access your web app. On Google Chrome, for example, you click Advanced and Proceed to `localhost` (unsafe) if your web app is being served to `https://localhost:8000/`.

   For more information, see *Serve a Web App to a HTTPS Server Using a Self-signed Certificate*.

Load the application in the media toolbar of your Fusion application

See *Configure the Media Toolbar*.

Once you load the application in your media toolbar, sign in to your Fusion application and click the phone icon to open the media toolbar window. You can see that the media toolbar application is loaded with your application.

Verify your progress

Once you load the application in your media toolbar, sign in to your Fusion application and click the Phone icon to open the media toolbar window. You'll see that the media toolbar application is loaded with your application.

References

- *Get started with OJET*
- *Create an OJET application*

Add the Call Panel component to your application

The previous topic showed how to build a basic JET application and how to deploy it in your Fusion media toolbar.

Next, we need to show incoming call notifications, and buttons to accept or disconnect a call in the media toolbar application. To do this, you can build your logic to render UI elements based on your requirements. As part of the steps in this series, the UI elements required for showing the call notification, and buttons to accept and disconnect calls are encapsulated as a custom JET component.

States of the Call Panel component

The Call Panel component has three states:

- **Ringing**: The Call Panel component is in the Ringing state when the agent receives an incoming call. The Call Panel component will also be in a ringing state when the agent starts an outbound call which isn't yet picked up by the customer. Your application. During this state, the Call Panel component has information about the incoming call such as contact name (which is retrieved from the Fusion application), IVR data, and 2 buttons for accepting or rejecting a call.
- **Accepted**: The Call Panel component is in the Accepted state when the agent accepts an incoming call from the customer. The Call Panel component is also in an Accepted state when a customer accepts an outbound call started by an agent. During this state, the call-panel component will have the call hangup button and a call-in-progress animation in addition to the incoming call information.

- **Disconnected**: The Call Panel component moves to the Disconnected state when an agent rejects an incoming call or disconnects an ongoing call.

Create a custom component in your application

1. Open the root folder of your toolbar application in the terminal and run the command: `ojet create component call-panel`.

   This creates the `call panel` custom component in the `src/ts/jet-composites` folder.

   For more information, see *Create an Oracle JET Web Component*.

Update the component's files based on your requirements

To update the component's code based on your requirements, see the following files:

- To update the `src/ts/jet-composites/call-panel-view.html` file:

```
<div class="oj-sm-margin-4x oj-flex oj-sm-justify-content-center">
 <div class="oj-flex oj-bg-neutral-170 oj-sm-padding-6x call-panel oj-color-invert">
 <div
 class="oj-sm-12 oj-flex oj-sm-flex-direction-column oj-sm-align-items-center oj-sm-justify-content-
space-between">
 <div class="oj-sm-margin-4x-top oj-flex oj-sm-flex-direction-column oj-sm-align-items-center top-
panel">
 <div class="oj-flex-item oj-typography-heading-xs oj-sm-margin-5x-bottom">
 <!-- Shows the text Incoming Call / Calling in the component based on the call direction -->
 <oj-bind-if test="[[callContext().direction === 'inbound']]">
 Incoming Call
 </oj-bind-if>
 <oj-bind-if test="[[callContext().direction === 'outbound']]">
 Calling
 </oj-bind-if>
 </div>

 <!-- Adds a phone icon as avatar in the component -->
 <oj-c-avatar
 class="oj-avatar-8xl"
 role="img"
 icon-class="oj-ux-ico-call-incoming"
 background="green"></oj-c-avatar>
 <br/><br/>
 <div id="progressBarContainer" class="oj-flex-item">
 <oj-bind-if test="[[callContext().state === 'ACCEPTED']]">
 <!-- Show a call in progress animation during the ACCEPTED state -->
 <oj-c-progress-bar
 id="progressBar"
 aria-label="basic progress bar"
 value="-1"></oj-c-progress-bar>
 </oj-bind-if>
 </div>

 </div>
 <!-- Show caller information -->
 <div class="oj-sm-flex oj-sm-flex-direction-column call-details">
 <div class="oj-flex-item oj-typography-heading-md oj-sm-margin-2x-vertical">
 <oj-bind-text value="[[callContext().callerName]]"></oj-bind-text>
 </div>
 <div class="oj-flex-item oj-typography-subheading-xs">
 <oj-bind-text value="[[callContext().phonenumber]]"></oj-bind-text>
 </div>
```

```
<table class="ivr-data-table oj-typography-body-sm oj-sm-margin-4x oj-helper-text-align-left">
<oj-bind-for-each data="[[ivrDataProvider]]">
<template>
<tr>
<td><oj-bind-text value="[[$current.data.key]]"></oj-bind-text></td>
<td>:</td>
<td><oj-bind-text value="[[$current.data.value]]"></oj-bind-text></td>
</tr>
</template>
</oj-bind-for-each>
</table>
</div>
<div class="oj-flex oj-sm-justify-content-center oj-sm-margin-2x-vertical">
<oj-bind-if test="[[callContext().state === 'ACCEPTED']]">
<oj-c-button id="hold" display="icons" label="hold">
<span slot="startIcon" class="oj-ux-ico-pause-circle"></span>
</oj-c-button>
</oj-bind-if>

</div>
<div class="oj-flex oj-sm-justify-content-center">
<!-- Show call accept and disconnect buttons in the component -->
<oj-bind-if test="[[callContext().state === 'RINGING' && callContext().direction === 'inbound']]">
<!-- Show call accept button only during inbound call RINGING state -->
<oj-c-button id="acceptCall" display="icons" label="Accept" chroming="borderless"
on-oj-action="[[ acceptClicked ]]"
class="oj-sm-margin-2x-horizontal">
<span slot="startIcon"
class="oj-ux-ico-call oj-sm-padding-4x oj-sm-padding-10x-horizontal call-accept"></span>
</oj-c-button>
</oj-bind-if>
<oj-c-button id="declineCall" display="icons" label="Decline" chroming="danger"
on-oj-action="[[ disconnectClicked ]]"
class="oj-sm-margin-2x-horizontal">
<span slot="startIcon" class="oj-ux-ico-call-end oj-sm-padding-4x oj-sm-padding-10x-horizontal"></span>
</oj-c-button>
</div>
</div>
</div>
```

- To update the `src/ts/jet-composites/call-panel-viewModel.ts` file:

```
"use strict";

import * as ko from "knockout";
import Context = require("ojs/ojcontext");
import Composite = require("ojs/ojcomposite");
import ArrayDataProvider = require('ojs/ojarraydataprovider');
import "oj-c/button";
import "ojs/ojknockout";
import "oj-c/avatar";
import "oj-c/progress-bar";

interface CallContext {
 phonenumber: string;
 callerName: string;
 direction: string;
 eventId: string;
 ivrData: { [key: string]: string };
 state: string;
}

export default class ViewModel implements Composite.ViewModel<Composite.PropertiesType> {
 busyResolve: (() => void);
 composite: Element;
```

```
properties: Composite.PropertiesType;
callContext: ko.Observable<CallContext>;
ivrDataProvider: ArrayDataProvider<string, { [key: string]: string }>;

constructor(context: Composite.ViewModelContext<Composite.PropertiesType>) {
//At the start of your viewModel constructor
const elementContext: Context = Context.getContext(context.element);
const busyContext: Context.BusyContext = elementContext.getBusyContext();
const options = { "description": "Web Component Startup - Waiting for data" };
this.busyResolve = busyContext.addBusyState(options);

this.composite = context.element;

// Properties passed to the component
this.properties = context.properties;

this.callContext = ko.observable(this.properties.callContext);

this.ivrDataProvider = new ArrayDataProvider(this.parseIvrData(), {
keyAttributes: 'key'
});

//Once all startup and async activities have finished, relocate if there are any async activities
this.busyResolve();
}

parseIvrData(): any {
let data: any[] = [];
if (this.callContext()?.ivrData) {
data = Object.keys(this.callContext().ivrData).map((key: string) => {
return {
key: key,
value: this.callContext().ivrData[key]
}
});
}
const ivrData: any = ko.observableArray(data);
return ivrData;
}

public acceptClicked: (event: any) => void = (event: any): void => {
// Logic to fire and event to the container that the accept button is clicked
const formattedEvent: object = {bubbles: true, cancelable: false, detail: {}};
const acceptButtonClickedEvent: CustomEvent = new CustomEvent('acceptButtonClicked', formattedEvent);
this.composite.dispatchEvent(acceptButtonClickedEvent);
}

public disconnectClicked: (event: any) => void = (event: any): void => {
// Logic to fire and event to the container that the disconecct button is clicked.
// If the call is in ACCEPTED state, the value 'HANGUP' is passed as the event payload to the
container,
// Otherwise, the value 'REJECT' is passed as the event payload
const disconnectionState: string = (this.callContext().state === 'ACCEPTED') ? 'WRAPUP' : 'REJECT';
const formattedEvent: object = {bubbles: true, cancelable: false, detail: { disconnectionState }};
const disconnectButtonClickedEvent: CustomEvent = new CustomEvent('disconnectButtonClicked',
formattedEvent);
this.composite.dispatchEvent(disconnectButtonClickedEvent);
}

//Lifecycle methods - implement if necessary

activated(context: Composite.ViewModelContext<Composite.PropertiesType>): Promise<any> | void {

};

connected(context: Composite.ViewModelContext<Composite.PropertiesType>): void {
```

```
 };

 bindingsApplied(context: Composite.ViewModelContext<Composite.PropertiesType>): void {

 };

 propertyChanged(context: Composite.PropertyChangedContext<Composite.PropertiesType>): void {
 if (context.property === 'callContext') {
 // Update callContext observable based on the propertyChange
 this.callContext({...this.callContext(), state: context.value.state});
 }
 };

 disconnected(element: Element): void {

 };
};
```

- To update the `src/ts/jet-composites/call-panel-styles.css` file:

```
call-panel:not(.oj-complete) {
 visibility: hidden;
}

call-panel[hidden] {
 display: none;
}

.call-panel {
 min-height: 450px;
 min-width: 300px;
 border-radius: 10px;
}

.top-panel {
 color: #dfdfdf;
}

.call-accept {
 background-color: #28b328;
}

.call-accept:hover {
 background-color: #239b23;
}

.call-details {
 text-align: center;
 color: #ffffff;
}

.ivr-data-table {
 color: #c4c4c4;
}

#progressBarContainer {
 width: 100%;
}
```

- To update the `src/ts/jet-composites/component.json` file:

```
{
 "name": "call-panel",
 "version": "1.0.0",
 "jetVersion": "^15.1.0",
 "displayName": "A user friendly, translatable name of the component.",
```

```
"description": "A translatable high-level description for the component.",
"properties": {
"callContext": {
"description": "The Call context details.",
"displayName": "Call Context",
"type": "object"
}
},
"methods": {},
"events": {
"acceptButtonClicked": {
"displayName": "acceptButtonClicked",
"description": "Will be called when accept button is clicked.",
"bubbles": true,
"cancelable": true,
"detail": {
}
},
"disconnectButtonClicked": {
"displayName": "disconnectButtonClicked",
"description": "Will be called when disconnect button is clicked.",
"bubbles": true,
"cancelable": true,
"detail": {
}
}
},
"slots": {}
}
```

References

OJET custom components

Create the project structure

Now you have your basic OJET application with the required UI components loaded in your Fusion media toolbar.

We'll now make updates to the basic application to handle inbound calls, outbound calls, and interact with your CTI supplier, and with your Fusion application. We'll add more files to our basic application to handle these scenarios. Dividing functionalities into different files makes our application more readable and maintainable.

Add Files

You can create the following folders and files in your application for this:

- **src/ts/cti/fusion/fusionHandler.ts** contains the `FusionHandler` class. This class contains functions having UEF API calls to handle scenarios mentioned here.
- **src/ts/cti/integrationActionHandler.ts** contains the `IntegrationActionHandler` class. This call acts as a bridge between your CTI supplier and your CTI accelerator application.
- **src/ts/cti/integrationEventsHandler.ts** contains the `IntegrationEventsHandler` class. This call acts as a bridge between your `FusionHandler` and your CTI accelerator application.
- **src/ts/cti/vendor/vendorHandler.ts** contains the supplier-specific classes for handling calls.

- **src/index.html** is the html page having the UI components loaded.

Integrate the UEF Library for Handling Call Flows

Overview of integrating UEF library for handling call flows

Now we'll move on to integrating the UEF library with your media toolbar application to handle inbound and outbound calls.

We'll cover the following topics:

- *Toggle agent availability*
- *Show Incoming Call Notification*
- *Accept an incoming call from the Fusion application or toolbar application*
- *Disconnect or reject an incoming call from the Fusion application or the toolbar application*
- *Outbound calls*

You can start the UEF integration by building the first requirement, which is to toggle the agent availability between your media toolbar application and your Fusion application.

Toggle agent availability

Before you can perform call-related operations in the CTI app we need to set up the capability to toggle the Agent availability status. For each supplier there's an API.

To make the agent available and unavailable once we toggle the agent availability we need to notify the Fusion application that the agent state has been changed so that the Fusion application can display the agent as available or unavailable.

Steps to implement the agent availability

1. Add a knockout variable to handle Agent availability status
2. Add a method `toggleAgentAvailability` in `RootViewModel` class
3. Replace the toolbar content with a button
4. Inject the UEF JavaScript library
5. Add a method to initialize UEF in the `FusionHandler` class
6. Add the `toggleAgentAvailability` method in `FusionHandler` class
7. Add the `makeAgentAvailable` and `makeAgentUnavailable` methods in `FusionHandler` class
8. Create a new file `ICtiVendorHandler`
9. Update the `IntegrationEventsHandler` class to handle agent availability
10. Update the `VendorHandler` class to implement the `ICtiVendorHandler` interface
11. Update the `IntegrationActionHandler` class
12. Initialize the `IntegrationActionHandler` and `IntegrationEventsHandler` in the `RootViewModel` class
13. Update the `toggleAgentAvailability` method in the `appController` file

1. Add a knockout variable to handle Agent availability status

Introduce a new knockout observable of type boolean to track the agent states. To do this, add the following lines of code to the `src/ts/appController.ts` file. You'll also need to initialize your app classification. ORA_SERVICE is set as the app classification here.

The following table lists the ready-to-use application classification codes that the application recognizes. The list of application classifications can be changed using Functional Setup Manager.

Application Classification Codes

Application Classification Code	Description
ORA_HRHD	Default classification for Human Resources Help Desk related setup for Lookup rules and Screen Pop rules.
ORA_SALES	Default classification for sales-related setup for Lookup rules and Screen Pop rules.
ORA_SERVICE	Default classification for service-related setup for Lookup rules and Screen Pop rules.

Here are the code lines to add to the `src/ts/appController.ts` file:

```
//.....
import "oj-c/button";
// ....

class RootViewModel {
 // ....
 agentState: ko.Observable<boolean>;
 appClassification: string;
 // ...

 constructor() {
 // ....

 // CTI app properties
 this.agentState = ko.observable(false);
 this.appClassification = 'ORA_SERVICE';
 //...
 }
}
```

2. Add a method toggleAgentAvailability in RootViewModel class

In the `appController.ts` file add the function `toggleAgentAvailability`to update the `agentState`.

```
public toggleAgentAvailability = async () => {
 this.agentState(!this.agentState());
}
```

3. Replace the toolbar content with a button

Replace the `<oj-toolbar> innerHtml` in `src/index.html` file with the following to show an icon button:

```
<oj-c-button id="toggleAgentAvailability" label="Agent" chroming="solid" on-oj-
action="[[toggleAgentAvailability]]"
 :class="[[{'oj-bg-success-30' : agentState(), 'oj-bg-danger-30' : !agentState()}]]">
```

```
 <span slot="endIcon"
 :class="[[{'oj-ux-ico-user-available' : agentState(), 'oj-ux-ico-user-not-available' : !agentState()}]]"></
span>
</oj-c-button>
```

4. Inject the UEF JavaScript library

Inject THE UEF JavaScript library into the `src/index.html` file at the end of the body tag.

```
<script src="https://static.oracle.com/cdn/ui-events-framework/libs/ui-events-framework-client.js"></script>
```

Download and copy the UEF type definition file to the path `src/types/uiEventsFramework.d.ts`. Update the file `fusionHandler.ts` in `src/ts/cti/fusion/fusionHandler.ts` path and add the following content to the file.

```
/// <reference path ="../../../types/uiEventsFramework.d.ts"/>

export class FusionHandler {

}
```

5. Add a method to initialize UEF in the FusionHandler class

Create a method `initializeUef` in `fusionHandler` class and add the following lines of code to initialize the UEF:

```
export class FusionHandler {
 private static frameworkProvider: IUiEventsFrameworkProvider;
 private static phoneContext: IPhoneContext;
 public static appClassification: string = '';

 public static async initializeUef(): Promise<void> {
 if (!FusionHandler.frameworkProvider) {
 FusionHandler.frameworkProvider = await CX_SVC_UI_EVENTS_FRAMEWORK.uiEventsFramework.initialize('cti-
accelerator', 'v1');
 }
 if (!FusionHandler.phoneContext) {
 const mcaContext: IMultiChannelAdaptorContext = await
 FusionHandler.frameworkProvider.getMultiChannelAdaptorContext();
 FusionHandler.phoneContext = await mcaContext.getCommunicationChannelContext('PHONE') as IPhoneContext;
 }
 }
}
```

6. Add toggleAgentAvailability method in FusionHandler class

Add a method `toggleAgentAvailability` in the `FusionHandler` class and write the logic to publish *agentStateEvent*. This method will update the agent availability based on the input parameter `isAgentAvailable`.

```
private static async toggleAgentAvailability(isAgentAvailable: boolean) {
 const requestObject: IMcaAgentStateEventActionRequest =
 FusionHandler.frameworkProvider.requestHelper.createPublishRequest('agentStateEventOperation') as
 IMcaAgentStateEventActionRequest;
 requestObject.setEventId('1');
 requestObject.setIsAvailable(isAgentAvailable);
 requestObject.setIsLoggedIn(isAgentAvailable);
 requestObject.setState(isAgentAvailable ? 'AVAILABLE' : 'UNAVAILABLE');
 requestObject.setStateDisplayString('Idle');
 requestObject.setReason('');
 requestObject.setReasonDisplayString('Idle');
 requestObject.setInData({ 'phoneLineId': '1' });
 await FusionHandler.phoneContext.publish(requestObject) as IMcaAgentStateEventActionResponse;
```

```
}
```

7. Add makeAgentAvailable and makeAgentUnavailable methods in the FusionHandler class

Add `makeAgentAvailable` and `makeAgentUnavailable` in the `FusionHandler` class and call the `toggleAgentAvailabilty` method.

```
public static async makeAgentAvailable(): Promise<void> {
 await FusionHandler.toggleAgentAvailability(true);
}

public static async makeAgentUnavailable(): Promise<void> {
 await FusionHandler.toggleAgentAvailability(false);
}
```

8. Create a new file ICtiVendorHandler

Copy below the file to the path `src/ts/cti/vendor/ICtiVendorHandler.ts`, this is an interface with methods that needs to be implemented to call the supplier API for supporting CTI flows. (NEED FILE)

9. Update IntegrationEventsHandler class to handle agent availability

Update the class file `integrationEventsHandler.ts` at the `src/ts/cti/integrationEventsHandler.ts` path and copy the following content to the file:

Add the `MakeAgentAvailable` method as shown in the following example:

```
import RootViewModel from "../appController";
import { FusionHandler } from "./fusion/fusionHandler";

export class IntegrationEventsHandler {
 public ctiAppViewModel: typeof RootViewModel;
 constructor(ctiAppViewModel: any) {
 this.ctiAppViewModel = ctiAppViewModel;
 }
}
```

Add the `makeAgentAvailable` method as shown in the following example:

```
public async makeAgentAvailable(): Promise<void> {
 try {
 await FusionHandler.makeAgentAvailable();
 } catch (err) {
 console.log("Error while making agent available", err);
 }
}
```

Add the `makeAgentUnavailable` method as shown in the following example:

```
public async makeAgentUnavailable(): Promise<void> {
 try {
 await FusionHandler.makeAgentUnavailable();
 } catch (err) {
 console.log("Error while making agent unavailable", err);
 }
}
```

10. Update VendorHandler Class to implement ICtiVendorHandler interface

Update the class file `vendorHandler.ts` at the `src/ts/cti/vendor/vendorHandler.ts` path to implement the `ICtiVendorHandler interface` as in the following example:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private integrationEventsHandler: IntegrationEventsHandler;
 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }
 public async makeAgentAvailable() {
 // TODO: call the vendor specific api to make the agent available
 }
 public async makeAgentUnavailable() {
 // TODO: call the vendor specific api to make the agent unavailable
 }
 public async makeOutboundCall(phoneNumber: string, eventId: string) {
 throw new Error('Method not implemented.');
 }
 public async acceptCall() {
 throw new Error('Method not implemented.');
 }
 public async rejectCall() {
 throw new Error('Method not implemented.');
 }
 public async hangupCall() {
 throw new Error('Method not implemented.');
 }
}
```

11. Update IntegrationActionHandler class

Update the class file `integrationActionHandler.ts` at the `src/ts/cti/integrationActionHandler.ts` path as in the following example:

```
import { IntegrationEventsHandler } from "./integrationEventsHandler";
import { ICtiVendorHandler } from "./vendor/ICtiVendorHandler";
import { VendorHandler } from "./vendor/vendorHandler";

export class IntegrationActionHandler {
 private vendor: ICtiVendorHandler;
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.vendor = new VendorHandler(integrationEventsHandler);
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async makeAgentAvailable(): Promise<void> {
 await this.vendor.makeAgentAvailable();
 await this.integrationEventsHandler.makeAgentAvailable();
 }

 public async makeAgentUnavailable(): Promise<void> {
 await this.vendor.makeAgentUnavailable();
 await this.integrationEventsHandler.makeAgentUnavailable();
 }
}
```

12. Initialize IntegrationActionHandler and IntegrationEventsHandler in RootViewModel class

Initialize the `integrationActionHandler` and `integrationEventsHandler` classes in `appController.ts` constructor with the following code:

```
//.....
import "oj-c/button";
import {IntegrationActionHandler} from "./cti/integrationActionHandler";
import {IntegrationEventsHandler} from "./cti/integrationEventsHandler";
import {FusionHandler} from "./cti/fusion/fusionHandler";
// ....

class RootViewModel {
 // ....
 agentState: ko.Observable<boolean>;
 appClassification: string;
 integrationActionHandler: IntegrationActionHandler;
 integrationEventsHandler: IntegrationEventsHandler;
 // ...

 constructor() {
 // ....

 // CTI app properties
 this.agentState = ko.observable(false);
 this.appClassification = 'ORA_SERVICE';
 FusionHandler.appClassification = this.appClassification;
 this.integrationEventsHandler = new IntegrationEventsHandler(this);
 this.integrationActionHandler = new IntegrationActionHandler(this.integrationEventsHandler);
 FusionHandler.initializeUef();
 //...
 }
}
```

13. Update toggleAgentAvailability method in appController file

```
public toggleAgentAvailability = async () => {
 if (!this.agentState()) {
 await FusionHandler.initializeUef();
 this.integrationActionHandler.makeAgentAvailable();
 } else {
 this.integrationActionHandler.makeAgentUnavailable();
 }
 this.agentState(!this.agentState());
}
```

Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the agent availability button from your media toolbar application. You'll see, the button color change to green and the phone icon status in the Fusion application change to **Available**. If you click the button again the status button color should change back to red and the phone icon status in the Fusion application will change to **Unavailable**.

Incoming Calls from the Fusion application

Show Incoming Call Notification

In the previous section, we saw, how to toggle the agent availability for calls. Once the agent's availability is set to Available for calls, the next setup step is to show the inbound call notification from your CTI supplier in the media toolbar and the Fusion application.

Steps to implement the incoming call notification

Part 1: Show incoming call notifications in your media toolbar application

1. Add a call-panel component in the media toolbar application.
2. Define the `callContext` property.
3. Define the function `handleIncomingCall` to update the call-panel component property during an incoming call event.
4. Add a generic function in the `src/ts/appController.ts` file to update the call-pane state.
5. Add a method called `incomingCallHandler` in the `integrationEventsHandler` class.

Part 2: Show incoming call notifications in the Fusion application

1. Add a method called `showIncomingCallNotificationInFusion` in the `FusionHandler` class and write the logic to publish `newCommEvent`.
2. Call the `showIncomingCallNotificationInFusion` function from the `incomingCallHandler` function.
3. Call the `incomingCallNotificationHandler` function from your `vendorHandler` file.

- Add a method called `showIncomingCallNotificationInFusion` in the `FusionHandler` class and write the logic to publish `newCommEvent`.
- Call the `showIncomingCallNotificationInFusion` function from the `incomingCallHandler` function.
- Call the `incomingCallNotificationHandler` function from your `vendorHandler` file.

Part 1: Show incoming call notifications in your media toolbar application

When an incoming called from the CTI supplier comes in, a UI component is displayed in the media toolbar UI.

This component is a JET component consisting of call information and two buttons, for accepting or rejecting the call. Call information consists of the contact name associated with the incoming phone number and the IVR data. The label Unknown Number is displayed if the incoming phone number isn't saved in the Fusion application. If you followed the previous steps, you might have already created this component in your media toolbar application. So, you only need to add this component to your application.

Add a call-panel component in the media toolbar application

You can add the call-panel component in the `index.html` file of your toolbar application. In the `src/index.html` file, add the component in the div with role `main` as shown in the following example:

```
<oj-bind-if test="[[callContext().state === 'RINGING' || callContext().state === 'ACCEPTED']]">
 <call-panel call-context="[[callContext]]"></call-panel>
</oj-bind-if>
```

In the code example, call-context is the only property that's passed to the call panel component. So, you need to define `callContext` in your `appController.ts` file. The `callContext` property contains call-related information like phone

number, caller name, direction, event id, IVR data, and state of the call. For simplicity, we can define three states for a call to our media toolbar application:

- RINGING: The toolbar application will be in the RINGING state when the agent receives an incoming call from a customer. The toolbar application will also be in a ringing state when an agent starts an outbound call to a customer, which isn't yet picked up.
- ACCEPTED: The toolbar application will be in the ACCEPTED state when the agent accepts an incoming call from the customer. The toolbar application will also be in an ACCEPTED state when a customer accepts an outbound call started by an agent.
- DISCONNECTED: The toolbar application will move to the DISCONNECTED state when an agent rejects an incoming call from a customer or disconnects an ongoing call with the customer.

The call component is loaded conditionally only when the call state is RINGING or ACCEPTED.

Define callContext property

The `callContext` property must be passed to the call-panel component. Define the `callContext` property in your `appController.ts` file found in `src/ts/appController.ts` using the following code:

```
//.....
import "oj-c/button";
import "./jet-composites/call-panel/loader";
// ....

interface CallContext {
 phonenumber: string;
 callerName: string;
 direction: string;
 eventId: string;
 ivrData: { [key: string]: string };
 state: string;
}

class RootViewModel {
 // ....
 callContext: ko.Observable<CallContext>;
 // ...

 constructor() {
 // ....

 // CTI app properties
 this.callContext = ko.observable({
 phonenumber: "+918921670701",
 callerName: "John Doe",
 direction: "inbound",
 eventId: this.mockEventId(),
 ivrData: {
 jobTitle: "Developer",
 phoneType: "Work Phone",
 additional: "Some more"
 },
 state: "AVAILABLE" //AVAILABLE RINGING
 } as CallContext);
 //...
 }
}
```

Define the function handleIncomingCall to update the call-panel component property during an incoming call event

```
//.....
import "oj-c/button";
import "./jet-composites/call-panel/loader";
// ....

interface CallContext {
 phonenumber: string;
 callerName: string;
 direction: string;
 eventId: string;
 ivrData: { [key: string]: string };
 state: string;
}

class RootViewModel {
 // ....
 callContext: ko.Observable<CallContext>;
 mockEventId: ko.Observable<string> = ko.observable(Date.now().toString());
 // ...

 constructor() {
 // ....

 // CTI app properties
 this.callContext = ko.observable({
 phonenumber: "+918921670701",
 callerName: "John Doe",
 direction: "inbound",
 eventId: this.mockEventId(),
 ivrData: {
 jobTitle: "Developer",
 phoneType: "Work Phone",
 additional: "Some more"
 },
 state: "AVAILABLE" //AVAILABLE RINGING
 } as CallContext);
 //...
 }
 public handleIncomingCall = (customerName: string, incomingPhoneNumber: string, eventId: string) => {
 this.callContext({...this.callContext(),
 direction: 'inbound',
 phonenumber: incomingPhoneNumber,
 callerName: customerName,
 eventId: eventId,
 state: 'RINGING'});
 };
 public updateCallPanelState = (state: string) => {
 this.callContext({...this.callContext(), state: state});
 }
}
```

Add a generic function in src/ts/appController.ts to update the call-pane state

During the life cycle of a phone call, you need to update the state of the call-pane component several times. You can define the following generic function `updateCallPanelState` to make it easy::

```
//.....
import "oj-c/button";
import "./jet-composites/call-panel/loader";
// ....
```

```
interface CallContext {
 phonenumber: string;
 callerName: string;
 direction: string;
 eventId: string;
 ivrData: { [key: string]: string };
 state: string;
}

class RootViewModel {
 // ....
 callContext: ko.Observable<CallContext>;
 mockEventId: ko.Observable<string> = ko.observable(Date.now().toString());
 // ...

 constructor() {
 // ....

 // CTI app properties
 this.callContext = ko.observable({
 phonenumber: "+918921670701",
 callerName: "John Doe",
 direction: "inbound",
 eventId: this.mockEventId(),
 ivrData: {
 jobTitle: "Developer",
 phoneType: "Work Phone",
 additional: "Some more"
 },
 state: "AVAILABLE" //AVAILABLE RINGING
 } as CallContext);
 //...
 }
 public updateCallPanelState = (state: string) => {
 this.callContext({...this.callContext(), state: state});
 }
}
```

Add a method incomingCallHandler to the integrationEventsHandler class

The `incomingCallHandler` method can be considered as a generic method that should be called during the incoming call event from your CTI supplier. From this method, you can call the `handleIncomingCall` function to show the incoming call notification component in your toolbar application.

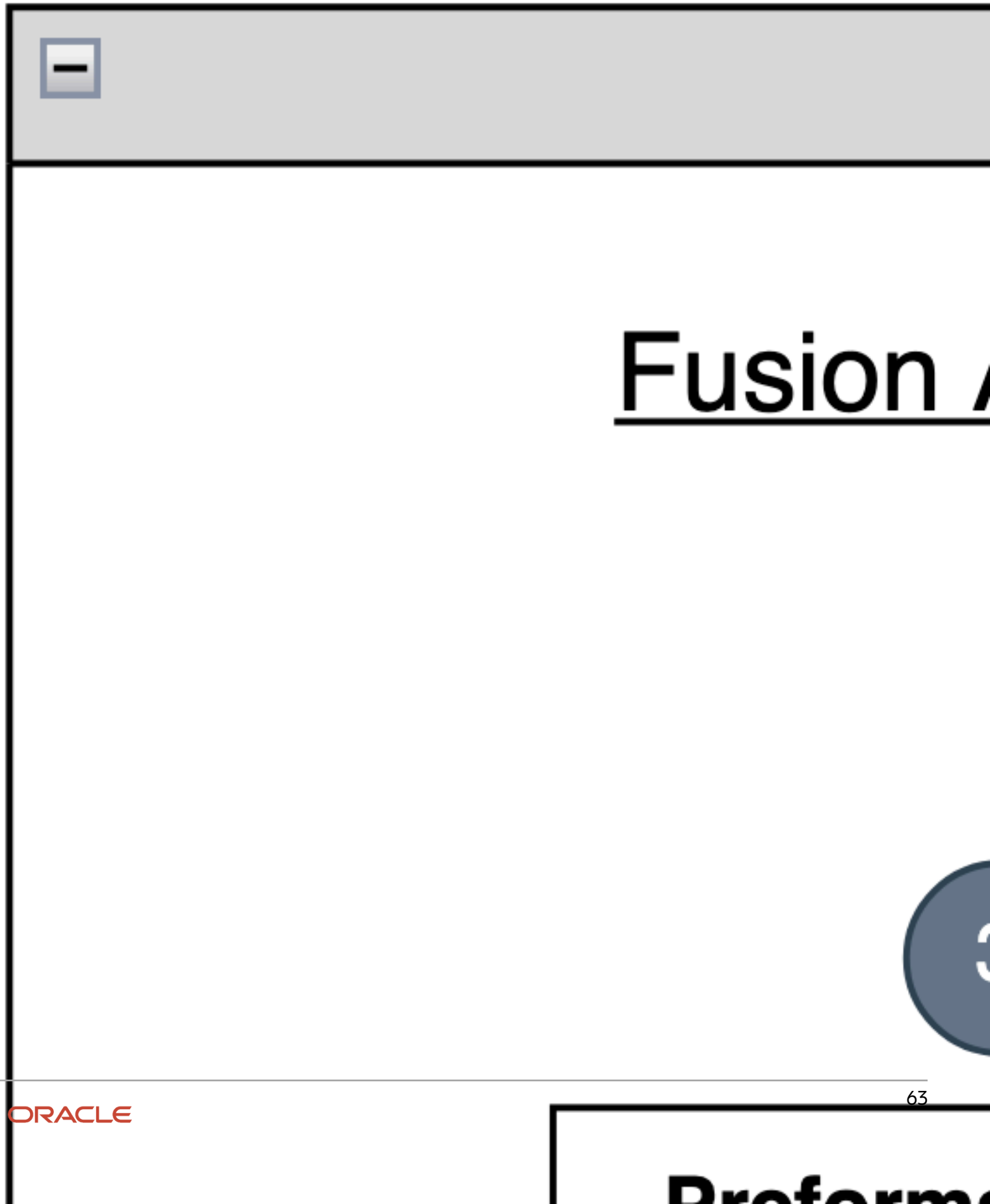
```
/**
 *
 * @param incomingPhoneNumber
 * @param connectionId
 * Invoke this function when you receive an incoming call notification from your CTI system.
 * Once this is invoked, an incoming call notification popup will be shown in the fusion application.
 * Please refer doc shared for more information on newCommEvent API.
 */
public async incomingCallHandler(incomingPhoneNumber: string, eventId: string): Promise<void> {
 try {
 this.ctiAppViewModel.handleIncomingCall('', incomingPhoneNumber, eventId);
 } catch {
 // Catch errors
 }
}
```

Note that, the first parameter of the`handleIncomingCall` function is the contact name associated with the phone number. For now, it's passed as an empty string. In the below part, you'll see, how you can get the contact name from the Fusion application so it can be passed here instead of an empty string.

Part 2: Showing incoming call notifications in the Fusion application

When there's an incoming call from your CTI supplier, a call notification dialog box will be displayed in the Fusion application.

The following ring received scenario handles this process.


Fusion A

2. Add a method showIncomingCallNotificationInFusion in the FusionHandler class and write the logic to publish the newCommEvent

You'll get the `newCommResponse` from the `showIncomingCallNotificationInFusion` function call. You can get the contact name from this response and can be passed to `handleIncomingCall` function as shown in the following example:

```
public async incomingCallHandler(incomingPhoneNumber: string, eventId: string): Promise<void> {
 try {
 // Add logic to render the incoming call component and to notify Fusion about the incoming call
 const newCommResponseFromFusion: IMcaOutData = await
 FusionHandler.showIncomingCallNotificationInFusion(incomingPhoneNumber, eventId);
 const incomingCallCustomerName: string = newCommResponseFromFusion['SVCMCA_CONTACT_NAME'] ?
 newCommResponseFromFusion['SVCMCA_CONTACT_NAME'] : 'Unknown Number';
 this.ctiAppViewModel.handleIncomingCall(incomingCallCustomerName, incomingPhoneNumber, eventId);
 } catch {
 // Catch errors
 }
}
```

Call showIncomingCallNotificationInFusion function from incomingCallHandler function

You'll get the `newCommResponse` from the `showIncomingCallNotificationInFusion` function call. You can get the contact name from this response and can be passed to `handleIncomingCall` function as shown in the following example:

```
public async incomingCallHandler(incomingPhoneNumber: string, eventId: string): Promise<void> {
 try {
 // Add logic to render the incoming call component and to notify Fusion about the incoming call
 const newCommResponseFromFusion: IMcaOutData = await
 FusionHandler.showIncomingCallNotificationInFusion(incomingPhoneNumber, eventId);
 const incomingCallCustomerName: string = newCommResponseFromFusion['SVCMCA_CONTACT_NAME'] ?
 newCommResponseFromFusion['SVCMCA_CONTACT_NAME'] : 'Unknown Number';
 this.ctiAppViewModel.handleIncomingCall(incomingCallCustomerName, incomingPhoneNumber, eventId);
 } catch {
 // Catch errors
 }
}
```

3. Call incomingCallNotificationHandler function from your vendorHandler file

See your CTI supplier's documentation to identify how an incoming call is notified to the CTI application using their APIs. The `incomingCallNotificationHandler` function should be called from there. You can add the logic in the `vendorHandler.ts` file:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';

export class VendorHandler implements ICtiVendorHandler {
 public async makeAgentAvailable() {
 // TODO: call the vendor specific api to make the agent available
 // TODO: call the vendor specific api to show incoming call notification
 }
 public async makeAgentUnavailable() {
 // TODO: call the vendor specific api to make the agent unavailable
 }
 public async makeOutboundCall(phoneNumber: string, eventId: string) {
 throw new Error('Method not implemented.');
 }
 public async acceptCall() {
 throw new Error('Method not implemented.');
 }
 public async rejectCall() {
 throw new Error('Method not implemented.');
```

```
  }
  public async hangupCall() {
  throw new Error('Method not implemented.');
  }
 }
```

Verify your progress

Once you complete the previous steps, use the OJET serves to start your application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking the agent availability button. Now, start a call to your customer care number. During this time, your media toolbar state will be changed to the RINGING state and you'll see the incoming call notification in your media toolbar application.

Accept an incoming call from the Fusion application or toolbar application

Now that you've set up incoming call notifications in your media toolbar application and Fusion application, now you set up accepting them from the media toolbar application and your Fusion application.

You can screen pop the corresponding UI (Customer 360, Service request details,and so on) to help the Agent during the call. Once you do the setup the following setup tasks, you can accept the incoming calls.

Overview of the tasks

Here's an overview of what you need to do:

- Part 1: Accepting incoming calls from your media toolbar application
  a. Add an event handler for the Accept button click event.
  b. Define the `callAcceptedEventHandler` function.
  c. Define the function `acceptIncomingCall in integrationActionHandler.ts` file.
  d. Implement the `acceptCall()` function in the `vendorHandler.ts` file based on your CTI supplier's SDK.
  e. Define the function `notifyCallAcceptedToFusion in fusionHandler.ts` file.
- Part 2: Accepting incoming calls from the Fusion application
  a. Add a method called `subscribeToToolbarInteractionCommandsFromFusion` in the `FusionHandler` class and write the logic to subscribe to `onToolbarInteractionCommand` event.
  b. Call the function from the `makeAgentAvailable` function in the `integrationEventsHandler.ts` file.
  c. Define the function `listenToToolbarInteractionCommandsFromFusion` in the `integrationEventsHandler.ts` file.

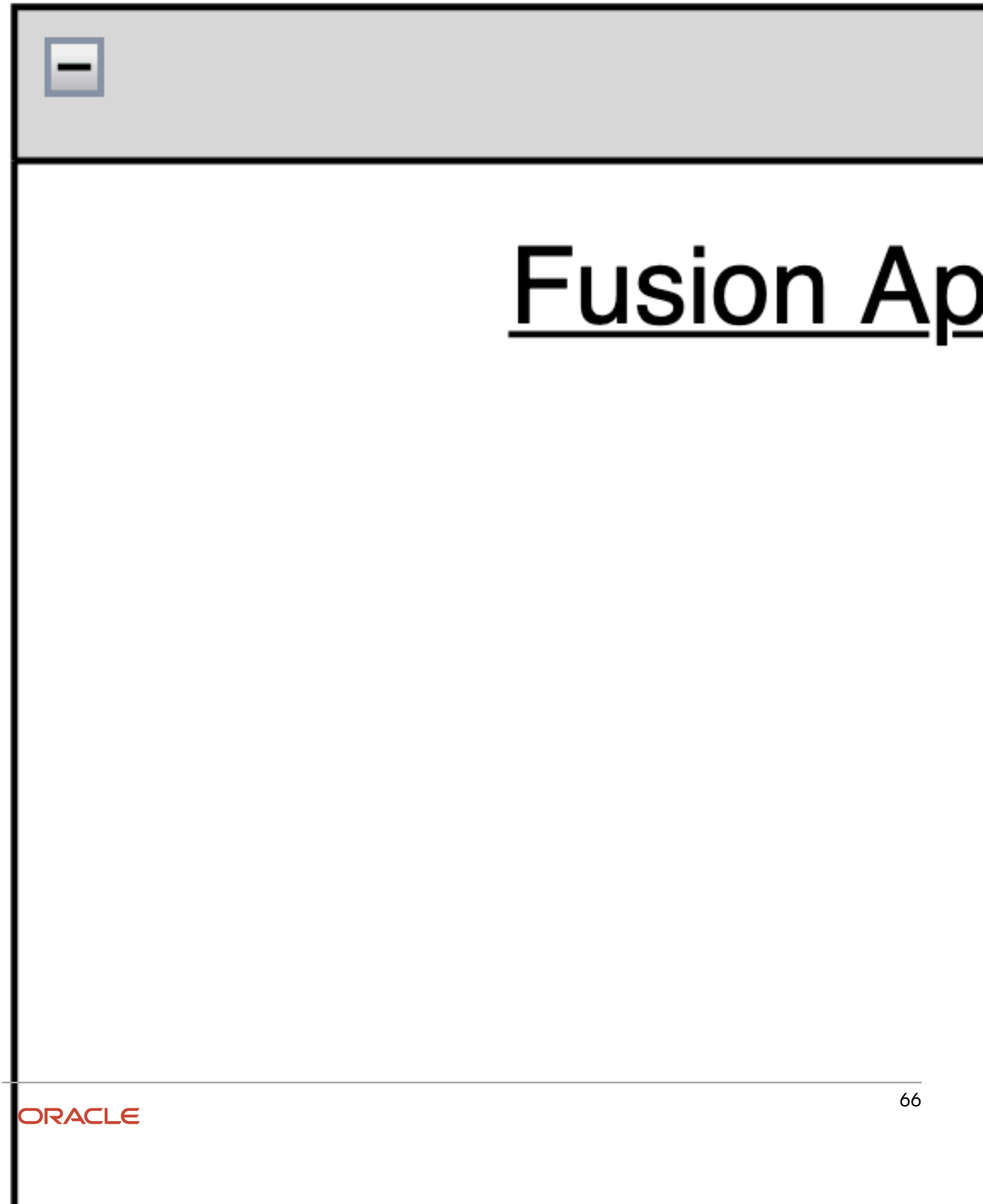
Part 1: Accepting incoming calls from your media toolbar application

Once the agent clicks on the Accept button, the call-panel component fires an event notifying its container where the component is loaded. In our case, the component is loaded in the media toolbar application. An event is similarly fired when the Reject button is selected.

Also, a screen pop of the corresponding UI (Customer 360, Service request details, and so on) appears to help the agent during the call.

From the container, an event handler must be defined to handle these events.

The following diagram shows the sequence of actions to be performed when the agent accepts the call from the media toolbar application:


Fusion Ap

Here's the agent call flow:

1. The agent clicks on the **Answer** button from the media toolbar application.
2. The Partner application notifies the CTI supplier to accept the call by calling the supplier-specific APIs.
3. Once the CTI supplier notifies the partner application that the call is accepted, the partner application can fire the `startCommEvent` action.
4. Once the Fusion application identifies this action, the matched contact and an engagement panel is opened based on the rules configured in the Fusion configuration management for the screen pop.

   To configure the screen pop rules, see *How do I configure screen pop pages?*.

