Oracle CX Service

Developing Live Experience

February 2024

Oracle CX Service
Developing Live Experience

February 2024

F52597-04

Author: B2C Service Documentation

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

Use help icons ⓘ to access help in the application. 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.

Get Support

You can get support at *My Oracle Support*. For accessible support, visit *Oracle Accessibility Learning and Support*.

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.

Learn About Accessibility

For information about Oracle's commitment to accessibility, visit the *Oracle Accessibility Program*. Videos included in this guide are provided as a media alternative for text-based topics also available in this guide.

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!

2 Add Live Experience Features to iOS Apps

Preparing Your iOS App for the Live Experience Widget

Set up your application environment in preparation for adding Oracle Live Experience to your iOS App.

There are a series of steps required to get Live Experience running in your iOS apps.

- Prepare Your iOS App for the Live Experience Widget: *Disable Bitcode in Your App Project*, *Add the Live Experience iOS Frameworks to Your Xcode Project*, *Configure the Required Background Modes*, and *Set the Appropriate App Permissions in Info.plist*.
- *Authenticate with Live Experience for iOS*
- *Adding and Configuring the Live Experience Widget for your iOS App*

Before integrating your iOS app with Oracle Live Experience, verify that your existing iOS app and development environment meet the following criteria:

- Xcode version 10.2 or higher.
- Swift version 5.0 or higher or Objective-C version 2.0.
- A copy of the Live Experience iOS SDK downloaded to your development machine.
- Credentials from the Live Experience Admin Console needed to obtain a JSON Web Token (JWT) from Live Experience.
- An existing iOS app targeted at iOS version 10 or higher.
- A valid tenant account for Live Experience.
- An actual iOS hardware device running iOS 10 or above.

  Note: While you can test the general flow and function of your Live Experience iOS application using the iOS simulator, a physical iOS device such as a phone or tablet is required to utilize audio or video functionality.

Disable Bitcode in Your App Project

Disable bitcode in your Live Experience app project.

1. In Xcode, select your project in the project navigation pane.
2. In the central pane, select **Build Settings**.
3. Select **All**
4. Scroll to Build Options.
5. Change the value of **Enable Bitcode** to **No**.

What to do next

Next, *Add the Live Experience iOS Frameworks to Your Xcode Project*.

Add the Live Experience iOS Frameworks to Your Xcode Project

Add the Live Experience iOS frameworks to your Xcode project.

> **Note:** If you update your version of Xcode, you need to remove and re-add the Live Experience frameworks to prevent language version mismatch errors.

1. Download and extract the Live Experience iOS SDK compressed (.zip) file.

   The **debug** folder contains debug frameworks. The **release** folder contains release frameworks.

2. Add the Live Experience frameworks to your project.
   a. Select your application target in the Xcode project navigator.
   b. Select the **General** tab in the top of the editor pane.
   c. Expand **Embedded Binaries**.
   d. Depending on your requirements, drag the framework files from either **release** or **debug** to Embedded Binaries. You can use the default options.

      If you're using the debug frameworks, make sure you also add **WebRTC.framework** as well from the release folder.

3. Import any other system frameworks you require.
   a. Select your project, and select the **Build Phases** tab.
   b. Expand **Link Binary With Libraries**.
   c. Click **+** to add new frameworks.
      - CFNetwork.framework: zero-configuration networking services. For more information, see *https://developer.apple.com/library/ios/documentation/CFNetwork/Reference/CFNetwork_Framework/index.html*.
      - Security.framework: General interfaces for protecting and controlling security access. For more information, see *https://developer.apple.com/library/ios/documentation/Security/Reference/SecurityFrameworkReference/index.html*.
      - CoreMedia.framework: Interfaces for playing audio and video assets in an iOS application. For more information, see *https://developer.apple.com/library/mac/documentation/CoreMedia/Reference/CoreMediaFramework/index.html*.
      - GLKit.framework: Library that facilitates and simplifies creating shader-based iOS applications (useful for video rendering). For more information, see *https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/DrawingWithOpenGLES/DrawingWithOpenGLES.html*.
      - AVFoundation.framework: A framework that facilities managing and playing audio and video assets in iOS applications. For more information, see *https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html#//apple_ref/doc/uid/TP40010188*.
      - AudioToolbox.framework: A framework containing interfaces for audio playback, recording, and media stream parsing. For more information, see *https://developer.apple.com/library/ios/*

documentation/MusicAudio/Reference/CAAudioTooboxRef/index.html#/apple_ref/doc/uid/TP40002089.

- VideoToolbox.framework: A low-level framework that provides direct access to hardware encoders and decoders. For more information, see *https://developer.apple.com/documentation/videotoolbox*.
- libicucore.tdb: A unicode support library. For more information, see *http://icu-project.org/apiref/icu4c40/*.
- libsqlite3.tdb: A framework providing a SQLite interface.

4. Drag the Live Experience frameworks to the **Frameworks** folder in your project.
5. If you are targeting iOS version 10 or above, add the **libstdc++.6.tbd** framework to prevent linking errors.

What to do next

Next, *Configure the Required Background Modes*.

Configure the Required Background Modes

Select appropriate background modes for your Live Experience app so it can handle various audio, video, and data communication tasks when it's not running as a foreground application.

1. Select your project in the project navigator.
2. Select the **Capabilities** tab in the top of the editor pane.
3. Expand **Background Modes**.
4. Click the switch from Off to **On** to enable background modes.
5. Enable the following modes:
    - Audio, AirPlay, and Picture in Picture
    - Voice over IP
    - External accessory communication
    - Background fetch

What to do next

Next, *Set the Appropriate App Permissions in Info.plist*.

Set the Appropriate App Permissions in Info.plist

The Live Experience widget requires iOS privacy permissions.

The Live Experience widget requires the following iOS permissions:

- Privacy - Camera Usage Description
- Privacy - Microphone Usage Description

1. Expand your application folder within the project view and select **Info.plist**.
2. Select any row in the Information Property List and click **+**.
3. From the drop down list, choose **Privacy - Camera Usage Description**.
4. Add a description of the way in which your application will use the camera in the **Value** field.
5. Click the **+** button again, and choose **Privacy - Microphone Usage Description** from the drop down list.

6. Add a description of the way in which your application will use the microphone in the **Value** field.

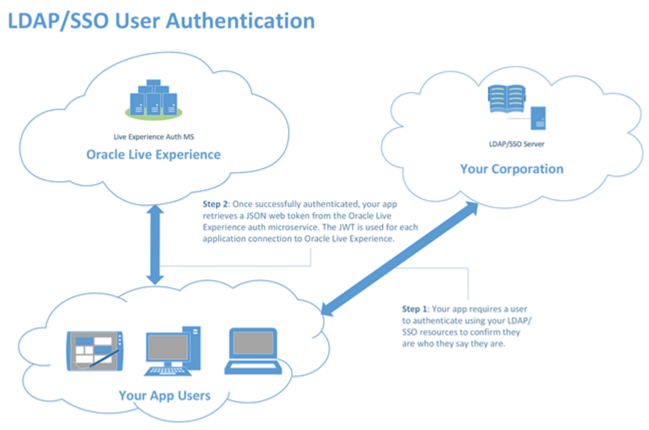
What to do next

Next, *Authenticate with Live Experience for iOS*.