   The following steps show you how to implement this in your toolbar application:

1. Add an event handler for the accept button click event

Add the Accept button clicked event handler in call-panel component as in the following example of the index.html file:

```
<oj-bind-if test="[[callContext().state === 'RINGING' || callContext().state === 'ACCEPTED']]">
 <call-panel call-context="[[callContext]]"
 on-accept-button-clicked="[[ callAcceptedEventHandler ]]"></call-panel>
</oj-bind-if>
```

2. Define callAcceptedEventHandler function

Define the `callAcceptedEventHandler` function in the `appController.ts` file as shown in the following example. The `callAcceptedEventHandler` function internally calls two other functions, `acceptIncomingCall`, which is defined in `integrationActionHandler`, and `updateCallPanelState`, which is defined in `appController.ts` itself.

```
//.....
import "oj-c/button";
// ....

class RootViewModel {
 // ....

 constructor() {
 // ....
 };

 public callAcceptedEventHandler: (event: any) => void = (event: any): void => {
 this.integrationActionHandler.acceptIncomingCall(this.callContext().eventId).then(() => {
 this.updateCallPanelState('ACCEPTED');
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 }

 //...
 }

}
```

3. Define the function acceptIncomingCall in integrationActionHandler.ts file

The `acceptIncomingCall` function contains the logic to notify your CTI supplier to accept the call and also it contains the logic to notify the Fusion application that the call is accepted by firing the `startCommEvent` action.

```
import { IntegrationEventsHandler } from "./integrationEventsHandler";
import { ICtiVendorHandler } from "./vendor/ICtiVendorHandler";
```

```
import { VendorHandler } from "./vendor/vendorHandler";
import { FusionHandler } from "./fusion/fusionHandler";

export class IntegrationActionHandler {
 private vendor: ICtiVendorHandler;
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.vendor = new VendorHandler();
 this.integrationEventsHandler = integrationEventsHandler;
 }

 //....

 public async acceptIncomingCall(eventId: string): Promise<IMcaStartCommEventOutData> {
 await this.vendor.acceptCall();
 const startCommResponseFromFusion: IMcaStartCommEventOutData = await
 FusionHandler.notifyCallAcceptedToFusion(eventId);
 return startCommResponseFromFusion;
 }

}
```

4. Implement acceptCall() function in vendorHandler.ts file based on your CTI provider's SDK

See your CTI supplier's documentation to identify the API used to accept a call. You can add the logic in the `vendorHandler.ts` file:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';

export class VendorHandler implements ICtiVendorHandler {
 // ....
 public async acceptCall() {
 // TODO: call the vendor specific api to accept a call
 }
 // ....
}
```

5. Define function notifyCallAcceptedToFusion in fusionHandler.ts file

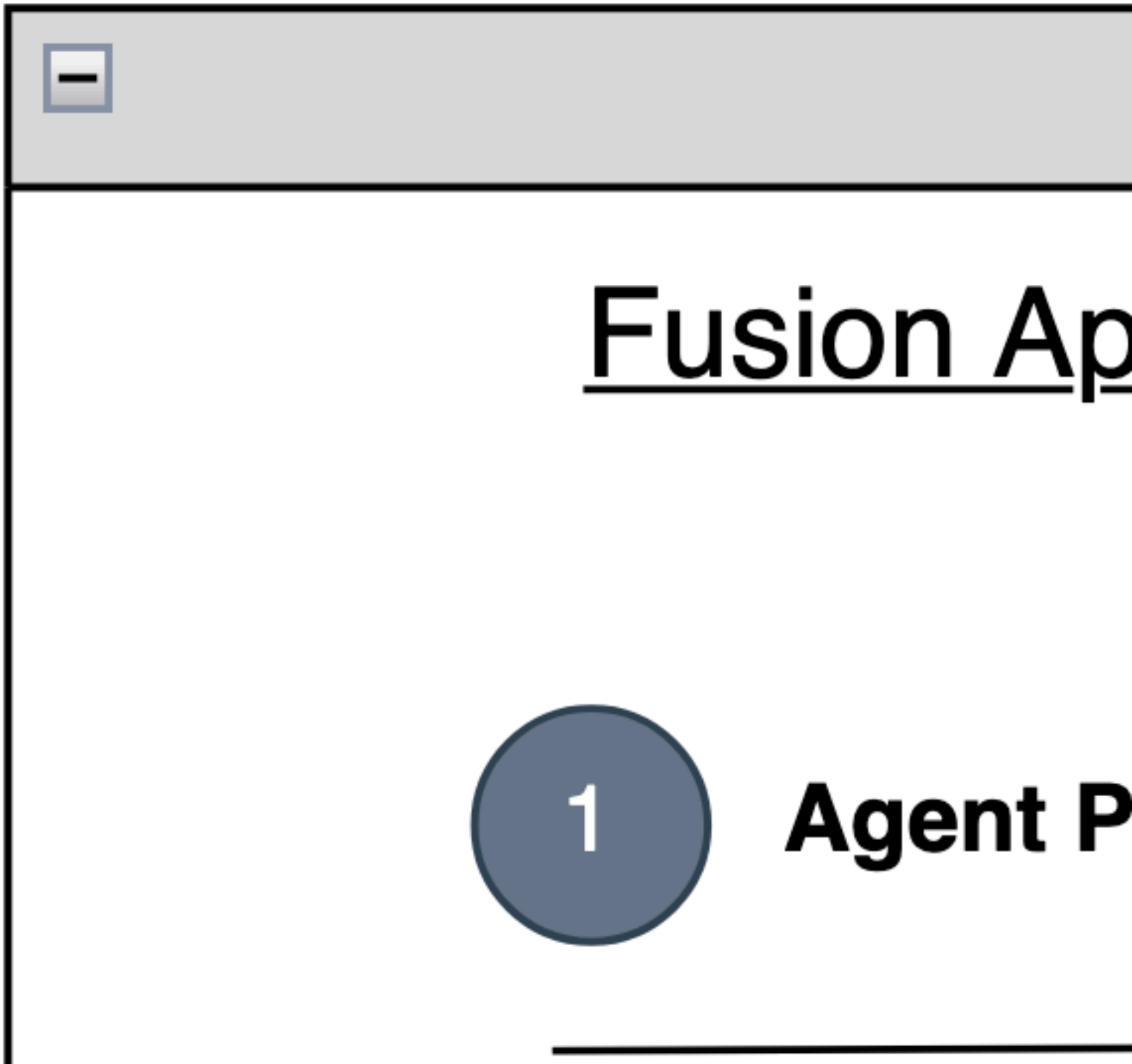
From the `fusionHandler.ts`, you notify the Fusion application that the call is accepted by firing the `startCommEvent` action.

```
export class FusionHandler {
 private static engagementContext: IEngagementContext;
 // ....
 public static async notifyCallAcceptedToFusion(eventId: string): Promise<IMcaStartCommEventOutData> {
 let request: IMcaStartCommEventActionRequest =
 FusionHandler.frameworkProvider.requestHelper.createPublishRequest('startCommEvent') as
 IMcaStartCommEventActionRequest;
 const newCommResponse: any = FusionHandler.callToNewCommResponseMap.get(eventId);
 if (newCommResponse) {
 for (const property in newCommResponse) {
 request.getInData().setInDataValueByAttribute(property, newCommResponse[property]);
 }
 }
 //TODO pass unique identifier from your CTI data
 request.setEventId(eventId);
 request.setAppClassification(FusionHandler.appClassification);
 const operationResponse: IMcaStartComActionResponse = await FusionHandler.phoneContext.publish(request) as
 IMcaStartComActionResponse;
 FusionHandler.engagementContext = operationResponse.getResponseData().getEngagementContext();
 return operationResponse.getResponseData().getOutData();
```

```
    }
    // ...
}
```

Part 2: Accepting incoming calls from the Fusion application

Now, when an agent clicks the Accept button from the Fusion call notification, you need to notify your media toolbar application and your CTI supplier that the call is accepted by the agent. You need to use the call accepted from the Fusion scenario to handle this. The following diagram shows the sequence of actions to be performed when an agent accepts the call from the Fusion application:



1. When the agent clicks the Answer button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as accept.

2. The partner application receives this event and if the event is to accept a call, the partner application notifies the CTI supplier to accept the call.
3. Once the CTI supplier notifies the partner application that the call is accepted, the partner application fires the `startCommEvent` action.
4. Once the Fusion application identifies this action, the matched contact and an engagement panel are opened based on the rules configured in the Fusion configuration management for the screen pop.

   To configure the screen pop rules, see *How do I configure screen pop pages?*.

1: Add a method subscribeToToolbarInteractionCommandsFromFusion in the FusionHandler class and write the logic to subscribe to the onToolbarInteractionCommand event

The agent's interactions, such as accepting, rejecting, and disconnecting phone calls can be identified by subscribing to the `onToolbarInteractionCommand` event through the UEF API. The event response consists of information about the operations performed by the agent in the Fusion application window. Add a function for the `onToolbarInteractionCommand` event in the `fusionHandler.ts` file as shown in the following example:

```
export class FusionHandler {
 // ....
 public static subscribeToToolbarInteractionCommandsFromFusion(callback: Function): void {
 const request: IMcaEventRequest =
 FusionHandler.frameworkProvider.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand') as
 IMcaEventRequest;
 FusionHandler.phoneContext.subscribe(request, (eventResponse: IEventResponse) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse = (eventResponse as
 IMcaOnToolbarInteractionCommandEventResponse).getResponseData();
 callback(eventResponseDetails.getCommand());
 });
 }
 // ....

}
```

When an event is received from the Fusion application window, it will execute a callback function by passing the `eventType` to the callback function.

2. Call the above function from makeAgentAvailable function in integrationEventsHandler.ts

The `onToolbarInteractionCommand` event subscription must be added during the initial step when an agent is made available. So, `subscribeToToolbarInteractionCommandsFromFusion` is called when making an agent available. A callback function is also passed as an argument to the `subscribeToToolbarInteractionCommandsFromFusion` function. The callback function calls another function `listenToToolbarInteractionCommandsFromFusion` which must be defined in the `interactionEventsHandler.ts` file itself.

```
export class IntegrationEventsHandler {
 // ....
 public async makeAgentAvailable(): Promise<void> {
 try {
 await FusionHandler.makeAgentAvailable();
 FusionHandler.subscribeToToolbarInteractionCommandsFromFusion((command: string) =>
 { this.listenToToolbarInteractionCommandsFromFusion(command); });
 this.ctiAppViewModel.agentState(true);
 } catch (err) {
 console.log("Error while making agent available", err)
 }
 }
 // ....
```

```
}
```

3. Define the function listenToToolbarInteractionCommandsFromFusion in integrationEventsHandler.ts

This function takes a command variable as a parameter. When a call is accepted, the value passed for the command variable is `accept`. During this case, `callAcceptedEventHandler` function is launched, which is defined in `appController` and this will in turn call the `acceptCall` function in your `vendorHandler.ts` file and call the `startCommEvent` action and also updates the call-panel component UI that a call is ACCEPTED.

```
export class IntegrationEventsHandler {
 // ....
 public listenToToolbarInteractionCommandsFromFusion(command: string): void {
 switch (command) {
 case "accept":
 this.ctiAppViewModel.callAcceptedEventHandler(null);
 break;
 }
 }
 // ....

}
```

Verify your progress

Once you complete the above steps, use the OJET serves to start your application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking Agent Availability button. Start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and your Fusion application window. You can accept the call from your media toolbar application or your Fusion application. Once you accept the call, your media toolbar state will be changed to ACCEPTED state and the engagement will be opened in your Fusion application.

Disconnect or reject an incoming call from the Fusion application or the toolbar application

Now you set up functionality to reject an incoming call or, disconnect the accepted call.

You can reject or disconnect a call from your media toolbar application and from the Fusion application.

Overview of the tasks

Here's an overview of what you need to do:

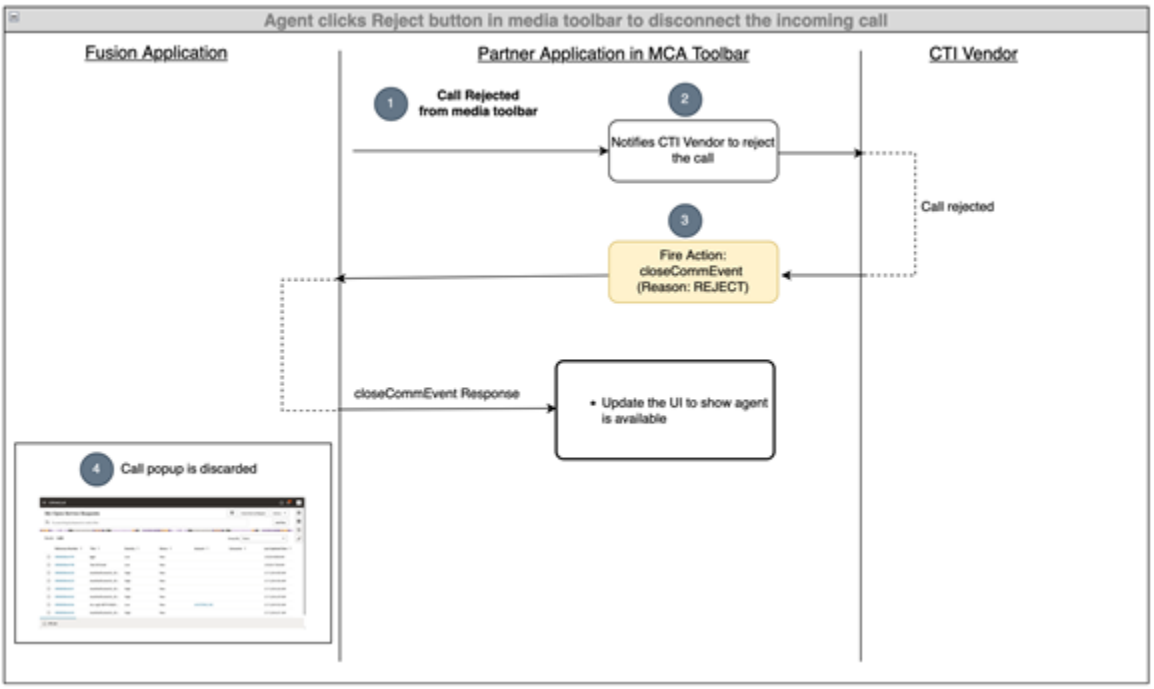
- Part 1: Rejecting/disconnecting a call from your media toolbar application
  a. Add an event handler for the Disconnect button click event.
  b. Define the `callDisconnectedEventHandler` function.
  c. Define the functions `disconnectCall` and `rejectCall` in the `integrationActionHandler.ts` file.
  d. Implement the `rejectCall()` and `hangupCall()` functions in the `vendorHandler.ts` file based on your CTI supplier's SDK.
  e. Define the function `notifyCallDisconnectedToFusion` in the `fusionHandler.ts` file.
- Part 2: Rejecting or disconnecting a call from the Fusion application
  a. Add cases for reject and disconnect to `listenToToolbarInteractionCommandsFromFusion` in the `integrationEventsHandler.ts` file.

Part 1: Rejecting or disconnecting a call from your media toolbar application

Once the agent clicks on the Reject or Disconnect button, the call-panel component fires an event notifying its container where the component is loaded. In our case, the component is loaded in the media toolbar application. An event is similarly fired when the Reject button is selected. When a call is disconnected before the call is accepted by the agent, an event with the payload `disconnectionState` as REJECT is fired and when a call is disconnected after the call is accepted by the agent, an event with payload `disconnectionState` as WRAPUP is fired.

From the container, an event handler must be defined to handle these events.

The following diagram shows the sequence of actions to be performed when the agent accepts the call from the media toolbar application:



Here's the agent call flow:

1. The agent clicks on the **Reject** button from the media toolbar application.
2. The Partner application notifies the CTI supplier to reject the call.
3. Once the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with the reason as REJECT.
4. Once the Fusion application identifies this action, the call dialog box will be discarded from the UI.

   The following steps show you how to implement this in your toolbar application:

1. Add an event handler for the disconnect button click event

Add the Reject button clicked event handler in call-panel component as in the following example of the index.html file:

```
<oj-bind-if test="[[callContext().state === 'RINGING' || callContext().state === 'ACCEPTED']]">
 <call-panel call-context="[[callContext]]"
 on-accept-button-clicked="[[ callAcceptedEventHandler ]]"
 on-disconnect-button-clicked="[[ callDisconnectedEventHandler ]]"></call-panel>
```

```
</oj-bind-if>
```

2. Define callDisconnectedEventHandler function

Define the `callDisconnectedEventHandler` function in the `appController.ts` file as shown in the following example. The `callDisconnectedEventHandler` function internally calls two other functions,`disconnectCall`, which is defined in `integrationActionHandler`, and`updateCallPanelState`, which is defined in `appController.ts` itself.

```
//.....
//.....
import "oj-c/button";
// ....

class RootViewModel {
 // ....

 constructor() {
 // ....
 };

 public callDisconnectedEventHandler: (event: any) => void = (event: any): void => {
 if (event.detail.disconnectionState === 'WRAPUP') {
 this.integrationActionHandler.disconnectCall(this.callContext().eventId,
 event.detail.disconnectionState).then(() => {
 this.updateCallPanelState('DISCONNECTED');
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 } else if (event.detail.disconnectionState === 'REJECT') {
 this.integrationActionHandler.rejectCall(this.callContext().eventId,
 event.detail.disconnectionState).then(() => {
 this.updateCallPanelState('DISCONNECTED');
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 }
 }

 public updateCallPanelState = (state: string) => {
 this.callContext({...this.callContext(), state: state});
 }

 //...
 }

}
```

3. Define the functions disconnectCall and rejectCall in integrationActionHandler.ts file

The `disconnectCall` function contains the logic to notify your CTI supplier to hang up or reject the call and also it contains the logic to notify the Fusion application that the call is disconnected by firing the `closeCommEvent` action.

```
import { IntegrationEventsHandler } from "./integrationEventsHandler";
import { ICtiVendorHandler } from "./vendor/ICtiVendorHandler";
import { VendorHandler } from "./vendor/vendorHandler";
import { FusionHandler } from "./fusion/fusionHandler";

export class IntegrationActionHandler {
 private vendor: ICtiVendorHandler;
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.vendor = new VendorHandler();
```

```
this.integrationEventsHandler = integrationEventsHandler;
}

// .....

public async disconnectCall(eventId: string, disconnectionState: string): Promise<void> {
await this.vendor.rejectCall();
await FusionHandler.notifyCallDisconnectedToFusion(eventId, disconnectionState);
}

public async rejectCall(eventId: string, disconnectionState: string): Promise<void> {
await this.vendor.rejectCall();
await FusionHandler.notifyCallDisconnectedToFusion(eventId, disconnectionState);
}

}
```

4. Implement rejectCall() and hangupCall() functions in vendorHandler.ts file based on your CTI provider's SDK

See your CTI supplier's documentation to identify how an incoming call is notified to the CTI application using their API.The incomingCallNotificationHandler function must be called from there. You can add the logic in `vendorHandler.ts` file:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';

export class VendorHandler implements ICtiVendorHandler {
// ....
public async rejectCall() {
// TODO: call the vendor specific api to reject a call
}
public async hangupCall() {
// TODO: call the vendor specific api to hangup a call
}
// ....
}
```

5. Define function notifyCallDisconnectedToFusion function in fusionHandler.ts file

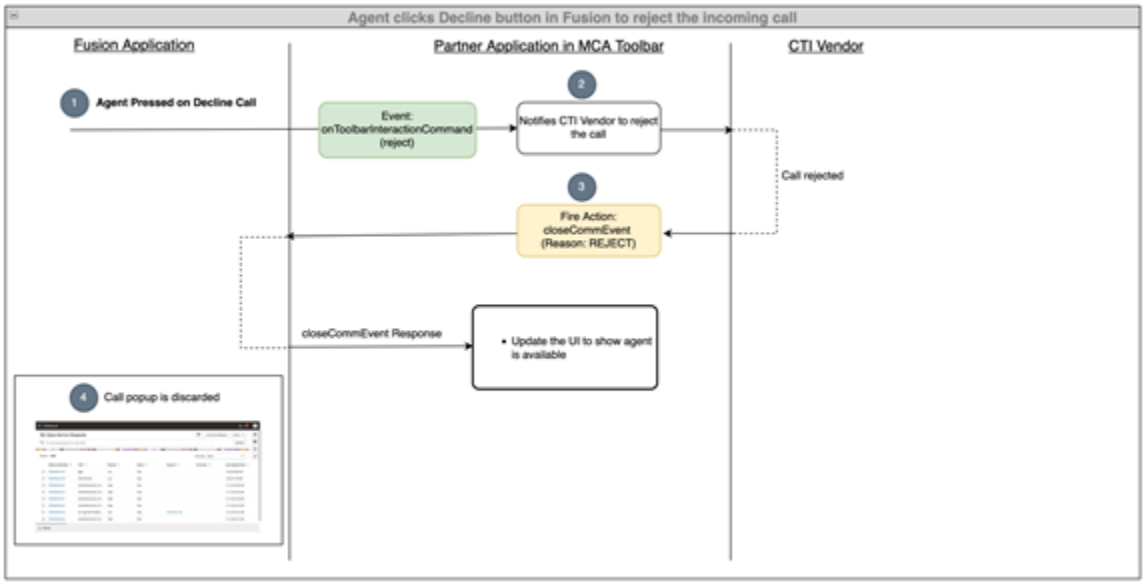
From the `fusionHandler.ts`, you notify the Fusion application that the call is rejected by firing the `closeCommEvent` action.

```
export class FusionHandler {
// ...

public static async notifyCallDisconnectedToFusion(eventId: string, disconnectionState: string):
Promise<void> {
let request: IMcaCloseCommEventActionRequest =
FusionHandler.frameworkProvider.requestHelper.createPublishRequest('closeCommEvent') as
IMcaCloseCommEventActionRequest;
request.setReason(disconnectionState);
const newCommResponse: any = FusionHandler.callToNewCommResponseMap.get(eventId);
if (newCommResponse) {
for (const property in newCommResponse) {
request.getInData().setInDataValueByAttribute(property, newCommResponse[property]);
}
}
//TODO pass unique identifier from your CTI data
request.setEventId(eventId);
request.setAppClassification(FusionHandler.appClassification);
const operationResponse: IMcaCloseComActionResponse = await FusionHandler.phoneContext.publish(request) as
IMcaCloseComActionResponse;
}
```

```
    // ...
}
```

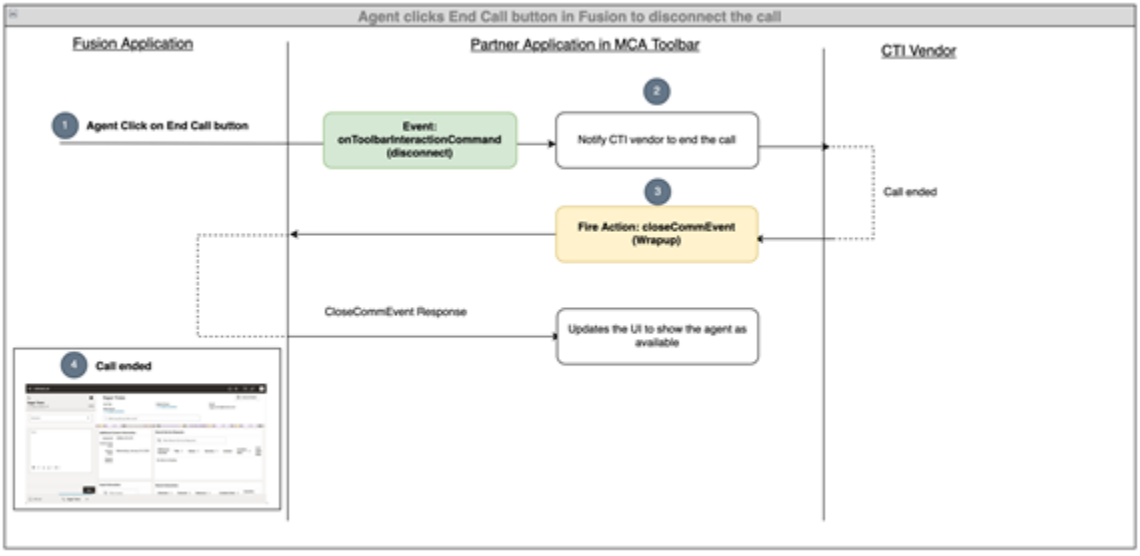
Part 2: Rejecting or disconnecting incoming calls from the Fusion application

Now, when an agent clicks on **Decline** button from the Fusion call notification, it notifies your media toolbar application and your CTI supplier that the call is disconnected by the agent. The following diagram shows the sequence of actions to be performed when an agent rejects the call from the Fusion application:



1. When the agent clicks on the **Decline** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `reject`.
2. The partner application receives this event and if the event is to reject a call, the partner application notifies the CTI supplier to reject the call.
3. Once the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with the reason as REJECT.
4. Once the Fusion application identifies this action, the call dialog box will be discarded from the UI.

The following diagram shows the sequence of actions performed when the agent disconnects an ongoing call from the Fusion application:



1. When the agent clicks on the **End Call** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `disconnect`.
2. The partner application receives this event and if the event is to disconnect a call, the partner application notifies the CTI supplier to disconnect the call.
3. Once the CTI supplier notifies the partner application that the call has been disconnected, the partner application can fire the `closeCommEvent` action.
4. Once the Fusion application identifies this action, it displays the wrap-up window in the UI.

1: Add cases for reject and disconnect in listenToToolbarInteractionCommandsFromFusion in integrationEventsHandler.ts

```
export class IntegrationEventsHandler {
 // ....
 public listenToToolbarInteractionCommandsFromFusion(command: string): void {
 switch (command) {
 case "accept":
 this.ctiAppViewModel.callAcceptedEventHandler(null);
 break;
 case "disconnect":
 this.ctiAppViewModel.callDisconnectedEventHandler({detail: {disconnectionState: "WRAPUP"}});
 break;
 case "reject":
 this.ctiAppViewModel.callDisconnectedEventHandler({detail: {disconnectionState: "REJECT"}});
 break;
 }
 }
 // ....

}
```

Verify your progress

Once you complete the above steps, use `ojet serve` to start your application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and your Fusion window. You can accept the call from your media toolbar application or your Fusion application. Once you accept the call, your media toolbar state will be changed to ACCEPTED state and the engagement will get opened in

your Fusion application. You can disconnect the call from your media toolbar application or your Fusion application. Once you disconnect the call, your media toolbar state will change to DISCONNECTED state.

Outbound calls from the media toolbar

Outbound calls

Now you set up functionality to reject an incoming call or, disconnect the accepted call.

Now let's look into how we can handle outbound calls in your CTI application. For example, when an agent clicks a Contact mobile number link from the Fusion application ON the Service Request Edit page, or the agent clicks a mobile link of a Contact from the Digital Sales Contact page. These actions trigger the `onOutgoingEvent` and the CTI application gets the notification if the CTI application has a subscription for this event. It's a non controllable event and by adding the listener CTI can wire up the logic to start a call. For example, to call the UEF `NewComm` API, based on the information in the event response of this event. Once you complete this section, you can start outbound calls from Fusion applications

Overview of the tasks

Here's an overview of what you need to do:

1. Add subscription to the `onOutgoingEvent` and add logic to fire the `newComm` event
2. Add the below method in the `integrationActionHandler`.
3. Update the `makeOutboundCall` method in the `VendorHandler` class.
4. Add the `triggerOutbound` call method in `appController`.
5. Call `subscribeToOutboundCallsFromFusion` from `IntegrationEventsHandler` in the `makeAgentAvailable` method
6. Add more methods to support the user accept and decline event.

1. Add subscription to onOutgoingEvent and add logic to fire the newComm event

In the `fusionHandler.ts` file located at: `src/ts/cti/fusion/fusionHandler.ts`, add the following methods to subscribe to `onOutgoingEvent`

```
public static async acknowledgeOutBoundCallEvent(eventResponse: IMcaonOutgoingEventResponse): Promise<void>
 {
 const request: IMcaNewCommEventActionRequest =
 FusionHandler.frameworkProvider.requestHelper.createPublishRequest('newCommEvent') as
 IMcaNewCommEventActionRequest;
 const eventId: string = eventResponse.getResponseData().getOutData().SVCMCA_CALL_ID;
 request.setEventId(eventId);
 const indata: any = {
 ...eventResponse.getResponseData().getOutData(),
 'callData': {
 "phoneLineId": "1",
 "eventId": eventId,
 },
 'SVCMCA_COMMUNICATION_DIRECTION': 'ORA_SVC_OUTBOUND',
 'SVCMCA_WRAPUP_TIMEOUT': '',
 'appClassification': ''
 // you can add additional indata here
 };
 FusionHandler.publishNewCommEvent(eventId, indata);
}

public static async subscribeToOutboundCallsFromFusion(onOutboundCall: Function): Promise<void> {
 const requestObject: IMcaEventRequest =
 FusionHandler.frameworkProvider.requestHelper.createSubscriptionRequest('onOutgoingEvent') as
 IMcaEventRequest;
```

```
 requestObject.setAppClassification(FusionHandler.appClassification);
 FusionHandler.phoneContext.subscribe(requestObject, async (eventResponse: IEventResponse) => {
 await onOutboundCall((eventResponse as IMcaonOutgoingEventResponse).getResponseData().getOutData());
 await FusionHandler.acknowledgeOutBoundCallEvent(eventResponse as IMcaonOutgoingEventResponse);
 });
}
```

2. Add the method in integrationActionHandler

```
public makeCall(phoneNumber: string, eventId: string): Promise<void> {
 return this.vendor.makeOutboundCall(phoneNumber, eventId);
}
```

3. Update the makeOutboundCall method in VendorHandler class

```
public async makeOutboundCall(phoneNumber: string, eventId: string) {
 // TODO: call the vendor specific api to make the make an outbound call
}
```

4. Add the triggerOutbound call method in appController

```
public triggerOutboundCall = (outdata: IMcaOnOutgoingEventOutData) => {
 this.callContext({
 ...this.callContext(),
 direction: 'outbound',
 phonenumber: outdata.SVCMCA_ANI,
 callerName: outdata.SVCMCA_DISPLAY_NAME,
 eventId: outdata.SVCMCA_CALL_ID,
 state: 'RINGING'
 });
 this.integrationActionHandler.makeCall(outdata.SVCMCA_ANI, outdata.SVCMCA_CALL_ID);
}
```

5. Call the subscribeToOutboundCallsFromFusion from IntegrationEventsHandler in the makeAgentAvailable method

```
public async makeAgentAvailable(): Promise<void> {
 try {
 await FusionHandler.makeAgentAvailable();
 this.ctiAppViewModel.agentState(true);
 FusionHandler.subscribeToToolbarInteractionCommandsFromFusion((command: string) =>
 { this.listenToToolbarInteractionCommandsFromFusion(command); });
 // code block starts
 FusionHandler.subscribeToOutboundCallsFromFusion((outdata: IMcaOnOutgoingEventOutData) => {
 this.ctiAppViewModel.triggerOutboundCall(outdata);
 });
 // code block ends
 } catch (err) {
 console.log("Error while making agent available", err)
 }
}
```

6. Add more methods to support the user accept, decline event

Add the following methods to the RootViewModel class:

```
public handleOutgoingCallAccepted = () => {
 this.callContext({
 ...this.callContext(),
 state: 'ACCEPTED'
```

```
 });
}

public handleOutgoingCallEnded = () => {
 this.callContext({
 ...this.callContext(),
 state: 'DISCONNECTED'
 });
}
```

Add the following methods to the IntegrationEventsHandler class:

```
public async outboundCallAcceptedHandler(eventId: string): Promise<void> {
 try {
 await FusionHandler.notifyCallAcceptedToFusion(eventId);
 this.ctiAppViewModel.handleOutgoingCallAccepted();
 } catch {
 console.log("Error: Unable to notify call accepted.")
 }
}

public async callHangupHandler(eventId: string): Promise<void> {
 try {
 await FusionHandler.notifyCallDisconnectedToFusion(eventId, "WRAPUP");
 this.ctiAppViewModel.handleOutgoingCallEnded();
 } catch {
 console.log("Error: Unable to notify Call disconnected.")
 }
}

public async callRejectedHandler(eventId: string): Promise<void> {
 try {
 await FusionHandler.notifyCallDisconnectedToFusion(eventId, "REJECT");
 this.ctiAppViewModel.handleOutgoingCallEnded();
 } catch {
 console.log("Error: Unable to notify Call disconnected.")
 }
}
```

Verify your progress

Once you complete these steps, use the OJET server to start your application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. To start an outbound call, open the contact from your Fusion application and click the phone number.

Enable outbound dialing from the media toolbar

You can enable outbound calls made from the media toolbar to trigger a screen pop on a chosen page, as specified in the screen pop configuration.

This displays relevant information during a call, ensuring agents have the necessary context and tools to provide a more personalized and responsive experiences.

Outbound calls started from the media toolbar can trigger a screen pop to a chosen page when the token `SVCMCA_DIALER_SCREEN_POP_REQUIRED` is passed in the IVR data.

Here's an example of the newCommEvent to enable screen pop actions if the IVR data passes the token:

```
const uiEventsFrameworkInstance = await CX_SVC_UI_EVENTS_FRAMEWORK.uiEventsFramework.initialize('appname',
 'v1');
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();const
 phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');let request:
```

```
 IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent') as
 IMcaNewCommEventActionRequest;
request.getInData().setInDataValueByAttribute('SVCMCA_ANI', phoneNumber);
request.setEventId('1234');
request.getInData().setInDataValueByAttribute("appClassification", "ORA_SERVICE");
request.getInData().setInDataValueByAttribute("SVCMCA_COMMUNICATION_DIRECTION", "ORA_SVC_OUTBOUND");
request.getInData().setInDataValueByAttribute('callStatus', "OUTGOING");
request.getInData().setInDataValueByAttribute('SVCMCA_DIALER_SCREEN_POP_REQUIRED', "Y"); // for enabling the
 screenpop for outbound call
request.setAppClassification("ORA_SERVICE");
const operationResponse: IMcaNewComActionResponse = await phoneContext.publish(request) as
 IMcaNewComActionResponse;
```

How to start your OJET application from a compressed file

You can either follow the steps in this playbook to build your media toolbar application or you can download the compressed file included at the beginning of each section.

1. Download the compressed file and extract it. The contents will be extracted to a folder named: cti-accelerator as shown below:
2. Generate and add the certificates as mentioned in the Start the application with a self signed certificate section of the *Create a JET application* topic.
3. Open `cti-accelerator` folder in your command prompt or terminal (in case of Mac OS) and run the command: `ojet restore` command to initialize the OJET application.
4. Once the `ojet restore` command is successful, run the `ojet serve` command to start your application

5 Build the CTI toolbar as a Visual Builder App UI

6 Manage Real Time Channel Configurations

Configure screen pop rules for Service Center

Now we'll configure screen pops to work in the Service Center console mode.

The application provides ready-to-use screen pop rules. You can choose to configure the screen pop rules based on your business requirements.

Overview of ready-to-use screen pop configuration rules for the Service Center application

The framework provides the following ready-to-use screen pop configuration rules for the Service Center application:

1. If the IVR data or the chat payload passes the SR details, such as the Service Request number for an incoming call or chat, the application screen pops to the SR details tab within the Service Center console.

   The communication panel is displayed to the side of the SR enabling easy interaction.
2. If the IVR data or the chat payload passes the contact details of a known contact, like the contact name or email for an incoming call or chat, the application screen pops to the Contact Details tab within the Service Center console.
3. If the IVR data or the chat payload passes the contact details of an unknown contact for an incoming call or chat, the application screen pops to the Create Contact form within the Service Center console.

Configure screen pop rules for the Service Center application

To configure screen pop configuration for the Service Center console, an external event `cxSvcNavigation` is exposed which can be subscribed to by the customer, and the customer can add business logic and drive the screen pop rules.

The event is defined at the application level which will be fired on every navigation that occurs in Service Center in MSI mode. The event is fired with the following payload which can help the customer make the decision:

```
{
 "extraAttributes": {
 "svcNavigateEvent": {
 "application": "",
 "flow": "",
 "tabInfo": ""
 },
 },
 "navigationType": "SCREENPOP|DRILLDOWN",
 "engagmentData": {
 "inData": "object",
 "SVCMCA_INTERACTION_ID": "string",
 "engagementType": "PHONE|CHAT",
 "engagementId": "string",
 "multiMatches": boolean
 }
```

```
}
```

- `svcNavigateEvent` is the original event
- `navigationType` can be either `SCREENPOP` or `DRILLDOWN`. For navigation when a chat or call is started it will be a `SCREENPOP`. For every other kind of navigation it will be `DRILLDOWN`.
- `engagmentData`: Chat or Call's IVR or Engagement data is based on which customers can add business logic.
- `Return Type` refers to the action chain which has the following return type. This decides whether the default actions also need to be executed or skipped, or stopped. If set to True, the action not execute the default behavior.

Prerequisites and setting up a project for screen pop

1. Sign in to Service Center as an agent with VB studio access.
2. From the Settings and Actions menu, select **Edit Page in Visual Builder**.
3. Choose an existing project or create a new one.

   To enable or disable screen pop with the agent console you change the Service application.
4. Once your project is open, from the App UIs menu, select **Oracle CX Service UI Extension App > service**.
5. In the main workspace, click the **Variables** tab.
6. From the **Constants** list, select **enableScreenPops**.

   In the Properties pane, choose **True** to enable screen pops and **False** to disable the functionality and take the viewer to the Contact page.

Create your own screen pop rules

1. To add screen pop rules you must add event listeners to the `cxSvcNavigation` object in VB Studio.
2. Select the Event Listeners tab.
3. Click the **+ Event Listener** button.
4. From the list, select `cxSvcNavigation`, and click **Next**.
5. Click the **+ Event Listener** button.

   All the events that can be listened to will be listed.

   , All the events that can be listened to will be listed, find and Choose `cxSvcNavigation` and click **Next**.
6. Choose `cxSvcNavigation`, and click **Next**.

   We'll now add an action chain to the event. We can either use the existing one or create a new one.

   > **Note:** If you're creating a new project, there won't be an existing action chain.

   After creating the action chain, we'll add it to the `cxSvcNavigation` event.
7. Find the action chain in the list and click **Go to Action Chain** to open and edit it.

   Note the list of actions you can add to the chain listed in the explorer.
8. Drag the `If` action to make sure the action chain runs only for the `SCREENPOP` navigation type.

   Once the action is added to the action chain, we can then add conditions to match our business use case. For this scenario, we'll add a condition from the properties pane of `event.navigationType === 'SCREENPOP'`.

9. Inside the `If` action we can drag another `If` action to match our use case. For the first case we'll render the case object: `if caseId is present in engagementData.`

   We're now ready to trigger the event which is going to screen pop the case object, for that we'll use `FireEventAction` by dropping it inside the inner `If` Action

   .The next step is to define `FireEvent` which you do from the Properties pane. Here are the properties required for the event:

   - **application** = `service` (root application name).
   - **containerPath** = `ec`.
   - **flow** = `case` (Object or flow which is required to open).
   - **tabInfo** = copied from `originalEvent`, it includes the `tab-title`, or `tab-icon` event.`extraAttributes.svcNavigateEvent.tabInfo`.
   - **key** = `interactionId` can be retrieved from `engagementData` `event.engagementData.inData.SVCMCA_INTERACTION_ID`.
   - **Page Params**: an object which is required by the page that needs to be rendered case page required `caseNumber`.

10. You can add other blocks to support various business use cases, but for this example, we'll add one more `If` action which will render work order `if` work order is present in the engagement data `workOrder` page requires `WoNumber`.
11. Here's an example of the resulting action chain:

```
define([
 'vb/action/actionChain',
 'vb/action/actions',
 'vb/action/actionUtils',
], (
 ActionChain,
 Actions,
 ActionUtils
) => {
 'use strict';

 class cxSvcNavigationListener extends ActionChain {

 /**
 * @param {Object} context
 * @param {Object} params
 * @param
 {{application:string,navigationType:string,flow:string,page:string,pageParams:object,engagementData:object,extr
 params.event
 * @return {{stopPropagation:boolean}}
 */
 async run(context, { event }) {
 const { $application, $base, $extension, $constants, $variables, $modules } = context;

 if (event.navigationType === 'SCREENPOP') {

 if (event.engagementData.inData.caseId) {
 await Actions.fireEvent(context, {
 event: 'oracle_cx_serviceUI/application:svcNavigate',
 payload: {
 application: 'service',
 flow: 'case',
 key: event.engagementData.inData.SVCMCA_INTERACTION_ID,
 page: 'edit',
 pageParams: {
 caseNumber: event.engagementData.inData.caseId
 },
```

```
    tabInfo: event.extraAttributes.svcNavigateEvent.tabInfo
    },
    });

    return {stopPropagation: true};
    } else if (event.engagementData.inData.workOrderNumber) {
    await Actions.fireEvent(context, {
    event: 'oracle_cx_serviceUI/application:svcNavigate',
    payload: {
    application: 'service',
    flow: 'fieldsvc',
    key: event.engagementData.inData.SVCMCA_INTERACTION_ID,
    page: 'detail',
    pageParams: {
    WoNumber: event.engagementData.inData.workOrderNumber
    },
    tabInfo: event.extraAttributes.svcNavigateEvent.tabInfo
    },
    });
    return {stopPropagation: true};
    }
    }
    }
    }

    return cxSvcNavigationListener;
   });
```

Add Screen Pop in case of a contact multi match scenario

Contact search engines provide search results if there are multiple contact matches for a chat or a phone call, which is then passed down to `engagementData.multiMatches`.

You can change the `cxSVCNavigation` listener using `engagementData.multiMatches` to navigate to the desired object. Here are steps which will enable you to redirect to the `svc-contact` page with create contact as a view

Here's same code for the action chain:

```
define([
 'vb/action/actionChain',
 'vb/action/actions',
 'vb/action/actionUtils',
], (
 ActionChain,
 Actions,
 ActionUtils
) => {
 'use strict';

 class cxSvcNavigationListener extends ActionChain {

 /**
 * @param {Object} context
 * @param {Object} params
 * @param
 {{application:string,navigationType:string,flow:string,page:string,pageParams:object,engagementData:object,extraAtt
 params.event
 * @return {{stopPropagation:boolean}}
 */
 async run(context, { event }) {
 const { $application, $base, $extension, $constants, $variables, $modules } = context;

 if (event.navigationType === 'SCREENPOP') {
```

```
if (event.engagementData.multiMatches) {
await Actions.fireEvent(context, {
event: 'oracle_cx_serviceUI/application:svcNavigate',
payload: {
application: 'service',
flow: 'sr',
key: event.engagementData.inData.SVCMCA_INTERACTION_ID,
page: 'svc-contact',
pageParams: {
...event.extraAttributes.svcNavigateEvent.pageParams,
selectedView: 'createContact'
},
tabInfo: event.extraAttributes.svcNavigateEvent.tabInfo
},
});

return { stopPropagation: true };
}

}
}
}

return cxSvcNavigationListener;
});
```

Required page parameters for an object

To screen pop a page or object we need to pass the required parameters to the page while calling the **svcNavigate** event. Each object has its input properties. Follow the steps to find out the parameters that need to be passed:

1. Open **API UI > Service Center > Service**.
2. Find the page where you want to get input variables. For example, **SR Details Service > ec > sr > edit**.
3. Once the page is loaded in VB Designer we can switch to the variables tab.
4. There can be n number of variables which can be filtered by their type.
5. Apply the filter as **Input parameter**.

Configure Screen Pop Rules for Help Desk, Fusion Sales, and CX for Utilities

You can configure screen pop pages to display pages of information that can aid an agent in starting a customer interaction efficiently. For example, you can configure a screen pop page to display information about an open ticket logged by the call-in customer.

You can create screen pop pages for ready-to-use standard business objects or user-defined objects. You can also screen pop for custom objects you've created, in Application Composer.

Here's a high level overview of how you configure screen pop pages in Setup and Maintenance:

1. Configure business objects: These business objects are associated with standard or custom objects. Standard objects include ready-to-use objects, such as Service Requests or Queues, and the custom objects are created by you.
2. Create tokens: You associate tokens with business object attributes. For example, you can create a token called **SVC_INVOICE_DATE** and associate it with the **Invoice_Date** field of the Invoices business object.
3. Map pages: Mapping associates a screen pop page with the pages of the underlying standard or custom object.

4. Create rules : Rules identify the page that's displayed when passing a token. Rules are defined in order of priority. For example, if a service request number is available, display the service request page. If no service request number is available, but there's a contact identified, display the Edit Contact page. If no service request or contact information is available, display the Create Contact page.

Manage Screen Pop Configuration

Business object configuration enables you to specify the business objects that can be used for reverse lookup and screen pop.

1. Sign in to your Fusion application as an administrator.
2. In the Setup and Maintenance work area, go to the following:
    - Offering: Service.
    - Functional Area: Communication Channels (select All Tasks from the Show drop-down list)
    - Task: Manage Screen Pop Configuration
3. On the Screen Pop Page Configuration page, click the **Business Objects** tab.
4. Click the **Systems Objects** tab.
5. View the objects in the **Mapped System Objects** list.
6. Click **Save**.

Create Business Objects

Configuring business objects enables you to specify those business objects that can be used for reverse lookup and screen pop. You can configure a standard business object or a user business object. A standard object is based on a ready-to-use business object and a user business object is based on a user-defined business object that's created using the Application Composer.

Business objects are associated with standard or user-defined objects. Standard objects include ready-to-use objects, such as Service Requests or Queues, and the user-defined objects are created by the user.

A standard business object is based on a preconfigured business object, such as Service or Queue. The standard or the predefined objects can't be changed or deleted.

Here's how you create a business object:

1. On the Screen Pop Page Configuration page, click the **Business Object**s tab.
2. Click the **User Business Objects** tab.
3. Click **Add**.
4. Select object type as **User-Defined**.
5. Select the application in which you created an object. For example, select Service application.
6. Type the name of the user-defined object that you created and click **Validate**. For example, type SRTickets.
   If the object name is valid, the object's full path is displayed.
7. Click **Add**, then click **Save** or **Save and Close**.

Create your own token

Tokens are associated with the attributes of a business object. This tasks assumes you're using a provided business object or a custom object you've created.

1. On the Screen Pop Page Configuration page, click the **Tokens** tab.

2. Click the System Token tab and view the predefined tokens. You can't change or delete these tokens.
3. Now, to create a new token, click the **User-Defined Tokens** tab.
4. Click the Add (**+**) icon.
5. Use the following list to specify your values for the new token:
    - **Name**: Enter a name for the token.
    - **Token Code**: A unique code that's used to represent a token.
    - **Description**: Provide any extra information about the token.
    - **Object Name**: Optional. The business object to which the token is associated.
    - **Object Attribute**: Optional. The attribute of the business object that's associated with the token.
6. Click **Save**.