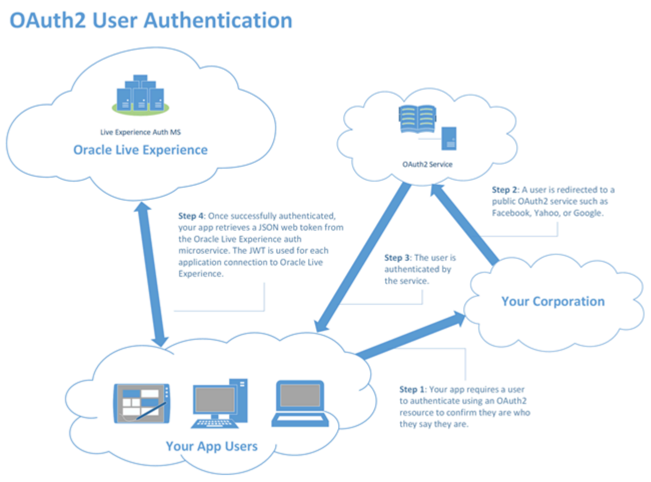
Authenticate with Live Experience for iOS

Set up User Authentication and Live Experience Authentication, which are two necessary steps for adding Live Experience to your iOS app.

User Authentication refers to confirming that a user who's accessing your application is who they say they are. Your greater organization will determine how that authentication is handled. For instance, you may need to use OAuth authentication through a third-party provider such as Google, Facebook, or Yahoo. In other cases, you may use your own organization's single sign-on (SSO) facilities, some sort of LDAP system, or during development, you may use a simple username and shared password. In any case, you'll need to design an appropriate user interface workflow along with the supporting backend code.
The following graphic shows the flow if you're using your own SSO/LDAP workflow.


LDAP/SSO User Authentication
Live Experience Auth MS
Oracle Live Experience
LDAP/SSO Server
Your Corporation
Step 2: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.
Step 1: Your app requires a user to authenticate using your LDAP/SSO resources to confirm they are who they say they are.
Your App Users

The following graphic shows the flow if you're using a third party OAuth platform such as Facebook, Google, or Yahoo.


OAuth2 User Authentication
Live Experience Auth MS
Oracle Live Experience
OAuth2 Service
Step 2: A user is redirected to a public OAuth2 service such as Facebook, Yahoo, or Google.
Step 4: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.
Step 3: The user is authenticated by the service.
Your Corporation
Step 1: Your app requires a user to authenticate using an OAuth2 resource to confirm they are who they say they are.
Your App Users

After you authenticate a user, assuming it's required for your application, you then need to handle Live Experience Authentication. To authenticate with Live Experience, you obtain a JSON Web Token (JWT) from Live Experience which you use when opening any connection. You use standard Swift APIs to communicate with a simple script that you deploy on a web server in your own domain. See *Deploy the Sample JWT Script*. While the supplied script is sufficient for development, you'll want to create something more secure for a production environment using the REST call described in *Retrieve a JWT Access Token Using the Auth REST Call*.

Follow the steps below to obtain a JWT from Live Experience.

1. Deploy the script as described in *Deploy the Sample JWT Script*.
2. Retrieve the JWT from the script's return value using code similar to the following:

```
let rqst = NSMutableURLRequest(url: URL(string: "https://your-server/cgi-bin/auth.sh")!)
let session = URLSession.shared
rqst.httpMethod = "GET"
rqst.addValue("application/json", forHTTPHeaderField: "Accept")
_ = session.dataTask(with: rqst as URLRequest,
 completionHandler: {data, response, error -> Void in
 guard let data = data, let _ = response, error == nil else { return }
 do {
 if let json = try JSONSerialization.jsonObject(with: data) as?
 [String: Any] {
 let access_token = json["access_token"] as? [[String: Any]] ?? []
 print(access_token)
 // Pass access_token to the authentication method...
 }
 } catch let error as NSError {
 print(error)
 }
})
```

Results:

After you obtain the JWT `access_token`, you can use it to authenticate with Live Experience. See *Adding and Configuring the Live Experience Widget for your iOS App*.

Adding and Configuring the Live Experience Widget for your iOS App

Your customers initiate engagements with your associates by tapping on the Oracle Live Experience widget.

Learn how to add and configure the widget so that it provides the exact experience you intend to deliver to your customers through your iOS app.

We've created an iOS SDK to make this task as simple and straightforward as possible.

1. Extend your existing iOS app to include the Live Experience framework.
2. Add the widget.
3. Configure the widget in as many views as you need.

These instructions are for the Swift programming language. For Objective-C, see *Add and Configure the Live Experience Widget for your iOS App in Objective-C*.

Start by configuring the widget with service information so it can connect to Oracle Live Experience. Then you configure the widget service settings.

Import the widget framework:

```
import OracleLive
```

```
Configure the widget service settings:
Controller.shared.service.userID = "bob@example.com
Controller.shared.service.tenantID = "MyTenantID"
Controller.shared.service.clientID = "clientID"
Controller.shared.service.authToken = "MyAuthToken"
Controller.shared.service.address = "cloudURL"
```

The example above sets the end user ID, the tenant ID, the Live Experience application client ID retrived from the Admin Console by selecting Application and then Details, the `authToken` retrieved from *Authenticate with Live Experience for iOS*, and the Live Experience cloud URL (either https://live.oraclecloud.com or https://emea.live.oraclecloud.com).

For more information on service configuration, see the *Live Experience iOS API Reference*.

> **Note:** The example above uses hard coded strings. In a production application, you'll want to store such items in a string table, or request them from a separate user interface layout and activity. When you test the integration of the mobile widget into your application and make your first call, ensure the initialized userID isn't the same as the email address used to log into the Associate Desktop.

Related Topics

- Configure Context Information for your iOS App

Configure Context Information for your iOS App

The Live Experience widget lets you define a variety of context information that's relayed to the Associate Desktop and displayed to an associate when in a call.

The context information can include the caller's name and email address, phone number, and geographical location, which is then displayed for each caller in the caller information pane. For more information on context configuration, see the *Live Experience iOS API Reference*.

1. To configure a particular piece of context information, you'll use the `contextAttributes setValue` method, and specify a key indicating the name of the context attribute to set and a value indicating the value of that context attribute.

   The following table lists the default context attribute key and value pairs you can configure.

   Default Context Attribute Key and Value Pairs

Key	Value Example	Notes
appLocation	"Support Tab"	The appLocation value should match the name of a value configured for Application Location in the Admin Console. For information on configuring an Application Location on an engagement scenario, see Manage Engagement Scenarios.
email	"bob@example.com"	Retrieved from your application's UI.
fullName	"Bob Dobbs"	Retrieved from your application's UI.
location	"Scottsdale, AZ"	Retrieved via the device's location API.
phone	"415-555-1212"	Retrieved from your application's UI.

Key	Value Example	Notes

2. To configure context information for the widget, use the `contextAttributes.setValue()` method:

   ```
   // Optional context attributes...
   Controller.shared.contextAttributes.setValue("Bob Dobbs", forKey: "fullName")
   Controller.shared.contextAttributes.setValue("bob@example.com", forKey: "email")
   Controller.shared.contextAttributes.setValue("202-555-0171", forKey: "phone")
   Controller.shared.contextAttributes.setValue("Support Tab", forKey: "appLocation")
   ```

   The example above sets context information for a user's name, email address, and phone number, which are displayed to associates when calls come into the Associate Desktop. The example also sets the location in the app from where the call originates.

What to do next

Next, *Add the Live Experience Widget to your iOS App*.

Add the Live Experience Widget to your iOS App

Follow this procedure for any controller in which you'd like the Live Experience widget to appear.

1. If you haven't already, import the Live Experience iOS framework: `import OracleLive`.
2. For each view controller from which a user can either start a call or navigate to while a call is in progress, update `viewDidAppear` with the following lines of code:

   ```
   override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    Controller.shared.addComponent(viewController: self)
   }
   ```

> **Note:** If you'd like to use a different `appLocation` for different view controllers (for example, to indicate that a user has contacted an associate from a support tab as opposed to a sales tab), you can update the `appLocation` context attribute in `viewDidAppear` adding the statement before `addComponent`.

What to do next

Next, *Add Meeting Support to Your iOS App*.

Add Meeting Support to Your iOS App

When you enable Live Experience meeting support in your app, you provide a URL to a customer that they can click to join a meeting instance that you configured.

Meeting URLs are composed of a base URL and a meeting code. The base URL is application-specific and is entered in the Admin Console, for example, orcl://www.example.com/meeting. When an associate creates a meeting and sends the URL to the user, the final URL might look like orcl://www.example.com/meeting/DGGD2KAU.
In order to process meeting URLs, you need configure the base URL to automatically load your Live Experience-enabled app, and pass the meeting code to it using either *https://developer.apple.com/documentation/uikit#//apple_ref/doc/uid/TP40007072-CH6-SW1* or *https://developer.apple.com/library/archive/documentation/General/Conceptual/AppSearch/UniversalLinks.html*.

Follow this procedure to build a single viewcontroller meeting app and process meeting URLs using the URL schemes method.

1. Create a new Single View App project in Xcode.
2. Configure the project settings as described in *Preparing Your iOS App for the Live Experience Widget*.
3. Expand your application folder within the project view and select **Info.plist**.
4. Select any row in the Information Property List and click **+**.
5. Enter **URL types**.
6. Expand **URL types**, then **Item 0**, and click **+**.
7. Enter **URL Schemes**, expand **Item 0**, and set `orcl` as a string value.
8. Configure the Controller shared service settings as described in *Adding and Configuring the Live Experience Widget for your iOS App* in the UIViewController `viewDidLoad` callback.
9. Set the following additional controller shared settings:

   ```
   Controller.shared.settings.startVideoInFullScreen = true
   Controller.shared.settings.startVideoWithFrontCamera = true
   ```

10. Handle authentication in the `viewDidLoad` callback as described in *Authenticate with Live Experience for iOS*.
11. In the `viewDidLoad` callback, initialize the `appLocation` as Meeting, a default scenario which enables Live Experience meeting functionality: `Controller.shared.contextAttributes.setValue("Meeting", forKey: "appLocation")`
12. Configure your app to extract the meeting code when the URL is tapped by overriding the `application(_:open:options:)` method in `UIApplicationDelegate` (generated by Xcode as `AppDelegate.swift` by default). That method is called when the app is opened for the URL scheme you defined earlier.

    The following example extracts the meeting code and posts it to a notification:

    ```
    func application(_ app: UIApplication,
     open url: URL,
     options: [UIApplicationOpenURLOptionsKey : Any] = [:])
     -> Bool {
     let components = url.absoluteString.components(separatedBy:"/")
     if components.count > 2 {
     var notification = Notification.init(name: NSNotification.Name("MeetingNotification"))
     notification.object = components.last
     NotificationCenter.default.post(notification)
     }
     return true
    }
    ```

13. Configure the view controller to display the widget when the notification comes in. In the view controller `viewDidLoad` callback, register yourself as a listener for the meeting notifications:

    ```
    NotificationCenter.default.addObserver(
     self,
     selector: #selector(MeetingViewController.onMeeting(notification:)),
     name: NSNotification.Name("MeetingNotification"),
     object: nil)
    ```

14. Implement the meeting processing method callback, `onMeeting`:

    ```
    @objc private func onMeeting(notification: NSNotification) {
     if let meetingCode = notification.object as? String {
     Controller.shared.contextAttributes.setValue(meetingCode,
     forKey: "meetingCode")
     Controller.shared.addComponent(viewController: self)
     }
    }
    ```

15. You'll need to handle timing issues, since there's no guarantee that you've been authenticated before you try to start the meeting. Add the following variables to the view controller:

```
private var authenticated = false
private var meetingCode: String? = nil
```

16. Modify the `completionHandler` for the authentication step as well as the `onMeeting` callback so that either of them start the meeting depending on which one completes first.

    Add the following to `completionHandler`:

```
// Last part of the authentication
Controller.shared.service.authToken = token
Controller.shared.contextAttributes.setValue("Meeting", forKey: "appLocation")
self.authenticated = true
if let _ = self.meetingCode {
 Controller.shared.addComponent(viewController: self)
}
```

    Revise `onMeeting` as follows:

```
// Revised version of onMeeting
@objc private func onMeeting(notification: NSNotification) {
 if let meetingCode = notification.object as? String {
 Controller.shared.contextAttributes.setValue(meetingCode,
 forKey: "meetingCode")
 if self.authenticated == true {
 Controller.shared.addComponent(viewController: self)
 } else {
 self.meetingCode = meetingCode
 }
 }
}
```

What to do next

Next, *Add Notification Support to your iOS App*.

Add Notification Support to your iOS App

This task builds on Apple notification documentation.

Before you start

Familiarize yourself with Apple notification documentation before undertaking any development. See *https://developer.apple.com/documentation/pushkit/supporting_pushkit_notifications_in_your_app*.

The app needs to register the device to receive notifications by obtaining a unique token, as shown in the following example:

```
final public class Controller : NSObject {
 /**Register this device to our server with Apple APNS server when we get the deviceToken, then we can
 receive notification from server.*/
 public func registerDeviceForPushNotifiction(applicationID: String, deviceID: String, deviceToken: String,
 userID: String)
 /** Report the incoming call to OracleLive widget.*/
 @available(iOS 10.0, *)
 public func reportIncomingCall(payload: PKPushPayload)
}
```

Then, the app needs to identify itself by providing details about the device and the configured user name, which Live Experience uses to do a lookup. Also, expand the Controller to make it catch incoming calls, as shown in the following example:

```
class AppDelegate: UIResponder, UIApplicationDelegate, PKPushRegistryDelegate {
 func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
 [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
 self.voipRegistration()
 ...
 return true
 }
 // Register for VoIP notifications
 private func voipRegistration() {
 let mainQueue = DispatchQueue.main
 // Create a push registry object
 let voipRegistry: PKPushRegistry = PKPushRegistry(queue: mainQueue)
 // Set the registry's delegate to self
 voipRegistry.delegate = self
 // Set the push type to VoIP
 voipRegistry.desiredPushTypes = [PKPushType.voIP]
 }
 // implement this from PKPushRegistryDelegate, called when we got the device token
 func pushRegistry(_ registry: PKPushRegistry, didUpdate pushCredentials: PKPushCredentials, for type:
 PKPushType) {
 let pushtoken = pushCredentials.token
 let token = pushtoken.map { String(format: "%02.2hhx", $0) }.joined()
 AppState.deviceToken = token
 Log.debug("voip get pushCredentials: \(token)")
 let userID = UserDefaults.userEmailAddress
 let clientID = UserDefaults.applicationId
 let deviceID = UIDevice.current.identifierForVendor!.uuidString
 if !AppState.authToken.isEmpty {
 Controller.shared.registerDeviceForPushNotifiction(applicationID: clientID, deviceID: deviceID,
 deviceToken: token, userID: userID)
 }
 }
 // implement this from PKPushRegistryDelegate, called when we receive the incoming call notification
 func pushRegistry(_ registry: PKPushRegistry, didReceiveIncomingPushWith payload: PKPushPayload, for type:
 PKPushType, completion: @escaping () -> Void) {
 Log.debug("voip notice got.....")
 if #available(iOS 10.0, *) {
 if AppState.authToken.isEmpty {
 let settings = AuthenticationUtils.retrieveAuthSettings()
 let authCompletion: (String?, Error?) -> Void = { (token, error) in
 guard token != nil else {
 return
 }
 AppState.authToken = token!
 Controller.shared.service.authToken = AppState.authToken
 Controller.shared.reportIncomingCall(payload: payload)
 }
 AuthenticationUtils.authenticate(withType: AuthType.guest, withSettings: settings, withPassword: nil,
 completionHandler: authCompletion)
 }else{
 Controller.shared.reportIncomingCall(payload: payload)
 }
 } else {
 // Fallback on earlier versions
 }
}
```

What to do next

Next, *Additional Configuration Options for your iOS App*.

Additional Configuration Options for your iOS App

In addition to context attributes, you can control other widget behaviors.

Additional configuration options for your iOS App are listed in the following table.