Map a page

Mapping associates a screen pop page with the pages of the underlying standard or custom object.

1. On the Screen Pop Page Configuration page, click the **Pages** tab.
2. Deselect **Show only used**.
3. You can view the list of pages available for mapping. Standard pages can't be changed or deleted.
4. To create a new mapping, click the Add (**+**) icon.
5. Click the **User Interface Type** drop-down list and select **Redwood**, then do the following:
    a. **Business Object:** Click the drop-down list and select the business object you want to map to.
    b. **Page Name**: Enter a name for your page.
    c. Page Title: Enter a title for the screen pop page. A title can contain a title prefix, the name of the token, and a title suffix. One of these values is required.
    d. Page Path: If you've selected a user-defined type of mapping, the page path is displayed automatically when you select the page. However, for a standard mapping, you must specify the page path. After specifying a page path, click **Inspect** to validate the page path and to list the page parameters.
6. In the **Page Parameters Mapping** section, associate required parameters with a token or a user-defined value. Based on the information passed to one or more page parameters, the result is displayed in the page. For example, based on the invoice number parameter, the invoice details screen is displayed.
7. Click **Save**.

Overview of creating an defining rules

Rules identify the page that's displayed when passing a token. Rules are defined in order of priority. For example, if a service request number is available, display the service request page. If no service request number is available, but there's a contact identified, display the Edit Contact page. If no service request or contact information is available, display the Create Contact page.

Screen pop configuration rules identify which set of rules must be applied when a screen pop logic is called. Different screen pop rules can be called based on several different variables, such as application classification and channel. Based on the input parameters, you can choose from several different pages to screen pop to the agent. For example, pages such as the Contact Edit, Account Edit, and Service Request edit pages can be displayed to the customer. Also, you can choose to create an object, such as a service request. This framework also allows for user-defined objects to be presented to the agent as part of the screen pop process.

Define rules in order of priority to display a screen pop page, when an associated token value is available. If a rule in a higher priority isn't satisfied, the next in the order is checked.

Note: There can be only one active Rule set for a combination of application classification and UI. If there are multiple active rule sets with the same combination of application classification and UI, you'll receive an error.

View standard ready-to-use rules

1. On the Screen Pop Page Configuration page, Click the Rules tab.
2. View list of all ready-to-use Rule set for Redwood and ADF interface
3. To activate ready-to-use Rule set, check the checkbox 'Active' for the required Rule specific to App classification and UI
4. View the list of ready-to-use rules of each rule set. By clicking on Rule set a table will be displayed below with details like channel, priority, token, and page to pop

Define new rules

1. On the Screen Pop Page Configuration page, click the **Rules** tab.

   We'll now create a rule set. A rule set consists of one or more rules that are defined in an order of priority.
2. Create a rule set: by doing the following:
   a. Click the Add (**+**) icon.

      You can also select **Duplicate** from the **Actions** menu to duplicate an existing rule set.
   b. Enter a name for the rule set.
   c. Select an application classification to which the rule set belongs.
   d. Click the **User Interface Type** drop-down list, and select **Redwood** depending on your implementation.
   e. A rule set is Active by default. To deactivate a rule set, clear the Active option.
   f. Enter a description for the rule set.
3. Next, add rules to the rule set by doing the following:
   a. Click the Add (**+**) icon.

      The priority column displays the order of priority in which the rules are checked. You can change the priority by clicking up arrow and down arrow icons.
   b. A rule is enabled by default. Clear the **Enable** option to disable a rule.
   c. Select the communication channel to which the rule is applied.

      For example, a rule is applied only when an agent receives a service phone call or when there's a chat alert from a customer. You can add or change the channels list by changing the associated lookup values.
   d. Select a token name.

      You can select ready-to-use token or select the created token as specified in the previous section.
   e. Select a page to display, when a token value is available.

      You can select ready-to-use page or select the created page as specified in the previous section.
   f. You can change the priority of a rule by clicking on the up or down arrow.
   g. Click **Save**.
4. When you're finished creating the rule sets, click **Done**.

Configure the Reverse Lookup logic using Lookup Filters

Customer details are extracted from the database based on the availability of the token value, starting from the token with the highest priority. When customers make service calls, they provide information such as Customer ID, Email address, Contact name, Phone number, Date of Birth, First Name, and so on. Based on this information, customer details are extracted from the database. Use the Lookup Filters page to configure reverse lookups and lookup filters for channel-driven interactions with customers. Order the filters by the priority in which the tokens are evaluated.

Let's understand the reverse lookup logic with an example where there are two customer contacts with the same name but different email addresses:

- Customer Contact #1: Adam Parker - adam.parker@testing.com
- Customer Contact #2: Adam Parker - adam2.parker2@testing.com

In the first scenario, the administrator has set **`Contact Name`** as a higher priority in the order of the lookup filters than **`Email`**.

Both customer contacts have the same name but a different email address. Let's say Customer Contact #1 starts a chat where the First Name is Adam, and the Last Name is Parker, and the Email address is adam.parker@testing.com. Because **`Contact Name`** has higher priority, the reverse lookup logic will extract customer details from the database based on the availability of the token value. In this case, the database will extract two records with matching names. The next priority is **`Email`**, so the database will further filter out and extract Customer Contact #1 details. Here the lookup filtering was done twice.

In the next scenario, the administrator has set **`Email`** as a higher priority in the order of the lookup filters than**`Contact Name`**.

Let's say Customer Contact #1 starts a chat with the First Name is Adam, and the Last Name is Parker, and the Email address is adam.parker@testing.com. Because **`Email`** has a higher priority, the reverse lookup logic will extract customer details from the database based on the availability of the token value. In this case, the database will extract one record with the matching email. Here the lookup filtering was done once.

Lookup sets are grouped by application classification. They contain one or more lookup filters defined in an order of priority, and each lookup filter is associated with a token. To create a new lookup filter, you can duplicate an existing one and make changes.

Here's how you configure lookup filters:

1. Sign in to your Fusion application as an administrator.
2. In the Setup and Maintenance work area, go to the following:
    - Offering: Service.
    - Functional Area: Communication channels. Select **All Tasks** from the **Show** drop-down list.
    - Task: Manage Screen Pop Configuration
3. To view the ready-to-use lookup filters, click the **Lookup Filters** tab.

    You can view all active lookup sets for the different application classifications. These ready-to-use look up filters can't be changed or deleted.
4. Set the ready-to-use Lookup Set to **Active** or **Inactive** by using the checkbox for the selected record.

5. View the list of ready-to-use lookup rules of each lookup set by clicking Lookup Set to view a table which displays details such as channel, priority, token, and object name.
6. Create a new lookup filter by first clicking the **Lookup Filters** tab on the Screen Pop Page Configuration page.
7. Click the Add (**+**) icon to create a lookup set, then do the following:
    a. Enter a name.
    b. Select an application classification from the drop-down list.
    c. Select the **Active** option to activate a lookup set.
    d. Enter a description.
8. To add filters to the lookup set, click the Add (**+**) icon then do the following:
    a. Enter a name.
    b. Select the channel from the **Channels** drop-down list. The filter is applied to the selected channel.
    c. Select a token from the drop-down list. When you select a token, the associated object name is displayed in the **Object Name** column.
    d. Add more filters if required.

    You can reorder the filters by clicking the up and down arrow icons. You can also click the **Actions** drop-down list or select the Move Up or Move Down option.
9. Click **Save** or **Save and Close**.

> **Note:** There can be only one active lookup set for an application classification. If there are multiple active lookup sets with the same application classification, you'll receive an error.

> **Note:** There can be only one active lookup set for an application classification. You'll receive an error if there are multiple active lookup sets with the same application classification.

7 Extend Real Time Channel UIs

Pre populate the Contact form fields from the chat launch form or an IVR

In this section, we'll discuss pre populating the Create Contact Form fields from the chat launch form or IVR.

When an agent receives a chat or call from an unknown contact, on accepting the chat, the agent can see a Create Contact form loaded with First Name, Last Name, Email, along with other fields or custom fields. Once the form is loaded, instead of manually filling in the details, the agent can automate this process by updating the fields in the new contact form with the data from the DCS information the customer enters during the initialization of the chat.

Here's how you pre populate the Create Contact form using Engagement data:

1. Open the Oracle Cloud Application page containing the component you want to extend.
2. Click the Settings and Actions menu and select **Edit Page in Visual Builder** to open the page in the VB Studio Page Designer.
3. Create a project or open an existing project.
4. Add required dependencies from the **Dependencies** sub tab.
5. Click the **Layouts** sub tab, then select **Contacts** from the list.
6. From the Dynamic Forms list, select **Create Contact**.

   You can either duplicate existing default layouts or create your own layout.
7. Duplicate ready-to-use rules by doing the following:
   a. In the Display Logic area, click the **Duplicate Rule** icon to clone the default rule.
   b. Duplicate the existing default rules UI.
   c. Enter a name for the duplicate rule then click the **Duplicate** button to create a copy of rule and layout.

   > **Note:** Default layouts from dependencies are read-only, you can't edit them
8. In the newly copied layout, select **Click to add condition**.

   You can add multiple conditions, or you can add multiple rules based on your requirements.
9. Create a new rule by doing the following:
   a. In the Display Logic area, click the **+ Rule** button.
   b. Add new rules and conditions.
   c. Add new fields or use existing form fields.
   d. Create a custom field, click **+ Custom Field**.
   e. In the Properties section, update the value of any custom fields you created.
   f. Select the variable you want to see as this field, or use the function to work custom code. For example, if you wanted the `firstName` field to be populated using engagement data, your function would look like this:

```
firstNameUnknown = [[ $componentContext.outData.firstName ]]

emailUnknown = [[ $componentContext.outData.SVCMCA_EMAIL ]]

lastNameUnknown = [[ $componentContext.outData.lastName ]]
```

```
phoneUnknown = [[ $componentContext.outData.SVCMCA_ANI ]]

queueUnknown = [[ $componentContext.outData.SVCMCA_QUEUE_NAME ]]
```

10. Now click the **Rule Sets** tab, and add the field to your layout.
11. Click **Preview** to view the updates, or **Publish** to make them available.

Extend the Phone Call or Chat Header

There are two ready-to-use layouts to display customer information to agents.

There's a layout for known customers and a layout for unknown customers. You can extend the header UI in your application to add or remove standard or custom fields to the layout, or mark a standard or custom field as required. You can also display fields conditionally, and make them required conditionally. You can configure a conditional Call or Chat header UI layout based on attributes such as Application Classification, Channel, and Communication Direction.

By default, the Chat header for a known contact has the following details:

- Email address of end-user
- Mobile phone number of end-user
- Queue name of chat
- Account name

By default, the Phone Call header of known contact has the following details:

- Work phone number
- Email of end-user
- Mobile phone number
- Account name

By default, the Chat header for an unknown contact has the following details:

- Email address of end-user
- Queue name of the chat

By default, the Call header for an unknown contact has the following details:

- Phone

Use the following these steps to extend the Call or Chat header UI:

Navigate to the Chat or Call header layout

1. Sign in to the Oracle Cloud Application page containing the component you want to extend.
2. Click the Settings and Actions menu and select Edit Page in Visual Builder to open the page in the VB Studio Page Designer.
3. Create a project or open an existing project.
4. Add required dependencies in the dependencies subtab.
5. Click the Layouts subtab.

6. Select contacts from the Layouts list.

Overview of Extending the Chat or Call header

1. To Extend the Chat header in VB Studio, go to **Layouts > Contacts**, and select the **Chat Header** dynamic form.
2. To Extend Call header go to **Layouts > Contacts** then select the **Cti Header** dynamic form.

Duplicate the default rule or create a new rule and define conditions

1. In the VB Studio Display Logic area, click the **Duplicate Rule** icon to clone the default rule.
2. Duplicate existing default rules UI.
3. Enter a name for the duplicate rule then click the **Duplicate** button to create a copy of the rule and layout.

   > **Note:** Default layouts from dependencies are read-only, you can't edit them.

4. In the newly copied layout, select **Click to add condition** to add conditions for when to use this rule and layout.
5. Then select **Click to add conditions UI**.

   You can add one of multiple conditions depending on your requirements. You can base them on the following:
   - Context
   - Fields
   - Responsive
   - User
6. Create you own rules by clicking the **+ Rule** button to add your own rules.

Add standard fields to the Chat or Call header

1. Select your new layout.

   All available fields are displayed.
2. Select fields in the **All Fields list**.
3. Reorder the list by selecting then dropping them where you want them.

Add custom fields to the Chat or Call header

1. Create a custom field from Application Composer for the **Contacts** object and publish it.
2. In Visual Builder, open the **Chat/Call** header.
3. From the **All Fields** list, find your custom field and select it to add it to the **Select field to display** list.

Add data from the chat launch form or IVR to the Chat or Call header

The data entered in chat launch form or collected in IVR will be available in engagement `indata/outdata` of the `processevent` API which can be accessed in this form using:`componentContext.inData` or `componentContext.outData`. For example: to see interaction ID in the call or chat header, use: `[[ $componentContext.outData.SVCMCA_INTERACTION_ID ]]` as the Value.

1. In the **Contacts** layouts, click the **Fields** tab.
2. Click the **+ Custom field**.

3. Enter an ID and label, and then click **Create**.
4. Select the newly created field from the **Fields** list.
5. In the Properties area, click the **Value** field.

   Say you want to have the contact's job title to be seen. For example:
   `[[ $componentContext.inData.SVCMCA_CONTACT_JOB_TITLE ]].`
6. Select the variable that you want to see as this field or use `function` to write custom code
7. Now click the **Rule Sets** tab and add this field to your layout.
8. Preview or publish to see the newly created field and tagged values.

Delete a field from the Chat or Call header

1. In the Layout workspace, note the list of fields.
2. To delete a field, hover over the field, and click the **Delete** button.

Conditionally display a field in the Chat or Call header

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, select **Show Field** and select **Always**.
5. Update the condition, then save to view the updated Show Field condition.

Mark a field required in the Chat or Call header

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, select **Required** checkbox.

Make fields conditionally required in the Chat or Call header

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, hover over the Expression Editor icon `fx`.
5. Enter a condition in editor, then enter the required key and value.

   Here are some examples:
   - `[[ $componentContext.callDirection == 'ORA_SVC_OUTBOUND' ]]`
   - `[[ $user.userName == 'ctiautouisvr3' ]]`

After making changes you can publish the changes and start using the updated Chat or Call header.

Extend the Contact Verification card

The Contact Verification card is displayed to agents only for known customers.

The UI includes a **Verify** button for agents to confirm whether the customer information is correct. With Contact Verification extensibility, we can extend the Contact Verification UI in our application to add or remove standard or custom fields to the layout. You can also mark a standard or custom field required. You can display fields conditionally and make them required conditionally. You can configure a conditional Contact Verification UI layout based on attributes such as Application Classification, Channel, and Communication Direction.

Note: You can't remove the Notes field from the Wrap Up layout.

Navigate to the Wrap Up layout

1. In Visual Builder, create a new project or open an existing project.
2. Click the **Dependencies** tab, and add the required dependencies.
3. Click the **Layouts** tab.
4. From the **Dependencies** list, click **WrapUps**.
5. In the Dynamic Form list, select the **wrapup** dynamic layout.
6. In the Rule Sets properties area, click the **Parameters** tab.
7. The Wrapup component has the following three contexts:
    - appClassification
    - callDirection
    - channelType
8. Hover over each context to highlight it, then hover over the **?** symbol to view possible values for each component context.
9. Possible values are:
    - `appClassification`: Possible Values: ORA_SALES, ORA_SERVICE, ORA_HRHD
    - `callDirection`: Possible Values: ORA_SVC_INBOUND, ORA_SVC_OUTBOUND
    - `channelType`: Possible Values: ORA_SVC_PHONE, ORA_SVC_CHAT

Navigate to the Wrap Up layout

1. In the **Display Logic** workspace, click the Duplicate Rule icon to clone the default rule.
2. Click the **Layouts** tab.
3. From the **Dependencies** list, click **WrapUps**.
4. In the Dynamic Form list, select the **wrapup** dynamic layout.
5. In the Rule Sets properties area, click the **Parameters** tab.
6. The Wrap up component has the following three contexts:
    - appClassification
    - callDirection
    - channelType

7. Hover over each context to highlight it, then hover over the **?** symbol to view possible values for each component context.
8. Possible values are:
    - `appClassification`: Possible Values: ORA_SALES, ORA_SERVICE, ORA_HRHD
    - `callDirection`: Possible Values: ORA_SVC_INBOUND, ORA_SVC_OUTBOUND
    - `channelType`: Possible Values: ORA_SVC_PHONE, ORA_SVC_CHAT

Add standard fields to the Contact Verification card

1. Click your newly created layout.
2. From the All Fields list, view the ready-go-use fields.
3. You can add new fields to the list by selecting the checkbox from the list.

Add custom fields to the Contact Verification card

1. Create a custom field from Application Composer for the **Contacts** object and publish it.
2. In Visual Builder, open the **Contact Verification** header.
3. From the **All Fields** list, find your custom field and select it to add it to the **Select field to display** list.

Add data from the chat launch form or IVR to the Contact Verification card

The data entered in chat launch form or collected in IVR will be available in engagement `indata/outdata` of the `processevent` API which can be accessed in this form using:`componentContext.inData` or `componentContext.outData`. For example: to see interaction ID in the call or chat header, use: `[[ $componentContext.outData.SVCMCA_INTERACTION_ID ]]` as the Value.

1. In the **Contacts** layouts, click the **Fields** tab.
2. Click the **+ Custom field**.
3. Enter an ID and label, and then click **Create**.
4. Select the newly created field from the **Fields** list.
5. In the Properties area, click the **Value** field.

   Say you want to have the contact's job title to be seen. For example:
   `[[ $componentContext.inData.SVCMCA_CONTACT_JOB_TITLE ]].`
6. Select the variable that you want to see as this field or use `function` to write custom code
7. Now click the **Rule Sets** tab and add this field to your layout.
8. Preview or publish to see the newly created field and tagged values.

Delete a field from the Contact Verification card

1. In the Layout workspace, note the list of fields.
2. To delete a field, hover over the field, and click the **Delete** button.

Conditionally display a field in the Contact Verification card

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list

3. Click the added field, and expand the properties panel
4. In the properties panel, select **Show Field** and select **Always**.
5. Update the condition, then save to view the updated Show Field condition.

Mark a field required in the Contact Verification card

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, select **Required** checkbox.

Make fields conditionally required in the Contact Verification card

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, hover over the Expression Editor icon (**fx**)
5. Enter a condition in editor, then enter the required key and value.

   Here are some examples:
   - `[[ $componentContext.callDirection == 'ORA_SVC_OUTBOUND' ]]`
   - `[[ $user.userName == 'ctiautouisvr3' ]]`

After making changes you can publish the changes and start using the updated Contact Verification card.

Extend the Wrap Up UI

The Wrap Up UI is displayed to agents only for known customers.

The UI includes a **Verify** button for agents to confirm whether the customer information is correct. With Wrap Up extensibility, we can extend the Wrap Up UI in our application to add or remove standard or custom fields to the layout. You can also mark a standard or custom field required. You can display fields conditionally and make them required conditionally. You can configure a conditional Wrap Up UI layout based on attributes such as Application Classification, Channel, and Communication Direction.

> **Note:** The Wrap Up UI is displayed only for known customers.

Duplicate the default rule and define conditions

This example will use the Wrap Up form.

1. In Visual Builder, create a new project or open an existing project.
2. Click the **Dependencies** tab, and add the required dependencies.
3. Click the **Layouts** tab.
4. Click **Contacts** in the explorer.

5. In the Dynamic Form list, click **Contact Verify**.
6. In the Display Logic area, click the **Duplicate Rule** icon to clone the default rule.
7. Enter a name for the duplicate rule then click the Duplicate button to create a copy of the rule and layout.

   > **Note:** Default layouts from dependencies are read-only, you can't edit them.

8. In the newly copied layout, select **Click to add condition** to add any conditions you need for use of this rule and layout basing your conditions on the following:
   - Context
   - Fields
   - Responsive
   - User

   You can give multiple conditions, or you can add multiple rules based on your requirements.

Add standard fields to the Wrap Up UI

1. Click your newly created layout.
2. From the All Fields list, view the ready-go-use fields.
3. You can add new fields to the list by selecting the checkbox from the list.

Add custom fields to the Wrap Up UI

1. Create a custom field from Application Composer for the **Contacts** object and publish it.
2. In Visual Builder, open the **Wrap Up** header.
3. From the **All Fields** list, find your custom field and select it to add it to the **Select field to display** list.

Add data from the chat launch form or IVR to the Wrap Up UI

The data entered in chat launch form or collected in IVR will be available in engagement `indata/outdata` of the `processevent` API which can be accessed in this form using:`componentContext.inData` or `componentContext.outData`. For example: to see interaction ID in the call or chat header, use: `[[ $componentContext.outData.SVCMCA_INTERACTION_ID ]]` as the Value.

1. In the **Contacts** layouts, click the **Fields** tab.
2. Click the **+ Custom field**.
3. Enter an ID and label, and then click **Create**.
4. Select the newly created field from the **Fields** list.
5. In the Properties area, click the **Value** field.

   Say you want to have the contact's job title to be seen. For example:
   `[[ $componentContext.inData.SVCMCA_CONTACT_JOB_TITLE ]].`
6. Select the variable that you want to see as this field or use `fx(function)` to write custom code
7. Now click the **Rule Sets** tab and add this field to your layout.
8. Preview or publish to see the newly created field and tagged values.

Delete a field from the Wrap Up UI

1. In the Layout workspace, note the list of fields.

2. To delete a field, hover over the field, and click the **Delete** button.

Conditionally display a field in the Wrap Up UI

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, select **Show Field** and select **Always**.
5. Update the condition, then save to view the updated Show Field condition.

Mark a field required in the Wrap Up UI

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, select **Required** checkbox.

Make fields conditionally required in the Wrap Up UI

1. In the Layout workspace, note the list of fields.
2. Select the checkbox of the desired field from the **All Fields** list
3. Click the added field, and expand the properties panel
4. In the properties panel, hover over the Expression Editor icon (**fx**)
5. Enter a condition in editor, then enter the required key and value.

   Here are some examples:
   - `[[ $componentContext.callDirection == 'ORA_SVC_OUTBOUND' ]]`
   - `[[ $user.userName == 'ctiautouisvr3' ]]`

After making changes you can publish the changes and start using the updated Wrap Up UI.

Pre populate SR fields from chats or phone calls

Here's a list of field that come pre populated in SR forms:

Pre populated fields

Field	Value Source
Title	SVC_MCA_INTRACTION_DESCRIPTION
Problem description	Summary
Product	`engagementData` > Product ID

Field	Value Source
Category	`engagementData` > Category ID
Channel Type Code	INTERACTION_TYPE

How to pre populate SR field from a phone call or chat

1. In VB Studio, choose `svc-contact` from App UIs list, then **Service Center > Service > ec > sr > svc-contact**.
2. Click the Event Listeners tab, and then click **+ Event Listener**.
3. From the list of events, choose **OpenCreateServiceRequest**, and click Next.

   This event will be fired when the agent initiates the create SR action.
4. From the Action Chains list, select **Get Engagement Data** to retrieve the engagement data.
5. Pass `$base.variables.uefContext` to first argument `(uefContext)`.

   Now we need to set the SR field values from the engagement data we retrieved from the previous step,
6. Use a global function with the label `Set Fields Value` from the CX Service list.

   The method expects following arguments. from the Fields And Value List, we'll construct an Array like the following:

```
[
 {
 fieldName: 'ServiceRequest.Title',
 value: callGetEngagementDataByUefPageContext.inData.my_text_customfield,
 },
]

uefContext: we will pass the drawers uef context which is available as
 $base.variables.drawerObjectUefContext
```

   And here's a sample action chain from this implementation:

```
define([
 'vb/action/actionChain',
 'vb/action/actions',
 'vb/action/actionUtils',
], (
 ActionChain,
 Actions,
 ActionUtils
) => {
 'use strict';

 class openCreateServiceRequestListener extends ActionChain {

 /**
 * @param {Object} context
 * @return {{stopPropagation:boolean}}
 */
 async run(context) {
 const { $page, $flow, $application, $base, $extension, $constants, $variables, $modules } = context;

 const callGetEngagementDataByUefPageContext = await
 $modules.mcaUtils.getEngagementDataByUefPageContext($base.variables.uefContext);
```

```
const callSetFieldsValue = $modules.uiEventsFramework.setFieldsValue([{
fieldName: 'ServiceRequest.Title',
value: callGetEngagementDataByUefPageContext.inData.my_text_customfield,
}], $base.variables.drawerObjectUefContext, false);

return { stopPropagation: false };
}
}

return openCreateServiceRequestListener;
});
```

Extend Screen Pop

When a chat or call panel is accepted it appears over an agent's screen giving the agent more information and opportunities to help their customer while in chat or call. You can extend this screen pop dialog to fit your business needs.

Here are the default rules and behaviors:

Rules

Rule priority	Rules	Screen pop	Behavior
1	Chat, call with SR information	SR Details page	Based on the service center settings - show SR details or SR foldout view
2	Unknown contact with SR details	SR Details page	Based on the service center settings - show SR details or SR foldout view
3	Unknown contact without SR details	SVC-Contact Createpage	Option to search for another contact Create contact form Fields should be auto populated
4	Known contact without SR	SVC-Contact page	Just contact details with Chat / call panel

When a chat or call panel is accepted it appears over an agent's screen giving the agent more information and opportunities to help their customer while in chat or call. You can extend this screen pop dialog to fit your business needs.

The event is defined at Application level and is fired when a navigation within Service occurs in MSI mode.

The event is fired with following payload which can help with customer decision making:

```
{
 "extraAttributes": {
 "svcNavigateEvent": {
 "application": "",
 "flow": "",
 "tabInfo": ""
 },
 },
 "navigationType": "SCREENPOP|DRILLDOWN",
 "engagmentData": {
 "inData": "object",
 "SVCMCA_INTERACTION_ID": "string",
 "engagementType": "PHONE|CHAT",
 "engagementId": "string",
 "multiMatches": boolean
 }
}
```

- `svcNavigateEvent`: The original event
- `navigationType`: Can be `SCREENPOP` or `DRILLDOWN`, for navigation when a chat or call is started. It will be `SCREENPOP`, and for every other kind of navigation it will be `DRILLDOWN`.
- `engagmentData`: A chat or call's IVR or Engagement data based on business logic you add.

The action chain contains the following return type which decides whether default actions also need to be executed or skipped. To enable this behavior, set the `stopPropagation` property to True.

```
{
 "stopPropagation": true,
}
```

Note: If `stopPropagation` is set to False, the action will execute the customer Listener first and then execute the default behavior.

Required page parameters for an object

To screen pop a page or object we need to pass required parameters to the page while calling the `svcNavigate` event, and each object as its own input properties must follow the steps to find the parameters which must be passed.

1. Open **API UI > Oracle CX Service UI Extension App > Service**.
2. In VB Studio find the page you want to get input variables. For example: **SR Details Service > ec > sr > edit**.
3. When the page is loaded in VB Studio Designer, click the **Variables** tab.

   There can be number of variables which you can filter by type
4. Apply the filter as **Input parameter**.

How to add your own screen pop

Here are three screen pop use cases:

- Navigating to a Case page where `caseNumber` is provided in the pre launch form.
- Navigating to a Work Order page where the `WorkOrder` number is provided in the pre launch form

- Navigating to a `svc-contact` in case of `multiMatches` (If a contact search returns `multiMatches`.)

To add Screen Pop custom rules we need to add event listeners to `cxSvcNavigation`.

1. In VB Studio, select APP UIs, then, choose **Oracle CX Service UI Extension App > service**.
2. Click the **Event Listeners** tab.
3. Click **+ Event Listener**.
   All events that can be listened to are shown.
4. Select **cxSvcNavigation** and click **Next**.
5. Add an action chain, or use an existing one.
6. After you create the action chain, it will be added to the **cxSvcNavigation** event
7. Drag an **if** Action which means the action chain runs only for the `SCREENPOP` navigation type.
8. Now you can add conditions to match your business use case. For this example, we'll add the `event.navigationType = 'SCREENPOP'.`
9. Inside the **if** action we can now drag another **if** action to render the case object if the `caseId` present in the `engagementData.`
   We're now ready to trigger the event which is going to screen pop the case object.
10. Drop the **FireEventAction** inside the inner **if** action.
11. Define the `FireEvent` from Properties Panel. Here are the required properties:
    - `application`: service (root application name).
    - `containerPath`: ec
    - `flow`: case (Object or flow which is required to open)
    - `tabInfo`: Copies from originalEvent. It includes tab-title, tab-icon event.extraAttributes.svcNavigateEvent.tabInfo
    - `key`: `interactionId` can be retrieved from `engagementData` `event.engagementData.inData.SVCMCA_INTERACTION_ID.`
    - `Page Params`: An object which is required by the page which needs to be rendered case page required `caseNumber.`
12. Add a return block to avoid executing the default behavior. For this example we'll add one more **if** Action which will render `workorder` if `workorder` is present in the engagement data and required the work order page to require the work order number.
    Here's the example action chain:

```
define([
 'vb/action/actionChain',
 'vb/action/actions',
 'vb/action/actionUtils',
], (
 ActionChain,
 Actions,
 ActionUtils
) => {
 'use strict';

 class cxSvcNavigationListener extends ActionChain {

 /**
 * @param {Object} context
 * @param {Object} params
 * @param
 {{application:string,navigationType:string,flow:string,page:string,pageParams:object,engagementData:object,extr
 params.event
```

```
 * @return {{stopPropagation:boolean}}
 */
 async run(context, { event }) {
 const { $application, $base, $extension, $constants, $variables, $modules } = context;

 if (event.navigationType === 'SCREENPOP') {

 if (event.engagementData.inData.caseId) {
 await Actions.fireEvent(context, {
 event: 'oracle_cx_serviceUI/application:svcNavigate',
 payload: {
 application: 'service',
 flow: 'case',
 key: event.engagementData.inData.SVCMCA_INTERACTION_ID,
 page: 'edit',
 pageParams: {
 caseNumber: event.engagementData.inData.caseId
 },
 tabInfo: event.extraAttributes.svcNavigateEvent.tabInfo
 },
 });

 return {stopPropagation: true};
 } else if (event.engagementData.inData.workOrderNumber) {
 await Actions.fireEvent(context, {
 event: 'oracle_cx_serviceUI/application:svcNavigate',
 payload: {
 application: 'service',
 flow: 'fieldsvc',
 key: event.engagementData.inData.SVCMCA_INTERACTION_ID,
 page: 'detail',
 pageParams: {
 WoNumber: event.engagementData.inData.workOrderNumber
 },
 tabInfo: event.extraAttributes.svcNavigateEvent.tabInfo
 },
 });
 return {stopPropagation: true};
 }
 }
 }
 }

 return cxSvcNavigationListener;
});
```

8 Accelerators

Overview of using accelerators with CTI

You can use different CTI providers in your media toolbar application.

The following examples show sample integrations with Twilio, Genesys and Amazon Connect. We'll be using the concepts described in Develop your own media toolbar application for integrating the CTI providers. If you haven't reviewed Develop your own media toolbar application, check it out to understand concepts we'll now review.

Add all supplier specific code to the `vendorHandler.ts` file in your media toolbar application.

> **Note:** The source code of the accelerators is written as a sample reference with the intention of allowing third-party partners or customers to build their own custom integrations. These samples aren't certified or supported by Oracle and are intended for educational or testing purposes only. Use of these samples implies acceptance of the Oracle Technology Network Developer License Terms.

Twilio

Overview of integrating with Twilio

Twilio is a CTI supplier that provides programmable communication tools for making and receiving phone calls, sending and receiving text messages, and other communication functions through its web service APIs.

In this section we'll walk through the integration with Twilio with the application we built.

Note that this integration isn't a ready-to-use integration. The steps which follow will show a sample implementation based on few concepts described in *Configure the Media Toolbar*. This implementation isn't scaled for production use cases and can't be considered as a production ready implementation.

Before moving into the implementation, you need to have a Twilio account. If you don't have a Twilio account, you can create a trial account from *Twilio's website*.

We'll be using the Twilio's JavaScript SDK to handle calls from your browser. Read the information on Twilio's JavaScript SDK on their website.

> **Note:** As a prerequisite, you'll need to set up a Twilio account.

Getting a Twilio phone number

After you've created your Twilio account, you set up a *phone number* using the Twilio console. This phone number can be used by your customers to call your service center.

Setup Twilio Voice SDK Quick start application

See details on the Twilio SDK, see *Voice JavaScript SDK*.

You'll then do the following:

- Download and deploy the Quick start application. Follow *these* steps.
- Then, you'll create a a TwiML application from your Twilio console.

Add the Twilio dependency and load the script file in the Jet application

Here's how you add a Twilio dependency to your JET application.

Install node package manager (npm) to Voice SDK as a dependency

Run the following command to do the install:

```
@twilio/voice-sdk

npm install @twilio/voice-sdk --save
```

Load the voice-sdk.js file in index.html

1. Add the following line of code to this file: `src/js/path_mapping.json`.

```
"@twilio/voice-sdk": {
 "cdn": "3rdparty",
 "cwd": "node_modules/@twilio/voice-sdk/dist/",
 "debug": {
 "src": "twilio.min.js",
 "path": "libs/twilio/voice-sdk/twilio.min.js",
 "cdnPath": "twilio/voice-sdk/twilio.min.js"
 },
 "release": {
 "src": "twilio.min.js",
 "path": "libs/twilio/voice-sdk/twilio.min.js",
 "cdnPath": "twilio/voice-sdk/twilio.min.js"
 }
 }
```

2. Add the script tag to load the `twilio.min.js` file to index.html:

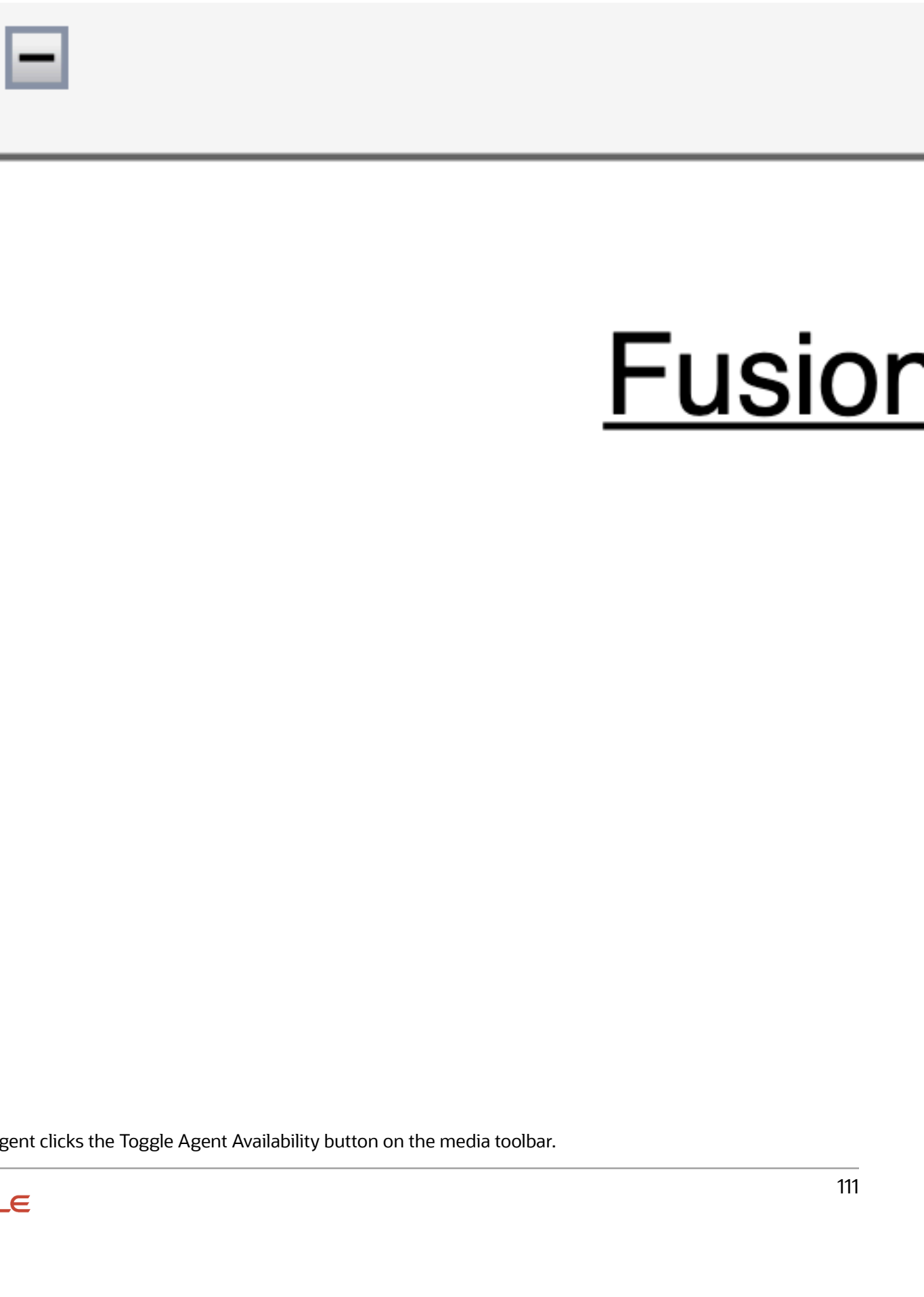
```
<script type="text/javascript" src="js/libs/twilio/voice-sdk/twilio.min.js"></script>
```

Set up Toggle Agent availability in Twilio

To make the agent available or unavailable, we need to use the Device object in the Twilio SDK. The `Device` object represents a softphone that communicates with Twilio to help inbound and outbound audio connections.

Overview

Here's an overview of the sequence of operations performed once an agent marks themselves as available by clicking the Toggle Agent Availability button in the media toolbar application:



1. The agent clicks the Toggle Agent Availability button on the media toolbar.

2. The media toolbar application fires a REST API call to fetch the ID and toked required for authentication.
3. The Twilio service authenticates the request and returns the ID and token.
4. The media toolbar application uses this token to initialize the Device object provided by Twilio SDK.
5. Once the device object is registered in Twilio, it fires a `registered` event.
6. When the registered event is received in your media toolbar application, the `invoke agentStateEvent` UEF operation to make the agent state available.
7. After the fusion application identifies this action, the agent state is made as Available in the Fusion application also.

Update makeAgentAvailable method in VendorHandler to call the supplier API to make the agent available

1. Create a method in the `vendorHandler` class (`src/ts/cti/vendor/vendorHandler.ts`) with the following code:

```
private async getIdAndToken(): Promise<any> {
 const headers: Headers = (new Headers()) as Headers;
 const url: string = 'https://twilio-node-voice-stream.com/token'; // Replace this url with the url of
 the deployed node app
 headers.set('Accept', 'application/json');
 const request: Request = new Request(url, {
 method: 'GET',
 headers: headers
 }) as Request;
 const idAndToken: Response = await fetch(request);
 this.idAndToken = await idAndToken.json();
 return this.idAndToken;
}
```

2. Add the following import statement:

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';
```

3. Create two class properties in Twilio, `device` and `idAndToken` as shown:

```
export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 public idAndToken: any;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.twilio = (window as any).Twilio;
 this.device = null;
 this.idAndToken = null;
 this.integrationEventsHandler = integrationEventsHandler;
 }
// ...
}
```

4. Instantiate a device with `DeviceOptions`:

```
const idAndToken = await this.getIdAndToken(); // get the id token
this.device = new this.twilio.Device(idAndToken.token, {
 logLevel: 1,
 codecPreferences: ["opus", "pcmu"],
 enableRingingState: true
});
```

5. To make the agent available call the register on the device instance as shown:

```
this.device.register();
```

6. Use the `registered` event and `error` event to get the result for registration event as shown:

```
this.device.on("registered", () => {
```

```
 // Do your logic when registration is completed.
});
this.device.on("error", (deviceError) => {
 // Do your logic when the registration fails
});
```

7. Update the `makeAgentAvailable` method in `vendorHandler` as shown:

```
public async makeAgentAvailable(): Promise<void> {
 await this.getIdAndToken();
 this.device = this.twilio.Device(this.idAndToken.token, {
 logLevel: 1,
 codecPreferences: ["opus", "pcmu"],
 enableRingingState: true
 });
 let resolve: Function;
 let reject: Function;
 if (this.device) {
 this.device.on("registered", () => {
 console.log("Registration completed ...")
 resolve();
 });
 this.device.on("error", (deviceError) => {
 console.error("Registration Failed ...", deviceError);
 reject();
 });
 this.device.register();
 }
 return new Promise((res: Function, rej: Function) => {
 resolve = res;
 reject = rej;
 });
}
```

8. Update the `makeAgentUnavailable` method in `VendorHandler` as shown:

```
public async makeAgentUnavailable(): Promise<void> {
 let resolve: Function;
 let reject: Function;
 if (this.device) {
 this.device.on("unregistered", () => {
 console.log("Successfully UnRegistered ...");
 resolve();
 });
 this.device.on("error", (deviceError) => {
 console.error("Failed to unregister ...", deviceError);
 reject();
 });
 this.device.unregister();
 }
 return new Promise((res: Function, rej: Function) => {
 resolve = res;
 reject = rej;
 });
}
```

9. View the complete code:

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 public idAndToken: any;
```

```
constructor() {
this.twilio = (window as any).Twilio;
this.device = null;
this.idAndToken = null;
this.integrationEventsHandler = integrationEventsHandler;
}

public async makeAgentAvailable(): Promise<void> {
const idAndToken = await this.getIdAndToken();
this.device = new Device(idAndToken.token, {
logLevel: 1,
codecPreferences: [Call.Codec.Opus, Call.Codec.PCMU],
//enableRingingState: true
});
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("registered", () => {
console.log("Registration completed ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Registration Failed ...", deviceError);
reject();
});
this.device.register();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeAgentUnavailable(): Promise<void> {
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("unregister", () => {
console.log("Successfully UnRegistered ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Failed to unregister ...", deviceError);
reject();
});
this.device.unregister();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
throw new Error('Method not implemented.');
}
public async acceptCall(): Promise<void> {
throw new Error('Method not implemented.');
}
public async rejectCall(): Promise<void> {
throw new Error('Method not implemented.');
}
public async hangupCall(): Promise<void> {
throw new Error('Method not implemented.');
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
```

```
headers.set('Accept', 'application/json');
const request: Request = new Request('https://twilio-node-voice-stream.com/token', {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}
}
```

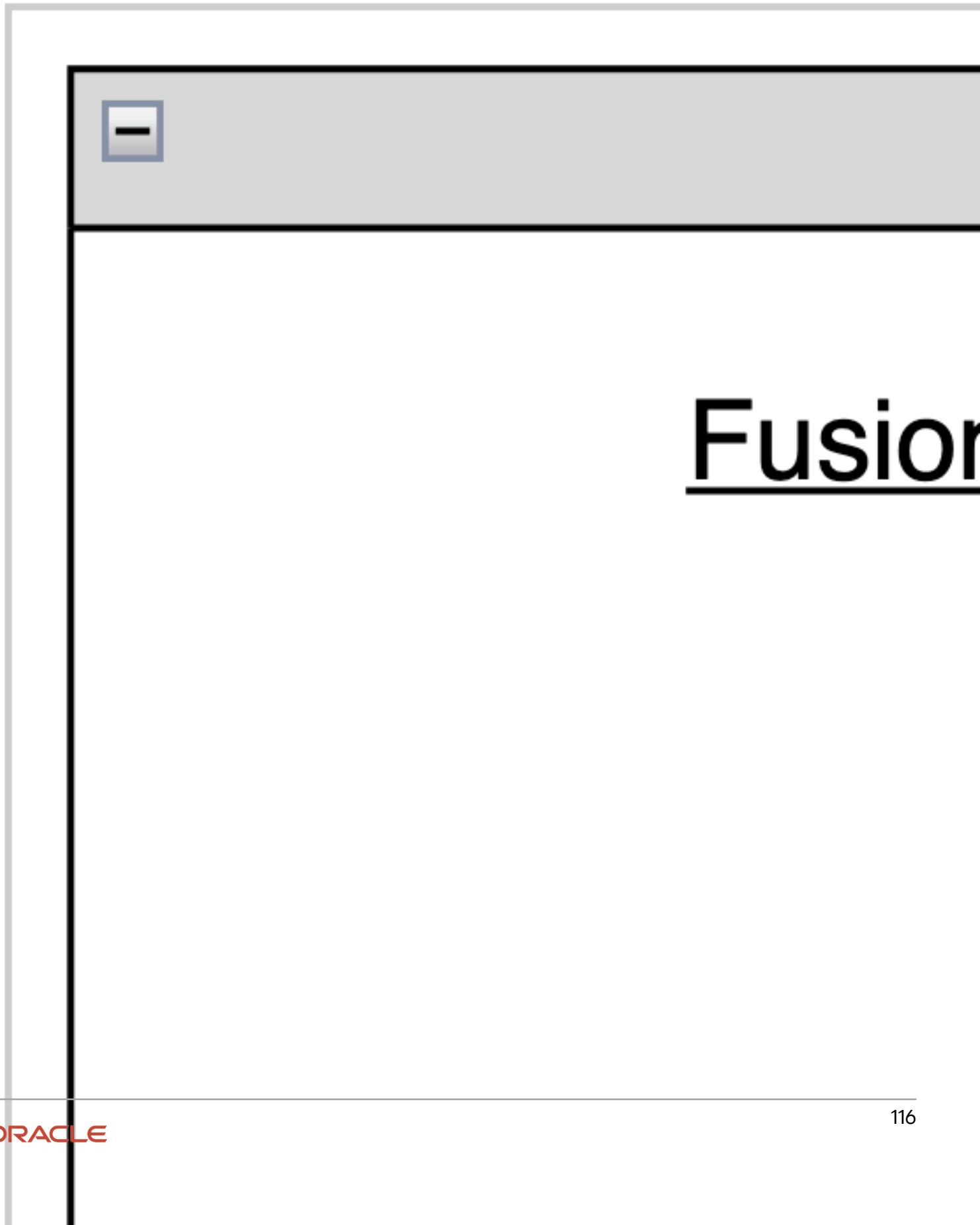
Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the Agent Availability button from your media toolbar application. You'll see that the button color changes to and the phone icon status in the Fusion application is changed to Available.