Additional Configuration Options for your iOS App

Method	Description
Controller.shared.settings.startVideoInFullS	Choose whether the video engagement starts in full-screen mode (true), or as an overlay (false). Default value: false
Controller.shared.settings.startVideoWithFr	Choose whether to start video engagements with the device's front camera (true), or the back camera (false). Default value: true
Controller.shared.settings.startCallDirectly	Choose whether to start an engagement automatically when the customer opens the view containing the widget. When set to true, the widget is hidden from the customer. Default value: false
Controller.shared.settings.hidePauseButton	Choose whether to hide the pause button from the customer on the mobile widget, preventing them from pausing the call. Default value: false
Controller.shared.settings.hideEndCallButto	Choose whether to hide the end-call button from the customer in the mobile widget, preventing them from ending the call themselves. Default value: false
Controller.shared.settings.hideQueueLengt	Choose whether to hide the queue length indicator from the customer as they wait to be connected to an associate. Default value: false
Controller.shared.settings.attemptFaceMatc	In the context of verifying the identity of your customers, use this setting to enable the facial recognition capability provided by Jumio. See Implementing Identity Verification in a Web or Mobile App for more info. Default value: false
Controller.shared.service.skipRecordingPerı	Disable the recording permission request warning presented to customers before they can join a recorded call. Default value: false

For more information on context configuration, see the *Live Experience iOS API Reference*.

Add and Configure the Live Experience Widget for your iOS App in Objective-C

This topic provides guidelines and an overview for using Objective-C to add and configure the widget in your iOS app.

Before you start

Make sure your development environment is properly set up (see *Preparing Your iOS App for the Live Experience Widget*). This procedure builds on the concepts and detailed procedure for including and configuring the widget using the Swift programming language. You should read and familiarize yourself with *Adding and Configuring the Live Experience Widget for your iOS App* before continuing.

Your customers initiate engagements with your associates by tapping on the Oracle Live Experience widget. Add and configure the widget so that it provides the exact experience you intend to deliver to your customers through your iOS app.
In our experience, Xcode is slow to fully initialize the Live Experience framework when working in Objective-C.

Use the following procedure as a reference for educating Xcode into progressively exploring the framework. You should only need to run through this procedure one time. Once Xcode has fully discovered the Live Experience framework in one project, you can add and configure the Live Experience widget in your other projects.

Here's what to do

1. In Xcode, create a new, simple Objective-C project such as a "Hello-World" project.
2. In the general settings for your project, under Linked Frameworks and Libraries, include all your frameworks and libraries, and include the Live Experience frameworks.
3. In your project, import the file **OracleLive/OracleLive-Swift.h**: `#import "OracleLive/OracleLive-Swift.h"`
4. In your Xcode build settings, set Enable Bitcode to **No** and the project to build on a real device or on a generic iOS device.
5. Build the project.
6. In your project, in the `viewDidLoad` method, enter **Cont**.
   Xcode should suggest to complete the `Controller` class for you.
7. Enter **.sha**.
   Xcode should suggest attaching the shared property on the class so that it becomes `Controller.shared`.

Results:

If Xcode is suggesting classes and properties belonging to the Live Experience framework, then you can carry on adding and configuring the Live Experience widget to your other Objective-C projects.
If Xcode does not suggest auto-completing the classes and properties from the Live Experience frameworks, check that you are building the project on a real device or on a generic iOS device, not an iOS simulator. If that doesn't work, try restarting Xcode.

The following sample code shows you how to configure the widget service settings in your Objective-C project:

```
Controller.shared.service.userID = @"bob@example.com";
Controller.shared.service.tenantID = @"MyTenantID";
Controller.shared.service.clientID = @"clientID";
Controller.shared.service.authToken = @”MyAuthtoken”;
Controller.shared.service.address = @"cloudURL";
```

The following sample code shows you how to set optional context attributes in your Objective-C project:

```
[Controller.shared.contextAttributes setValue: @"Bob Dobbs" forKey: @"fullName"];
[Controller.shared.contextAttributes setValue: @"bob@example.com" forKey: @"email"];
[Controller.shared.contextAttributes setValue: @"202-555-0171" forKey: @"phone"];
[Controller.shared.contextAttributes setValue: @"Support Tab" forKey: @"appLocation"];
```

iOS Client Development Quick Start

Quickly embed the Oracle Live Experience widget into your iOS app.

Before continuing, make sure you meet the following prerequisites:

- Xcode version 10.2 or higher.
- Swift version 5.0 or higher.
- The Live Experience iOS SDK downloaded and extracted to a folder on your computer.
- Credentials (client_id/client_secret) to access a provisioned application on an active Live Experience Tenant account.
- An Apple device running iOS 10 or above.
- Some experience with the Swift programming language.

To embed the Live Experience widget in your iOS app, complete the following tasks:

1. *Create a New Xcode Project*
2. *Authenticate and Add the Live Experience Widget*
3. *Test Your App*
4. *Experiment with Live Experience Features*

Create a New Xcode Project

Create a new Xcode project.

1. Select **File** > **New** > **Project...**.
2. Select **Single View App**.
3. Ensure that Swift is selected as the development language.
4. Name your project and location, and click **Create**.
   Xcode will start in the Project Navigator with the General tab already open.
5. Scroll to the Embedded Binaries section.
6. Click **+** and then **Add Other...**.
7. Find the **live-experience-ios-sdk** folder you extracted from the iOS SDK zip.
8. Open the release folder, select all three framework files inside, and click **Open**.
9. Make sure **Copy items if needed** is selected when prompted, and click **Finish**.
10. Click the **Build Settings** tab.
11. Enter **bitcode** into the **Search** field of the tab.

12. Change the Enable Bitcode build option from Yes to **No**.
13. Request permission to access the camera and microphone in our app.
    a. Click the **Info.plist** file from the tree view list of the project's files on the left pane of the Xcode window.
       The file displays in the center pane.
    b. Hover your mouse over **Information Property List** and click **+**.
    c. Start entering **Privacy** (with a capital P) and select the option named **Privacy - Camera Usage Description**.
    d. Set the value of this option to the text you want to display to the end-user when requesting camera access. For example, **Can I use your camera?**.
    e. Hover your mouse again over **Information Property List** and click **+**.
    f. Start entering **Privacy** (with a capital P) and select the option named **Privacy - Microphone Usage Description**.
    g. Set the value of this option to the text you want to display to the end-user when requesting microphone access. For example, **Can I use your microphone?**.

Authenticate and Add the Live Experience Widget

Now that your Xcode project is prepared, you are ready to authenticate to Live Experience and add the widget to the UI of your iOS app.

1. Click the **ViewController.swift** file from the tree view list of the project's files on the left pane of the Xcode window.
   The file displays in the center pane.
2. Locate the line `import UIKit` and add `import OracleLive` below it.
3. Locate the `viewDidLoad()` function and paste the following code below the line that says `// Do any additional setup after loading the view, typically from a nib`, and replace the following values with your own:
   - `client_id`: replace with your client ID
   - `client_secret`: replace with your client secret
   - `tenantID`: replace with your assigned tenant name
   - `userID`: replace with your user ID

```
// Application credentials from LX console
let client_id = "1fntfqg6k45nh4o8t3rt"
let client_secret = "379709dd-c230-42d4-9f7a-7564501ccfb9"
let loginString = String(format: "%@:%@", client_id, client_secret)
let loginData = loginString.data(using: String.Encoding.utf8)!
let base64LoginString = loginData.base64EncodedString()
let url = URL(string: "https://live.oraclecloud.com/auth/apps/api/access-token?
grant_type=client_credentials&nonce=&state=12345&scope=optional")!
var request = URLRequest(url: url)
// REST call requires Basic Auth
request.httpMethod = "GET"
request.setValue("Basic \(base64LoginString)", forHTTPHeaderField: "Authorization")
let task = URLSession.shared.dataTask(with: request) { data, response, error in
 guard let data = data, error == nil else {
 print("XXXXXXXXX URLRequest error: \(error!.localizedDescription)")
 return
 }
 if let httpStatus = response as? HTTPURLResponse {
 // Check status code returned by the http server. Should be 200
 print("XXXXXXXXX HTTP status code: \(httpStatus.statusCode)")
 // Process result
 do {
```

```
 let json = try JSONSerialization.jsonObject(with: data, options: .allowFragments) as? [String:Any]
 if let token = json?["access_token"] as? String {
 Controller.shared.service.authToken = token
 // LX Configuration
 Controller.shared.service.tenantID = "Your_Tenant_Name"
 Controller.shared.service.userID = "yourid@example.com"
 Controller.shared.contextAttributes.setValue("Collaboration", forKey: "appLocation")
 // Display the Widget
 Controller.shared.addComponent(viewController: self)
 } else {
 print("XXXXXXXXX Error: \(json?["description"] ?? "Unknown")")
 }
 } catch let error as NSError {
 print("XXXXXXXXX JSON parse errors: \(error)")
 }
 }
}
task.resume()
```

Results:

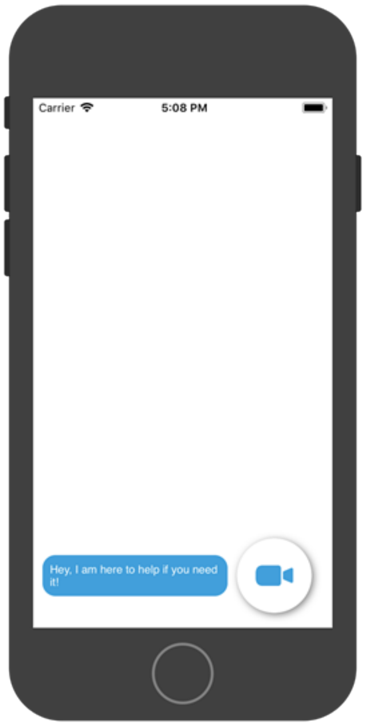
This code authenticates you with Live Experience using your tenant name (Your_Tenant_ID, which you replace with your own tenant name). After the authentication is successful, the Live Experience widget is added to your ViewController and displays to your customers using your app.

Test Your App

Test your app on your device or simulator.

1. Click **Play** in the Xcode interface to run your project.

   If all went well, your device or simulator looks like this:



2. Call an associate.

   You were emailed an Associate account and instructions to install the Live Experience demo app. Follow those instructions and make sure your phone, using the Live Experience demo app, and browser can talk to each other.

Experiment with Live Experience Features

Now that you have a working iOS app, start experimenting with Live Experience features.

1. Add the following lines to your code to configure camera options.

```
Controller.shared.settings.startVideoInFullScreen = true

Controller.shared.settings.startVideoWithFrontCamera = true
```

2. The authentication code in *Authenticate and Add the Live Experience Widget* selects the Collaboration scenario. Scenarios control which channels (video, audio, screen share) are available. Try other pre-configured scenarios by replacing `Collaboration` in the code with one of the following:
   - `Basic_Guidance`
   - `Remote_Support`
   - `Personal_Shopper`
   - `Concierge`
   - `Short_Code`
   - `Meeting`
3. When you call an associate with your app, you see Not Available and Unknown User in the engagement information pane. Add context attributes to customize and pass that information to the associate:

```
Controller.shared.contextAttributes.setValue("John Smith", forKey: "fullName")
Controller.shared.contextAttributes.setValue("john.smith@ example.com", forKey: "email")
Controller.shared.contextAttributes.setValue("+1-202-555-0171", forKey: "phone")
Controller.shared.contextAttributes.setValue("Redwood City, CA", forKey: "location")
```

3 Add Live Experience Features to Android Apps

Preparing Your Android App for the Live Experience Widget

Set up your application environment in preparation for adding Live Experience to your Android app.

This is one part in a series of steps required to get Live Experience up and running in your Android apps.

- Prepare your Android app for the Live Experience widget: *Add the Live Experience Framework to Your Android Studio Project*, *Import the Mobile Component Framework into Your Android Application*, and *Add the Mobile Component to Your Android Layouts*
- *Authenticate with Live Experience for Android*
- *Adding and Configuring the Live Experience Widget for your Android App*

Before integrating your Android app with Oracle Live Experience, verify that your existing Android app and development environment meet the following criteria:

- Android Studio version 3.0.1 or above installed, with Android SDK version 28 or above and Android Build Tools version 28.0.3 or above.
- A valid tenant account for Live Experience.
- Credentials from the Live Experience Admin Console needed to obtain a JSON Web Token (JWT) from Live Experience.
- A copy of the Live Experience Android SDK downloaded to your development machine.
- An existing Android app targeted at Android version 5.1 (Lollipop) or higher.
- Physical and virtual devices targeting the ARM architecture:
  - An actual Android hardware device.
  - Virtual devices as required for your application. See *https://developer.android.com/studio/run/managing-avds.html* and *https://developer.android.com/studio/run/device.html*.

    Note: You can test the general flow and function of your Live Experience Android application using the Android simulator, however, a physical Android device such as a phone or tablet is required for audio or video functionality.

Add the Live Experience Framework to Your Android Studio Project

Add the Live Experience support framework to your Android Studio project.

1. Extract the following files from the Android SDK ZIP file and add them to your project **libs** directory:

  - **oracle.wsc.feature.clientsdk.android-7.2.1.1-SNAPSHOT.jar**
  - **peerconnection_android-66.0.3359.117-b20aef0d47bf3924adfc7bfbee707f6f137670fb.jar**
  - **tyrus-standalone-client-1.13.jar**
  - **oracle.live.api-release.aar or oracle.live.api-debug.aar**

2. Update your project dependencies to include the extracted files in the **libs** directory. Add one of the following blocks of code to your **build.gradle** file.

   -
     ```
     dependencies {
      implementation fileTree(dir: 'libs', include: '*.jar')
      implementation fileTree(dir: 'libs', include: '*.aar')
     }
     ```

   -
     ```
     dependencies {
      implementation files('libs/oracle.wsc.feature.clientsdk.android-7.2.1.1-SNAPSHOT.jar')
      implementation files('libs/peerconnection_android-66.0.3359.117-
     b20aef0d47bf3924adfc7bfbee707f6f137670fb.jar')
      implementation files('libs/tyrus-standalone-client-1.13.jar')
      implementation files('libs/oracle.live.api-release.aar')
     }
     ```
3. Add third-party library dependencies to your project. Add the following block of code to your **build.gradle** file.
   ```
   dependencies {
    implementation 'androidx.appcompat:appcompat:versionA'
    implementation 'com.google.android.gms:play-services-location:versionB'
    implementation 'org.slf4j:slf4j-log4j12:versionD'
   }
   ```
   Where `versionA` is 1.0.0-beta01 or greater, `versionB` is 17.0.0 or greater, `versionC` is version 74.0.3729.108 or greater, and `versionD` is 1.7.25 or greater.
4. Open your Android Studio project and from the File menu, select **New** > **New Module**.
5. In New Module, select **Import JAR/.AAR Package** and click **Next**.
6. Navigate to the location of the AAR file, and choose either the debug or release version depending on your requirements.
   Optionally, enter a different name in the Subproject name edit box if you like.
7. Click **Finish**.
8. In the Project pane, right-click your application name and select **Open Module Settings**.
9. Select the **Dependencies** tab, click **+** on the right side and then select **Module dependency**.
10. Add your new module to the project dependencies and then click **OK**.
11. Add the following Android permissions to **AndroidManifest.xml**:
    ```
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
    ```

What to do next

Next, *Import the Mobile Component Framework into Your Android Application*.

Import the Mobile Component Framework into Your Android Application

After you add the mobile component framework in your project, you need to import it in each activity that will use the mobile component before you can access its functionality.