Show incoming call notification in Twilio

Twilio notifies incoming calls to your media toolbar application by firing an incoming event from `Twilio.Device` object. From your media toolbar application listens to this event and does the necessary steps once this event is received.

The following flow diagram shows the sequence of operations performed once a customer dials your service center phone number:


Fusion

Here's how it works:

1. Twilio receives an incoming call notification and fires an incoming event along with the call information as payload with type as `Twilio.Call`. Twilio is responsible for this step.
2. The incoming event handler in your toolbar application receives this event along with the Call payload.)
3. The `newComm` event is fired from your toolbar application.
4. Once the Fusion application identifies this action, it performs a contact number lookup and returns the contact details in the action response.
   To create lookup filters to create your own the default reverse lookup, see *Create Lookup Filters*.
5. Finally, the Fusion application notifies the agent about the incoming call offer notification through a dialog box.
6. The agent can now either Answer or Decline the call.

All you need to do this is to add the following code once the Twilio.Device object is initialized in the makeAgentAvailable function in vendorHandler.ts file.

> **Note:** Add the following code once the `Twilio.Device` object is initialized in the `makeAgentAvailable` function in `vendorHandler.ts` file.

```
this.device.on("incoming", this.incomingCallCallback);
```

From the code shown, `incomingCallCallback` is the callback function which gets executed during the `Twilio.Device` incoming event. Define this function in your `vendorHandler.ts` file as:

```
public incomingCallCallback = (call: Call) => {
 this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
 this.call = call;
}
```

Call related operations such as `acceptCall`, `rejectCall` and `hangupCall` are performed on the call object received in the incoming event listener.

Sample code for the vendorHandler.ts file

Here's the complete code of the `vendorHandler.ts` file for showing the incoming call notification:

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 private call: Call | null;
 public idAndToken: any;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.twilio = (window as any).Twilio;
 this.device = null;
 this.idAndToken = null;
 this.integrationEventsHandler = integrationEventsHandler;
 this.call = null;
 }

 public async makeAgentAvailable(): Promise<void> {
 this.idAndToken = await this.getIdAndToken();
```

```
this.device = new this.twilio.Device(this.idAndToken.token, {
logLevel: 1,
codecPreferences: ["opus", "pcmu"],
enableRingingState: true
});
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("registered", () => {
console.log("Registration completed ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Registration Failed ...", deviceError);
reject();
});

this.device.on("incoming", this.incomingCallCallback);

this.device.register();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}

public incomingCallCallback = (call: Call) => {
this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
this.call = call;
}
public async makeAgentUnavailable(): Promise<void> {
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("unregister", () => {
console.log("Successfully UnRegistered ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Failed to unregister ...", deviceError);
reject();
});
this.device.unregister();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
throw new Error('Method not implemented.');
}
public async acceptCall(): Promise<void> {
throw new Error('Method not implemented.');
}
public async rejectCall(): Promise<void> {
throw new Error('Method not implemented.');
}
public async hangupCall(): Promise<void> {
throw new Error('Method not implemented.');
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Accept', 'application/json');
```

```
const request: Request = new Request('https://twilio-voice-stream.com/token', {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}
}
```

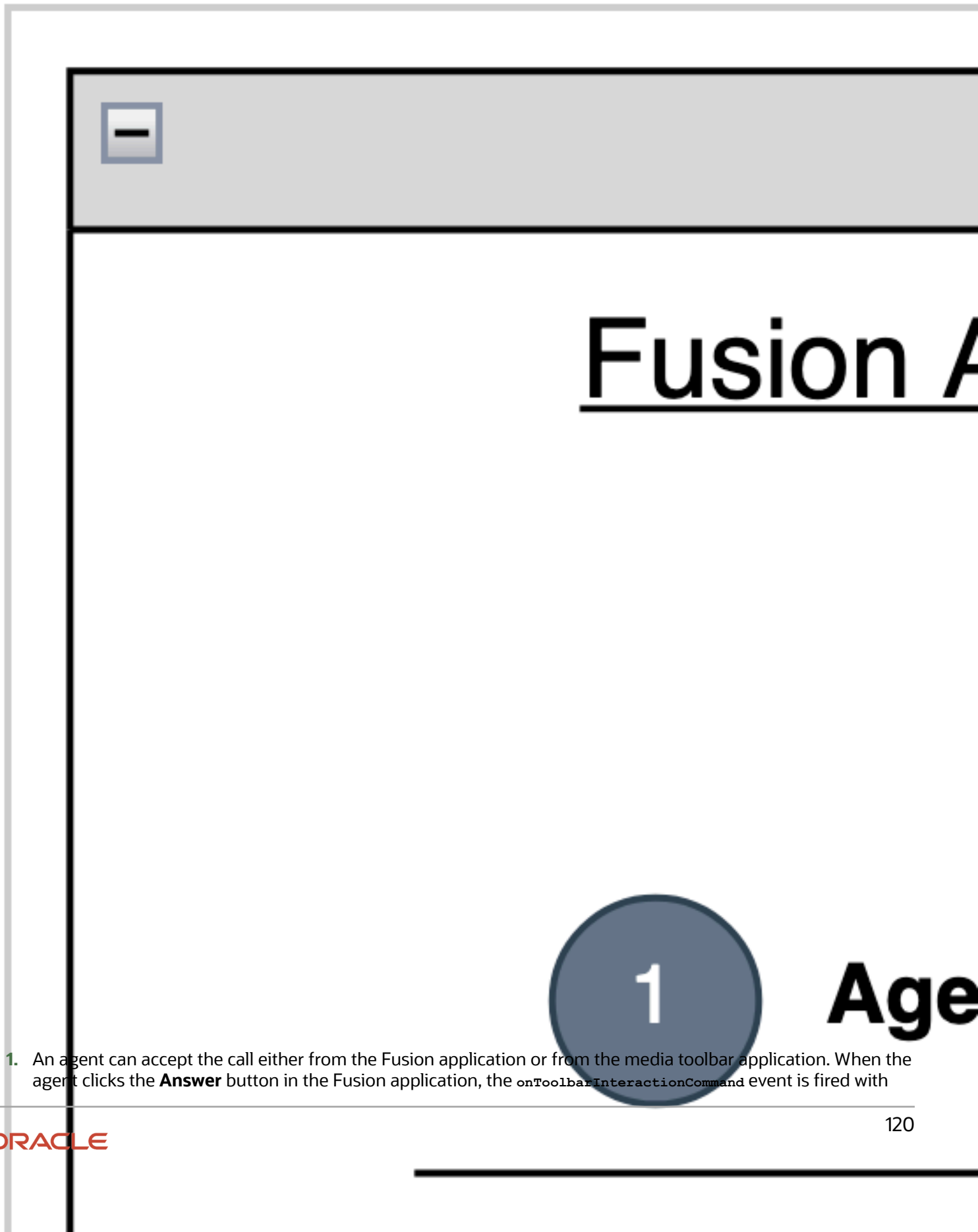
Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. Then, start a call to your customer care number. During this time, your media toolbar state will be changed to RINGING state and you can see the incoming call notification in your media toolbar application.

Accept incoming call in Twilio

All the call related operation such as call accept, reject, disconnect, mute and so on, must be performed on the Call object. You use the `call.accept()` function to accept a call.

The following flow diagram shows the sequence of operations performed once an agent accepts the call from Fusion or from the media toolbar application:



1. An agent can accept the call either from the Fusion application or from the media toolbar application. When the agent clicks the **Answer** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with

the command as accept. When the agent accepts the call from media toolbar application, the supplier API to accept the call is directly called.

2. The media toolbar application receives this event and if the event is to accept a call, the Twilio API to accept the call can be called.
3. Once the CTI supplier notifies the media toolbar application that the call is accepted, the partner application can fire the `startCommEvent` action.
4. Once the Fusion application identifies this action, the matched contact and an engagement panel is opened based on the rules configured in the Fusion configuration management for the screen pop. See *Configure Screen Pop Pages* for more information.

During the incoming event, you'll receive the `Twilio.Call` object from the event payload. You can use `call.accept()` function to accept a call as shown in the following example:

```
public async acceptCall(): Promise<void> {
 if (this.call) {
 this.call.accept();
 }
 }
```

Change your `acceptCall` function in `vendorHandler.ts` file to call the `call.accept()` function.

Sample code for the vendorHandler.ts file

Here's the complete code of the `vendorHandler.ts` file for accepting an incoming call:

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 private call: Call | null;
 public idAndToken: any;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.twilio = (window as any).Twilio;
 this.device = null;
 this.idAndToken = null;
 this.integrationEventsHandler = integrationEventsHandler;
 this.call = null;
 }

 public async makeAgentAvailable(): Promise<void> {
 this.idAndToken = await this.getIdAndToken();
 this.device = new this.twilio.Device(this.idAndToken.token, {
 logLevel: 1,
 codecPreferences: ["opus", "pcmu"],
 enableRingingState: true
 });
 let resolve: Function;
 let reject: Function;
 if (this.device) {
 this.device.on("registered", () => {
 console.log("Registration completed ...")
 resolve();
 });
 this.device.on("error", (deviceError) => {
 console.error("Registration Failed ...", deviceError);
 reject();
```

```
});

this.device.on("incoming", this.incomingCallCallback);

this.device.register();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}

public incomingCallCallback = (call: Call) => {
this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
this.call = call;
}
public async makeAgentUnavailable(): Promise<void> {
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("unregister", () => {
console.log("Successfully UnRegistered ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Failed to unregister ...", deviceError);
reject();
});
this.device.unregister();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
throw new Error('Method not implemented.');
}
public async acceptCall(): Promise<void> {
if (this.call) {
this.call.accept();
}
}
public async rejectCall(): Promise<void> {
throw new Error('Method not implemented.');
}
public async hangupCall(): Promise<void> {
throw new Error('Method not implemented.');
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Accept', 'application/json');
const request: Request = new Request('https://twilio-voice-stream.com/token', {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. Then, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion application window. You can accept the call from your media toolbar application or from your Fusion application or from your softphone application. Once you accept the call, your media toolbar state will change to the ACCEPTED state and the engagement is opened in your Fusion application.

Disconnect or reject an incoming call in Twilio

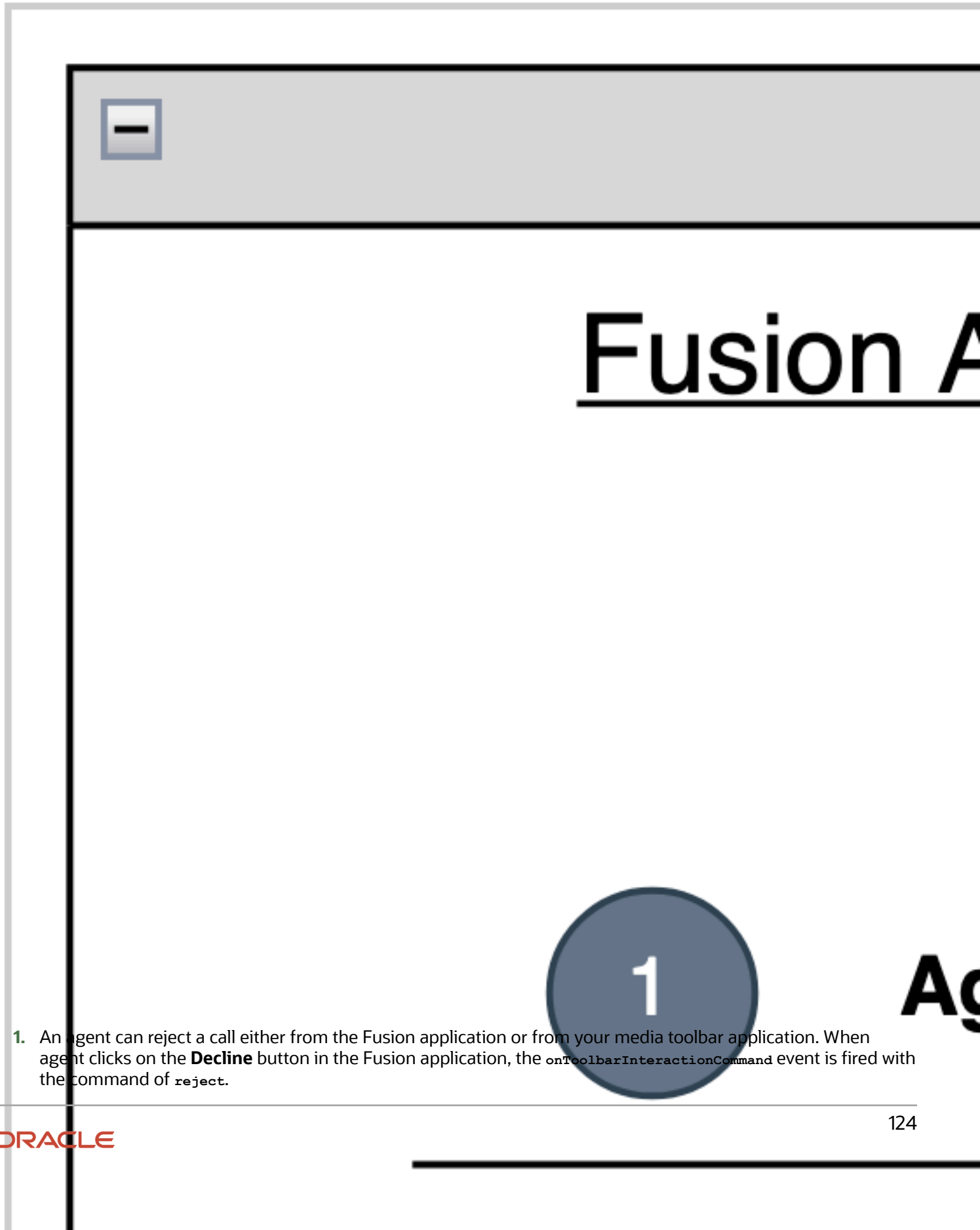
All the call related operations such as call accept, reject, disconnect, mute and so on, must be performed on the Call object. You use the `call.disconnect()` function to disconnect a call and the `call.reject()` function to reject a call.

Call disconnection can happen in any of the following three ways:

- Agent rejects the incoming call (before the call is accepted).
- Agent disconnects the call once the conversation is complete.
- Customer disconnects the call.

Scenario 1: Agent rejects the incoming call (before the call is accepted)

The following flow diagram shows the sequence of operations performed once an agent rejects the call from Fusion or from the media toolbar application:



1. An agent can reject a call either from the Fusion application or from your media toolbar application. When agent clicks on the **Decline** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command of `reject`.

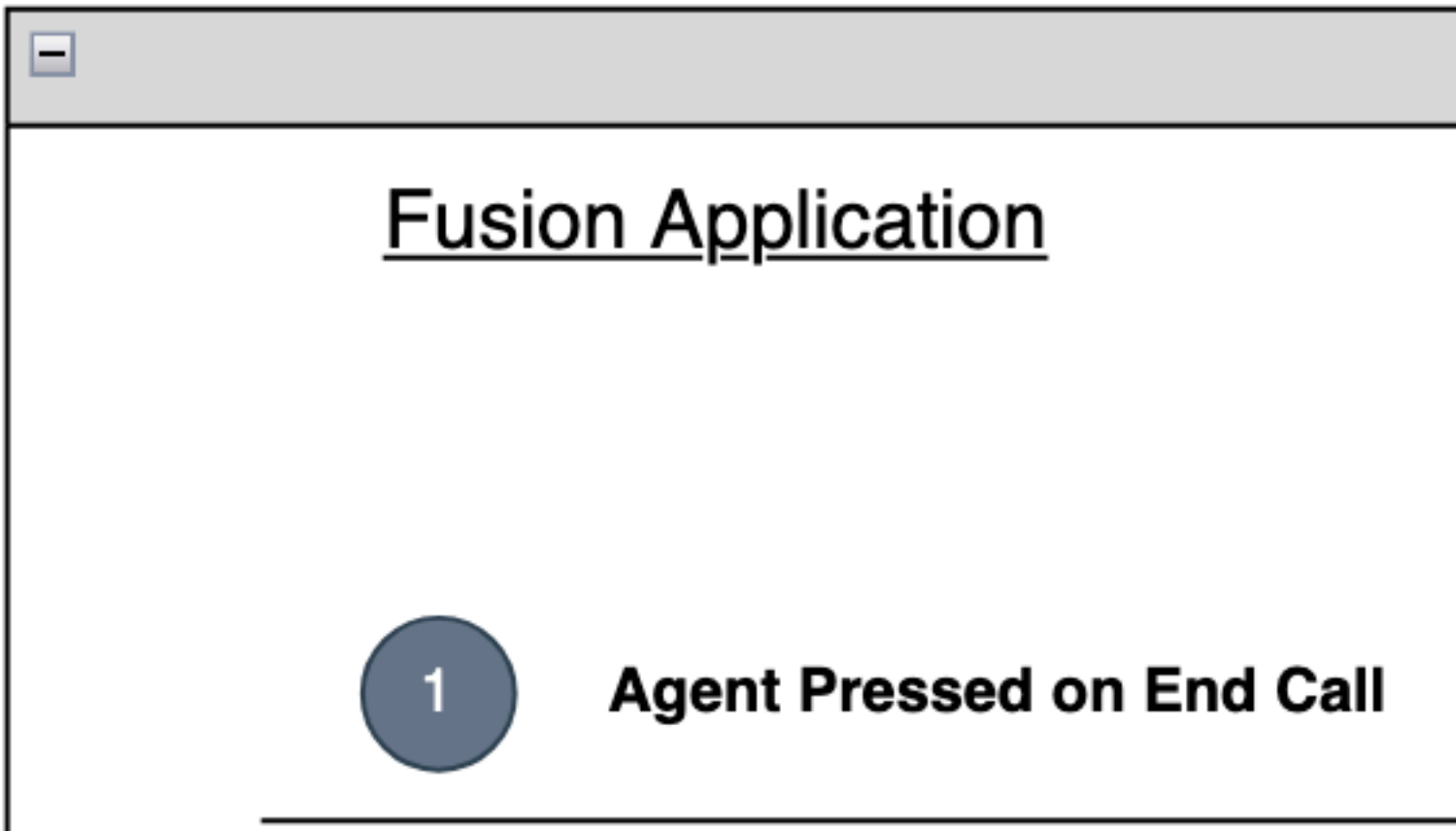
2. The partner application receives this event and if the event is to `reject` the call, a reject call to the Twilio API is made.
3. When the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with reason as REJECT.
4. When the Fusion application identifies this action, the call dialog is discarded from the UI.

During the incoming event, the `Twilio.Call` object is included in the event payload. You can use the `call.reject()` function to reject a call as shown in the following example:

```
public async rejectCall(): Promise<void> {
 if (this.call) {
 this.call.reject();
 }
 }
```

Scenario 2: Agent disconnects the call once the conversation is complete

The following flow diagram shows the sequence of operations performed once an agent disconnects the call from Fusion or from the media toolbar application:



1. An agent can reject a call either from the Fusion application or from your media toolbar application. When agent clicks on the **Decline** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command of `disconnect`.

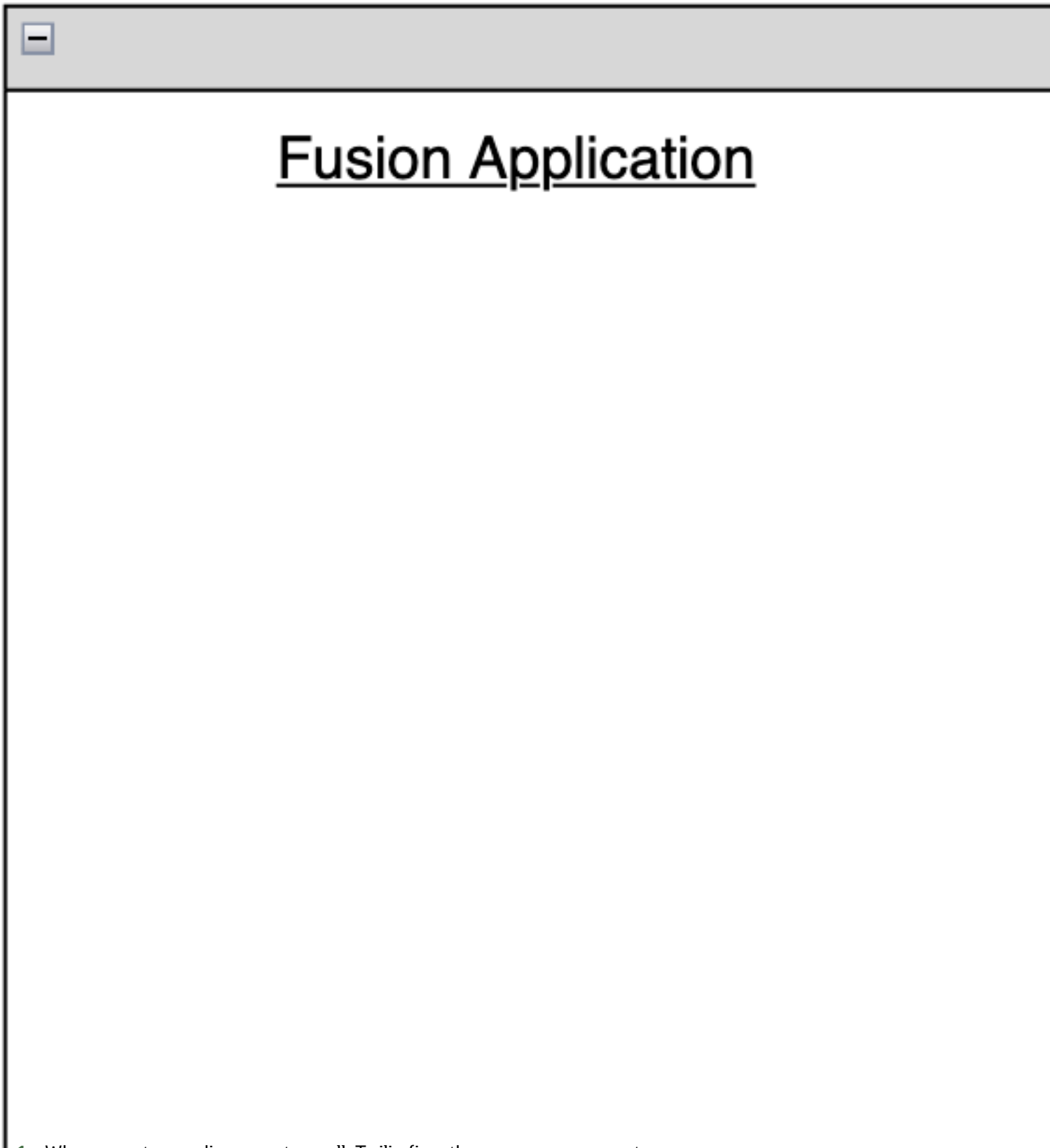
2. The partner application receives this event and if the event is to `disconnect` the call, a reject call to the Twilio API is made.
3. When the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with reason as HANGUP.
4. When the Fusion application identifies this action, it renders the wrap up window in the UI.

You can add the disconnect and reject event listeners in the call object received in the `incoming` event handler as shown in the following example:

```
public incomingCallCallback = (call: Call) => {
 this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
 this.call = call;
 this.call.on("cancel", () =>
 { this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
 this.call.on("disconnect", () =>
 { this.integrationEventsHandler.callHangupHandler(call.parameters.CallSid) });
 this.call.on("reject", () =>
 { this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
}
```

Scenario 3: Customer disconnects the call

The following flow diagram shows the sequence of operations performed when the customer disconnects the call:



1. When a customer disconnects a call, Twilio fires the `disconnected` event.

2. You need to add an event listener in your media toolbar application for the reject and disconnect events
3. The `closeCommEvent` action is fired from your media toolbar application with reason as REJECT or WRAPUP.
4. When the Fusion application identifies this action, it renders the wrap up window in the UI.

You can add the disconnect and reject event listeners in the call object received in the `incoming` event handler as shown in the following example:

```
public incomingCallCallback = (call: Call) => {
 this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
 this.call = call;
 this.call.on("cancel", () =>
 { this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
 this.call.on("disconnect", () =>
 { this.integrationEventsHandler.callHangupHandler(call.parameters.CallSid) });
 this.call.on("reject", () =>
 { this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
}
```

Complete code

Here's the complete code for the `vendorHandler.ts` file for disconnecting a call.

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 private call: Call | null;
 public idAndToken: any;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.twilio = (window as any).Twilio;
 this.device = null;
 this.idAndToken = null;
 this.integrationEventsHandler = integrationEventsHandler;
 this.call = null;
 }

 public async makeAgentAvailable(): Promise<void> {
 this.idAndToken = await this.getIdAndToken();
 this.device = new this.twilio.Device(this.idAndToken.token, {
 logLevel: 1,
 codecPreferences: ["opus", "pcmu"],
 enableRingingState: true
 });
 let resolve: Function;
 let reject: Function;
 if (this.device) {
 this.device.on("registered", () => {
 console.log("Registration completed ...")
 resolve();
 });
 this.device.on("error", (deviceError) => {
 console.error("Registration Failed ...", deviceError);
 reject();
 });

 this.device.on("incoming", this.incomingCallCallback);

 this.device.register();
```

```
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}

public incomingCallCallback = (call: Call) => {
this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
this.call = call;
this.call.on("cancel", () =>
{ this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
this.call.on("disconnect", () =>
{ this.integrationEventsHandler.callHangupHandler(call.parameters.CallSid) });
this.call.on("reject", () =>
{ this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
}
public async makeAgentUnavailable(): Promise<void> {
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("unregister", () => {
console.log("Successfully UnRegistered ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Failed to unregister ...", deviceError);
reject();
});
this.device.unregister();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
throw new Error('Method not implemented.');
}
public async acceptCall(): Promise<void> {
if (this.call) {
this.call.accept();
}
}
public async rejectCall(): Promise<void> {
if (this.call) {
this.call.reject();
}
}
public async hangupCall(): Promise<void> {
if (this.call) {
this.call.disconnect();
}
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Accept', 'application/json');
const request: Request = new Request('https://twilio-voice-stream.com/token', {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}
```

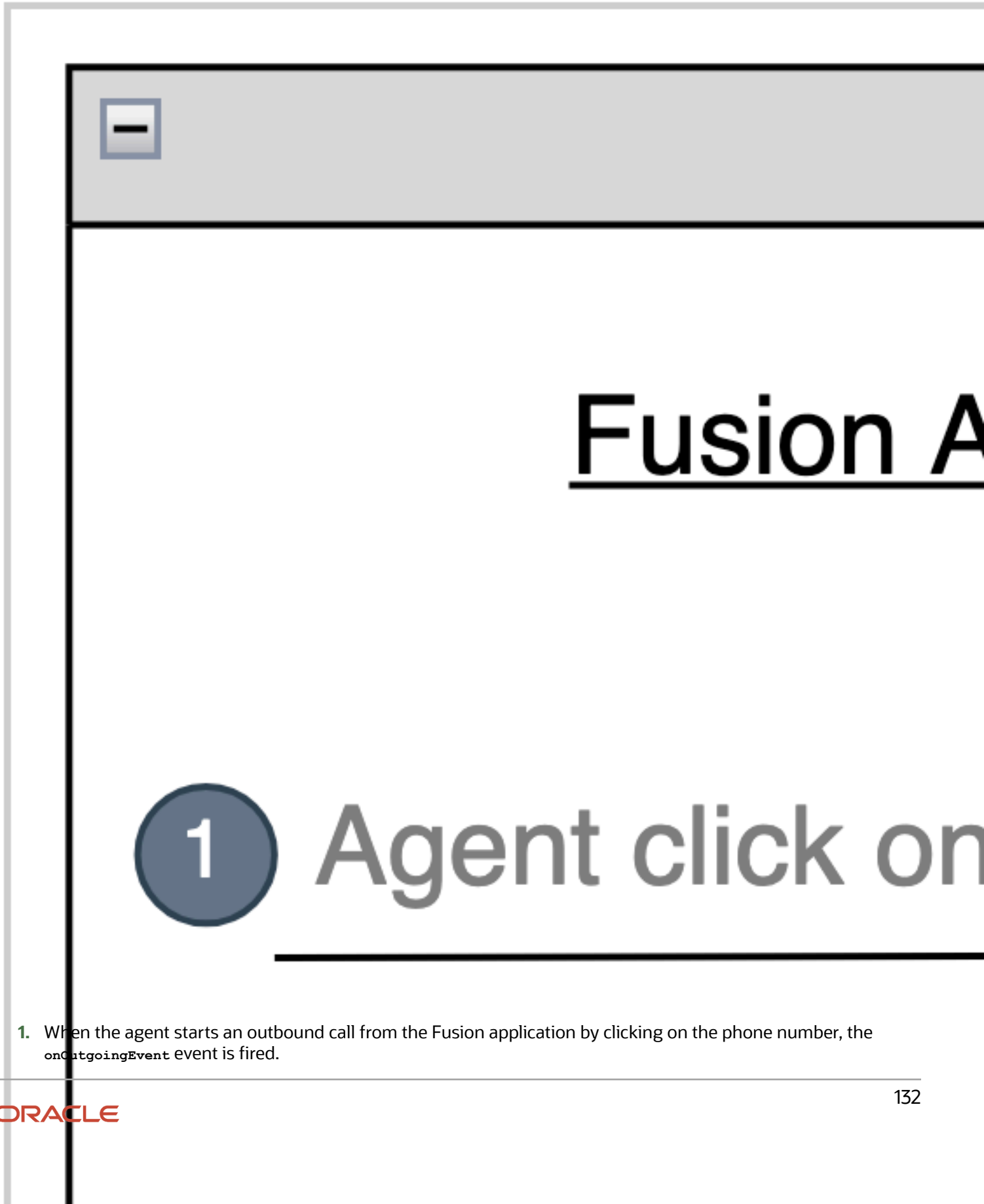
```
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. Then, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion application window. You can accept the call from your media toolbar application or from your Fusion application or from your softphone application. Once you accept the call, your media toolbar state will change to the ACCEPTED state and the engagement is opened in your Fusion application. You can disconnect the call from your media toolbar application or from your Fusion application or from the softphone. Once you disconnect the call, your media toolbar state will be changed to the DISCONNECTED state.

Make an outbound call with Twilio

The following flow diagram shows the sequence of operations performed once an agent starts an outbound call from the Fusion application:



1. When the agent starts an outbound call from the Fusion application by clicking on the phone number, the `onOutgoingEvent` event is fired.

2. The `onOutgoingEvent` event listener in your media toolbar application receives this event and starts the outbound call by calling the Twilio `device.connect` API by passing the phone number to be dialed.
3. Twilio starts the outbound call and returns the `Call` object.
4. The `newCommEvent` action is called from your media toolbar application.
5. The Fusion application shows the dialing panel once the `newCommEvent` action is received

To make an outbound call using Twilio , we need to call the connect method of the Twilio `Device` object. The code looks like this:

```
const device = new Device(token);

let call = await device.connect({
 params: {
 To: '+15551234567'
 }
});
```

This example shows an try to create a new connection with the Twilio application that you associated with the Access Token used when instantiating the `Device` instance. This method returns a `Promise` with the `Call` object. You can track this `Call` object to monitor or change the active call. You can listen to user actions by listening to the to the `accept`, `disconnect` and `cancel` events on the `Call` object.

You can update the `makeOutboundCall` method in `VendorHandler` class for this as shown in the following example:

```
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
 const params = {
 To: phoneNumber,
 };
 if (this.device) {
 this.call = await this.device.connect({ params });
 const callId = this.call.parameters.CallSid
 this.call.on("accept", () => { this.integrationEventsHandler.outboundCallAcceptedHandler(callId) });
 this.call.on("disconnect", () => { this.integrationEventsHandler.callHangupHandler(callId) });
 this.call.on("cancel", () => { this.integrationEventsHandler.callRejectedHandler(callId) });
 }
 }
```

Complete code

Here's the complete code of the `vendorHandler.ts` file for starting an outbound call:

```
import { Call, Device } from '@twilio/voice-sdk';
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private twilio: any;
 private device: Device | null;
 private integrationEventsHandler: IntegrationEventsHandler;
 private call: Call | null;
 public idAndToken: any;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.twilio = (window as any).Twilio;
 this.device = null;
 this.idAndToken = null;
 this.integrationEventsHandler = integrationEventsHandler;
 this.call = null;
 }

 public async makeAgentAvailable(): Promise<void> {
```

```
this.idAndToken = await this.getIdAndToken();
this.device = new this.twilio.Device(this.idAndToken.token, {
logLevel: 1,
codecPreferences: ["opus", "pcmu"],
enableRingingState: true
});
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("registered", () => {
console.log("Registration completed ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Registration Failed ...", deviceError);
reject();
});

this.device.on("incoming", this.incomingCallCallback);

this.device.register();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}

public incomingCallCallback = (call: Call) => {
this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
this.call = call;
}
public async makeAgentUnavailable(): Promise<void> {
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("unregister", () => {
console.log("Successfully UnRegistered ...")
resolve();
});
this.device.on("error", (deviceError) => {
console.error("Failed to unregister ...", deviceError);
reject();
});
this.device.unregister();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
const params = {
To: phoneNumber,
};
if (this.device) {
this.call = await this.device.connect({ params });
const callId = this.call.parameters.CallSid
this.call.on("accept", () => { this.integrationEventsHandler.outboundCallAcceptedHandler(callId) });
this.call.on("disconnect", () => { this.integrationEventsHandler.callHangupHandler(callId) });
this.call.on("cancel", () => { this.integrationEventsHandler.callRejectedHandler(callId) });
}
}
public async acceptCall(): Promise<void> {
if (this.call) {
this.call.accept();
```

```
}
}
public async rejectCall(): Promise<void> {
if (this.call) {
this.call.reject();
}
}
public async hangupCall(): Promise<void> {
if (this.call) {
this.call.disconnect();
}
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Accept', 'application/json');
const request: Request = new Request('https://twilio-voice-stream.com/token', {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. To start an outbound call, open the contact from your Fusion application and click the phone number, which starts the outbound call.

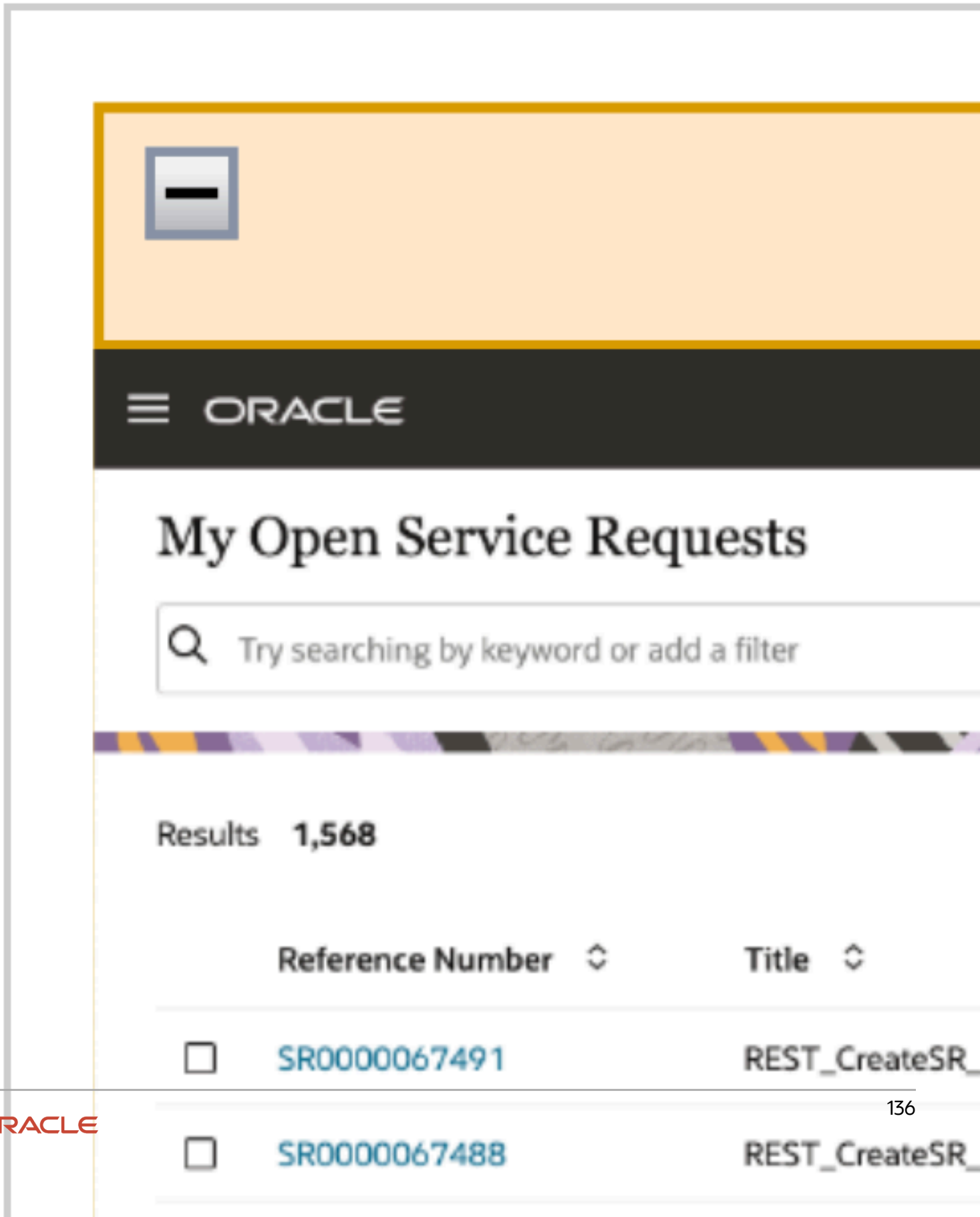
Genesys

Overview of Genesys integration

Genesys is another CTI provider which can be integrated with your Fusion application.

You need to use the *Platform SDK* provided by Genesys for the integration.

Here's an overview of the components that make up the integration:


ORACLE
My Open Service Requests
Try searching by keyword or add a filter
Results 1,568
Reference Number
Title
SR0000067491
REST_CreateSR_
SR0000067488
REST_CreateSR_

About the Java Microservice

For the current integration with Genesys PureEngage, you need to use a Java microservice that communicates to Genesys using the Genesys PSDK. This Java Microservice acts as a bridge between the Genesys server and your media toolbar application. Events from Genesys are propagated to your media toolbar application through this microservice using web sockets. So, you need to initialize this web socket in your media toolbar application before making the agent available for calls. Here are the events that are part of the integration:

Events

Event	Description
EventRegistered	The agent has signed in and is ready to receive calls.
EventRinging	Incoming call notification.
EventEstablished	Call accepted notification.
EventReleased	Call ended notification.

There are also call operations such as accept call, disconnect call, initiate call from your media toolbar application are propagated to Genesys by firing REST APIs which are served by the microservice. Here's the call operations that can serve the microservice:

Call operations

Type	Description
Register	Register the agent and make the agent available to receive the calls.
Accept	Accept the incoming call.
Reject	Reject the incoming call.
Hangup	Hangup the ongoing call.
MakeCall	Start an outbound call.

About the Softphone

Genesys integration as part of this tutorial series uses VoIP based integration and you need to configure a softphone with your Genesys credentials. *Zoiper*, *PhonerLite*, *Bria* are a few of the softphone applications that can be configured with your Genesys account. Once you configure your softphone, the incoming calls to your Genesys number will be received in your softphone and you can accept it from the softphone itself. Outgoing calls are also made through the softphone.

Configure the softphone application

Genesys uses VoIP based protocol for communication, so the agent needs to configure a softphone application in their computer or smart phone to accept or start calls.

A softphone is a software-based app that enables you to make and receive phone calls over the internet using a computer, tablet, or smartphone. Various softphone applications are available Windows, MacOS, Android or iOS platforms. You can download any of then based on your convenience. Some commonly used softphones are Zoiper, Bria and Phonerlite.

Download and install the softphone of your choice, and set it up.

Microservice using PSDK

Now you need to build a Java Microservice for CTI session management using the PSDK provided by Genesys.

Starting the microservice

To start the microservice, you must have the following:

- Java 21
- Maven

After downloading the code, follow these steps:

1. Open the project root folder in your terminal.
2. Run the following commands:

```
mvn install:install-file -Dfile=./external/com/gensyslab/platform/voiceprotocol.jar -
DgroupId=com.genesyslab.platform -DartifactId=voiceprotocol -Dversion=1.0 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/commons.jar -
DgroupId=com.genesyslab.platform -DartifactId=commons -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/protocol.jar -
DgroupId=com.genesyslab.platform -DartifactId=protocol -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/connection.jar -
DgroupId=com.genesyslab.platform -DartifactId=connection -Dversion=1.0 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/kvlists.jar -
DgroupId=com.genesyslab.platform -DartifactId=kvlists -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-transport.jar -DgroupId=io.netty
 -DartifactId=netty-transport -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-buffer.jar -DgroupId=io.netty -
DartifactId=netty-buffer -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-codec-http.jar -
DgroupId=io.netty -DartifactId=netty-codec-http -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-codec-socks.jar -
DgroupId=io.netty -DartifactId=netty-codec-socks -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-codec.jar -DgroupId=io.netty -
DartifactId=netty-codec -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-common.jar -DgroupId=io.netty -
DartifactId=netty-common -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-handler-proxy.jar -
DgroupId=io.netty -DartifactId=netty-handler-proxy -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
```

```
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-handler.jar -DgroupId=io.netty -
DartifactId=netty-handler -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-resolver.jar -DgroupId=io.netty
 -DartifactId=netty-resolver -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/netty-transport-native-unix-common.jar
 -DgroupId=io.netty -DartifactId=netty-transport-native-unix-common -Dversion=1.0 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/connection.jar -
DgroupId=com.genesyslab.platform -DartifactId=connection -Dversion=1.0 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/commons.jar -
DgroupId=com.genesyslab.platform -DartifactId=commons -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/voiceprotocol.jar -
DgroupId=com.genesyslab.platform -DartifactId=voiceprotocol -Dversion=1.0 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/protocol.jar -
DgroupId=com.genesyslab.platform -DartifactId=protocol -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn install:install-file -Dfile=./external/com/gensyslab/platform/kvlists.jar -
DgroupId=com.genesyslab.platform -DartifactId=kvlists -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
mvn clean package
```

3. The command will generate a jar file in the target folder.
4. Run the jar file using the following command:

```
java -jar target/genesys-service.jar
```

Concepts

The Java Microservice for CTI session management acts as a bridge between the Genesys service provider and your media toolbar application. Events from Genesys are listened to through the PSDK APIs and your media toolbar application is notified through a web socket. Similarly, operations from your media toolbar application are passed on to Genesys through this microservice.

Message Structure

Once your microservice receives events from your Genesys service, the microservice sends it to your media toolbar application through a web socket. The following example shows the structure of the message which is passed from the microservice to your media toolbar application:

```
public class EventData {
 String eventName;
 long connectionId;
 String ANI;
 String direction;
 String rep_score;
 String eventId;
 String sessionId;
}
```

The following table shows the description of the message properties:

SI Number	Property Name	Description
1	eventname	Name of the event which is fired from Genesys. It can be **EventRegistered**, **EventRinging**, **EventEstablished**, **EventReleased** as shown in table which follows.

SI Number	Property Name	Description
2	connectionId	Connection ID of the call.
3	ANI	Phone number associated with the call.
4	direction	Direction of the call such as `Inbound`, `Outbound`, and so on.
5	eventId	Event ID of the call.
6	sessionId	ID of the current session.

See the *Genesys documentation* for a list of Genysys-supported events.

The following table shows the mapping between the `PSDK` events and the events fired from the microservice:

SI Number	Event Name	Description
1	EventRegistered	Agent sign in has completed and the agent is ready to receive calls.
2	EventRinging	Incoming call notification
3	EventEstablished	Call accepted notification.
4	EventReleased	Call ended notification.

Code Samples

All functions shown in the following examples are defined in `src/main/java/com/oracle/genesys/handler/GenesysHandler.java` in the shared project.

Initializing the Channel code sample

The following example shows how to initialize the channel of your Genesys service:

```
private void initChannel() {
 PropertyConfiguration options = new PropertyConfiguration();
 options.setUseAddp(true);
 options.setAddpClientTimeout(3);
 options.setAddpServerTimeout(4);
 options.setAddpTraceMode(ClientADDPOptions.AddpTraceMode.Local);
 channel = new TServerProtocol(new Endpoint("PSDK-Sample-App", "10.138.194.168", 8080, options));
 ChannelEventsListener channelListener = new ChannelEventsListener();
 channel.addChannelListener(channelListener);
 channel.setMessageHandler(new ReceivedMessageHandler(this));
 channel.setInvoker(new SwingInvoker());
 if (channel.getState() != ChannelState.Closed) {
 return;
 }
 channel.setClientName("test");
 channel.setClientPassword("");
 System.out.println("Connecting...");
```

```
}
```

Request Register Address code sample

The following shows how to register your Genesys service:

```
private void requestRegisterAddress() {
 RequestRegisterAddress request = RequestRegisterAddress.create(this.DN,
 RegisterMode.ModeShare, ControlMode.RegisterDefault, AddressType.DN);
 try {
 channel.requestAsync(request, this, new CompletionHandler<Message, Object>() {
 public void completed(Message message, Object obj) {
 System.out.println("Received: \n" + message.toString());
 if (EventRegistered.ID == message.messageId()) {
 EventRegistered eventRegistered = (EventRegistered) message;
 System.out.println("####" + eventRegistered.getThisDN());
 EventData data = new EventData();
 data.eventName = message.messageName();
 GenesysHandler.setAniAndScore(data, eventRegistered.getThisDN());
 WebHookHandler.sendResponse(data);
 requestAgentLogin();
 } else {
 System.out.println("\n Can't register.");
 }
 }
 public void failed(Throwable arg0, Object obj) {
 System.out.println(arg0.getMessage());
 }
 });
 } catch (ChannelClosedOnSendException closedEx) {
 System.out.println("Channel should be opened before sending request");
 System.out.println(closedEx.getMessage());
 } catch (Throwable e) {
 System.out.println(e.getMessage());
 }
 }
```

Request Agent Login code sample

The following shows how to request agent sign in to your Genesys service:

```
private void requestAgentLogin() {
 RequestAgentLogin requestAgentLogin = RequestAgentLogin.create(this.DN, AgentWorkMode.AutoIn, "1010",
 this.DN, null, null, null);
 try {
 channel.requestAsync(requestAgentLogin, this, new CompletionHandler<Message, Object>() {
 public void completed(Message message, Object obj) {
 System.out.println("Received: \n" + message.toString());
 if (EventAgentLogin.ID == message.messageId()) {
 System.out.println("login");
 }
 else {
 System.out.println("\n Can't login.");
 }
 }
 public void failed(Throwable arg0, Object obj) {
 arg0.printStackTrace();
 }
 });
 } catch (ChannelClosedOnSendException closedEx) {
 System.out.println("Channel should be opened before sending request");
 } catch (Throwable e) {
 e.printStackTrace();
```

```
}
}
```

Request Answer Call

The following shows how to request your Genesys service to answer a call:

```
public void acceptCall(String connectionId) {
 RequestAnswerCall requestAnswerCall =
 RequestAnswerCall.create(this.DN, eventRingingSaved.getConnID());
 try {
 Message response = channel.request(requestAnswerCall);
 System.out.println(response.toString());
 } catch (ProtocolException e) {
 System.out.println(e.getMessage());
 e.printStackTrace();
 }
 }
```

Request Release Call

The following shows how to request your Genesys service to disconnect a call:

```
public void rejectCall(String connectionId) {
 RequestReleaseCall requestReleaseCall =
 RequestReleaseCall.create(this.DN, eventRingingSaved.getConnID());
 try {
 Message response = channel.request(requestReleaseCall);
 System.out.println(response.toString());
 } catch (ProtocolException e) {
 System.out.println(e.getMessage());
 e.printStackTrace();
 }
 }
```

Define ReceivedMessageHandler class

While initializing the channel, you have the line : `channel.setMessageHandler(new ReceivedMessageHandler(this));` added in your `initChannel` function. This call is responsible for handling the events fired from your Genesys service.

The `ReceivedMessageHandler` class is defined as follows:

```
package com.oracle.genesys.handler;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
import com.genesyslab.platform.commons.protocol.Message;
import com.genesyslab.platform.commons.protocol.MessageHandler;
import com.genesyslab.platform.voice.protocol.tserver.events.EventAbandoned;
import com.genesyslab.platform.voice.protocol.tserver.events.EventAgentLogin;
import com.genesyslab.platform.voice.protocol.tserver.events.EventAgentReady;
import com.genesyslab.platform.voice.protocol.tserver.events.EventDialing;
import com.genesyslab.platform.voice.protocol.tserver.events.EventEstablished;
import com.genesyslab.platform.voice.protocol.tserver.events.EventLinkConnected;
import com.genesyslab.platform.voice.protocol.tserver.events.EventNetworkReached;
import com.genesyslab.platform.voice.protocol.tserver.events.EventRegistered;
import com.genesyslab.platform.voice.protocol.tserver.events.EventReleased;
import com.genesyslab.platform.voice.protocol.tserver.events.EventRinging;
import com.oracle.genesys.SessionHanlder;
import com.oracle.genesys.model.EventData;
```

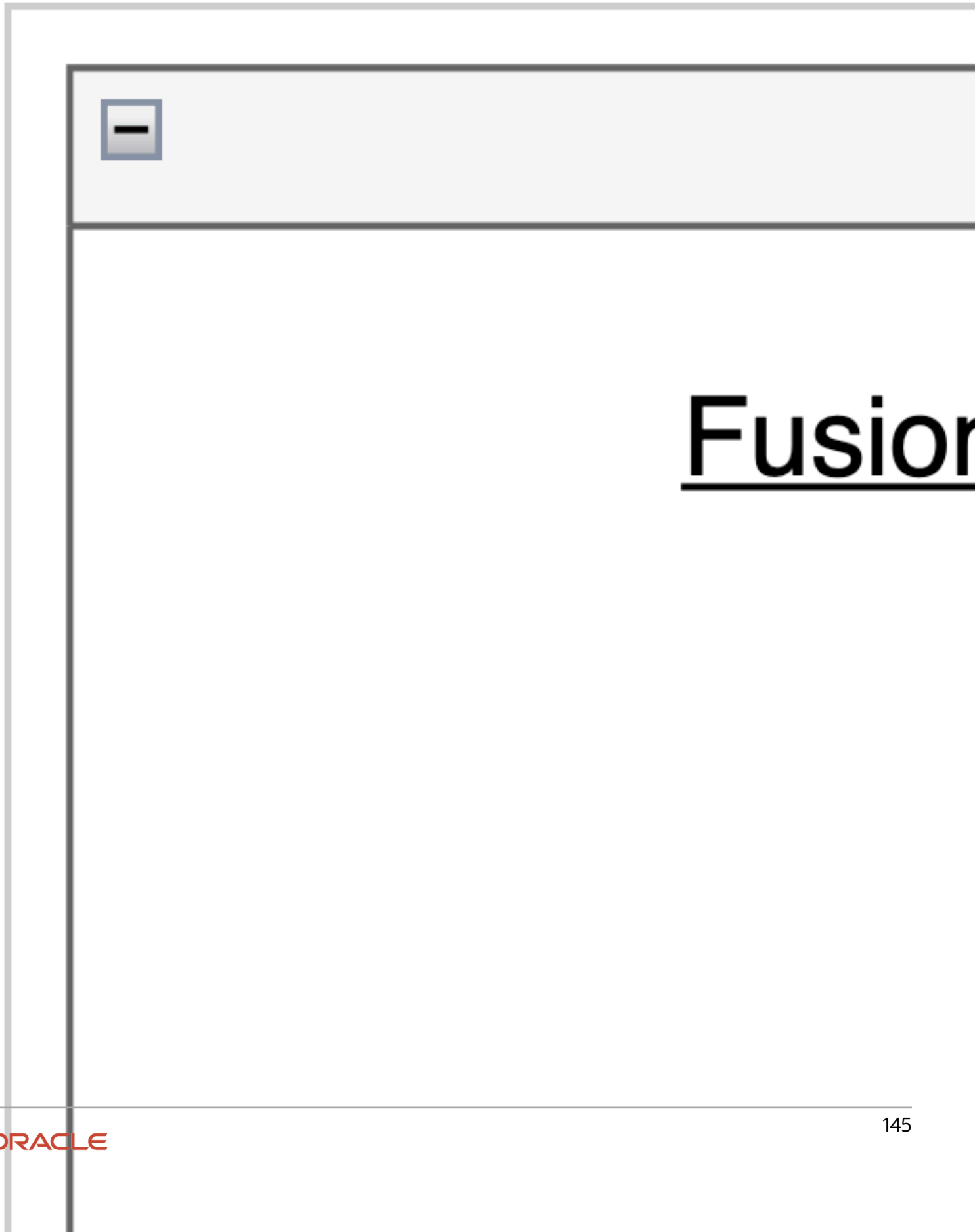
```
import java.io.IOException;

public class ReceivedMessageHandler implements MessageHandler {
 private final GenesysHandler genesysHandler;

 public ReceivedMessageHandler(GenesysHandler genesysHandler) {
 this.genesysHandler = genesysHandler;
 }
 public void onMessage(Message message) {
 System.out.println("Unsolicited event: \n" + message.toString());
 EventData data = new EventData();
 data.eventName = message.messageName();
 data.sessionId = this.genesysHandler.sessionId;
 String tempAni;
 switch (message.messageId()) {
 case EventLinkConnected.ID:
 case EventAgentLogin.ID:
 break;
 case EventRegistered.ID:
 break;
 case EventAgentReady.ID:
 break;
 case EventRinging.ID:
 EventRinging eventRinging = (EventRinging) message;
 data.eventId = eventRinging.getCallUuid().toLowerCase();
 tempAni = eventRinging.getANI() != null ? eventRinging.getANI() : eventRinging.getOtherDN();
 GenesysHandler.setAniAndScore(data, tempAni);
 data.direction = eventRinging.getCallType().toString();
 data.connectionId = eventRinging.getConnID().toLong();
 genesysHandler.eventRingingSaved = eventRinging;
 break;
 case EventEstablished.ID:
 EventEstablished eventEstablished = (EventEstablished) message;
 data.eventId = eventEstablished.getCallUuid().toLowerCase();
 tempAni = eventEstablished.getANI() != null ? eventEstablished.getANI() : eventEstablished.getOtherDN();
 GenesysHandler.setAniAndScore(data, tempAni);
 data.direction = eventEstablished.getCallType().toString();
 data.connectionId = eventEstablished.getConnID().toLong();
 break;
 case EventReleased.ID:
 EventReleased eventReleased = (EventReleased) message;
 data.eventId = eventReleased.getCallUuid().toLowerCase();
 tempAni = eventReleased.getANI() != null ? eventReleased.getANI() : eventReleased.getOtherDN();
 GenesysHandler.setAniAndScore(data, tempAni);
 data.direction = eventReleased.getCallType().toString();
 data.connectionId = eventReleased.getConnID().toLong();
 break;
 case EventDialing.ID:
 EventDialing eventDialing = (EventDialing) message;
 data.eventId = eventDialing.getCallUuid().toLowerCase();
 tempAni = eventDialing.getANI() != null ? eventDialing.getANI() : eventDialing.getOtherDN();
 GenesysHandler.setAniAndScore(data, tempAni);
 data.direction = eventDialing.getCallType().toString();
 data.connectionId = eventDialing.getConnID().toLong();
 break;
 case EventNetworkReached.ID:
 EventNetworkReached eventNetworkReached = (EventNetworkReached) message;
 data.eventId = eventNetworkReached.getCallUuid().toLowerCase();
 tempAni = eventNetworkReached.getANI() != null ? eventNetworkReached.getANI() :
 eventNetworkReached.getOtherDN();
 GenesysHandler.setAniAndScore(data, tempAni);
 data.direction = eventNetworkReached.getCallType().toString();
 data.connectionId = eventNetworkReached.getConnID().toLong();
 break;
 case EventAbandoned.ID:
```

```
EventAbandoned eventAbandoned = (EventAbandoned) message;
data.eventId = eventAbandoned.getCallUuid().toLowerCase();
tempAni = eventAbandoned.getANI() != null ? eventAbandoned.getANI() : eventAbandoned.getOtherDN();
GenesysHandler.setAniAndScore(data, tempAni);
data.direction = eventAbandoned.getCallType().toString();
data.connectionId = eventAbandoned.getConnID().toLong();
break;
}
WebHookHandler.sendResponse(data);
}
}
```

Events fired from your Genesys service are received by the `onMessage` function. The event payload is mapped to a common message structure as defined here and the payload is send to a web socket. From your media toolbar application you can listen to the messages from this web socket.

Set up toggle Agent Availability in Genesys

The following flow diagram shows the sequence of operations performed once agent marks themselves available by clicking the toggle Agent Availability button in the media toolbar application:


Fusion

Overview

Here's an overview of the sequence of operations performed once an agent marks themselves as available by clicking the toggle Agent Availability button in the media toolbar application:

1. The agent clicks the Toggle Agent Availability button on the media toolbar.
2. Media toolbar application starts a web socket connection to your Java microservice.
3. The microservice accepts the connection and listens for web socket messages.
4. If the web socket connection is successful from your media toolbar application, it notifies your microservice to register an agent.
5. When your microservice receives this notification, Your microservice should:
    a. Register a DN for your agent to use by using the `com.genesyslab.platform.voice.protocol.tserver.requests.dn.RequestRegisterAddress` request
    b. Sign in your agent using the `com.genesyslab.platform.voice.protocol.tserver.requests.agent.RequestAgentLogin.RequestAgentLogin` request
    c. Once the login is success, EventRegistered event is fired from your microservice.
6. When the `RequestRegisterAddress` request is success, the `EventRegistered` message is propagated from Genesys through your microservice to your media toolbar application.
7. After the `EventRegistered` event is received in your media toolbar application, the `agentStateEvent` UEF operation is launched to make the agent state available.
8. Finally, the Fusion application identifies the action, and the agent state is made as available in Fusion also.

Agent clicks the toggle Agent Availability button on the media toolbar

First the agent clicks the Agent Availability button on the media toolbar to become available.

Media toolbar application starts a web socket connection to your Java microservice

The microservice publishes events through a web socket. You need to initialize this web socket in your media toolbar application as your first step. This can be done when the agent selects the Agent Availability button in your application. Here are some examples:

Define static variables in the `vendorHandler.ts` file for keeping the microservice endpoint URLs as shown in the following example:

```
export class VendorHandler implements ICtiVendorHandler {
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
}
```

You can initialize the web socket from `makeAgentAvailable` function in your `vendorHandler.ts` file as shown in the following example:

```
public async makeAgentAvailable(): Promise<void> {
 let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
 webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
 webSocket.onmessage = this.webSocketOnMessage.bind(this);
 webSocket.onclose = this.webSocketCloseHandler.bind(this);
 webSocket.onerror = this.webSocketErrorHandler.bind(this);
 }
```

Define `webSocketOnOpenHandler`, `webSocketCloseHandler`, `webSocketErrorHandler` and `webSocketOnMessage` functions as:

```
public async webSocketOnOpenHandler(): Promise<void> {
```

```
 console.log("WebSocket opened");
}
public webSocketErrorHandler(error): void {
 console.log("WebSocket error", error);
}
public webSocketCloseHandler(event: Event): void {
 console.log("WebSocket is closed", event);
}
public webSocketOnMessage(event: MessageEvent): void {
 // webSocketOnMessage function acts as the listener for events published from the Java Microservice.
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready through the Java microservice
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
}
```

The `webSocketOnMessage` function acts as the listener for events published from the Java microservice.

The microservice accepts the connection and listens for web socket messages and notifies your microservice to register an agent

After making a successful web socket connection from your media toolbar application, you need to notify the microservice to register an agent. This can be done from the `webSocketOnOpenHandler` function. Update your `webSocketOnOpenHandler` function as shown below to notify your microservice to register the agent by firing a REST API call:

```
public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "initialize"
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
}
```

Microservice creates request for registering an agent through PSDK and notifies that the Agent is registered

The microservice uses `PSDK` for creating requests. After the request for registering an agent succeeds, the Genesys server responds with the `EventRegistered` event which propagates to your media toolbar application through the microservice.

Microservice receives EventRegistered message

Once the request for registering the DN succeeds the Genesys server responds with the `EventRegistered` message which is propagated to your media toolbar application through the microservice.

Invoke agentStateEvent from media toolbar application

When the `EventRegistered` event is received through web socket, the `makeAgentAvailable` function in started in the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
 }
```

Agent state is enabled in the Fusion application

You'll see the agent state as enabled in the Fusion application.

Complete code

Here's the complete code of the `vendorHandler.ts` file for making the agent status as available.

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "initialize"
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
 public webSocketErrorHandler(error: any): void {
 console.log("WebSocket error", error);
```

```
}
public webSocketCloseHandler(event: Event): void {
console.log("WebSocket is closed", event);
}

public webSocketOnMessage(event: MessageEvent): void {
const jsonMessage = JSON.parse(event.data);
console.log(jsonMessage);
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
}
console.log("Message is received");
}
public async makeAgentAvailable(): Promise<void> {
let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
webSocket.onmessage = this.webSocketOnMessage.bind(this);
webSocket.onclose = this.webSocketCloseHandler.bind(this);
webSocket.onerror = this.webSocketErrorHandler.bind(this);
}

public async makeAgentUnavailable(): Promise<void> {
// TODO: call the vendor specific api to make the agent available
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
// TODO: call the vendor specific api to make the make an outbound call
}
public async acceptCall(): Promise<void> {
// TODO: call the vendor specific api to accept a call
}
public async rejectCall(): Promise<void> {
// TODO: call the vendor specific api to reject a call
}
public async hangupCall(): Promise<void> {
// TODO: call the vendor specific api to hangup a call
}
}
```

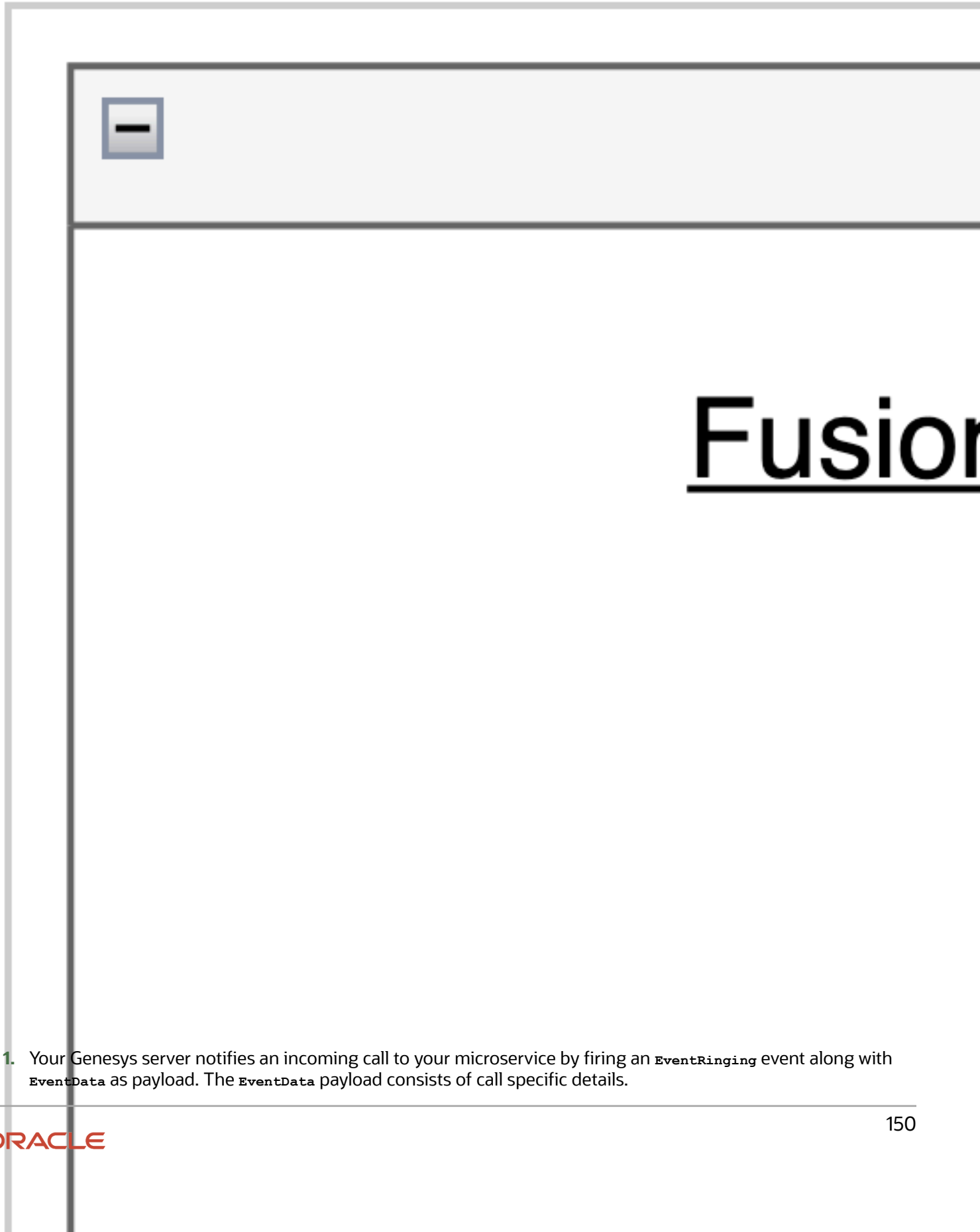
Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the Agent Availability button from your media toolbar application. You'll see that the button color changes to and the phone icon status in the Fusion application is changed to Available.

Show incoming call notification in Genesys

our media toolbar application listens to different events published from the microservice.

The following flow diagram shows the sequence of operations performed when a customer dials your service center phone number from their phone:



1. Your Genesys server notifies an incoming call to your microservice by firing an `EventRinging` event along with `EventData` as payload. The `EventData` payload consists of call specific details.

2. The web socket `onmessge` listener in your toolbar application receives this `EventRinging` event propagated from the microservice along with the EventData payload.
3. The newComm event is fired from your toolbar application.
4. When the Fusion application identifies this action, it performs the contact number lookup and returns the contact details in the action response.

   To configure the default reverse look up to suit your requirements, see *Create Lookup Filters*.
5. Finally the Fusion application notifies the agent about the incoming call offer notification through a dialog box, and the agent can answer or decline the call.

Genesys server notifies an incoming call to your microservice by firing an EventRinging message

When an incoming call is received by Genesys, an `EventRinging` message is propagated to your media toolbar application through the microservice. The message will be having an `EventData` payload which consists of call specific details.

EventRinging message is received in your media toolbar application

The `EventRinging`g message is received by your web socket listener in your media toolbar application.

Call the newCommEvent action from media toolbar application

When the `EventRinging` message is received through web socket, the `incomingCallHandler` function is called from the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
 }
```

The Fusion application performs the contact number lookup

Once the `newCommEvent` action is received by the Fusion application, it performs the contact number lookup and returns the contact details in the action response. You can render your media toolbar application UI with the returned data.

Incoming call offer notification is shown in Fusion

On receiving the `newCommEvent`, the Fusion application also shows the incoming call offer notification in the UI.

Agent accepts or rejects the call

From the incoming call offer notification, agent can either answer or reject the call.

Complete code

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 public static connectionId: string;
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "initialize"
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
 public webSocketErrorHandler(error: any): void {
 console.log("WebSocket error", error);
 }
 public webSocketCloseHandler(event: Event): void {
 console.log("WebSocket is closed", event);
 }

 public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
 }
 public async makeAgentAvailable(): Promise<void> {
 let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
 webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
 webSocket.onmessage = this.webSocketOnMessage.bind(this);
 webSocket.onclose = this.webSocketCloseHandler.bind(this);
 webSocket.onerror = this.webSocketErrorHandler.bind(this);
 }

 public async makeAgentUnavailable(): Promise<void> {
 // TODO: call the vendor specific api to make the agent available
 }
 public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
```

```
    // TODO: call the vendor specific api to make the make an outbound call
    }
    public async acceptCall(): Promise<void> {
    // TODO: call the vendor specific api to accept a call
    }
    public async rejectCall(): Promise<void> {
    // TODO: call the vendor specific api to reject a call
    }
    public async hangupCall(): Promise<void> {
    // TODO: call the vendor specific api to hangup a call
    }
  }
```

Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the Agent Availability button from your media toolbar application. Now, start a call to your customer care number. Your media toolbar state will be changed to RINGING state, and you'll see the incoming notification in your media toolbar application.

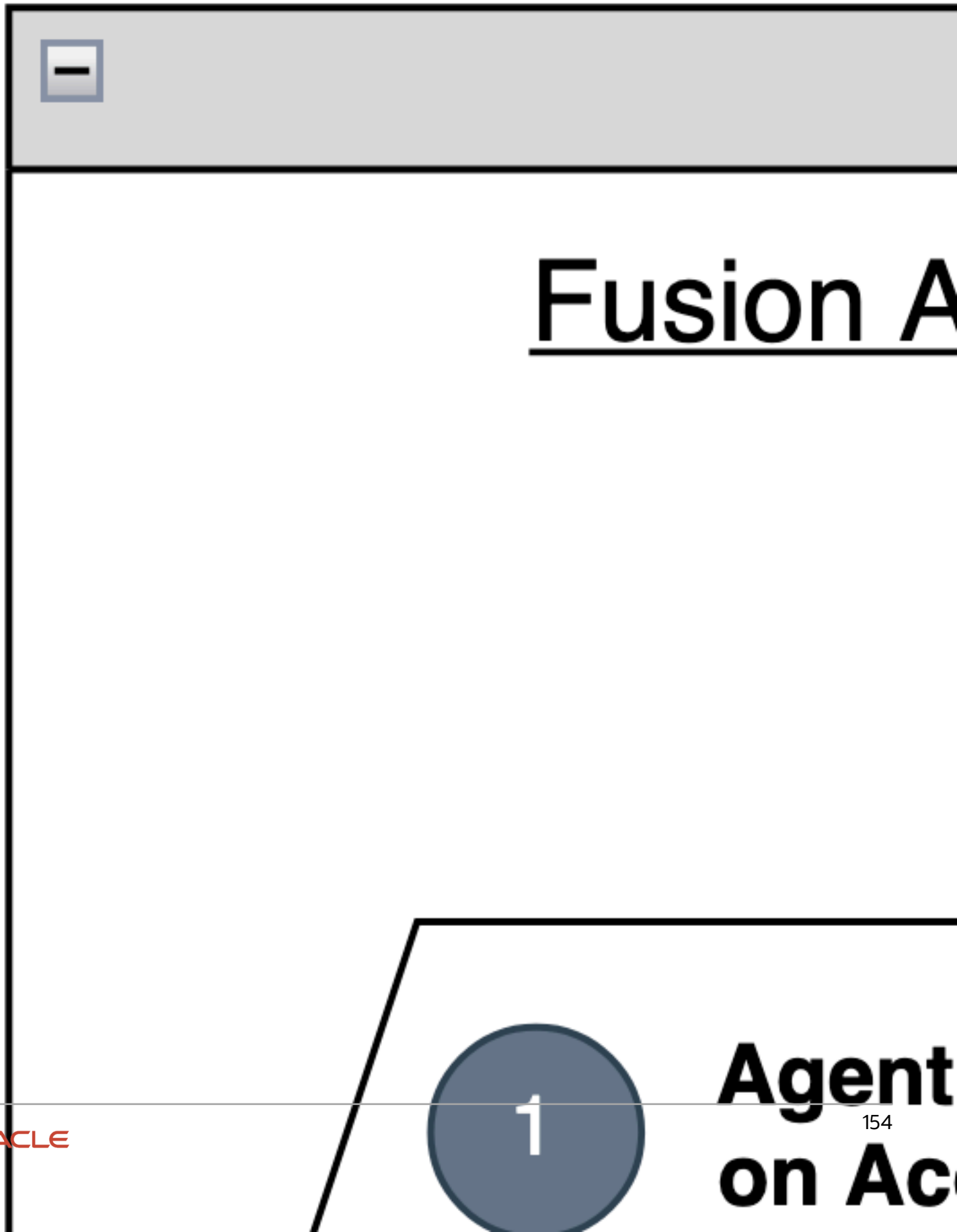
Accept incoming call in Genesys

Your media toolbar application listens to different events published from the microservice.

There are three ways to accept a call:

1. From your media toolbar application.
2. From your Fusion application.
3. From your softphone configured for Genesys.

The following flow diagram shows the sequence of operations performed once an agent accepts an incoming call from media toolbar or from the Fusion application:


Fusion A
1
Agent
on Ac

Scenario 1 and 2: Accept call from media toolbar and Fusion applications

1. An agent can accept the call either from the Fusion application or from the media toolbar application. When agent clicks the **Answer** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `accept`. When the agent accepts the call from media toolbar application,your microservice is notified that the call is accepted.
2. The media toolbar application receives this event and if the event is to accept a call, your microservice is notified to accept the call.
3. When your microservice receives this notification, it requests the Genesys server to accept the call using the `com.genesyslab.platform.voice.protocol.tserver.requests.party.RequestAnswerCall` request.
4. When the `RequestAnswerCall` request is success, the `EventEstablished` message is propagated from Genesys through your microservice to your media toolbar application.
5. The media toolbar application receives the `EventEstablished` message through the web socket and fires the `startCommEvent` action.
6. When the Fusion application identifies this action, the matched contact and an engagement panel are opened based on the rules configured in the fusion configuration management for the screen pop.

Agent accepts the call either from Fusion or from the media toolbar application

The incoming call notification is shown in your media toolbar and your Fusion application. You can answer the call by accepting it from anywhere.

Notify your microservice to accept the call

When the agent answers the call from media toolbar application or from the Fusion application, the microservice is notified that the agent has accepted the call. To enable this, you must update your `acceptCall` function in the `vendorHandler.ts` file as shown in the following example:

```
public async acceptCall(): Promise<void> {
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "Accept",
 "connectionId": VendorHandler.connectionId
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
```

Microservice creates request for accepting the call

The microservice then uses the `PSDK` to create request for answering the call.

Microservice receives EventEstablished message

Once the request for answering the call succeeds, the Genesys server responds with the `EventEstablished` message which is propagated to your media toolbar application through the microservice.

Call the startCommEvent from media toolbar application

When the `EventEstablished` message is received through the web socket, the `callAcceptedHandler` function is called from the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
 }
```

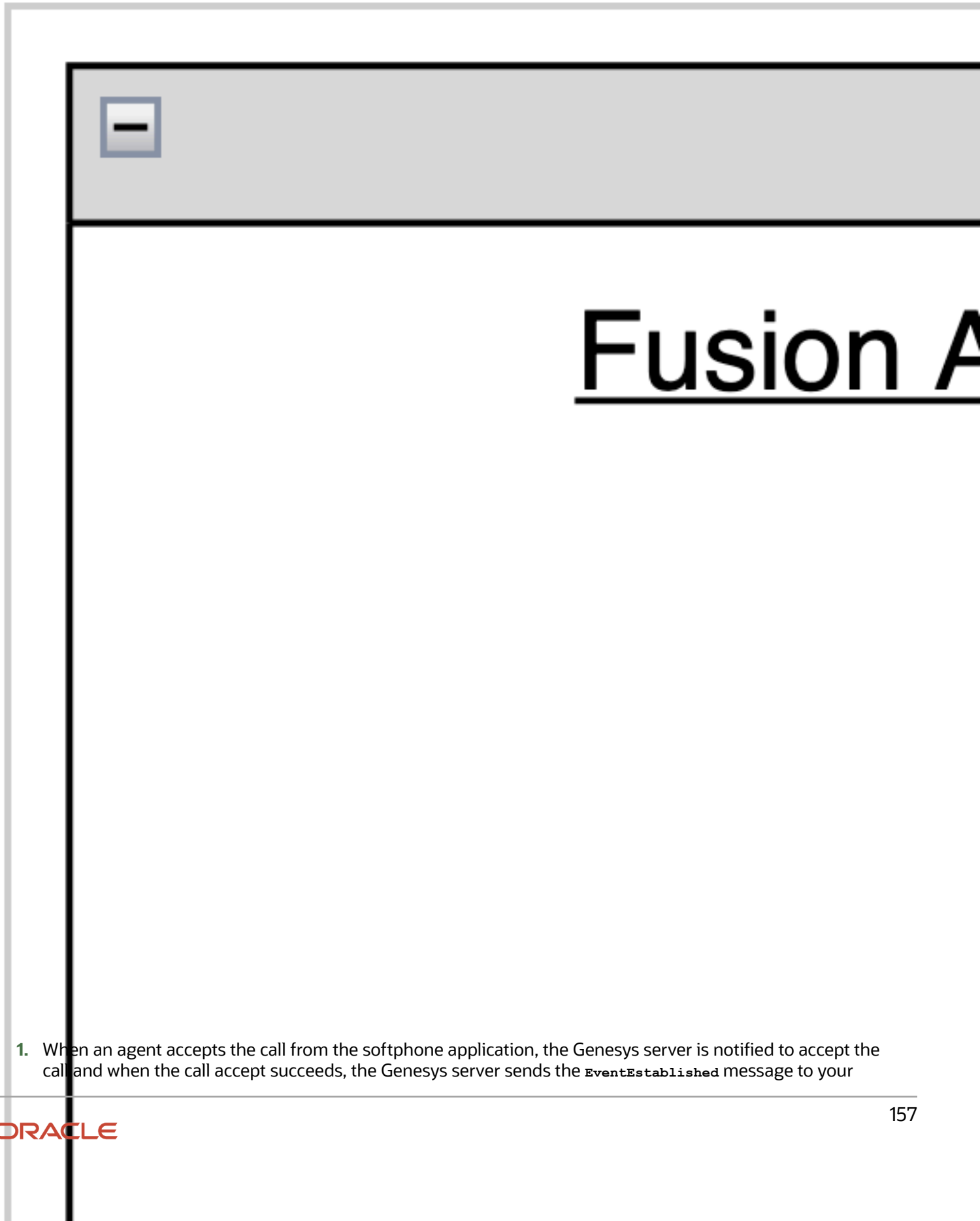
Call is accepted in Fusion application

Your Fusion application shows the call is accepted and the media toolbar application UI updated.

Scenario 3: Accept call from softphone

When the agent accepts the call from softphone, the `EventEstablished` event sends and accepted notification. You can add logic for handling call accepted scenario in the `EventEstablished` event.

The following flow diagram shows the sequence of operations performed once an agent accepts an incoming call from the softphone:



1. When an agent accepts the call from the softphone application, the Genesys server is notified to accept the call and when the call accept succeeds, the Genesys server sends the `EventEstablished` message to your

microservice and your microservice propagates this message to your media toolbar application through the web socket.

2. The media toolbar application receives this event, and fires the `startCommEvent` action.
3. Once the Fusion application identifies this action, the matched contact and an engagement panel will be opened based on the rules configured in the Fusion configuration management for the screen pop.

Microservice propagates EventEstablished message to your media toolbar application

When the agent accepts the call from the softphone application, the Genesys server is notified to accept the call, and once the call is accepted, the Genesys server sends the `EventEstablished` message to your microservice and your microservice propagates this message to your media toolbar application through the web socket.

Call the startCommEvent from media toolbar application

When the `EventEstablished` message is received through the web socket, the `callAcceptedHandler` function is called from the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 }
 console.log("Message is received");
 }
```

Call is accepted in Fusion application

Your Fusion application will show that the call is accepted and the media toolbar application UI will be updated.

Complete Code

Here's the complete code for the `vendorHandler.ts` file for accepting an incoming call.

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 public static connectionId: string;
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
```

```
headers.set('Content-type', 'application/json');
const message: any = {
"type": "initialize"
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public webSocketErrorHandler(error: any): void {
console.log("WebSocket error", error);
}
public webSocketCloseHandler(event: Event): void {
console.log("WebSocket is closed", event);
}

public webSocketOnMessage(event: MessageEvent): void {
const jsonMessage = JSON.parse(event.data);
console.log(jsonMessage);
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
}
console.log("Message is received");
}
public async makeAgentAvailable(): Promise<void> {
let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
webSocket.onmessage = this.webSocketOnMessage.bind(this);
webSocket.onclose = this.webSocketCloseHandler.bind(this);
webSocket.onerror = this.webSocketErrorHandler.bind(this);
}

public async makeAgentUnavailable(): Promise<void> {
// TODO: call the vendor specific api to make the agent available
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
// TODO: call the vendor specific api to make the make an outbound call
}
public async acceptCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Accept",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async rejectCall(): Promise<void> {
```

```
 // TODO: call the vendor specific api to reject a call
 }
 public async hangupCall(): Promise<void> {
 // TODO: call the vendor specific api to hangup a call
 }
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. Then, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion application window. You can accept the call from your media toolbar application or from your Fusion application or from your softphone application. Once you accept the call, your media toolbar state will change to the ACCEPTED state and the engagement is opened in your Fusion application.

Disconnect or reject an incoming call in Genesys

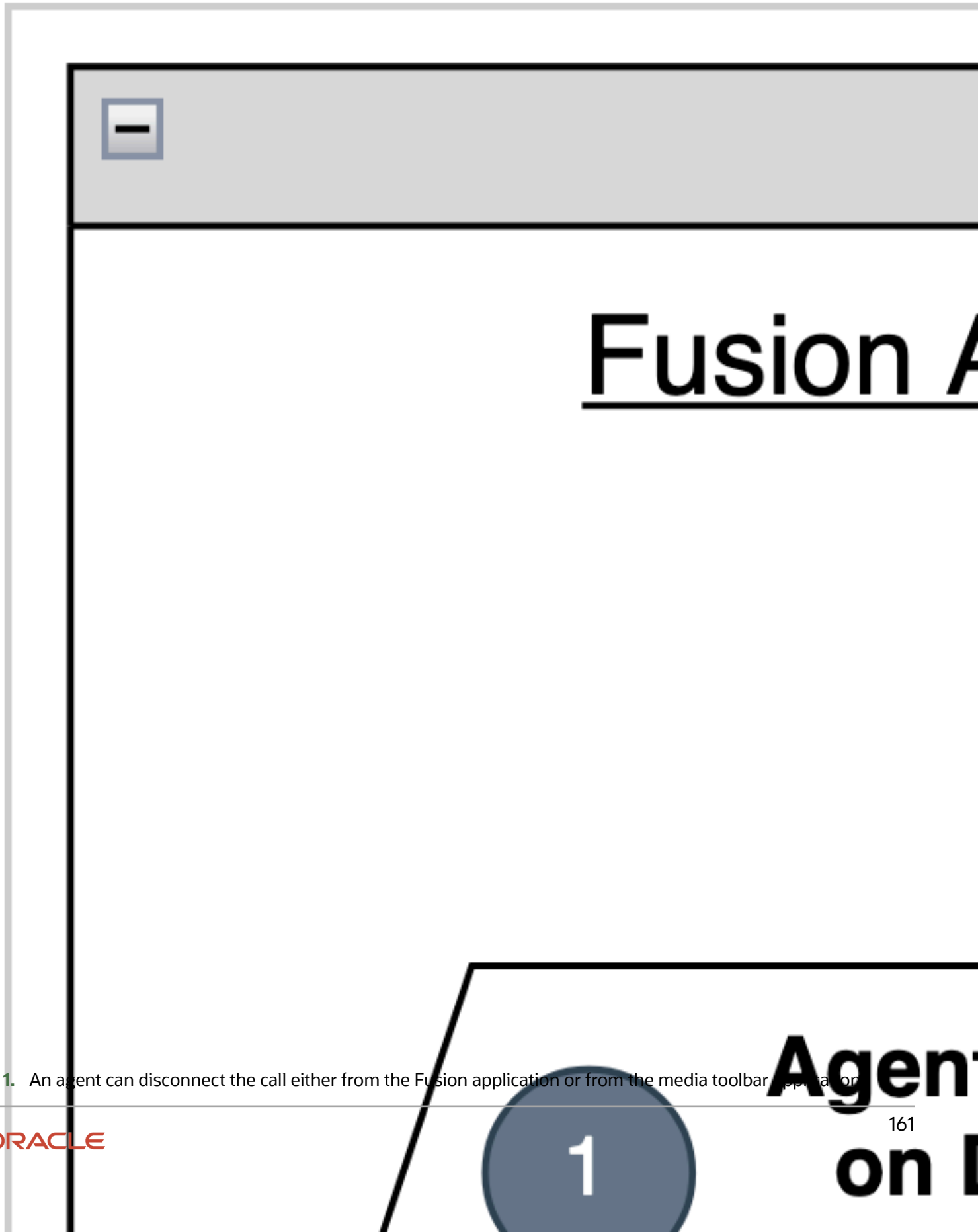
Your media toolbar application listens to different events published from the microservice.

There are three ways to disconnect a call:

1. From your media toolbar application.
2. From your Fusion application.
3. From your soft phone configured for Genesys.

Scenario 1 and 2: Disconnect call from media toolbar and Fusion applications

The following flow diagram shows the sequence of operations performed once an agent disconnects a call from Fusion or from the media toolbar application:



1. An agent can disconnect the call either from the Fusion application or from the media toolbar application.

- When the agent clicks the **Decline** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with command as `reject`.
  When the agent clicks the **End Call** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with command as `wrapup`.
  When the agent declines the call from media toolbar application, your microservice is notified that the call has been declined.
  When the agent ends the call from media toolbar application, your microservice is notified that the call has ended.

2. The media toolbar application receives this event, and if the event is to disconnect a call, your microservice is notified to disconnect the call.
3. When your microservice receives the notification, it should request the Genesys server disconnect the call using the `com.genesyslab.platform.voice.protocol.tserver.requests.party.RequestReleaseCall` request.
4. When the `RequestReleaseCall` request is succeeds, the `EventReleased` message is propagated from Genesys through your microservice to your media toolbar application.
5. The media toolbar application receives the `EventReleased` message through the web socket and fires the `closeCommEvent` action with reason as `reject` if call is declined or `wrapup` if call is ended.
6. When the Fusion application identifies this action, the call dialog box is discarded from the UI if reason is `reject`, or the wrap up window displayed in the UI if reason is `wrapup`.

Agent disconnects the call either from the application Fusion or from the media toolbar application

An agent can disconnect the call either from the Fusion application or from the media toolbar application.

- When agent clicks the **Decline** button in the Fusion application, `onToolbarInteractionCommand` event is fired with command as `reject`.
- When agent clicks the **End Call** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `wrapup`.
- When an agent declines the call from media toolbar application, the microservice is notified that the call has been declined.
- When agent ends the call from media toolbar application, the microservice is notified that the call has ended.

Notify your microservice to disconnect the call

When the agent disconnects the call from media toolbar application or from the Fusion application, you need to notify your microservice that the agent disconnected the call. For that, update your `rejectCall` function in the `vendorHandler.ts` file as shown in the following example:

```
public async rejectCall(): Promise<void> {
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "Reject",
 "connectionId": VendorHandler.connectionId
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
```

```
public async hangupCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
```

Microservice creates request for disconnecting the call

The microservice then uses `PSDK` to create request the call be disconnected.

Microservice receives EventReleased message

After the request for disconnecting the call succeeds, the Genesys server responds with the `withEventReleased` message and it will be propagated to your media toolbar application through the microservice.