1. In Android Studio, open the Java source file for the activity.
2. Add the following line to the import section of your the activity: `import oracle.live.api.CommunicationFragment;`.
3. Save the file.

What to do next

Next, *Add the Mobile Component to Your Android Layouts*.

Add the Mobile Component to Your Android Layouts

Add the Live Experience widget to the following types of Android layouts.

You can add the Live Experience widget to the following types of Android layouts:

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`
- `GridLayout`

To add the widget to a layout, add the following element:

```
<LinearLayout android:id="@+id/cx_container"
 android:orientation="horizontal"
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:layout_alignParentBottom="true">
</LinearLayout>
```

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 xmlns:tools="http://schemas.android.com/tools"
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:background="@android:color/white">
 ... other layout bits as required for your application...
 <LinearLayout android:id="@+id/cx_container"
 android:orientation="horizontal"
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:layout_alignParentBottom="true">
 </LinearLayout>
</RelativeLayout>
```

Note: You can customize the layout where you've added the Live Experience widget. For information on customizing Android layouts, see *https://developer.android.com/guide/topics/ui/themes.html*.

What to do next

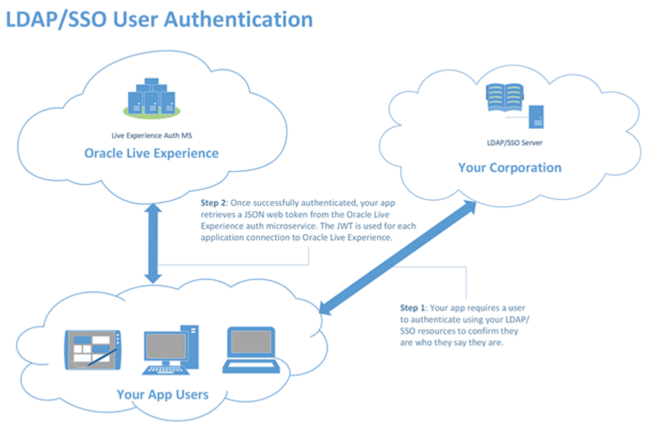
Next, *Authenticate with Live Experience for Android*.

Authenticate with Live Experience for Android

Set up User Authentication and Live Experience Authentication, which are two necessary steps for adding Live Experience to your Android app.

User Authentication refers to confirming that a user who's accessing your application is who they say they are. Your greater organization will determine how that authentication is handled. For instance, you may need to use OAuth authentication through a third-party provider such as Google, Facebook, or Yahoo. In other cases, you may use your own organization's single sign-on (SSO) facilities, some sort of LDAP system, or, during development, you may use a simple username and shared password. In any case, you'll need to design an appropriate user interface workflow along with the supporting backend code.

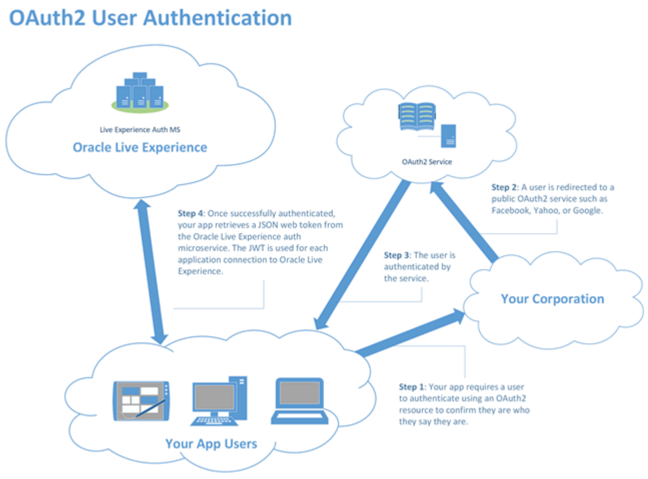
The following graphic shows the flow if you're using your own SSO/LDAP workflow.



Step 2: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.

Step 1: Your app requires a user to authenticate using your LDAP/SSO resources to confirm they are who they say they are.

And the following graphic shows the flow if you're using a third party OAuth platform such as Facebook, Google, or


OAuth2 User Authentication
Live Experience Auth MS
Oracle Live Experience
OAuth2 Service
Step 2: A user is redirected to a public OAuth2 service such as Facebook, Yahoo, or Google.
Step 4: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.
Step 3: The user is authenticated by the service.
Your Corporation
Step 1: Your app requires a user to authenticate using an OAuth2 resource to confirm they are who they say they are.
Your App Users

Yahoo.

After you authenticate a user, assuming it's required for your application, you then need to handle Live Experience Authentication. To authenticate with Live Experience, you obtain a JSON Web Token (JWT) from Live Experience which you use when opening any connection. You use standard Android Java APIs to communicate with a simple script that you deploy on a web server in your own domain. See *Deploy the Sample JWT Script* for info about deploying the sample script. While the supplied script is sufficient for development, you'll want to create something more secure for a production environment using the REST call described in *Retrieve a JWT Access Token Using the Auth REST Call*.

Follow the steps below to obtain a JWT from Live Experience.

1. Deploy the script as described in *Deploy the Sample JWT Script*.
2. Retrieve the JWT from the script return value using code similar to the following:

```
URL url = new URL("https://your-server/cgi-bin/auth.sh");
String response = new StringBuilder();
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
try {
 BufferedReader input = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));
 String line = null;
 while ((line = input.readLine()) != null) {
 response.append(line);
 }
 input.close();
 JSONArray arr = new JSONArray(response);
 JSONObject jObj = arr.getJSONObject(0);
 String access_token = jObj.getString("access_token");
 Log.i("Access token: ", access_token)
 // Pass access_token to the authentication method...
} finally {
 urlConnection.disconnect();
}
```

Results:

After you obtain the JWT access_token, you can use it to authenticate with Live Experience as described in *Adding and Configuring the Live Experience Widget for your Android App*.

Adding and Configuring the Live Experience Widget for your Android App

Tapping on the Oracle Live Experience widget is how your customers initiate engagements with your associates.

Add and configure the widget so that it provides the exact experience you intend to deliver to your customers through your Android app.

We created an Android SDK to make this task as simple and straightforward as possible.

1. Extend your existing Android app to include the Live Experience framework.
2. Configure the widget with service information so it can connect to Live Experience.
3. Retrieve a `CommunicationFragment` object and modify the required settings, typically in the `onCreate` event handler in your application's main activity.

To get started, initialize a `CommunicationFragment` in your activity to use when you actually want to display the widget:

```
protected static final String FRAGMENT_TAG = CommunicationFragment.class.getName();
private CommunicationFragment mCxFragment;
```

Set the service parameters in your `onCreate` event handler:

```
public final void onCreate() {

CommunicationFragment.service.setUserID("user-name");
CommunicationFragment.service.setTenantID("myTenantName");
CommunicationFragment.service.setClientID("myClientID");
CommunicationFragment.service.setAuthToken("myAuthToken");
CommunicationFragment.service.setAddress("cloudURL")

}
```

The example above sets the user ID, the tenant ID, the Live Experience application client ID (retrieved from **Application** > **Details** on the Admin Console), the `authToken` retrieved from *Authenticate with Live Experience for Android*, and the Live Experience cloud URL (either https://live.oraclecloud.com or https://emea.live.oraclecloud.com).

For more information on service configuration, see the *Live Experience Android API Reference*.

Note: The example above uses hard coded strings. In a production application, you'll want to store such items in a string table, or request them from a separate user interface layout and activity.

When you test the integration of the mobile widget into your application and make your first call, ensure the initialized userID isn't the same as the email address used to log into the Associate Desktop.

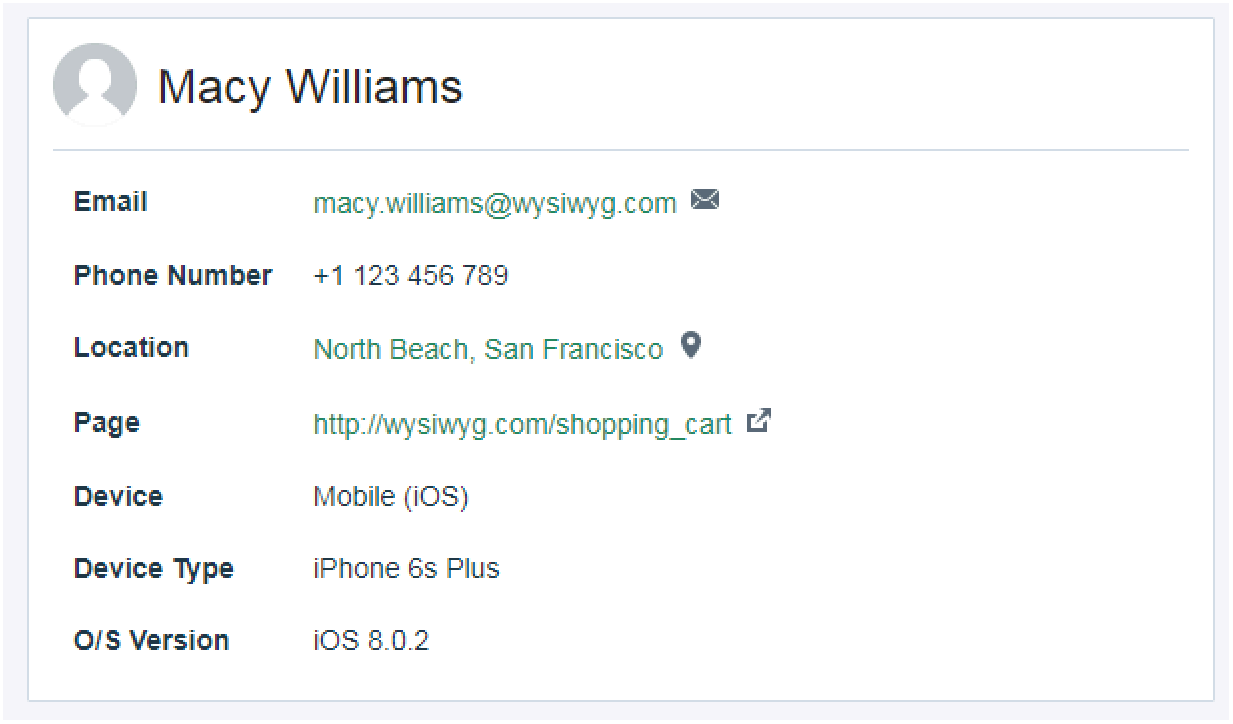
Related Topics

- Additional Configuration Options for your Android App
- Configure Context Information for your Android App
- Initialize and Display the Live Experience Widget for your Android App
- Add Meeting Support to your Android App
- Add Notification Support to your Android App

Configure Context Information for your Android App

The Live Experience widget lets you define a variety of context information that's relayed to the Associate Desktop and displayed to an associate when in a call.

The context information can include the caller's name and email address, phone number, and geographical location, which is then displayed for each caller in the caller information pane.



Macy Williams

Email	macy.williams@wysiwyg.com
Phone Number	+1 123 456 789
Location	North Beach, San Francisco
Page	http://wysiwyg.com/shopping_cart
Device	Mobile (iOS)
Device Type	iPhone 6s Plus
O/S Version	iOS 8.0.2

1. To configure a particular piece of context information, use the `contextAttributes` set method, and specify a key indicating the name of the context attribute to set and a value indicating the value of that context attribute.

   The following table lists the default context attribute key/value pairs you can configure.

   Default Context Attribute Key and Value Pairs

Key	Value Example	Notes
appLocation	"Support Tab"	The appLocation value should match the name of a value configured for Application Location in the Admin Console. For information on configuring an Application Location on an engagement scenario, see Manage Engagement Scenarios.
email	"bob@example.com"	Retrieved from your application's UI.

Key	Value Example	Notes
fullName	"Bob Dobbs"	Retrieved from your application's UI.
location	"Scottsdale, AZ"	Retrieved via the device's location API.
phone	"415-555-1212"	Retrieved from your application's UI.

2. To configure context information for the widget, add the following code to your `onCreate` event handler following the service configuration lines:

```
// Optional context attributes...
CommunicationFragment.contextAttributes.set("fullName","Bob Dobbs");
CommunicationFragment.contextAttributes.set("email","customer@example.com");
CommunicationFragment.contextAttributes.set("phone","415-555-1212");
CommunicationFragment.contextAttributes.set("appLocation","Support Tab");
```

   This sets context information for a user's name, email address, and phone number, which are displayed to associates when calls come into the Associate Desktop. The example also sets the location in the app from where the call originates. For more information on context configuration, see the *Live Experience Android API Reference*.

What to do next

Next, *Initialize and Display the Live Experience Widget for your Android App*

Initialize and Display the Live Experience Widget for your Android App

Initialize and display the widget in your Android app.

Create a new instance of the component fragment and insert it into the application layout in your `onCreate` handler, following the service and context attribute configuration:

```
final FragmentManager manager = getSupportFragmentManager();
private CommunicationFragment lcesFrag = (CommunicationFragment)
 manager.findFragmentByTag(FRAGMENT_TAG);
if (lcesFrag == null) {
 lcesFrag = CommunicationFragment.newInstance();
 getSupportFragmentManager()
 .beginTransaction()
 .add(R.id.cx_container, lcesFrag, FRAGMENT_TAG)
 .commit();
}
```

Note: If you'd like to use a different `appLocation` for different activities, for instance, to indicate that a user has contacted an associate from a support tab as opposed to a sales tab, you can update the `appLocation` context attribute in `onResume`, adding the statement before the `add` method.

What to do next

Next, *Add Meeting Support to your Android App*.

Add Meeting Support to your Android App

When you enable Live Experience meeting support in your app, you provide a URL to a user that they can click to join a meeting instance that you configure.

Meeting URLs are composed of a base URL and a meeting code. The base URL is application-specific and is entered in the Admin Console, for example, `orcl://www.example.com/meeting`.
When an associate creates a meeting and sends the URL to the user, the final URL might look like `orcl://www.example.com/meeting/DGGD2KAU`.

In order to process meeting URLs, you need to configure the base URL to automatically load your Live Experience-enabled app, and pass the meeting code to it using either *https://developer.android.com/training/app-links/deep-linking* or *https://developer.android.com/training/app-links/index.html*.

1. Create a new activity in your Android Studio project and define a new URI scheme:

```
<activity
 android:name=".MeetingActivity"
 android:label="title_activity_meeting"
 android:theme="@style/AppTheme.NoActionBar"
 android:parentActivityName=".HomeActivity">
 <intent-filter android:label="@string/filter_view_https">
 <action android:name="android.intent.action.VIEW" />
 <category android:name="android.intent.category.DEFAULT" />
 <category android:name="android.intent.category.BROWSABLE" />
 <!-- Accepts URIs that begin with "orcl://www.example.com/meeting" -->
 <data android:scheme="orcl"
 android:host="www.example.com"
 android:pathPrefix="/meeting" />
 </intent-filter>
</activity>
```

2. Initialize the widget. See *Initialize and Display the Live Experience Widget for your Android App*.
3. Extract the meeting code from the URL:

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 Intent intent = getIntent();
 Uri data = intent.getData();
 List<String> pathSegments = data.getPathSegments();
 CommunicationFragment.contextAttributes.set(MEETING_CODE, pathSegments.get(pathSegments.size() - 1));
}
```

4. Set the `appLocation` and `meetingCode` context attributes:

```
CommunicationFragment.contextAttributes.set("appLocation", "Meeting");
CommunicationFragment.contextAttributes.set("meetingCode", "DGGD2KAU");
```

   See *Configure Context Information for your Android App* for more information on context attributes.