Call the closeCommEvent from media toolbar application

When the `EventReleased` message is received through the web socket, the `callHangupHandler` function is called from the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
 }
 console.log("Message is received");
 }
```

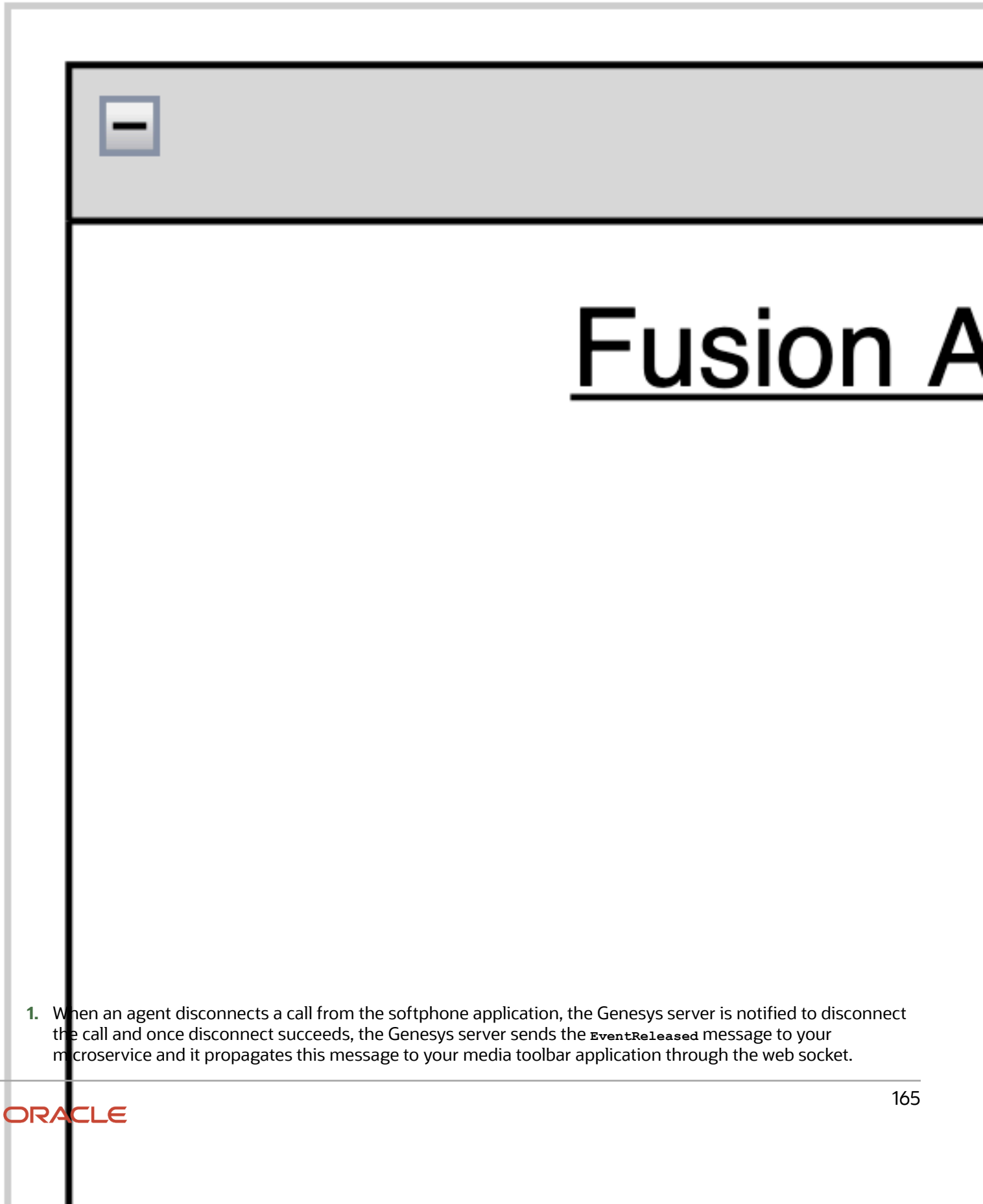
Call is disconnected in the Fusion application

Your Fusion application will remove the incoming call notification if the call is declined or it will show the `WRAPUP`window if the call is ended. The media toolbar application UI will be updated to show that the agent is available for calls.

Scenario 3: Disconnect call from softphone

When the agent accepts the call from a softphone, the `EventEstablished` event sends a notification that the call has been accepted. You can add logic for handling the call accepted scenario in the `EventEstablished` event as shown in the

following diagram. The flow diagram shows the sequence of operations performed once an agent accepts an incoming call from the softphone:



1. When an agent disconnects a call from the softphone application, the Genesys server is notified to disconnect the call and once disconnect succeeds, the Genesys server sends the `EventReleased` message to your microservice and it propagates this message to your media toolbar application through the web socket.

2. The media toolbar application receives this event, and fires the `closeCommEvent` action.
3. After the Fusion application identifies the action, the call dialog box will be discarded from the UI if the reason is `reject` or the `WRAPUP` window if the call is ended.

Microservice propagates EventReleased message to your media toolbar application

When the agent disconnects a call from the softphone application, the Genesys server is notified to disconnect the call and after the call is disconnected, the Genesys server sends the `EventReleased` message to your microservice and it propagates this message to your media toolbar application through the web socket.

Call the closeCommEvent from media toolbar application

When the `EventReleased` message is received through web socket, `callHangupHandler` function is started in the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
 if (jsonMessage.eventName === "EventRegistered") {
 // Genesys notifies that the agent is ready
 this.integrationEventsHandler.makeAgentAvailable();
 } else if (jsonMessage.eventName === "EventRinging") {
 // Show incoming call notification
 this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
 VendorHandler.connectionId = jsonMessage.connectionId;
 } else if (jsonMessage.eventName === "EventEstablished") {
 // Genesys notifies that the call is accepted
 this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
 } else if (jsonMessage.eventName === "EventReleased") {
 // Genesys notifies that the call is disconnected
 this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
 }
 console.log("Message is received");
 }
```

Call is disconnected in the Fusion application

You Fusion application will remove the incoming call notification if the call is declined or it will show the `WRAPUP` window if the call is ended. The media toolbar application UI will be updated to show that the agent is available for calls.

Complete code

Here's the complete code for the `vendorHandler.ts` file for disconnecting a call.

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 public static connectionId: string;
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
```

```
headers.set('Content-type', 'application/json');
const message: any = {
"type": "initialize"
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public webSocketErrorHandler(error: any): void {
console.log("WebSocket error", error);
}
public webSocketCloseHandler(event: Event): void {
console.log("WebSocket is closed", event);
}

public webSocketOnMessage(event: MessageEvent): void {
const jsonMessage = JSON.parse(event.data);
console.log(jsonMessage);
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
}
console.log("Message is received");
}
public async makeAgentAvailable(): Promise<void> {
let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
webSocket.onmessage = this.webSocketOnMessage.bind(this);
webSocket.onclose = this.webSocketCloseHandler.bind(this);
webSocket.onerror = this.webSocketErrorHandler.bind(this);
}

public async makeAgentUnavailable(): Promise<void> {
// TODO: call the vendor specific api to make the agent available
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
// TODO: call the vendor specific api to make the make an outbound call
}
public async acceptCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Accept",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
```

```
public async rejectCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}

public async hangupCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
}
```

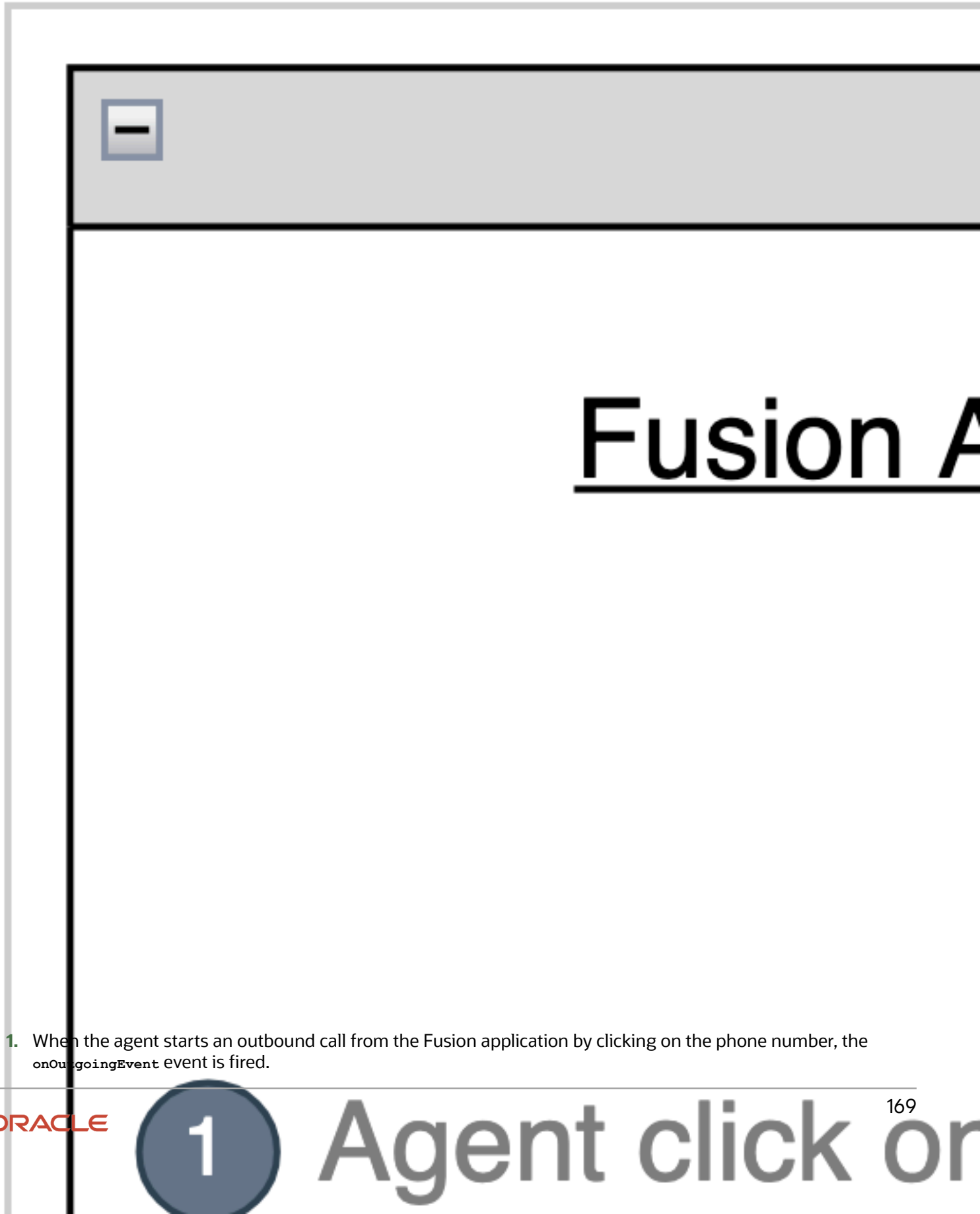
Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. Then, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion application window. You can accept the call from your media toolbar application or from your Fusion application or from your softphone application. Once you accept the call, your media toolbar state will change to the ACCEPTED state and the engagement is opened in your Fusion application. You can disconnect the call from your media toolbar application or from your Fusion application or from the softphone. Once you disconnect the call, your media toolbar state will be changed to the DISCONNECTED state.

Make an outbound call in Genesys

The following flow diagram shows the sequence of operations performed once an agent starts an outbound call from the Fusion application:

The following flow diagram shows the sequence of operations performed once an agent starts an outbound call from the Fusion application:



1. When the agent starts an outbound call from the Fusion application by clicking on the phone number, the `onOutgoingEvent` event is fired.

2. The `onOutgoingEvent` event listener in your media toolbar application receives this event and notifies your microservice to start an outbound call.
3. Your microservice starts the call by creating request for making a call using `PSDK`.
4. If the request is successful, the `EventRinging` event is fired from Genesys and propagated to your media toolbar application through the microservice.
5. When the `RequestMakeCall` request succeeds, the `EventRinging` message is propagated from Genesys through your microservice to your media toolbar application.
6. When the `EventEstablished` message is received by the media toolbar application, the `newCommEvent` action notifies the Fusion application.
7. The Fusion application show the dialing panel when the `newCommEvent` action is received.

Agent starts an outbound call from fusion application

Agent starts an outbound call from the Fusion application by clicking the phone number, and the `onOutgoingEvent` event is fired.

Notify Genesys to start a call

The `onOutgoingEvent` event listener in your media toolbar application receives this event and notifies your microservice to start an outbound call. You can update the `makeOutboundCall` function as shown in the following example to notify your microservice to start a call:

```
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "makeCall",
 "toNumber": phoneNumber,
 "connectionId": VendorHandler.connectionId
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
```

Microservice creates request for starting a call

The microservice uses `PSDK` for creating requests for making a call using `com.genesyslab.platform.voice.protocol.tserver.requests.party.RequestMakeCall` request class. If the request succeeds, the `EventRinging` message is received, and when the customer responds, the `EventEstablished` message will be received by your microservice.

Microservice receives EventRinging message

Once the request to start the outbound call succeeds, the Genesys server responds with the `EventRinging` message and it propagated it to your media toolbar application through the microservice.

Call newCommEvent from media toolbar application

Once you receive the `EventRinging` message from your microservice, you can call the `newCommEvent` action by executing the function `incomingCallHandler` in the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
```

```
console.log(jsonMessage);
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
}
console.log("Message is received");
}
```

The Fusion application displays the dialing panel

The Fusion application shows the dialing notification once the `newCommEvent` action is received.

Complete code

Here's the complete code of of the `vendorHandler.ts` file for starting an outbound call:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 public static connectionId: string;
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "initialize"
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
 public webSocketErrorHandler(error: any): void {
 console.log("WebSocket error", error);
 }
 public webSocketCloseHandler(event: Event): void {
 console.log("WebSocket is closed", event);
 }

 public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
```

```
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
}
console.log("Message is received");
}
public async makeAgentAvailable(): Promise<void> {
let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
webSocket.onmessage = this.webSocketOnMessage.bind(this);
webSocket.onclose = this.webSocketCloseHandler.bind(this);
webSocket.onerror = this.webSocketErrorHandler.bind(this);
}

public async makeAgentUnavailable(): Promise<void> {
// TODO: call the vendor specific api to make the agent available
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "makeCall",
"toNumber": phoneNumber,
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async acceptCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Accept",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async rejectCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
```

```
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async hangupCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. To start an outbound call, open the contact from your Fusion application and click the phone number, which starts the outbound call.

Amazon

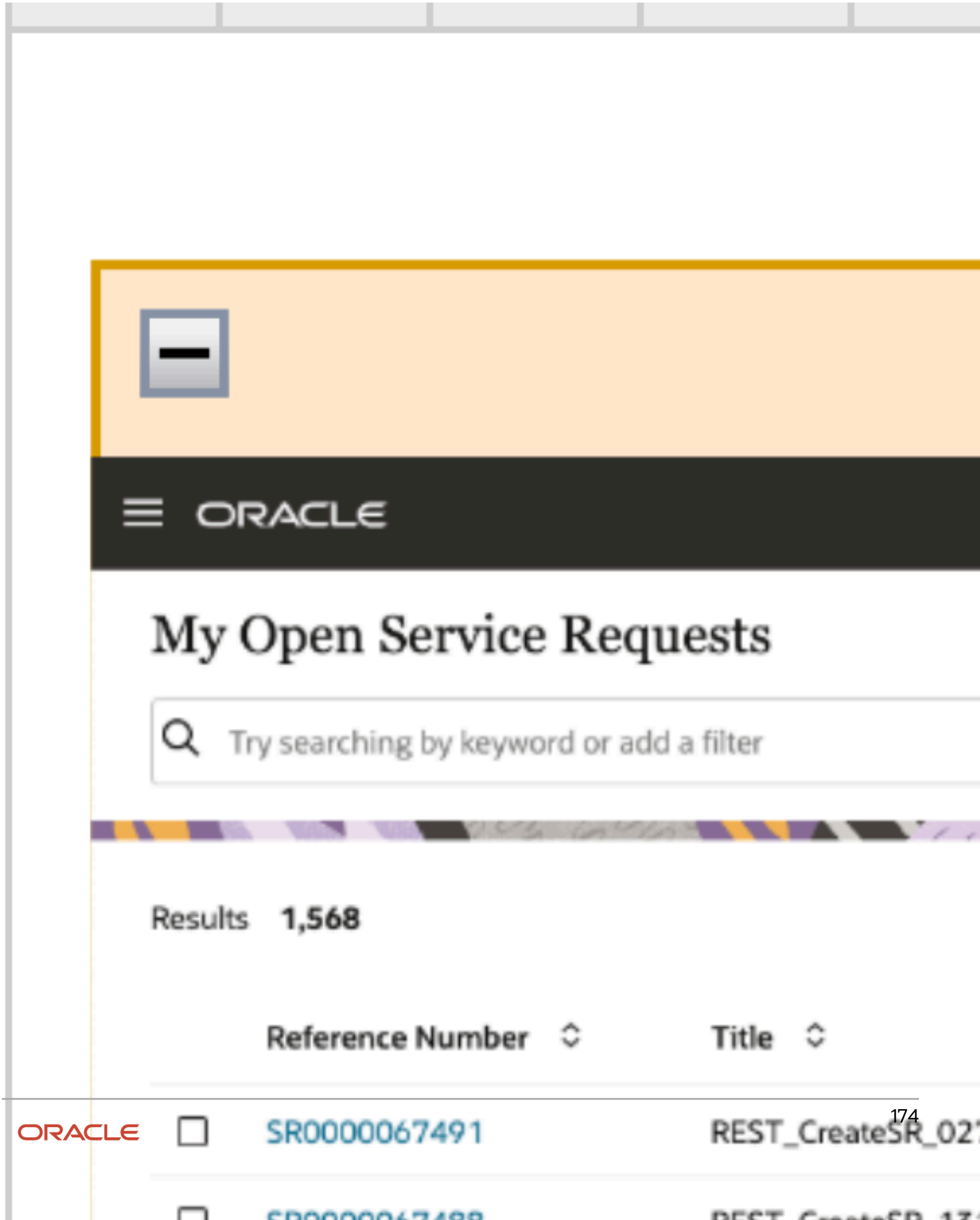
Overview of Amazon integration

The Amazon Connect Streams API (Streams) enables you to integrate your CTI web applications with Amazon Connect. Streams.

You can also embed the Contact Control Panel (CCP) and Customer Profiles app UI into your page. You can also handle agent and contact state events directly through an object oriented event driven UI. You can use the built in UI or build your own.

> **Note:** This isn't a ready to use means to integrate with Amazon Connect. Well just walk through a sample implementation. This implementation isn't scaled for production use cases and must not be considered as a production ready implementation.

Here's an overview of the components:


ORACLE
My Open Service Requests
Try searching by keyword or add a filter
Results 1,568
Reference Number
Title
SR0000067491
REST_CreateSR_02

Configure Amazon Connect

To use Amazon connect you must create an Amazon Web Services (AWS) account and use the Amazon Connect service.

Note: At this time, Amazon Connect service is not supported for accounts created from India.

You can have multiple instances of Amazon Connect. Each instance contains all the resources related to your contact center, such as phone numbers, agent accounts, and queues.

To configure Amazon Connect, see the Amazon set up documentation.

Configure Amazon Connect Streams API Dependency

The Amazon Connect Streams API (Streams) enables you to integrate your existing web applications with Amazon Connect.

Use Streams to embed the Contact Control Panel (CCP) and Customer Profiles app UI into your page. It also enables you to handle agent and contact state events directly through an object oriented event driven UI. You can use the built in UI or build your own from scratch.

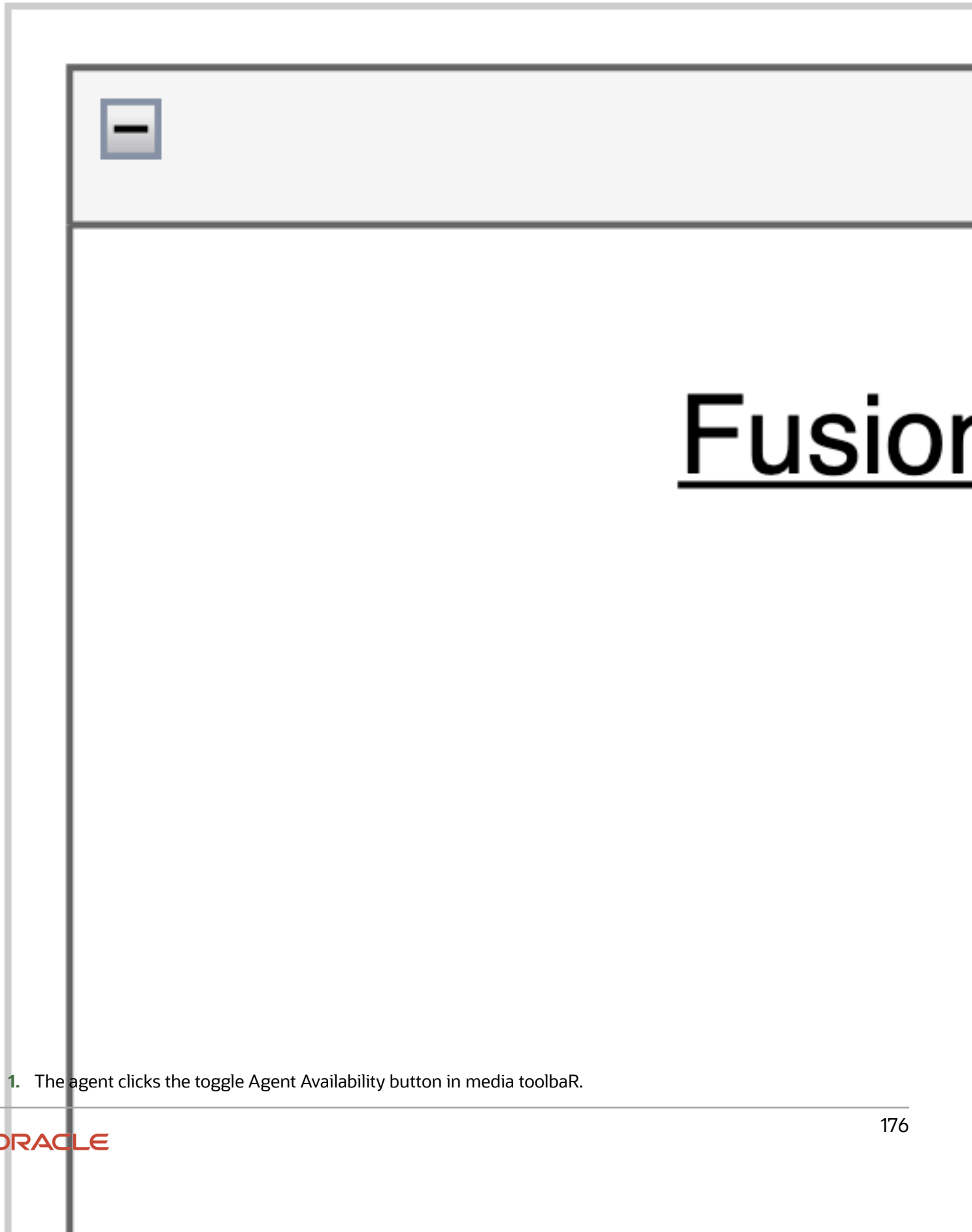
To configure Streams, see the Amazon set up documentation.

Set up Toggle Agent Availability in Amazon

To make an agent available and unavailable, we need to use the `intccp` method in the core object in the Amazon Connect stream API.

Initializing the `streams` API is the first step to verify that you have everything setup correctly and that you can listen for events.

The following flow diagram shows the sequence of operations performed once an agent marks themselves as available by clicking the toggle Agent Availability button in the media toolbar application:



1. The agent clicks the toggle Agent Availability button in media toolbaR.

2. Initialize the Amazon Connect CCP and call the `setState` function of agent object in Amazon Connect `Streams` API.
3. Amazon Connect updates the agent state and resolves the promise.
4. From the media toolbar fire the `agentStateEventOperation` to update the agent status in the Fusion application.
5. Agent state is updated in the Fusion application.

Update VendorHandler Class to call the Amazon Connect Streams API to toggle agent available

1. Add the import for amazon-connect-streams: `import "amazon-connect-streams";`
2. Add the following properties and update the connect property in the constructor as shown in the following example:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import "amazon-connect-streams";
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private connect: any;
 private agent: any;
 private contact: any;
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 this.connect = (window as any)["connect"];
 }
//....
}
```

3. Create a method in the `vendorHandler` class `src/ts/cti/vendor/vendorHandler.ts` with `codeInitializing` the `Streams` API is the first step to verify that you have everything setup correctly and that can listen for events.

```
// Intialize Amazon connect ccp
private init() {
 const containerDiv = document.getElementById("amazon-connect-cca-container");
 this.connect.core.initCCP(containerDiv, {
 ccpUrl: 'https://cti-amazon-connect-demo.my.connect.aws/ccp-v2', // REQUIRED
 loginPopup: true, // optional, defaults to `true`
 loginPopupAutoClose: true, // optional, defaults to `false`
 loginOptions: {
 // optional, if provided opens login in new window
 autoClose: true, // optional, defaults to `false`
 height: 600, // optional, defaults to 578
 width: 400, // optional, defaults to 433
 top: 0, // optional, defaults to 0
 left: 0, // optional, defaults to 0
 },
 region: "eu-west-2", // REQUIRED for `CHAT`, optional otherwise
 softphone: {
 // optional, defaults below apply if not provided
 allowFramedSoftphone: true, // optional, defaults to false
 disableRingtone: false, // optional, defaults to false
 },
 pageOptions: {
 //optional
 enableAudioDeviceSettings: false, //optional, defaults to 'false'
 enableVideoDeviceSettings: false, //optional, defaults to 'false'
 enablePhoneTypeSettings: true, //optional, defaults to 'true'
 },
 ccpAckTimeout: 5000, //optional, defaults to 3000 (ms)
```

```
 ccpSynTimeout: 3000, //optional, defaults to 1000 (ms)
 ccpLoadTimeout: 10000, //optional, defaults to 5000 (ms)
 });
}
```

4. Add a method to initialize `agentSubscribe`, a method that's called when the agent is initialized. If the agent has already been initialized, the call is synchronous and the callback is called immediately. Otherwise, the callback is made when the first agent data is received from upstream. This callback is provided with an `Agent` API object, which can also be created at any time after initialization is complete using `new connect.Agent()`.

```
private subscribeForAmazonEvents() {
 this.connect.agent((agent: any) => {
 this.agent = agent;
 let state = this.agent.getAgentStates()[0];
 this.agent.setState(state);
 });
}
```

5. For updating the agent's state use the `setState` method of the agent. Update the `makeAgentAvailable` method as shown in the following example:

```
public async makeAgentAvailable(): Promise<void> {
 if (!this.agent) {
 this.init();
 this.subscribeForAmazonEvents();
 } else {
 let state = this.agent.getAgentStates()[0];
 this.agent.setState(state);
 }
}
```

6. Update the `makeAgentUnavailable` method as shown in the following example:

```
public async makeAgentUnavailable(): Promise<void> {
 let state = this.agent.getAgentStates()[1];
 this.agent.setState(state);
}
```

Add a div for loading the Contact Control Panel

```
<div id="amazon-connect-cca-container" style="height: 400px; width: 300"></div>
```

On initializing the `Stream` API the Amazon Contact Control Panel (CCP) will be embedded in a iFrame inside the div. In this example, we're using the CCP instead of the custom call panel component, so we can hide the call panel by adding the display none style on the call-panel component or comment out the call-panel in index.html as shown in the following example:

```
<!--
<oj-bind-if test="[[callContext().state === 'RINGING' || callContext().state === 'ACCEPTED']]">
 <call-panel call-context="[[callContext]]"
 on-accept-button-clicked="[[ callAcceptedEventHandler ]]"
 on-disconnect-button-clicked="[[ callDisconnectedEventHandler ]]"></call-panel>
</oj-bind-if>
-->
```

Complete Code

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import "amazon-connect-streams";
import { IntegrationEventsHandler } from '../integrationEventsHandler';
```

```
export class VendorHandler implements ICtiVendorHandler {
 private connect: any;
 private agent: any;
 private contact: any;
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 this.connect = (window as any)["connect"];
 }
 public async makeAgentAvailable(): Promise<void> {
 if (!this.agent) {
 this.init();
 this.subscribeForAmazonEvents();
 } else {
 let state = this.agent.getAgentStates()[0];
 this.agent.setState(state);
 }
 }
 public async makeAgentUnavailable(): Promise<void> {
 let state = this.agent.getAgentStates()[1];
 this.agent.setState(state);
 }
 public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
 }
 public async acceptCall(): Promise<void> {
 }
 public async rejectCall(): Promise<void> {
 }
 public async hangupCall(): Promise<void> {
 }

 // Intialize Amazon connect ccp
 private init() {
 const containerDiv = document.getElementById("amazon-connect-cca-container");
 this.connect.core.initCCP(containerDiv, {
 ccpUrl: 'https://cti-amazon-connect-demo.my.connect.aws/ccp-v2', // REQUIRED
 loginPopup: true, // optional, defaults to `true`
 loginPopupAutoClose: true, // optional, defaults to `false`
 loginOptions: {
 // optional, if provided opens login in new window
 autoClose: true, // optional, defaults to `false`
 height: 600, // optional, defaults to 578
 width: 400, // optional, defaults to 433
 top: 0, // optional, defaults to 0
 left: 0, // optional, defaults to 0
 },
 region: "eu-west-2", // REQUIRED for `CHAT`, optional otherwise
 softphone: {
 // optional, defaults below apply if not provided
 allowFramedSoftphone: true, // optional, defaults to false
 disableRingtone: false, // optional, defaults to false
 },
 pageOptions: {
 //optional
 enableAudioDeviceSettings: false, //optional, defaults to 'false'
 enableVideoDeviceSettings: false, //optional, defaults to 'false'
 enablePhoneTypeSettings: true, //optional, defaults to 'true'
 },
 ccpAckTimeout: 5000, //optional, defaults to 3000 (ms)
 ccpSynTimeout: 3000, //optional, defaults to 1000 (ms)
 ccpLoadTimeout: 10000, //optional, defaults to 5000 (ms)
 });
 }
```

```
  private subscribeForAmazonEvents() {
  this.connect.agent((agent: any) => {
  this.agent = agent;
  let state = this.agent.getAgentStates()[0];
  this.agent.setState(state);
  });
  }
 }
```

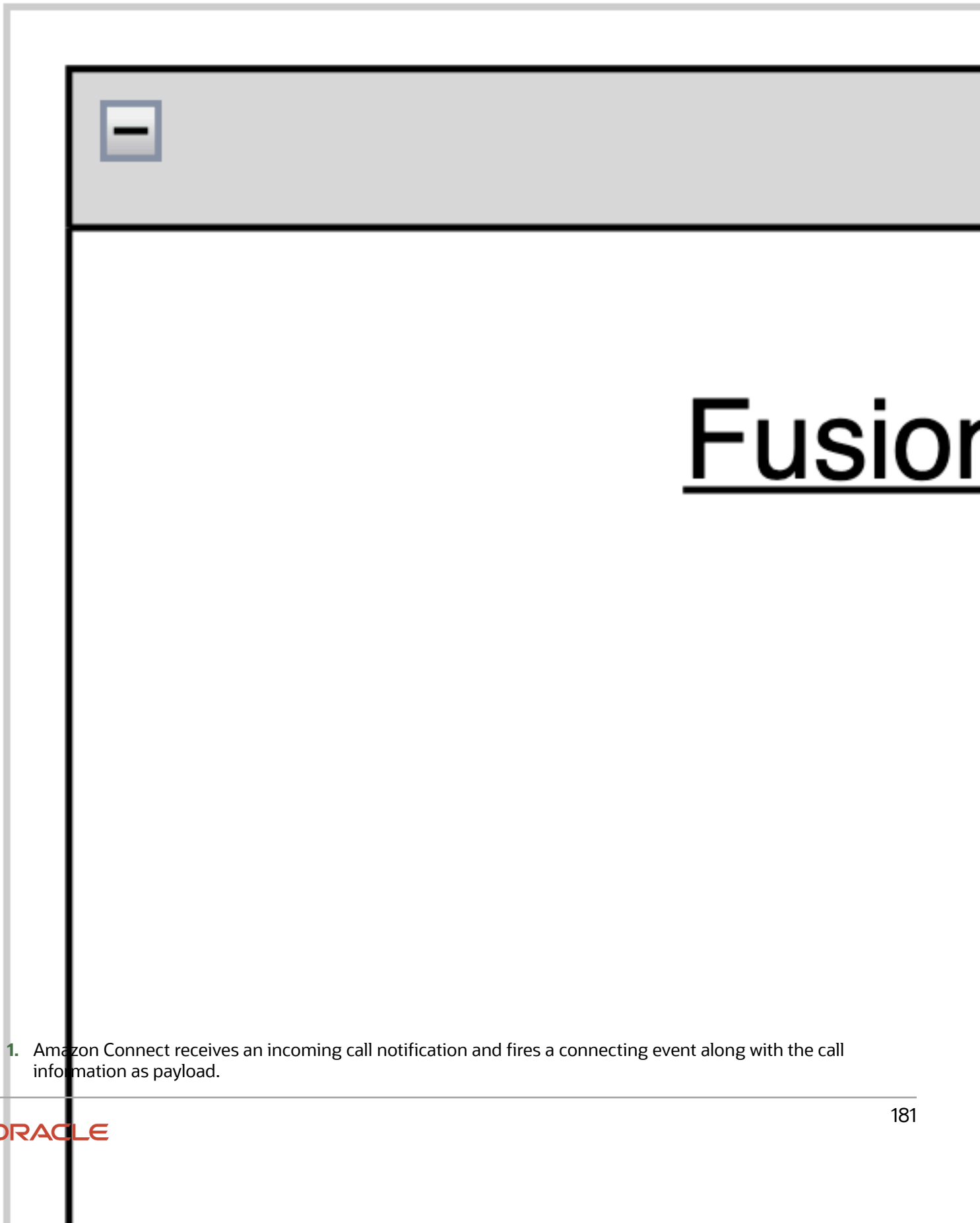
Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the Agent Availability button from your media toolbar application. You'll see, the button change colors and the phone icon status in the Fusion application changed to Available.

Show incoming call notification in Amazon

The `Contact` API provides event subscription methods and action methods which can be called on behalf of a specific contact. From your media toolbar application, you need to listen to `connecting event` and do the necessary steps once this event is received.

The following flow diagram shows the sequence of operations performed once a customer dials your service center phone number from their phone:



1. Amazon Connect receives an incoming call notification and fires a connecting event along with the call information as payload.

2. The connecting event listener on the contact object of `Amazon Connect Streams` API in your toolbar application receives this event along with the `call` payload.
3. The `newComm` is fired from your toolbar application.
4. Once the Fusion application identifies this action, it performs the contact number lookup and returns the contact details in the action response.

   To configure the default reverse look up to suit your requirements, see *Create Lookup Filters*.
5. Finally, the Fusion application notifies the agent about the incoming call offer notification through a dialog box.
6. The agent can either Answer or Decline the call.

Update VendorHandler Class to subscribe the Amazon Connect Streams API events

Add the following code in the `makeAgentAvailable` function in the `vendorHandler.ts` file and add the `eventId` property on the `VendorHandler` class.

```
private eventId: string = '';

private subscribeContactEvents() {
 this.connect.contact((contact: any) => {
 this.contact = contact;
 this.eventId = contact.contactId;
 contact.onConnecting((contact: any) => {
 console.log("Contact onConnecting >", contact);
 const phoneNumber = contact.getActiveInitialConnection().getEndpoint().phoneNumber;
 this.integrationEventsHandler.incomingCallHandler(phoneNumber, this.eventId);
 });

 contact.onAccepted((contact: any) => {
 console.log("Contact onAccepted >", contact);
 });

 contact.onEnded(async (contact: any) => {
 console.log("Contact onEnded >", contact);
 });

 contact.onMissed((contact: any) => {
 console.log("Contact onMissed >", contact);
 });
 });
}
```

Complete Code

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import "amazon-connect-streams";
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private connect: any;
 private agent: any;
 private contact: any;
 private integrationEventsHandler: IntegrationEventsHandler;
 private eventId: string = '';

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 this.connect = (window as any)["connect"];
 }
 public async makeAgentAvailable(): Promise<void> {
 if (!this.agent) {
 this.init();
```

```
this.subscribeForAmazonEvents();
this.subscribeContactEvents();
} else {
let state = this.agent.getAgentStates()[0];
this.agent.setState(state);
}
}
public async makeAgentUnavailable(): Promise<void> {
let state = this.agent.getAgentStates()[1];
this.agent.setState(state);
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
}
public async acceptCall(): Promise<void> {
}
public async rejectCall(): Promise<void> {
}
public async hangupCall(): Promise<void> {
}

// Intialize Amazon connect ccp
private init() {
const containerDiv = document.getElementById("amazon-connect-cca-container");
this.connect.core.initCCP(containerDiv, {
ccpUrl: 'https://cti-amazon-connect-demo.my.connect.aws/ccp-v2', // REQUIRED
loginPopup: true, // optional, defaults to `true`
loginPopupAutoClose: true, // optional, defaults to `false`
loginOptions: {
// optional, if provided opens login in new window
autoClose: true, // optional, defaults to `false`
height: 600, // optional, defaults to 578
width: 400, // optional, defaults to 433
top: 0, // optional, defaults to 0
left: 0, // optional, defaults to 0
},
region: "eu-west-2", // REQUIRED for `CHAT`, optional otherwise
softphone: {
// optional, defaults below apply if not provided
allowFramedSoftphone: true, // optional, defaults to false
disableRingtone: false, // optional, defaults to false
},
pageOptions: {
//optional
enableAudioDeviceSettings: false, //optional, defaults to 'false'
enableVideoDeviceSettings: false, //optional, defaults to 'false'
enablePhoneTypeSettings: true, //optional, defaults to 'true'
},
ccpAckTimeout: 5000, //optional, defaults to 3000 (ms)
ccpSynTimeout: 3000, //optional, defaults to 1000 (ms)
ccpLoadTimeout: 10000, //optional, defaults to 5000 (ms)
});
}

private subscribeForAmazonEvents() {
this.connect.agent((agent: any) => {
this.agent = agent;
let state = this.agent.getAgentStates()[0];
this.agent.setState(state);
});
}

private subscribeContactEvents() {
this.connect.contact((contact: any) => {
this.contact = contact;
this.eventId = contact.contactId;
contact.onConnecting((contact: any) => {
```

```
    console.log("Contact onConnecting >", contact);
    const phoneNumber = contact.getActiveInitialConnection().getEndpoint().phoneNumber;
    this.integrationEventsHandler.incomingCallHandler(phoneNumber, this.eventId);
    });
    contact.onAccepted((contact: any) => {
    console.log("Contact onAccepted >", contact);
    });

    contact.onEnded(async (contact: any) => {
    console.log("Contact onEnded >", contact);
    });

    contact.onMissed((contact: any) => {
    console.log("Contact onMissed >", contact);
    });
    });
    }
  }
```

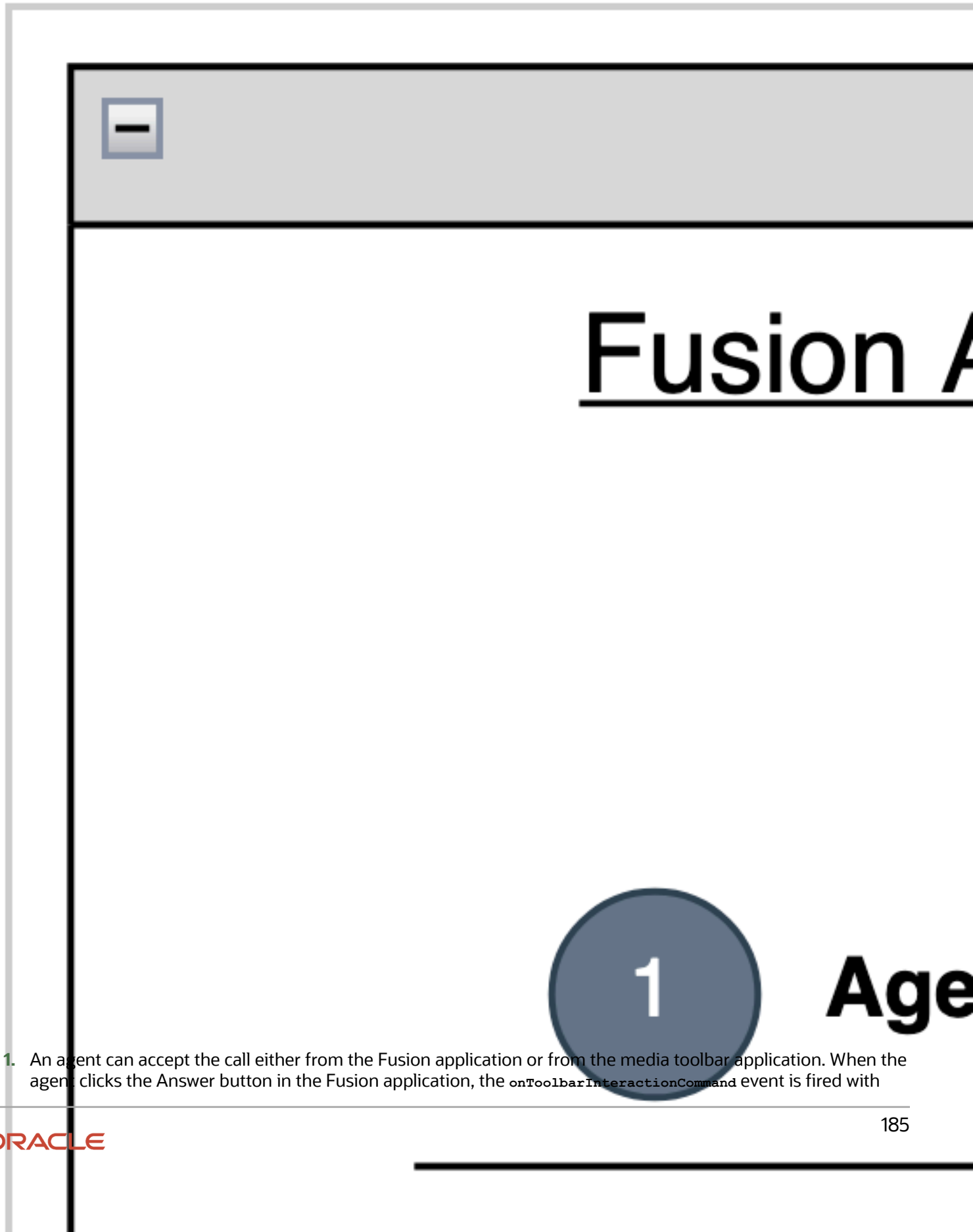
Verify your progress

Sign in to your Fusion application and open the media toolbar. Click the Agent Availability button from your media toolbar application. Make an inbound call and you'll see the incoming call notification on both the Fusion application and the toolbar window.

Accept incoming call in Amazon

All call related operations such as call accept, reject, disconnect, mute and so on are performed on the Connect Streams API. You can use `contact.accept()` function to accept a call.

The following flow diagram shows the sequence of operations performed once an agent accepts the call from the Fusion application or from the media toolbar application:



1. An agent can accept the call either from the Fusion application or from the media toolbar application. When the agent clicks the Answer button in the Fusion application, the `onToolbarInteractionCommand` event is fired with

command of `accept`. When the agent accepts the call from media toolbar application, supplier API `accept` can be called directly.

2. The media toolbar application receives this event and if the event is to accept a call, the `Amazon Connect Streams` API to accept the call is called.
3. When the CTI supplier notifies the media toolbar application that the call is accepted, the partner application can fire the `startCommEvent` action.
4. Once the Fusion application identifies this action, the matched contact and an engagement panel is opened based on the rules configured in the Fusion configuration management for the screen pop. See *Configure Screen Pop Pages* for more information.

During the incoming event, you can use contact.accept() function to accept a call as shown in the following example:

```
public async acceptCall(): Promise<void> {
 this.contact.accept();
}
```

Also update `subscribeContactEvents` as in the following example:

```
private subscribeContactEvents() {
 this.connect.contact((contact: any) => {
 this.contact = contact;
 this.eventId = contact.contactId;
 contact.onConnecting((contact: any) => {
 console.log("Contact onConnecting >", contact);
 const phoneNumber = contact.getActiveInitialConnection().getEndpoint().phoneNumber;
 this.integrationEventsHandler.incomingCallHandler(phoneNumber, this.eventId);
 });
 contact.onAccepted((contact: any) => {
 console.log("Contact onAccepted >", contact);
 this.integrationEventsHandler.outboundCallAcceptedHandler(this.eventId);
 });
 contact.onEnded(async (contact: any) => {
 console.log("Contact onEnded >", contact);
 });
 contact.onMissed((contact: any) => {
 console.log("Contact onMissed >", contact);
 });
 });
}
```

Complete code

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import "amazon-connect-streams";
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private connect: any;
 private agent: any;
 private contact: any;
 private integrationEventsHandler: IntegrationEventsHandler;
 private eventId: string = '';

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 this.connect = (window as any)["connect"];
 }
 public async makeAgentAvailable(): Promise<void> {
 if (!this.agent) {
 this.init();
 this.subscribeForAmazonEvents();
```

```
this.subscribeContactEvents();
} else {
let state = this.agent.getAgentStates()[0];
this.agent.setState(state);
}
}
public async makeAgentUnavailable(): Promise<void> {
let state = this.agent.getAgentStates()[1];
this.agent.setState(state);
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
}
public async acceptCall(): Promise<void> {
this.contact.accept();
}
public async rejectCall(): Promise<void> {
}
public async hangupCall(): Promise<void> {
}

// Intialize Amazon connect ccp
private init() {
const containerDiv = document.getElementById("amazon-connect-cca-container");
this.connect.core.initCCP(containerDiv, {
ccpUrl: 'https://cti-amazon-connect-demo.my.connect.aws/ccp-v2', // REQUIRED
loginPopup: true, // optional, defaults to `true`
loginPopupAutoClose: true, // optional, defaults to `false`
loginOptions: {
// optional, if provided opens login in new window
autoClose: true, // optional, defaults to `false`
height: 600, // optional, defaults to 578
width: 400, // optional, defaults to 433
top: 0, // optional, defaults to 0
left: 0, // optional, defaults to 0
},
region: "eu-west-2", // REQUIRED for `CHAT`, optional otherwise
softphone: {
// optional, defaults below apply if not provided
allowFramedSoftphone: true, // optional, defaults to false
disableRingtone: false, // optional, defaults to false
},
pageOptions: {
//optional
enableAudioDeviceSettings: false, //optional, defaults to 'false'
enableVideoDeviceSettings: false, //optional, defaults to 'false'
enablePhoneTypeSettings: true, //optional, defaults to 'true'
},
ccpAckTimeout: 5000, //optional, defaults to 3000 (ms)
ccpSynTimeout: 3000, //optional, defaults to 1000 (ms)
ccpLoadTimeout: 10000, //optional, defaults to 5000 (ms)
});
}

private subscribeForAmazonEvents() {
this.connect.agent((agent: any) => {
this.agent = agent;
let state = this.agent.getAgentStates()[0];
this.agent.setState(state);
});
}

private subscribeContactEvents() {
this.connect.contact((contact: any) => {
this.contact = contact;
this.eventId = contact.contactId;
contact.onConnecting((contact: any) => {
```

```
console.log("Contact onConnecting >", contact);
const phoneNumber = contact.getActiveInitialConnection().getEndpoint().phoneNumber;
this.integrationEventsHandler.incomingCallHandler(phoneNumber, this.eventId);
});
contact.onAccepted((contact: any) => {
console.log("Contact onAccepted >", contact);
this.integrationEventsHandler.outboundCallAcceptedHandler(this.eventId);
});
contact.onEnded(async (contact: any) => {
console.log("Contact onEnded >", contact);
});
contact.onMissed((contact: any) => {
console.log("Contact onMissed >", contact);
});
});
}

}
```

Verify your progress

Sign in to your Fusion application and open the media toolbar. Click Agent Availability button from your media toolbar application. Make an inbound call and you'll see the incoming call notification on both the Fusion application and the toolbar window. Accept the call and the screen pop appears with a new contact.

Disconnect or reject an incoming call in Amazon

All the call related operation such as call accept, reject, disconnect, mute and so on are performed on the `Contact` object of the `Connect Streams` API.

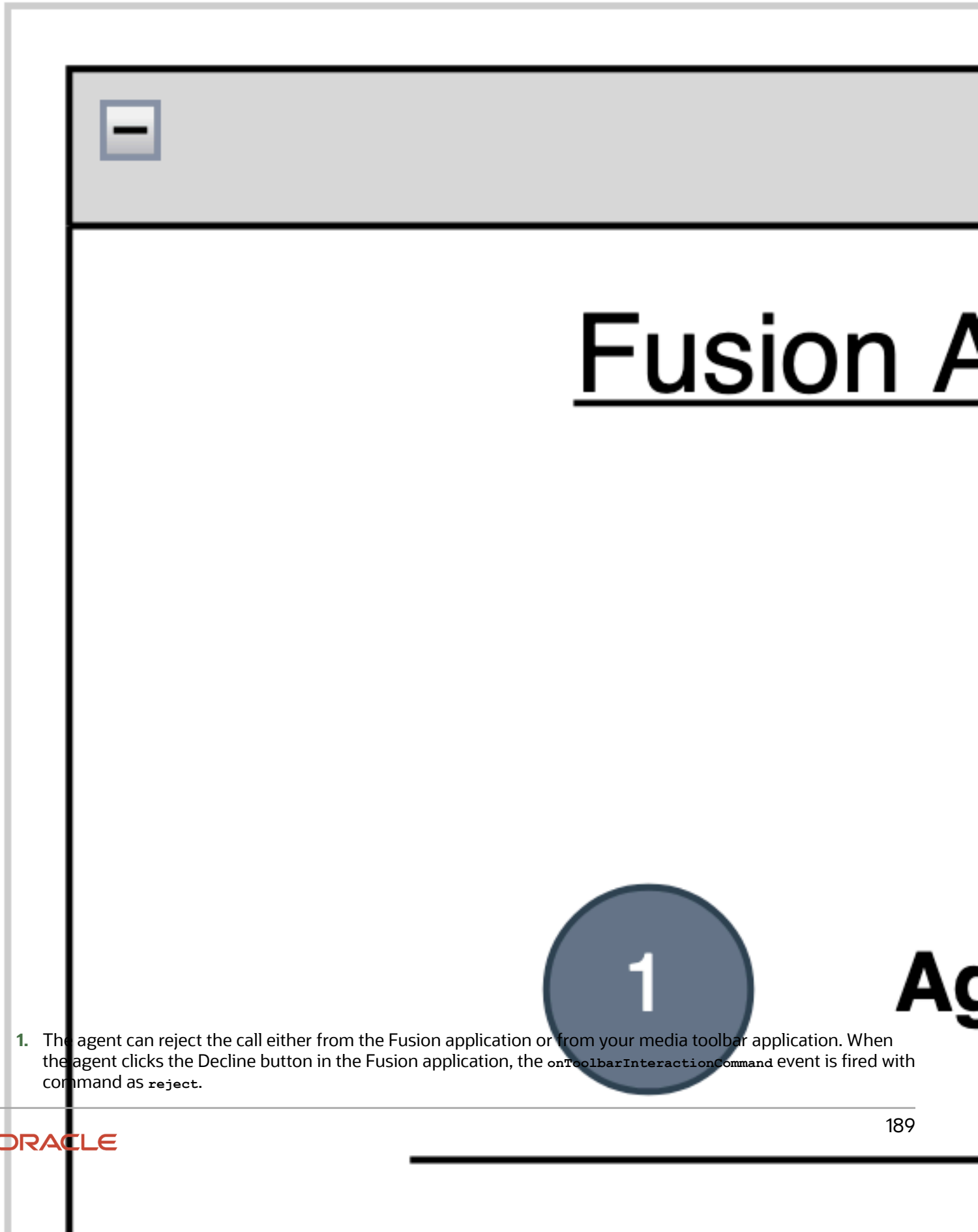
You can use the `contact.reject()` function to reject a call and the `contact.getAgentConnection().destroy()` function to disconnect the call.

Calls can be disconnected in one of three ways:

1. Agent rejects the incoming call before the call is accepted.
2. Agent disconnects the call after the conversation is complete.
3. Customer disconnects the call.

Scenario 1: Agent rejects the incoming call before the call is accepted

The following flow diagram shows the sequence of operations performed once an agent rejects the call in the Fusion application or from the media toolbar application:



1. The agent can reject the call either from the Fusion application or from your media toolbar application. When the agent clicks the Decline button in the Fusion application, the `onToolbarInteractionCommand` event is fired with command as `reject`.

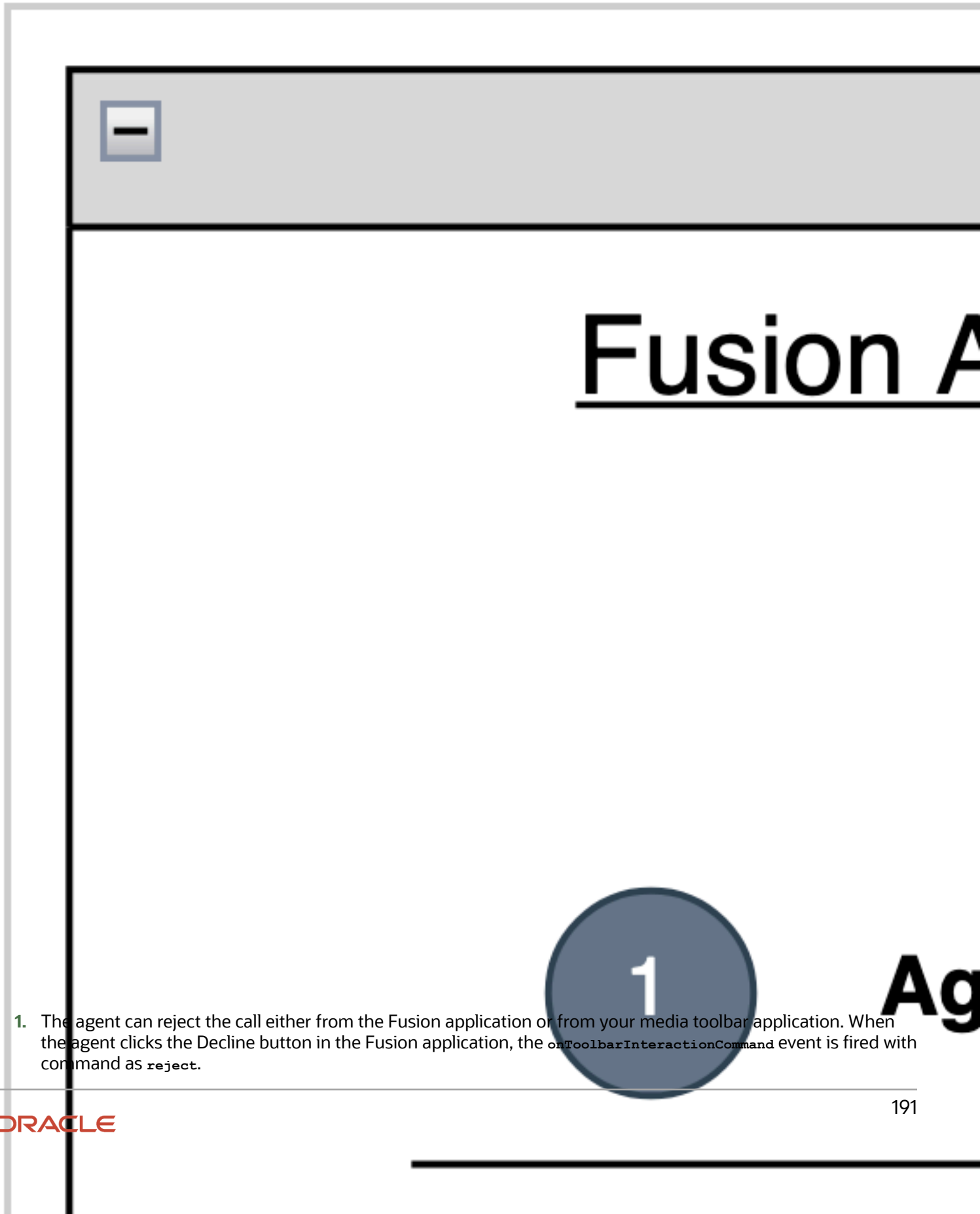
2. The partner application receives this event and if the event is to reject the call, the `Amazon Connect Streams` API can be called to reject the call.
3. After the CTI supplier notifies the partner application that the call has been rejected, the partner application can fire the `closeCommEvent` action with reason as `REJECT`.
4. Once the Fusion application identifies this action, the call dialog box is discarded from the UI.

Update the `rejectCall`example:

```
public async rejectCall(): Promise<void> {
 this.contact.reject({
 success: () => {
 console.log("Rejected call")
 },
 failure: (err: any) => {
 console.error("Reject error", err);
 }
 });
}
```

Scenario 2: Agent disconnects the call once the conversation is complete

The following flow diagram shows the sequence of operations performed once an agent disconnects the call from the Fusion application or from the media toolbar application:



1. The agent can reject the call either from the Fusion application or from your media toolbar application. When the agent clicks the Decline button in the Fusion application, the `onToolbarInteractionCommand` event is fired with command as `reject`.

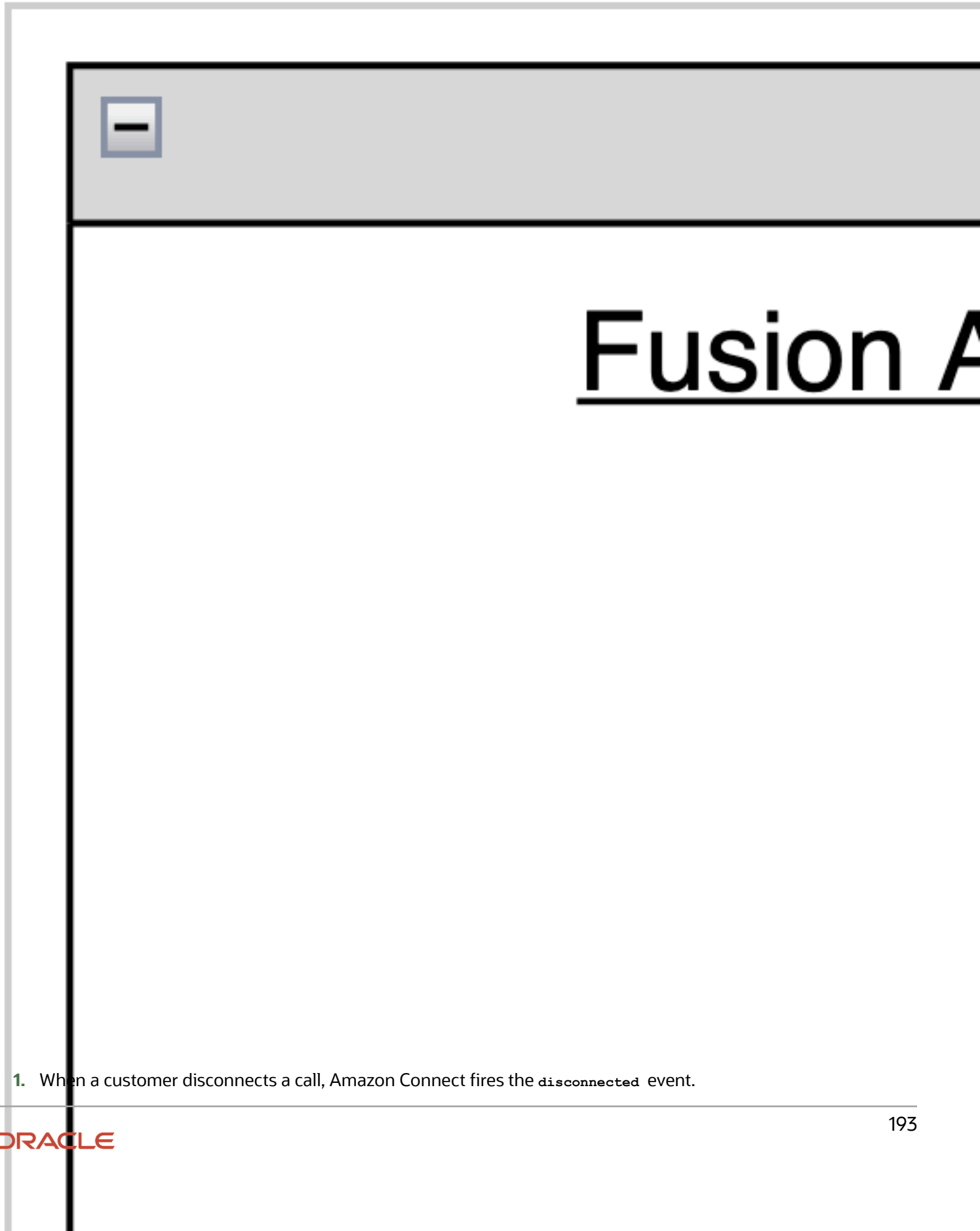
2. The partner application receives this event and if the event is to `disconnect` the call, the `Amazon Connect Streams` API to disconnect the call can be called.
3. After the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with reason as `HANGUP`.
4. After the Fusion application identifies the action, it displays the Wrap Up window.

Update the `hangupCall` example:

```
public async hangupCall(): Promise<void> {
 this.contact.getAgentConnection().destroy({
 success: function () {
 console.log("Contact complete disconnected");
 this.contact.clear({
 success: function () {
 console.log("Contact call clearContact");
 },
 failure: function (err: any) {
 console.log("Contact call clearContact failure", err);
 },
 });
 },
 failure: function (err: any) {
 console.log("Contact call complete failure", err);
 },
 });
}
```

Scenario 3: Customer disconnects the call

The following flow diagram shows the sequence of operations performed when the customer disconnects the call:



1. When a customer disconnects a call, Amazon Connect fires the `disconnected` event.

2. You add an event listener in your media toolbar application for the `reject` and `disconnect` events.
3. The `closeCommEvent` action is fired from your media toolbar application with reason as `REJECT` or `WRAPUP`.
4. After the Fusion application identifies the action, it displays the Wrap Up window.

Complete code

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import "amazon-connect-streams";
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 private connect: any;
 private agent: any;
 private contact: any;
 private integrationEventsHandler: IntegrationEventsHandler;
 private eventId: string = '';

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 this.connect = (window as any)["connect"];
 }
 public async makeAgentAvailable(): Promise<void> {
 if (!this.agent) {
 this.init();
 this.subscribeForAmazonEvents();
 this.subscribeContactEvents();
 } else {
 let state = this.agent.getAgentStates()[0];
 this.agent.setState(state);
 }
 }
 public async makeAgentUnavailable(): Promise<void> {
 let state = this.agent.getAgentStates()[1];
 this.agent.setState(state);
 }
 public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
 }
 public async acceptCall(): Promise<void> {
 this.contact.accept();
 }
 public async rejectCall(): Promise<void> {
 this.contact.reject({
 success: () => {
 console.log("Rejected call")
 },
 failure: (err: any) => {
 console.error("Reject error", err);
 }
 });
 }
 public async hangupCall(): Promise<void> {
 this.contact.getAgentConnection().destroy({
 success: function () {
 console.log("Contact complete disconnected");
 /*
 this.contact.clear({
 success: function () {
 console.log("Contact call clearContact");
 },
 failure: function (err: any) {
 console.log("Contact call clearContact failure", err);
 },
 });
 */
```

```
},
failure: function (err: any) {
console.log("Contact call complete failure", err);
},
});
}

// Intialize Amazon connect ccp
private init() {
const containerDiv = document.getElementById("amazon-connect-cca-container");
this.connect.core.initCCP(containerDiv, {
ccpUrl: 'https://cti-amazon-connect-demo.my.connect.aws/ccp-v2', // REQUIRED
loginPopup: true, // optional, defaults to `true`
loginPopupAutoClose: true, // optional, defaults to `false`
loginOptions: {
// optional, if provided opens login in new window
autoClose: true, // optional, defaults to `false`
height: 600, // optional, defaults to 578
width: 400, // optional, defaults to 433
top: 0, // optional, defaults to 0
left: 0, // optional, defaults to 0
},
region: "eu-west-2", // REQUIRED for `CHAT`, optional otherwise
softphone: {
// optional, defaults below apply if not provided
allowFramedSoftphone: true, // optional, defaults to false
disableRingtone: false, // optional, defaults to false
},
pageOptions: {
//optional
enableAudioDeviceSettings: false, //optional, defaults to 'false'
enableVideoDeviceSettings: false, //optional, defaults to 'false'
enablePhoneTypeSettings: true, //optional, defaults to 'true'
},
ccpAckTimeout: 5000, //optional, defaults to 3000 (ms)
ccpSynTimeout: 3000, //optional, defaults to 1000 (ms)
ccpLoadTimeout: 10000, //optional, defaults to 5000 (ms)
});
}

private subscribeForAmazonEvents() {
this.connect.agent((agent: any) => {
this.agent = agent;
let state = this.agent.getAgentStates()[0];
this.agent.setState(state);
});
}

private subscribeContactEvents() {
this.connect.contact((contact: any) => {
this.contact = contact;
this.eventId = contact.contactId;
contact.onConnecting((contact: any) => {
console.log("Contact onConnecting >", contact);
const phoneNumber = contact.getActiveInitialConnection().getEndpoint().phoneNumber;
this.integrationEventsHandler.incomingCallHandler(phoneNumber, this.eventId);
});
contact.onAccepted((contact: any) => {
console.log("Contact onAccepted >", contact);
this.integrationEventsHandler.outboundCallAcceptedHandler(this.eventId);
});
contact.onEnded(async (contact: any) => {
console.log("Contact onEnded >", contact);
this.integrationEventsHandler.callHangupHandler(this.eventId);
});
contact.onMissed((contact: any) => {
```

```
console.log("Contact onMissed >", contact);
this.integrationEventsHandler.callRejectedHandler(this.eventId);
});
});
}

}
```

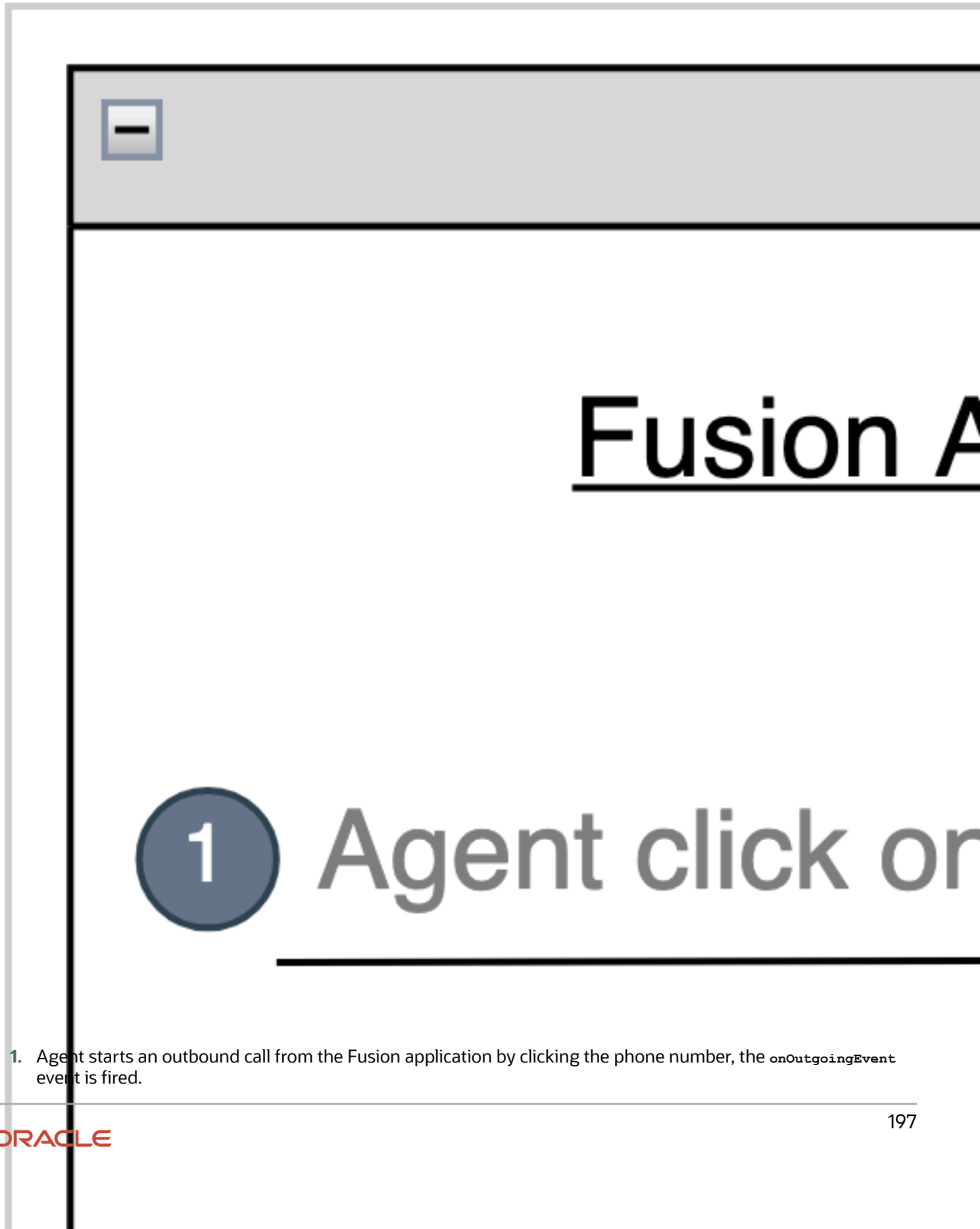
Verify your progress

Sign in to your Fusion application and open the media toolbar. Click Agent Availability button on your media toolbar application.

1. Make an inbound call and see the incoming call notification in both the Fusion application and the toolbar window. Reject the call and the call notification will close.
2. Make an inbound call and see the incoming call notification on both the Fusion application and the toolbar window. Accept the call and then end the call from Fusion application and the engagement will be changed to the `wrapup` state.
3. Make an inbound call and see the incoming call notification on both the Fusion application and the toolbar window. Accept a call and then end the call from user side and the engagement will be changed to the `wrapup` state.

Make an outbound call in Amazon

The following flow diagram shows the sequence of operations performed once an agent starts an outbound call from the Fusion application:



1. Agent starts an outbound call from the Fusion application by clicking the phone number, the `onOutgoingEvent` event is fired.

2. The `onOutgoingEvent` event listener in your media toolbar application receives this event and notifies your microservice to start an outbound call.
3. Your microservice starts the call by creating request using `PSDK`.
4. If the request is successful, the `EventRinging` event is fired from Genesys and propagated to your media toolbar application through the microservice.
5. Once the `RequestMakeCall` request is success, the `EventRinging` message is propagated from Genesys through your microservice to your media toolbar application.
6. Once the `EventEstablished` message is received by the media toolbar application, the `newCommEvent` action is started to notify Fusion application.
7. The Fusion application will show the dialing panel once this `newCommEvent` action is received.

Agent starts an outbound call from fusion application

The agent starts an outbound call from the Fusion application by clicking the phone number, and the `onOutgoingEvent` event is fired.

Notify Genesys to start a call

The `onOutgoingEvent` event listener in your media toolbar application receives this event and notifies your microservice to start an outbound call. You can update the `makeOutboundCall` function as shown in the following example to notify your microservice to start a call:

```
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "makeCall",
 "toNumber": phoneNumber,
 "connectionId": VendorHandler.connectionId
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
```

Microservice creates request for starting a call

The microservice uses `PSDK` to create requests for making a call using the `com.genesyslab.platform.voice.protocol.tserver.requests.party.RequestMakeCall` request class. If the request is successful, the `EventRinging` message is received and when the customer responds, the `EventEstablished` message will be received by your microservice.

Microservice receives EventRinging message

Once the request for starting outbound call is success, the Genesys server responds with the `EventRinging` message and it will be propagated to your media toolbar application through the microservice.

Start the newCommEvent from media toolbar application

When you receive the `EventRinging` message from your microservice, you can call the `newCommEvent` action by executing the function `incomingCallHandler` in the `integrationEventsHandler.ts` file as shown in the following example:

```
public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
```

```
console.log(jsonMessage);
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
}
console.log("Message is received");
}
```

Fusion will show dialing panel

The Fusion application shows the dialing notification once the `newCommEvent` action is received.

Complete code

Here's the complete code of the `vendorHandler.ts` file for starting an outbound call.

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { IntegrationEventsHandler } from '../integrationEventsHandler';

export class VendorHandler implements ICtiVendorHandler {
 public static connectionId: string;
 private static REST_ENDPOINT_URL: string = 'http://localhost:8087/genesys/events';
 private static WS_ENDPOINT_URL: string = 'ws://localhost:8087/genesysWs';
 private integrationEventsHandler: IntegrationEventsHandler;

 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.integrationEventsHandler = integrationEventsHandler;
 }

 public async webSocketOnOpenHandler(): Promise<void> {
 console.log("WebSocket opened");
 const headers: Headers = (new Headers()) as Headers;
 headers.set('Content-type', 'application/json');
 const message: any = {
 "type": "initialize"
 };
 const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
 method: 'POST',
 headers: headers,
 body: JSON.stringify(message)
 }) as Request;
 await fetch(request);
 }
 public webSocketErrorHandler(error: any): void {
 console.log("WebSocket error", error);
 }
 public webSocketCloseHandler(event: Event): void {
 console.log("WebSocket is closed", event);
 }

 public webSocketOnMessage(event: MessageEvent): void {
 const jsonMessage = JSON.parse(event.data);
 console.log(jsonMessage);
```

```
if (jsonMessage.eventName === "EventRegistered") {
// Genesys notifies that the agent is ready
this.integrationEventsHandler.makeAgentAvailable();
} else if (jsonMessage.eventName === "EventRinging") {
// Show incoming call notification
this.integrationEventsHandler.incomingCallHandler(jsonMessage.ANI, jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventEstablished") {
// Genesys notifies that the call is accepted
this.integrationEventsHandler.callAcceptedHandler(jsonMessage.eventId);
VendorHandler.connectionId = jsonMessage.connectionId;
} else if (jsonMessage.eventName === "EventReleased") {
// Genesys notifies that the call is disconnected
this.integrationEventsHandler.callHangupHandler(jsonMessage.eventId);
}
console.log("Message is received");
}
public async makeAgentAvailable(): Promise<void> {
let webSocket: WebSocket = new WebSocket(`${VendorHandler.WS_ENDPOINT_URL}`);
webSocket.onopen = this.webSocketOnOpenHandler.bind(this);
webSocket.onmessage = this.webSocketOnMessage.bind(this);
webSocket.onclose = this.webSocketCloseHandler.bind(this);
webSocket.onerror = this.webSocketErrorHandler.bind(this);
}

public async makeAgentUnavailable(): Promise<void> {
// TODO: call the vendor specific api to make the agent available
}
public async makeOutboundCall(phoneNumber: string, eventId: string): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "makeCall",
"toNumber": phoneNumber,
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async acceptCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Accept",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async rejectCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
```

```
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
public async hangupCall(): Promise<void> {
const headers: Headers = (new Headers()) as Headers;
headers.set('Content-type', 'application/json');
const message: any = {
"type": "Reject",
"connectionId": VendorHandler.connectionId
};
const request: Request = new Request(`${VendorHandler.REST_ENDPOINT_URL}`, {
method: 'POST',
headers: headers,
body: JSON.stringify(message)
}) as Request;
await fetch(request);
}
}
```

Verify your progress

After finishing these steps, use OJET server to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the Agent Availability button. To start an outbound call, open the contact from your Fusion application and click the phone number, which starts the outbound call.

Disconnect or reject an incoming call from the Fusion application or the toolbar application

Now you set up functionality to reject an incoming call or, disconnect the accepted call.

You can reject or disconnect a call from your media toolbar application and from the Fusion application.

Overview of the tasks

Here's an overview of what you need to do:

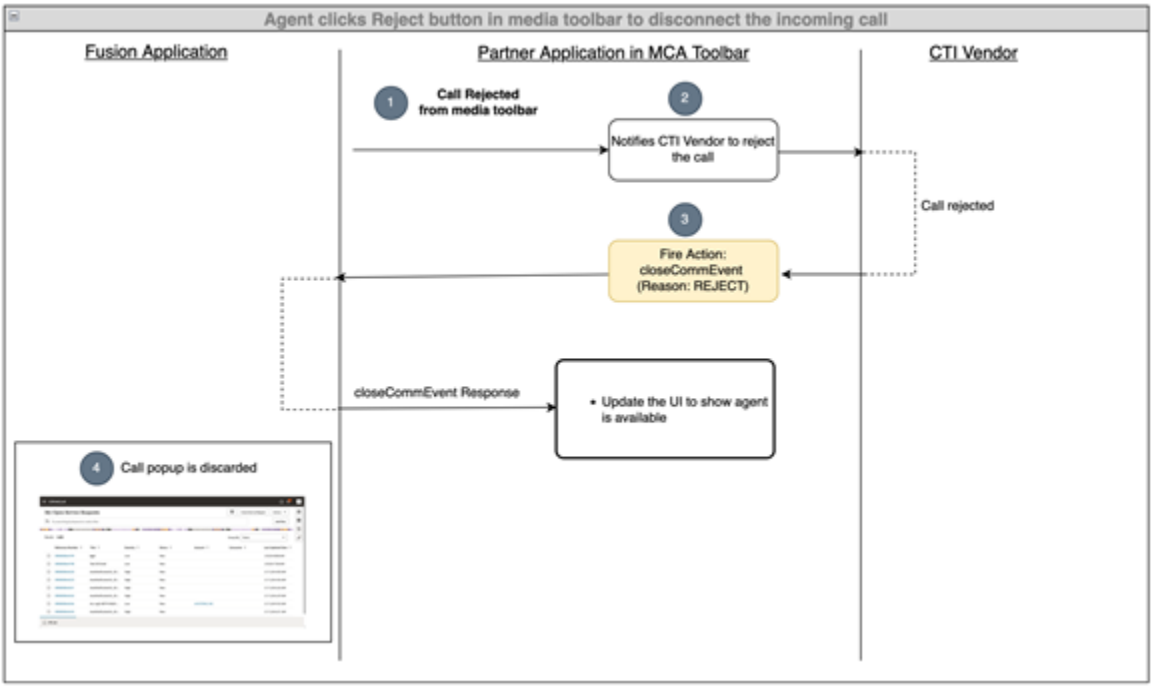
- Part 1: Rejecting/disconnecting a call from your media toolbar application
  a. Add an event handler for the Disconnect button click event.
  b. Define the `callDisconnectedEventHandler` function.
  c. Define the functions `disconnectCall` and `rejectCall` in the `integrationActionHandler.ts` file.
  d. Implement the `rejectCall()` and `hangupCall()` functions in the `vendorHandler.ts` file based on your CTI supplier's SDK.
  e. Define the function `notifyCallDisconnectedToFusion` in the `fusionHandler.ts` file.
- Part 2: Rejecting or disconnecting a call from the Fusion application
  a. Add cases for reject and disconnect to `listenToToolbarInteractionCommandsFromFusion` in the `integrationEventsHandler.ts` file.

Part 1: Rejecting or disconnecting a call from your media toolbar application

Once the agent clicks on the Reject or Disconnect button, the call-panel component fires an event notifying its container where the component is loaded. In our case, the component is loaded in the media toolbar application. An event is similarly fired when the Reject button is selected. When a call is disconnected before the call is accepted by the agent, an event with the payload `disconnectionState` as REJECT is fired and when a call is disconnected after the call is accepted by the agent, an event with payload `disconnectionState` as WRAPUP is fired.

From the container, an event handler must be defined to handle these events.

The following diagram shows the sequence of actions to be performed when the agent accepts the call from the media toolbar application:



Here's the agent call flow:

1. The agent clicks on the **Reject** button from the media toolbar application.
2. The Partner application notifies the CTI supplier to reject the call.
3. Once the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with the reason as REJECT.
4. Once the Fusion application identifies this action, the call dialog box will be discarded from the UI.

The following steps show you how to implement this in your toolbar application:

1. Add an event handler for the disconnect button click event

Add the Reject button clicked event handler in call-panel component as in the following example of the index.html file:

```
<oj-bind-if test="[[callContext().state === 'RINGING' || callContext().state === 'ACCEPTED']]">
 <call-panel call-context="[[callContext]]"
 on-accept-button-clicked="[[ callAcceptedEventHandler ]]"
 on-disconnect-button-clicked="[[ callDisconnectedEventHandler ]]"></call-panel>
```

```
</oj-bind-if>
```

2. Define callDisconnectedEventHandler function

Define the `callDisconnectedEventHandler` function in the `appController.ts` file as shown in the following example. The `callDisconnectedEventHandler` function internally calls two other functions,`disconnectCall`, which is defined in `integrationActionHandler`, and`updateCallPanelState`, which is defined in `appController.ts` itself.

```
//.....
//.....
import "oj-c/button";
// ....

class RootViewModel {
 // ....

 constructor() {
 // ....
 };

 public callDisconnectedEventHandler: (event: any) => void = (event: any): void => {
 if (event.detail.disconnectionState === 'WRAPUP') {
 this.integrationActionHandler.disconnectCall(this.callContext().eventId,
 event.detail.disconnectionState).then(() => {
 this.updateCallPanelState('DISCONNECTED');
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 } else if (event.detail.disconnectionState === 'REJECT') {
 this.integrationActionHandler.rejectCall(this.callContext().eventId,
 event.detail.disconnectionState).then(() => {
 this.updateCallPanelState('DISCONNECTED');
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 }
 }

 public updateCallPanelState = (state: string) => {
 this.callContext({...this.callContext(), state: state});
 }

 //...
 }

}
```

3. Define the functions disconnectCall and rejectCall in integrationActionHandler.ts file

The `disconnectCall` function contains the logic to notify your CTI supplier to hang up or reject the call and also it contains the logic to notify the Fusion application that the call is disconnected by firing the `closeCommEvent` action.

```
import { IntegrationEventsHandler } from "./integrationEventsHandler";
import { ICtiVendorHandler } from "./vendor/ICtiVendorHandler";
import { VendorHandler } from "./vendor/vendorHandler";
import { FusionHandler } from "./fusion/fusionHandler";

export class IntegrationActionHandler {
 private vendor: ICtiVendorHandler;
 private integrationEventsHandler: IntegrationEventsHandler;
```

```
 constructor(integrationEventsHandler: IntegrationEventsHandler) {
 this.vendor = new VendorHandler();
 this.integrationEventsHandler = integrationEventsHandler;
 }

 // .....

 public async disconnectCall(eventId: string, disconnectionState: string): Promise<void> {
 await this.vendor.rejectCall();
 await FusionHandler.notifyCallDisconnectedToFusion(eventId, disconnectionState);
 }

 public async rejectCall(eventId: string, disconnectionState: string): Promise<void> {
 await this.vendor.rejectCall();
 await FusionHandler.notifyCallDisconnectedToFusion(eventId, disconnectionState);
 }

}
```

4. Implement rejectCall() and hangupCall() functions in vendorHandler.ts file based on your CTI provider's SDK

See your CTI supplier's documentation to identify how an incoming call is notified to the CTI application using their API.The incomingCallNotificationHandler function must be called from there. You can add the logic in `vendorHandler.ts` file:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';

export class VendorHandler implements ICtiVendorHandler {
 // ....
 public async rejectCall() {
 // TODO: call the vendor specific api to reject a call
 }
 public async hangupCall() {
 // TODO: call the vendor specific api to hangup a call
 }
 // ....
}
```

5. Define function notifyCallDisconnectedToFusion function in fusionHandler.ts file

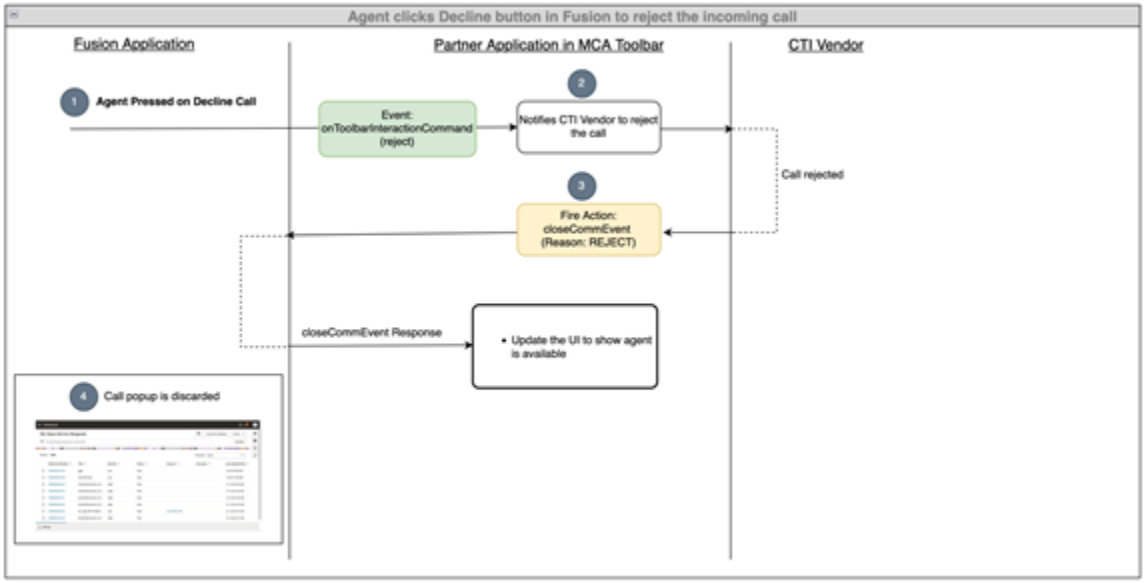
From the `fusionHandler.ts`, you notify the Fusion application that the call is rejected by firing the `closeCommEvent` action.

```
export class FusionHandler {
 // ...

 public static async notifyCallDisconnectedToFusion(eventId: string, disconnectionState: string):
 Promise<void> {
 let request: IMcaCloseCommEventActionRequest =
 FusionHandler.frameworkProvider.requestHelper.createPublishRequest('closeCommEvent') as
 IMcaCloseCommEventActionRequest;
 request.setReason(disconnectionState);
 const newCommResponse: any = FusionHandler.callToNewCommResponseMap.get(eventId);
 if (newCommResponse) {
 for (const property in newCommResponse) {
 request.getInData().setInDataValueByAttribute(property, newCommResponse[property]);
 }
 }
 //TODO pass unique identifier from your CTI data
 request.setEventId(eventId);
 request.setAppClassification(FusionHandler.appClassification);
```

```
    const operationResponse: IMcaCloseComActionResponse = await FusionHandler.phoneContext.publish(request) as
    IMcaCloseComActionResponse;
    }

    // ...
}
```

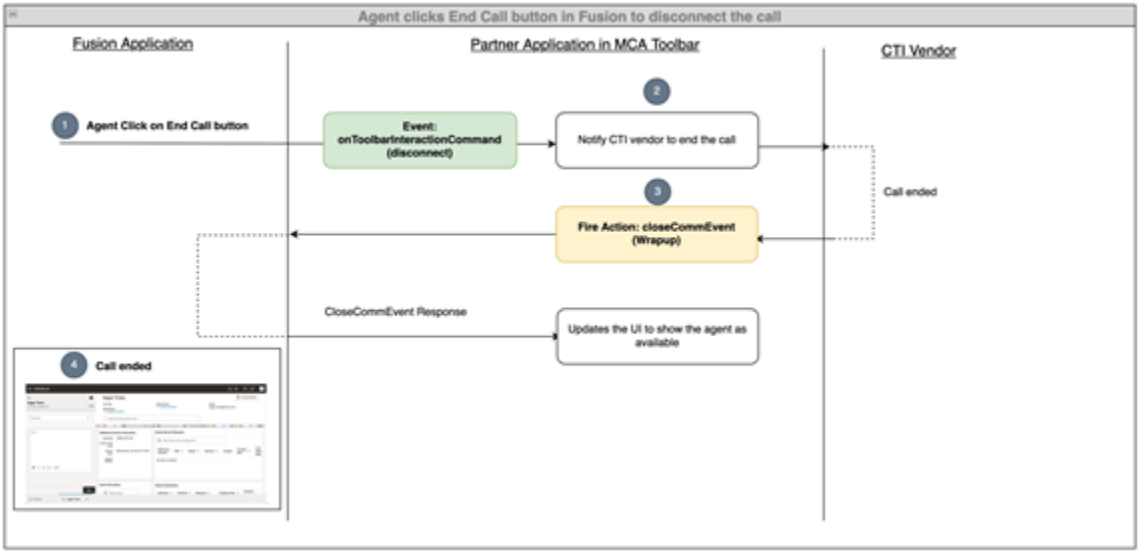
Part 2: Rejecting or disconnecting incoming calls from the Fusion application

Now, when an agent clicks on **Decline** button from the Fusion call notification, it notifies your media toolbar application and your CTI supplier that the call is disconnected by the agent. The following diagram shows the sequence of actions to be performed when an agent rejects the call from the Fusion application:



1. When the agent clicks on the **Decline** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `reject`.
2. The partner application receives this event and if the event is to reject a call, the partner application notifies the CTI supplier to reject the call.
3. Once the CTI supplier notifies the partner application that the call is rejected, the partner application can fire the `closeCommEvent` action with the reason as REJECT.
4. Once the Fusion application identifies this action, the call dialog box will be discarded from the UI.

The following diagram shows the sequence of actions performed when the agent disconnects an ongoing call from the Fusion application:



1. When the agent clicks on the **End Call** button in the Fusion application, the `onToolbarInteractionCommand` event is fired with the command as `disconnect`.
2. The partner application receives this event and if the event is to disconnect a call, the partner application notifies the CTI supplier to disconnect the call.
3. Once the CTI supplier notifies the partner application that the call has been disconnected, the partner application can fire the `closeCommEvent` action.
4. Once the Fusion application identifies this action, it displays the wrap-up window in the UI.

1: Add cases for reject and disconnect in listenToToolbarInteractionCommandsFromFusion in integrationEventsHandler.ts

```
export class IntegrationEventsHandler {
 // ....
 public listenToToolbarInteractionCommandsFromFusion(command: string): void {
 switch (command) {
 case "accept":
 this.ctiAppViewModel.callAcceptedEventHandler(null);
 break;
 case "disconnect":
 this.ctiAppViewModel.callDisconnectedEventHandler({detail: {disconnectionState: "WRAPUP"}});
 break;
 case "reject":
 this.ctiAppViewModel.callDisconnectedEventHandler({detail: {disconnectionState: "REJECT"}});
 break;
 }
 }
 // ....

}
```

Verify your progress

Once you complete the above steps, use `ojet serve` to start your application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and your Fusion window. You can accept the call from your media toolbar application or your Fusion application. Once you

accept the call, your media toolbar state will be changed to ACCEPTED state and the engagement will get opened in your Fusion application. You can disconnect the call from your media toolbar application or your Fusion application. Once you disconnect the call, your media toolbar state will change to DISCONNECTED state.

9 Use Gen AI with CTI

Introduction to Gen AI and CTI

Fusion CTI supports the following Gen AI features which are called Agent Assist features:

- Call summarization
- Suggestions during phone calls

Call Summarization

You can use call transcripts to generate a phone call summary. When a call is ended, the transcripts are passed to the Gen AI service which generates the summary of the phone call. The generated summary is added to the Wrap Up Notes field.

> **Note:** This is done only if the agent hasn't entered any notes in the field. But, once the summary has been generated, the agent can edit it.

Suggestions during phone calls

The live transcripts rendered in the Fusion application can also be used for showing suggestions to the agent based on the ongoing conversation. Once this feature is enabled, you can see agent assistance suggestions shown in the Fusion application based on the transcripts and the available KM sources. A maximum of three suggestions are presented to the agent.

There are two ways to share a suggestion to the customer from the drill down view. Suggestions can be shared using email or shared using text messages. Sharing with email is only possible when an SR is associated with the phone call. To share the suggestion using a text message, `shareSuggestionsExternally` should be enabled.

How to enable/disable summarization and share suggestions externally features in your media toolbar application

You must complete the prerequisites before implementing the following code. From the prerequisites section, you might have already enabled the transcripts from `subscribeToSupportedFeatures` function. For enabling summarization and share suggestions externally features, you need to update the `supportedFeatures` array to add `summaryEnabled` and `shareSuggestionsExternally` in your `subscribeToSupportedFeatures` function as shown in the following example:

```
public static async subscribeToSupportedFeatures(): Promise<any> {
 const request =
 FusionHandler.frameworkProvider.requestHelper.createSubscriptionRequest('onToolbarAgentCommand');
 FusionHandler.phoneContext.subscribe(request, (response) => {
 const agentCommandResponse: IMcaOnToolbarAgentCommandEventResponse = response as
 IMcaOnToolbarAgentCommandEventResponse;
 return new Promise((resolve, reject) => {
 const agentCommandResponseData = agentCommandResponse.getResponseData();
 const commandObject = agentCommandResponse.getResponseData().getData();
 const command = agentCommandResponse.getResponseData().getCommand();
 if (command === 'getActiveInteractionCommands') {
```

```
const outData = {
'supportedCommands': [],
'supportedFeatures': [
{
'name': 'transcriptEnabled',
'isEnabled': true // Set as true to enable transcripts
},
{
'name': 'summaryEnabled',
'isEnabled': true // Set as true to enable summarization
},
{
'name': 'shareSuggestionsExternally',
'isEnabled': true // Set as true to share suggestions externally
}
]
};
agentCommandResponseData.setOutdata(outData);
commandObject.result = 'success';
}
resolve(commandObject);
})
})
}
```

In this code, the `supportedFeatures` array in the `outData` variable is used to enable or disable the Gen AI features. You can set the `isEnabled` key to true or false to enable or disable the features.

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion window. You can accept the call from your media toolbar application or from your Fusion application. Once you accept the call, your media toolbar state will be changed to the ACCEPTED state and the engagement will be opened in your Fusion application. You'll see that a transcript component is loaded in the engagement panel instead of the notes field, which contains messages stating the agent joined the call and the customer joined the call.

Prerequisites for using Gen AI features with CTI

Complete the following prerequisites before you enable Gen AI.

Enable Generative AI

1. From the Opt In page, click the Pencil icon on the Service Adaptive Intelligence row to select specific features.
2. Select the **Enable** checkbox for **Use Generative AI features in the Service**.
3. Click **Done** when you're finished.

Associate the Generative AI duty role

Now you add a new duty role which has specific privileges for the available Fusion AI features. You add these privileges to your custom Generative AI Duty Role and then associate the role to your agents.

The privileges are:

- ORA_SVC_WRAP_UP_SUMMARIZATION_PRIV. Used for access to Wrap Up summary generation.
- ORA_SVC_CHAT_COLLABORATION_SUMMARIZATION_PRIV. Used for generating a chat transcript summary that will be included in chat transfer and conference offers to other agents.
- ORA_SVC_CHAT_KM_SUMMARIZATION_PRIV. Used for generating a knowledge article summary that can be included in a chat conversation.
- SVC_ACCESS_SR_SUMMARIZATION_APIS_PRIV – Used for SR-generated summaries.

To use the different Generative AI features, you can just associate the Service Generative AI User (ORA_SVC_GEN_AI_USER) duty role to your agents. But to enable only some new AI features, you'll need to create a custom Duty Role and associate the corresponding service privilege.

Add the privilege show summarization

To use the call and chat wrap up summarization, each user must be granted the ORA_SVC_WRAP_UP_SUMMARIZATION _PRIV privilege in Security Console.

The privilege includes a ready to use default duty role: ORA_SVC_GEN_AI_USER. You can use the ORA_SVC_GEN_AI_USER duty role or you can copy and edit the copy to only include the AI features required. If you've already created your own roles, you must use Security Console to add the ORA_SVC_WRAP_UP_SUMMARIZATION _PRIV privilege to the duty role.

Add the privilege to show summarization for transfers and conferences

To use chat transcript summarization, each user must be granted the ORA_SVC_CHAT_COLLABORATION_SUMMARIZATION_PRIV privilege in Security Console.

The privilege includes a ready to use default duty role: ORA_SVC_CHAT_COLLABORATION_SUMMARIZATION_PRIV. You can use the ORA_SVC_GEN_AI_USER duty role you can copy and edit the copy to only include the AI features required. If you've already created your own roles, you must use Security Console to add the ORA_SVC_CHAT_COLLABORATION_SUMMARIZATION_PRIV privilege to the duty role.

Enable call transcripts

After setting the required privileges and profile options, you then enable and add transcript messages. We'll do that in the next task.

Add a call transcript

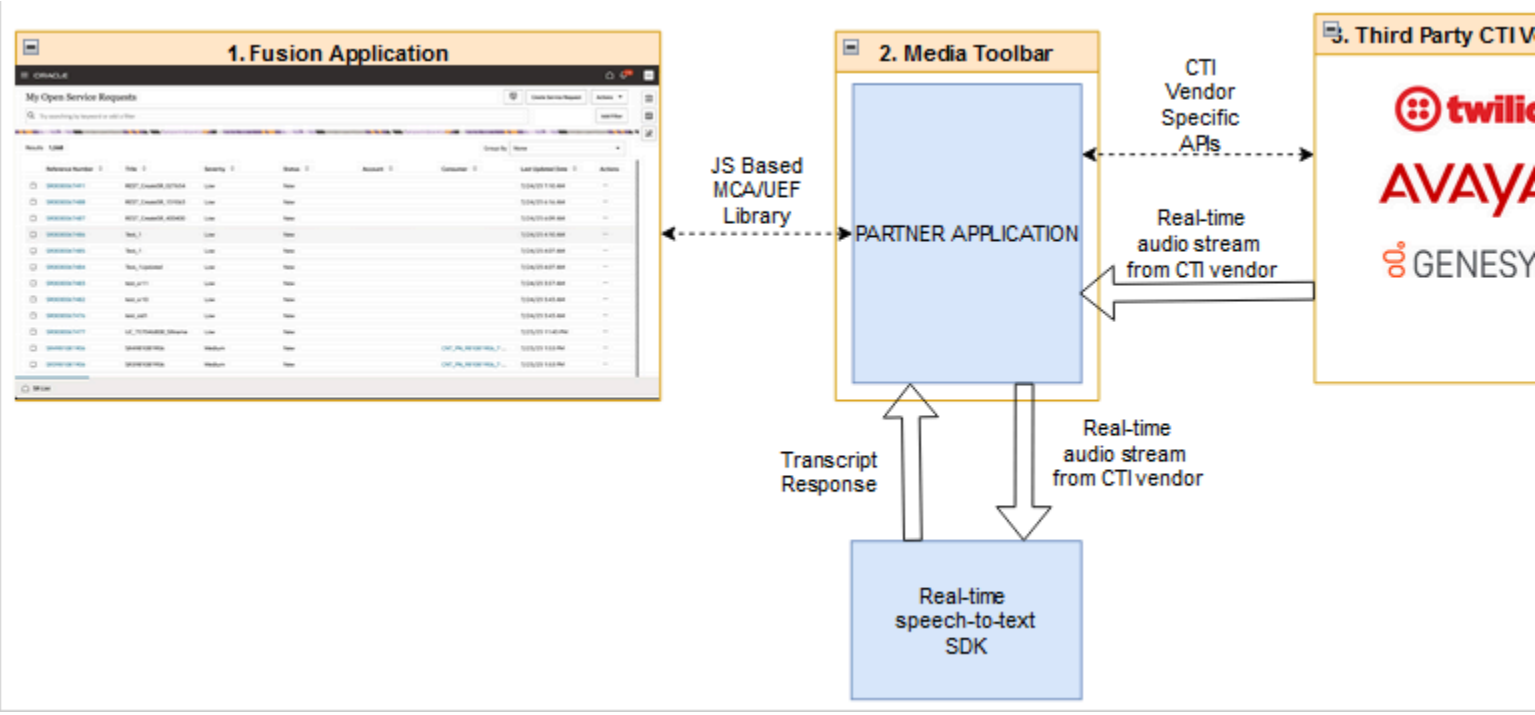
Call transcripts are used to provide suggestions during the call and for generating the summary of the phone call.

A call transcript within MCA is a written record of a phone call that's created by converting audio to text. If your CTI supplier provides the live transcription of a phone call in the form of text stream, you can render those transcript messages in the Fusion Engagement panel. Once a call is started, you can use the UEF `FeedLiveTranscript` action API to convert audio to text.

With this API the integrator can render transcript messages for both the agent and the customer. After it's enabled, the notes input box in Fusion Engagement panel is replaced with a transcript component and the messages "agent joined

the call" and "customer joined the call" are shown in the Engagement Panel along with a Call Details button. The agent can enter notes during the call by clicking the Call Details button.

Here's a diagram that shows a high level view of how to add call transcripts during a phone call.



As the diagram shows, your CTI supplier should provide the real-time audio stream of the ongoing call, which you then pass to a real-time speech-to-text service to get the textual data behind the ongoing call. OCI Realtime Speech SDK is one such speech to text service that you can use. If your CTI supplier directly provides real-time transcripts, you can pass them to the UEF APIs to render it on the Fusion screen.

How to enable or disable call transcripts in your media toolbar application

Define a function `subscribeToSupportedFeatures` in `fusionHandler.ts` file as shown in the following example:

```
public static async subscribeToSupportedFeatures(): Promise<any> {
 const request =
 FusionHandler.frameworkProvider.requestHelper.createSubscriptionRequest('onToolbarAgentCommand');
 FusionHandler.phoneContext.subscribe(request, (response) => {
 const agentCommandResponse: IMcaOnToolbarAgentCommandEventResponse = response as
 IMcaOnToolbarAgentCommandEventResponse;
 return new Promise((resolve, reject) => {
 const agentCommandResponseData = agentCommandResponse.getResponseData();
 const commandObject = agentCommandResponse.getResponseData().getData();
 const command = agentCommandResponse.getResponseData().getCommand();
 if (command === 'getActiveInteractionCommands') {
 const outData = {
 'supportedCommands': [],
 'supportedFeatures': [
 {
 'name': 'transcriptEnabled',
 'isEnabled': true // Set as true to enable transcripts
 }
 ]
 };
 agentCommandResponseData.setOutdata(outData);
```

```
commandObject.result = 'success';
}
resolve(commandObject);
})
})
}
```

In the code sample, the `supportedFeatures` array in the `outData` variable is used to enable or disable the transcript, summary and suggestions features. You can set the `isEnabled` key to true or false to enable or disable this feature.

Next, you need to call this function while initializing the application. For that, call the function shown in the previous example from the `makeAgentAvailable` function in the `integrationEventsHandler.ts` as shown in the following graphic:

You can use real-time-speech SDK's to convert the audio stream from your CTI supplier to text data, which are suitable for transcript.

Once you receive the text converted from the audio stream, you can use the `FeedLiveTranscript` API to render it in Fusion. Based on the audio stream source, its transcript can be rendered as and agent's message or a user's message.

Based on the `FeedLiveTranscript` request object, the transcript is rendered on the engagement panel.

Render Transcript Messages

The `FeedLiveTranscript` API should be executed from the engagement context, which is available from the `startComm` event action response. Once the engagement context is available, create the request object for `FeedLiveTranscript` API and set the required properties (message, message id, state, role). The role should be set as AGENT for adding the agent transcript messages. The role should be set as END_USER for adding the agent transcript messages. Once this API is executed, the agent's transcript message will be rendered in the fusion engagement panel.

In your media toolbar application, add the following function in `fusionHandler.ts`:

```
public static async addRealTimeTranscript(messageId: string, message: string, role: string, state: string):
 Promise<any> {
 var requestObject: IMcaFeedLiveTranscriptActionRequest =
 FusionHandler.frameworkProvider.requestHelper.createPublishRequest('FeedLiveTranscript') as
 IMcaFeedLiveTranscriptActionRequest;
 requestObject.setMessageId(messageId);
 requestObject.setMessage(message);
 requestObject.setRole(role);
 requestObject.setState(state);
 var res = await FusionHandler.engagementContext.publish(requestObject);
 return res;
}
```

Also, add another function in `integrationEventsHandler.ts` to invoke the previous function:

```
public async addRealTimeTranscript(messageId: string, message: string, role: string, state: string):
 Promise<any> {
 var resp: any = await FusionHandler.addRealTimeTranscript(messageId, message, role, state);
 return resp;
}
```

You need to invoke this function from your `vendorHandler.ts` file upon receiving a transcription result from your real-time speech to text service.

See *Introduction to the Gen AI accelerator* for more information.

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion window. Once you accept the call, your media toolbar state will be changed to ACCEPTED state and the engagement will get opened in your Fusion application. You'll see that a transcript component is loaded in the engagement panel instead of the notes field, which contains messages, agent joined the call and customer joined the call. Start speaking and you'll see the transcripts are getting added to the transcript component.

Show agent assistance suggestions during phone calls

Here's we'll see how the agent assistance suggestions are shown during a phone call.

If you've all the setup, no more configuration is needed. The agent assistance suggestions will be automatically shown when there are enough transcript messages.

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion window. Once the conversation is started, you can see the real-time transcripts are getting rendered in the Fusion engagement panel. You'll also see agent assistance suggestions are shown based on the transcript.

Share agent assistance suggestions using text messages during phone calls

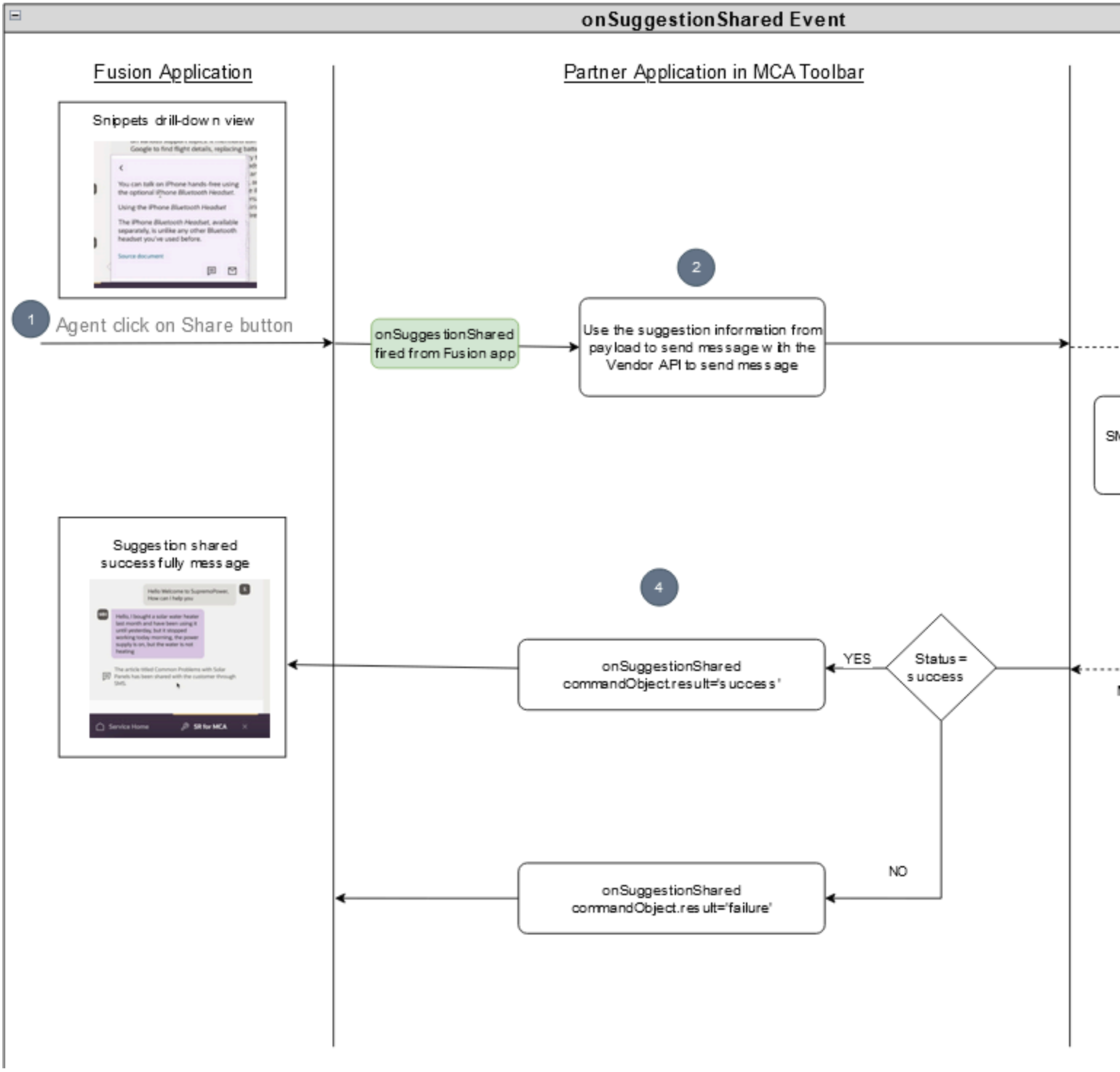
Agents can click each suggestion to go into the detail view of the suggestion and can click the link in the suggestion to open the associated KM article.

If the `shareSuggestionsExternally` setting is enabled, agent can see a share using the text message button in the detail view of the suggestion. If an SR is associated in the IVR data of the phone call, an agent can see a share using email button in the detail view of the suggestion.

It's assumed you've completed the prerequisites and added the code for sharing suggestions externally.

Once the call is in progress, you'll see agent assistance suggestions in the Fusion application. You can click a suggestion to go into the detail view of the suggestion where there will be a **Share via text message** icon there.

The following flow diagram shows the sequence of operations performed once the agent clicks on the **Share via text message** button in the snippet suggestion:



1. If the `shareSuggestionsExternally` parameter is set as true the agent can see a **Share via text message** button in the drill-down view of the snippet. The agent clicks the button to send the suggestion as text messages to the customer.
2. The `onToolbarInteractionCommand` event listener in the media toolbar application is executed with the command of `onSuggestionShared`. The event payload contains suggestion information such as `externalUrl`, suggestion

text, and so on. From this `eventlistener`, for the `onSuggestionShared` command, you need to call your supplier or SMS provider API to send the text message.

3. Your SMS supplier sends the SMS and responds whether the status is success or failure. If the SMS share status is success, a resolved promise is returned from the `onToolbarInteractionCommand` event listener with `commandObject.result='success'`, and if the SMS share status is failure, a resolved promise is returned from the `onToolbarInteractionCommand` event listener with `commandObject.result='failure'`.

   If the `commandObject.result` is set as success, the following message: `The article titled {ARTICLETITLE} has been shared with the customer through SMS` is shown in the call transcript as a message to the agent.

   If the `commandObject.result` is set as failure, the following message: `Failed to share the article titled {ARTICLETITLE} with the customer through SMS` is shown in the call transcript as a message to the agent.

How to send the suggestion as a SMS through your CTI supplier

1. In the `ICtiVendorHandler.ts` file, add the `shareSuggestionExternally` declaration for the `sendTextMessage` function:

```
export interface ICtiVendorHandler {
 //...
 sendTextMessage(suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function):
 Promise<void>;
 //...
}
```

2. Update `subscribeToToolbarInteractionCommandsFromFusion` in the `fusionHandler.ts` file with `shareSuggestionExternally`:

```
public static subscribeToToolbarInteractionCommandsFromFusion(callback: Function): void {
 const request: IMcaEventRequest =
 FusionHandler.frameworkProvider.requestHelper.createSubscriptionRequest('onToolbarInteractionCommand')
 as IMcaEventRequest;
 FusionHandler.phoneContext.subscribe(request, (eventResponse: IEventResponse) => {
 return new Promise((resolve, reject) => {
 const eventResponseDetails: IMcaOnToolbarInteractionCommandDataResponse = (eventResponse as
 IMcaOnToolbarInteractionCommandEventResponse).getResponseData();
 callback(eventResponseDetails.getCommand(), eventResponseDetails.getData(), resolve);
 })

 });
 }
```

3. Update the `subscribeToToolbarInteractionCommandsFromFusion` function call from the `makeAgentAvailable` function in the `integrationEventsHandler.ts` file to handle the new parameters as shown in the following example:

```
public async makeAgentAvailable(): Promise<void> {
 try {
 //...
 FusionHandler.subscribeToToolbarInteractionCommandsFromFusion((command: string,
 suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function) =>
 { this.listenToToolbarInteractionCommandsFromFusion(command, suggestionData, resolveRef); });
 //...
 } catch (err) {
 //...
 }
}
```

4. Update the `listenToToolbarInteractionCommandsFromFusion` function signature to handle the new parameters and add a case for the `onSuggestionShared` command as shown example:

```
public listenToToolbarInteractionCommandsFromFusion(command: string, suggestionData:
 IMcaOnToolbarInteractionCommandData, resolveRef: Function): void {
 switch (command) {
 //...
 case "onSuggestionShared":
 this.ctiAppViewModel.shareSuggestionExternally(suggestionData, resolveRef);
 //...
 }
}
```

5. Define the `functionshareSuggestionExternally` in the `appController.ts` file as shown in the following example:

```
public shareSuggestionExternally: (suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef:
 Function) => void = (suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function): void
 => {
 this.integrationActionHandler.shareSuggestionExternally(suggestionData, resolveRef).then(() => {
 console.log("Requested to send message")
 }).catch(() => {
 console.log("Error: Unable to accept the call.")
 });
 }
```

6. Define the function `shareSuggestionExternally` in the `integrationActionHandler.ts` file as shown in the following example:

```
public async shareSuggestionExternally(suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef:
 Function): Promise<void> {
 await this.vendor.sendTextMessage(suggestionData, resolveRef);
 }
```

7. Implement the logic to send message using your CTI supplier in the `vendorHandler.ts` file as shown in the following example:

```
export class VendorHandler implements ICtiVendorHandler {
//...
public async sendTextMessage(suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function):
 Promise<void> {
 // TODO: call the vendor specific api to send text message
 // if(SMS SHARE STATUS FROM API RESPONSE IS 200 OR SUCCESS)
 suggestionData.result = 'success';
 // else
 suggestionData.result = 'failure';
 resolveRef(suggestionData);
 }
//...
}
```

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. You can accept the call from your media toolbar application or from your Fusion application. Once the conversation is started, you can see the real-time transcripts are getting rendered in the Fusion engagement panel. You'll also see agent assistance suggestions are shown based on the suggestions. When you go to the drill down view of a particular suggestion, you'll see a **Share via text message** button. And when you click the button, the suggestion will be shared as a text message to the customer.

Generate a call summary

If you've followed all the setup steps for enabling summarization from media toolbar, no more configuration is needed.

The summary will be automatically generated once the call is ended and if the transcript contains enough messages.

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. You'll receive the incoming call notification in your media toolbar application and your Fusion window. You can accept the call from your media toolbar application or your Fusion application. Once you accept the call, your media toolbar state will be changed to ACCEPTED state and the engagement will be opened in your Fusion application. You'll see the a transcript component is loaded in the engagement panel instead of the notes field, which contains messages, agent joined the call and customer joined the call. Start speaking and you'll see the transcripts are getting added to the transcript component. Once you end the call and have enough number of messages in transcript, wrapup window is opened and a notes field is auto-populated with the summary of the phone call.

10 Accelerators with Gen AI support

Integrating with Twilio

Add a call transcript, show suggestions and summarizations in Twilio

You can either use the following steps to perform your setup or you can quickly try it out by downloading a compressed file which has the code for adding the transcripts, and shows the suggestions and summarizations described in this topic. See *How to start your OJET application from a compressed file*for more information.

You can configure your Twilio setup to send the live raw audio streams of phone calls to a specific destination. From there, you can use any real-time speech SDK to convert the audio stream into text. This text is then used to render in the Fusion.

Here's a high level view of the architecture:

INSERT GRAPHIC

Here's an overview of the diagram:

1. Configure your Twilio App to enable the streams
2. Initialize the real time-transcription service for real-time transcription
3. Pass the real time audio stream to the `realtime-transcription` service
4. Call the `feedLiveTranscript` API to render the transcripts in Fusion

Configure your Twilio App to enable the streams

Streams should be enabled from the quick start application you've built in *Introduction to Gen AI and CTI*.

You can enable the streams from `voiceResponse` function in handler.js file by adding the following code after initializing the `voiceResponse` function:

```
const start = twiml.start();
 start.stream({
 name: 'Example Audio Stream',
 url: 'wss://twilio-node-voice-stream.onrender.com/',
 track: 'both_tracks'
 });
```

For more information on Twilio streams, see *https://www.twilio.com/docs/voice/twiml/stream*.

You also need to add the logic to forward the audio stream to media toolbar application from your quick start application. This is done by way of websockets.

For this, in index.js file of your quick start application, initialize a websocket as shown in the following example:

```
//...
const WebSocket = require('ws');
//...
```

```
// const server = http.createServer(app);
const wss = new WebSocket.Server({server});
var callSidWebSocketMap = new Map();
var streamSidCallSidMap = new Map();

//...
wss.on("connection", function connection(ws) {
 console.log("New Connection Initiated");

 ws.on("message", function incoming(message) {
 const msg = JSON.parse(message);
 switch (msg.event) {
 case "registerClient":
 console.log('New client registered: ' + JSON.stringify(msg));
 callSidWebSocketMap.set(msg.accountSid, ws);
 break;
 case "connected":
 console.log(`A new call has connected.`);
 break;
 case "start":
 console.log('New stream started: ' + JSON.stringify(msg));
 streamSidCallSidMap.set(msg.start.streamSid, msg.start.accountSid)
 break;
 case "media":
 var twilioData;
 if (msg?.media?.track === 'outbound') {
 twilioData = "{\"event\":\"media\",\"streamId\":\""+msg?.streamSid+"\",\"from\":\"agent\",\"payload\":\"" +
 msg?.media?.payload + "\"}";
 } else {
 twilioData = "{\"event\":\"media\",\"streamId\":\""+msg?.streamSid+"\",\"payload\":\"" +
 msg?.media?.payload + "\"}";
 }
 const clientWebSocket = callSidWebSocketMap.get(streamSidCallSidMap.get(msg.streamSid));
 if (clientWebSocket) {
 console.log(`Sending media payload to clientWebSocket`);
 clientWebSocket.send(twilioData);
 } else {
 console.log(`Client not found`);
 }
 console.log(twilioData);
 break;
 case "stop":
 console.log('Call Has Ended' + JSON.stringify(msg));
 callSidWebSocketMap.get(msg?.stop?.accountSid)?.close();
 break;
 }
 });
});
//...
```

Initialize the realtime-transcription service for real-time transcription

1. Download the real-time speech transcription app, and extract the contents to a directory.
2. Extract the contents to a directory. In `index.ts` file, update the `compartmentId` variable with your compartment ID having the OCI speech service.
3. Update `config.js` file with your OCI configuration.
4. Open the `oci-speech` directory in your terminal and run the `npm install` command.
5. Build the project with the `npm run build` command.

6. Start the service by running the `npm run start` command.

   The websocket for the `oci-speech` service is now running. Now you just need to send the audio stream coming from Twilio to this websocket to get the transcription results.

Pass the realtime audio stream to realtime-transcription service

In your `vendorHandler.ts` file, initialize the following class variables:

```
export class VendorHandler implements ICtiVendorHandler {
 // ...
 private static messageIds: string[] = [];
 private static TWILIO_SERVICE: string = 'https://twilio-node-voice-stream.onrender.com'; // Your quick-
start application URL
 private static RT_SPEECH_SERVICE: string = 'wss://
phoenix339284.appsdev.fusionappsdphx1.oraclevcn.com:8004/'; // Your real-time speech transcription service
 URL
 private static TWILIO_WS_URL: string = 'wss://twilio-node-voice-stream.onrender.com/'; // Your quick-start
 application WebSocket URL
 private static transcriptionServerWsForAgent: WebSocket;
 // ...
}
```

Define the following functions:

```
// This function is the entry point for transcription
private initTranscription(accountSid: string): void {
 VendorHandler.transcriptionServerWsForAgent = new WebSocket(VendorHandler.RT_SPEECH_SERVICE); // WebSocket
 connection to real-time speech transcription service
 let self = this;
 // 1. Initialize WebSocket connection to real-time speech transcription service
 VendorHandler.transcriptionServerWsForAgent.addEventListener("open", (event) => {
 // 2. Once the WebSocket connection to realtime speech transcript service is success,
 // Initialize WebSocket connection to Twilio to get the audio stream
 this.initializeWebsocketConnectionToTwilio(accountSid);
 });

 // 3. The transcription results from the real
 VendorHandler.transcriptionServerWsForAgent.addEventListener("message", async (event) => {
 await this.handleTranscriptResponseFromSpeechService(event, self);
 });
}

// This function initializes the WebSocket connection to Twilio
private initializeWebsocketConnectionToTwilio(accountSid: string): void {
 let twilioServerWs: WebSocket = new WebSocket(VendorHandler.TWILIO_WS_URL);
 // 2.1. Initialize WebSocket connection to your Twilio quick-start application
 twilioServerWs.addEventListener("open", (event) => {
 // 2.2. Once the WebSocket connection to Twilio is success,
 // Send registerClient message to Twilio to register the client for transcription
 twilioServerWs.send(JSON.stringify({"event":"registerClient", "accountSid": accountSid}));
 });
 twilioServerWs.addEventListener("message", async (event) => {
 // 2.3. Here, you will receive the audio stream payload and you need to pass this to
 // your real-time speech service websocket for getting the transcript results.
 this.handleAudioStreamFromTwilio(event);
 });
 twilioServerWs.addEventListener("error", (err) => {
 console.log("Message from server ", err);
 });
}

// This function forwards the audio stream to real-time transcript service
```

```
private handleAudioStreamFromTwilio(event: any): void {
 const msg = JSON.parse(event.data);
 switch (msg.event) {
 case "media":
 // 2.3.1. Send the audio stream to your real-time transcript service in a specific format as returned from
 generatePayloadForTranscriptServer function
 VendorHandler.transcriptionServerWsForAgent.send(JSON.stringify(this.generatePayloadForTranscriptServer(msg)));
 break;
 }
}

// This function generates the payload to transcription function in a specific format
private generatePayloadForTranscriptServer(message: any): any {
 return {
 "callId": message.streamId,
 "role": message.from === 'agent' ? 'AGENT' : 'END_USER',
 "message": message.payload
 }
}

// This function handles the results generated from the real-time speech transcript service
private async handleTranscriptResponseFromSpeechService(event: any, self: any): Promise<void> {
 let state = "STARTED";
 const responseFromServer = JSON.parse(event.data);
 const role: string = responseFromServer.role == 'AGENT' ? 'AGENT' : 'END_USER'
 if (responseFromServer.final) {
 state = "CLOSED"
 } else {
 if (VendorHandler.messageIds.includes(responseFromServer?.messageId)) {
 state = "INPROGRESS";
 } else {
 VendorHandler.messageIds.push(responseFromServer?.messageId)
 }
 }
 // Invoke UEF API to add the transcript message to the engagement panel.
 await self.integrationEventsHandler.addRealTimeTranscript(responseFromServer?.messageId,
 responseFromServer?.transcript, role, state);
}
```

Invoke the initTranscription function from incoming and outgoing event handlers:

```
public incomingCallCallback = (call: Call) => {
 this.initTranscription(call.parameters.AccountSid);
 //...
}

public async makeOutboundCall(phoneNumber: string, eventId: string) {
 //...
 // if (this.device) {
 // this.call = await this.device.connect({ params });
 this.initTranscription(this.call.parameters.AccountSid);
 //...
 // }
}
```

Complete Code

Here's the complete code of the `vendorHandler.ts` file for accepting an incoming call:

```
import { ICtiVendorHandler } from './ICtiVendorHandler';
import { Device, Call } from '@twilio/voice-sdk';
import {IntegrationEventsHandler} from "../integrationEventsHandler";

export class VendorHandler implements ICtiVendorHandler {
```

```
private twilio: any;
private device: Device | null;
private integrationEventsHandler: IntegrationEventsHandler;
private call: Call | null;
public idAndToken: any;
private static messageIds: string[] = [];
private static TWILIO_SERVICE: string = 'https://twilio-node-voice-stream.onrender.com';
private static RT_SPEECH_SERVICE: string = 'wss://
phoenix339284.appsdev.fusionappsdphx1.oraclevcn.com:8004/';
private static TWILIO_WS_URL: string = 'wss://twilio-node-voice-stream.onrender.com/';
private static transcriptionServerWsForAgent: WebSocket;

constructor(integrationEventsHandler: IntegrationEventsHandler) {
this.twilio = (window as any).Twilio;
this.device = null;
this.idAndToken = null;
this.integrationEventsHandler = integrationEventsHandler;
this.call = null;
}

public async makeAgentAvailable(): Promise<void> {
this.idAndToken = await this.getIdAndToken();
this.device = new this.twilio.Device(this.idAndToken.token, {
logLevel: 1,
codecPreferences: ["opus", "pcmu"],
enableRingingState: true
});
let resolve: Function;
let reject: Function;
if (this.device) {
this.device.on("registered", () => {
console.log("Registration completed ...")
resolve();
});
this.device.on("error", (deviceError: any) => {
console.error("Registration Failed ...", deviceError);
reject();
});
this.device.on("incoming", this.incomingCallCallback);
this.device.register();
}
return new Promise((res: Function, rej: Function) => {
resolve = res;
reject = rej;
});
}
public async makeAgentUnavailable() {
throw new Error('Method not implemented.');
}
public async makeOutboundCall(phoneNumber: string, eventId: string) {
const params = {
To: phoneNumber,
};
if (this.device) {
this.call = await this.device.connect({ params });
this.initTranscription(this.call.parameters.AccountSid);
this.call.on("accept", () => { this.integrationEventsHandler.outboundCallAcceptedHandler(eventId) });
this.call.on("disconnect", () => { this.integrationEventsHandler.callHangupHandler(eventId) });
this.call.on("cancel", () => { this.integrationEventsHandler.callRejectedHandler(eventId) });
}
}
public async acceptCall() {
if (this.call) {
this.call.accept();
}
}
```

```
public async rejectCall() {
if (this.call) {
this.call.reject();
}
}
public async hangupCall() {
if (this.call) {
this.call.disconnect();
}
}

private initializeWebsocketConnectionToTwilio(accountSid: string): void {
let twilioServerWs: WebSocket = new WebSocket(VendorHandler.TWILIO_WS_URL);
twilioServerWs.addEventListener("open", (event) => {
twilioServerWs.send(JSON.stringify({"event":"registerClient", "accountSid": accountSid}));
});
twilioServerWs.addEventListener("message", async (event) => {
this.handleAudioStreamFromTwilio(event);
});
twilioServerWs.addEventListener("error", (err) => {
console.log("Message from server ", err);
});
}

private handleAudioStreamFromTwilio(event: any): void {
const msg = JSON.parse(event.data);
switch (msg.event) {
case "media":
VendorHandler.transcriptionServerWsForAgent.send(JSON.stringify(this.generatePayloadForTranscriptServer(msg)));
break;
}
}

private generatePayloadForTranscriptServer(message: any): any {
return {
"callId": message.streamId,
"role": message.from === 'agent' ? 'AGENT' : 'END_USER',
"message": message.payload
}
}

private async handleTranscriptResponseFromSpeechService(event: any, self: any): Promise<void> {
let state = "STARTED";
const responseFromServer = JSON.parse(event.data);
const role: string = responseFromServer.role == 'AGENT' ? 'AGENT' : 'END_USER'
if (responseFromServer.final) {
state = "CLOSED"
} else {
if (VendorHandler.messageIds.includes(responseFromServer?.messageId)) {
state = "INPROGRESS";
} else {
VendorHandler.messageIds.push(responseFromServer?.messageId)
}
}
await self.integrationEventsHandler.addRealTimeTranscript(responseFromServer?.messageId,
responseFromServer?.transcript, role, state);
}


private initTranscription(accountSid: string): void {
VendorHandler.transcriptionServerWsForAgent = new WebSocket(VendorHandler.RT_SPEECH_SERVICE);
let self = this;
VendorHandler.transcriptionServerWsForAgent.addEventListener("open", (event) => {
this.initializeWebsocketConnectionToTwilio(accountSid);
});
```

```
VendorHandler.transcriptionServerWsForAgent.addEventListener("message", async (event) => {
await this.handleTranscriptResponseFromSpeechService(event, self);
});
}

private async getIdAndToken(): Promise<any> {
const headers: Headers = (new Headers()) as Headers;
const url: string = `${VendorHandler.TWILIO_SERVICE}/token`; // Replace this url with the url of the
deployed node app
headers.set('Accept', 'application/json');
const request: Request = new Request(url, {
method: 'GET',
headers: headers
}) as Request;
const idAndToken: Response = await fetch(request);
this.idAndToken = await idAndToken.json();
return this.idAndToken;
}

public incomingCallCallback = (call: Call) => {
this.initTranscription(call.parameters.AccountSid);
this.integrationEventsHandler.incomingCallHandler(call.parameters.From, call.parameters.CallSid);
this.call = call;
this.call.on("cancel", () =>
{ this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
this.call.on("disconnect", () =>
{ this.integrationEventsHandler.callHangupHandler(call.parameters.CallSid) });
this.call.on("reject", () =>
{ this.integrationEventsHandler.callRejectedHandler(call.parameters.CallSid) });
}
public async sendTextMessage(suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function):
Promise<void> {
var myHeaders = new Headers();
myHeaders.append("Authorization", 'Basic
QUM0NzJjNjZmYTU0ZTRiNzNhYWExZTg1Yzk4Nzc1YmRjZjo3Mzg5NjlkYzBkMjNjMTVhMGEwNzE1NDY0N2ZiNjNhYg==='); // Add your
authorization header here
myHeaders.append("Content-Type", "application/x-www-form-urlencoded");

var urlencoded = new URLSearchParams();
urlencoded.append("To", this.call?.parameters.From || '');
urlencoded.append("From", "+13087374071");// Your TWILIO Number
urlencoded.append("Body", suggestionData.inData.metadata.originalSuggestionText + " Please refer: " +
suggestionData.inData.metadata.externalUrl);

var requestOptions: any = {
method: 'POST',
headers: myHeaders,
body: urlencoded,
redirect: 'follow'
};

fetch(`${VendorHandler.TWILIO_SERVICE}`, requestOptions)
.then(response => response.text())
.then(result => console.log(result))
.catch(error => console.log('error', error));
}
}
```

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your

Fusion window. You can accept the call from your media toolbar application or from your Fusion application. Once the conversation is started, you can see the real-time transcripts are getting rendered in the Fusion engagement panel.

Share Gen AI suggestions with Twilio using SMS

You can share agent assistance suggestions shown to the agent during phone call to a customer.

Twilio provides a REST API to send text messages. You need to use the REST API in your media toolbar application to send the text message.

For this, update sendTextMessage function in your vendorHandler.ts file as shown in the following example:

```
public async sendTextMessage(suggestionData: IMcaOnToolbarInteractionCommandData, resolveRef: Function):
 Promise<void> {
 var myHeaders = new Headers();
 myHeaders.append("Authorization", 'Basic AC8b3a4e9ea72839df7xxxxxx')); // Add your authorization header
 here
 myHeaders.append("Content-Type", "application/x-www-form-urlencoded");

 var urlencoded = new URLSearchParams();
 urlencoded.append("To", this.call.parameters.From);
 urlencoded.append("From", "+1234567890");// Your TWILIO Number
 urlencoded.append("Body", body);

 var requestOptions: any = {
 method: 'POST',
 headers: myHeaders,
 body: urlencoded,
 redirect: 'follow'
 };

 fetch(`${VendorHandler.TWILIO_SERVICE}`, requestOptions)
 .then(response => response.text())
 .then(result => console.log(result))
 .catch(error => console.log('error', error));
 }
```

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion window. You can accept the call from your media toolbar application or from your Fusion application. Once the conversation is started, you can see the real-time transcripts are getting rendered in the Fusion engagement panel. You'll also see agent assistance suggestions which are shown based on the suggestions. When you go to the drill down view of a particular suggestion, you'll see a share via text message button. Click the button and the suggestion will be shared as a text message to the customer.

Integrating with Genesys

Add a call transcript, show suggestions, and summarizations in Genesys

For Genesys transcripts, it's assumed that you'll be providing the transcript text for the ongoing conversation through a websocket.

This section expects, you'll be sending the transcription results from your transcription service as shown in the following example:

INSERT GRAPHIC

The transcription results from the transcription service should follow this format:

```
{
 "role": "AGENT" OR "END_USER", // Notifies fusion whether to render this transcript as Agent message or End
 user message.
 "transcript": "Hello", // The message to be rendered.
 "final": true OR false, // If final is set as false, a transcription in progress animation will be shown in
 Fusion.
 "messageId" :"12345"

}
```

Setup Steps

1. In your vendorHandler.ts file, initialize the following class variables:

```
export class VendorHandler implements ICtiVendorHandler {
 // ...
 private static TRANSCRIPT_ENDPOINT_URL: string = 'ws://localhost:8080/transcriptResult'; // Your
 endpoint that sends transcript results
 private static messageIds: string[] = [];
 // ...
}
```

2. In makeAgentAvailable function of vendorHandler.ts file, initialize the websocket as shown in the following example:

```
public async makeAgentAvailable(): Promise<void> {
 //....

 // Transcript result web socker
 let transcriptWebsocket: WebSocket = new WebSocket(`${VendorHandler.TRANSCRIPT_ENDPOINT_URL}`);
 transcriptWebsocket.onopen = this.transcriptWebSocketOnOpenHandler.bind(this);
 transcriptWebsocket.onmessage = this.transcriptWebSocketOnMessage.bind(this);
 transcriptWebsocket.onclose = this.transcriptWebSocketCloseHandler.bind(this);
 transcriptWebsocket.onerror = this.transcriptWebSocketErrorHandler.bind(this);
 }
```

3. Define the above event listeners as shown in the following example:

```
public async transcriptWebSocketOnOpenHandler(): Promise<void> {
```

```
console.log("WebSocket opened");
}
public transcriptWebSocketErrorHandler(error: any): void {
console.log("WebSocket error", error);
}
public transcriptWebSocketCloseHandler(event: Event): void {
console.log("WebSocket is closed", event);
}

public transcriptWebSocketOnMessage(event: MessageEvent): void {
const jsonMessage = JSON.parse(event.data);
console.log(jsonMessage);
if (jsonMessage.eventName === "TranscriptResult") {
await this.handleTranscriptResponseFromSpeechService(event);
}
}
```

4. Define the function handleTranscriptResponseFromSpeechService:

```
private async handleTranscriptResponseFromSpeechService(event: any): Promise<void> {
 let state = "STARTED";
 const responseFromServer = JSON.parse(event.data);
 const role: string = responseFromServer.role == 'AGENT' ? 'AGENT' : 'END_USER'
 if (responseFromServer.final) {
 state = "CLOSED"
 } else {
 if (VendorHandler.messageIds.includes(responseFromServer?.messageId)) {
 state = "INPROGRESS";
 } else {
 VendorHandler.messageIds.push(responseFromServer?.messageId)
 }
 }
 await this.integrationEventsHandler.addRealTimeTranscript(responseFromServer?.messageId,
 responseFromServer?.transcript, role, state);
 }
```

Verify your progress

Once you complete these steps, use OJET serve to start you application and sign in to your Fusion application. Open the media toolbar and make your agent available for calls by clicking on the agent availability button. Now, start a call to your customer care number. You'll receive the incoming call notification in your media toolbar application and in your Fusion window. You can accept the call from your media toolbar application or from your Fusion application. Once the conversation is started, you can see the real-time transcripts are getting rendered in the Fusion engagement panel.

Enable outbound dialing from the media toolbar

You can enable outbound calls made from the media toolbar to trigger a screen pop on a chosen page, as specified in the screen pop configuration.

This displays relevant information during a call, ensuring agents have the necessary context and tools to provide a more personalized and responsive experiences.

Outbound calls started from the media toolbar can trigger a screen pop to a chosen page when the token `SVCMCA_DIALER_SCREEN_POP_REQUIRED` is passed in the IVR data.

Here's an example of the newCommEvent to enable screen pop actions if the IVR data passes the token:

```
const uiEventsFrameworkInstance = await CX_SVC_UI_EVENTS_FRAMEWORK.uiEventsFramework.initialize('appname',
 'v1');
const multiChannelAdaptorContext = await uiEventsFrameworkInstance.getMultiChannelAdaptorContext();const
 phoneContext = await multiChannelAdaptorContext.getCommunicationChannelContext('PHONE');let request:
 IMcaNewCommEventActionRequest =
 uiEventsFrameworkInstance.requestHelper.createPublishRequest('newCommEvent') as
 IMcaNewCommEventActionRequest;
request.getInData().setInDataValueByAttribute('SVCMCA_ANI', phoneNumber);
request.setEventId('1234');
request.getInData().setInDataValueByAttribute("appClassification", "ORA_SERVICE");
request.getInData().setInDataValueByAttribute("SVCMCA_COMMUNICATION_DIRECTION", "ORA_SVC_OUTBOUND");
request.getInData().setInDataValueByAttribute('callStatus', "OUTGOING");
request.getInData().setInDataValueByAttribute('SVCMCA_DIALER_SCREEN_POP_REQUIRED', "Y"); // for enabling the
 screenpop for outbound call
request.setAppClassification("ORA_SERVICE");
const operationResponse: IMcaNewComActionResponse = await phoneContext.publish(request) as
 IMcaNewComActionResponse;
```