Results:

The meeting starts automatically after the widget gets initialized.

What to do next

Next, *Add Notification Support to your Android App*.

Add Notification Support to your Android App

This task builds on Android existing documentation.

Before you start

Familiarize yourself with Android Firebase Cloud Messaging service and Android notification documentation before undertaking any development. See *https://developer.android.com/studio/write/firebase.html*.

You need to implement a service that extends FirebaseMessagingService. This service should override the `onNewToken` and `onMessageReceived` call-back functions, passing both of them onto the widget, as shown in the following example.

```
public class CommunicationFragment {
 /**
 * If it's the in-app call ringing.
 */
 public static final String IS_IN_APP_CALL_RINGING = "isInAppCallRinging";
 ...
 /**
 * Set client information.
 *
 * @param clientInfo the client information
 * @param context the android application context
 */
 public static void setClientInfo(ClientInfo clientInfo, Context context);
 /**
 * Set in-app call content.
 *
 * @param notificationData the notification data
 * @param context the android application context
 * @param cls the communication fragment parent activity class
 */
 public static void setNotificationContent(Map<String, String> notificationData, Context context, Class<?>
 cls);
 ...
}
public class ClientInfo {
 String tenantKey;
 String phoneNumber;
 String applicationId;
 String deviceToken;
 public ClientInfo(String tenantKey, String userName, String applicationId, String deviceToken);
}
```

Update the widget to expose APIs for `onNewToken` and `onMessageReceived` and update your mobile app to register the mobile device with the Firebase Cloud Messaging service to receive notification messages. The mobile app needs to obtain a device token, then call the Live Experience widget API to pass the token onto Live Experience, as shown in the following example.

```
public class NotificationFirebaseService extends FirebaseMessagingService {
 @Override
 public void onMessageReceived(RemoteMessage remoteMessage) {
 super.onMessageReceived(remoteMessage);
 if (remoteMessage.getData() != null) {
 CommunicationFragment.settings.forceVideoInFullScreen = false;
 CommunicationFragment.settings.startVideoInFullScreen = false;
 CommunicationFragment.settings.noEndUserCallControls = false;
 CommunicationFragment.settings.suppressAssociateInformation = false;
 CommunicationFragment.settings.omitPreview = false;
 CommunicationFragment.setNotificationContent(remoteMessage.getData(), this.getApplicationContext(),
 HomeActivity.class);
 }
 }
 ...
}
public final class MceDemoApplication extends MultiDexApplication {
 public void setClientInfo() {
 FirebaseInstanceId.getInstance().getInstanceId()
 .addOnCompleteListener(new OnCompleteListener<InstanceIdResult>() {
 @Override
 public void onComplete(@NonNull Task<InstanceIdResult> task) {
 if (!task.isSuccessful()) {
 Log.i(TAG, "getInstanceId failed", task.getException());
 return;
 }
 // Get new Instance ID token
 String token = task.getResult().getToken();
 Log.d(TAG, "Refreshed token: " + token);
 final SharedPreferences sharedPreferences =
 PreferenceManager.getDefaultSharedPreferences(MceDemoApplication.this.getApplicationContext());
 final String serverAddress = sharedPreferences.getString(MceDemoApplication.SERVER_URL, "");
 final String tenant = sharedPreferences.getString(MceDemoApplication.TENANT, "");
 final String email = sharedPreferences.getString(MceDemoApplication.EMAIL, "john.smith@example.com");
 final String fullName = sharedPreferences.getString(FULL_NAME, "John Smith");
 final String phoneNumber = sharedPreferences.getString(PHONE_NUMBER, "+1-202-555-0171");
 String appId = sharedPreferences.getString(MceDemoApplication.APPLICATION_ID, "");
 appId = SecurityHelper.getInstance().decrypt(appId);
 CommunicationFragment.service.setAddress(serverAddress);
 CommunicationFragment.service.setTenantID(tenant);
 CommunicationFragment.service.setClientID(appId);
 CommunicationFragment.service.setUserID(getUserName());
 CommunicationFragment.contextAttributes.set("email", email);
 CommunicationFragment.contextAttributes.set("fullName", fullName);
 CommunicationFragment.contextAttributes.set("phone", phoneNumber);
 final ClientInfo clientInfo = new
 ClientInfo(CommunicationFragment.service.getTenantID(),phone,CommunicationFragment.service.getApplicationID(),token
 CommunicationFragment.setClientInfo(clientInfo, MceDemoApplication.this.getApplicationContext());
 }
 });
 }
}
public class HomeActivity extends AppCompatActivity implements TaskCompletionNotifier,
 NavigationView.OnNavigationItemSelectedListener {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 Log.d(TAG, "onCreate...");
 if (getIntent().getExtras() != null
 && getIntent().getExtras().getBoolean(CommunicationFragment.IS_IN_APP_CALL_RINGING)) {
 initCxFragment();
 }
 ...
 }
 private void initCxFragment() {
 final FragmentManager manager = getSupportFragmentManager();
 mCxFragment = (CommunicationFragment) manager.findFragmentByTag(FRAGMENT_TAG);
 if (mCxFragment == null) {
 mCxFragment = CommunicationFragment.newInstance();
 getSupportFragmentManager()
 .beginTransaction()
 .add(R.id.cx_container, mCxFragment, FRAGMENT_TAG)
 .commit();
 }
 mCxFragment.setNotificationHandler(((MceDemoApplication) getApplication()).getNotificationHandler());
 }
}
```

What to do next

Next, see *Additional Configuration Options for your Android App*.

Additional Configuration Options for your Android App

In addition to context attributes, you can control other widget behaviors.

Additional configuration options are listed in the following table.

Configuration Options for Your Android App

Method	Description
CommunicationFragment.settings.startVide	Choose whether the video engagement starts in full-screen mode (true), or as an overlay (false). Default value: false
CommunicationFragment.settings.startVide	Choose whether to start video engagements with the device's front camera (true), or the back camera (false). Default value: true
CommunicationFragment.settings.startCall	Choose whether to start an engagement automatically when the customer opens the page containing the widget. When set to true, the widget is hidden from the customer. Default value: false
CommunicationFragment.settings.hidePau	Choose whether to hide the pause button from the customer on the mobile widget, preventing them from pausing the call. Default value: false
CommunicationFragment.settings.hideEnd	Choose whether to hide the end-call button from the customer in the mobile widget, preventing them from ending the call themselves. Default value: false
CommunicationFragment.settings.hideQue	Choose whether to hide the queue length indicator from the customer as they wait to be connected to an associate. Default value: false
CommunicationFragment.settings.attemptI	In the context of verifying the identity of your customers, use this setting to enable the facial recognition capability provided by Jumio. See Implementing Identity Verification in a Web or Mobile App for more info. Default value: false
CommunicationFragment.service.setSkipRe	Disable the recording permission request warning presented to customers before they can join a recorded call. Default value: false

For more information on context configuration, see the *Live Experience Android API Reference*.

4 Add Live Experience Widget to Digital Customer Service Application

Before You Begin

This chapter explains how to add the Live Experience widget to the Digital Customer Service (DCS) application.

Here's what you need to complete before you can add the Live Experience widget to the DCS application:

- *Create a New Digital Customer Service Application*.
- *Download the Live Experience JavaScript SDK*.

Create an Application in Live Experience

Create an application in Live Experience to serve as a base to configure the Live Experience Web SDK to the DCS application. Complete these steps to create an application in Live Experience:

1. Sign in to Oracle Live Experience.
2. In the navigator, click **Applications**.
3. In the Applications drawer, click **Add New Application**.

   The Application Name field appears.
4. In the **Application Name** field, enter a name and click **Register New Application**.

   The Client Secret dialog box appears.

   > **Note:** Oracle recommends that the application name matches the DCS application name.

5. Note down the Client ID and Client Secret.
6. Click **OK I've made a note**.

   The application home page opens.
7. Click the Origin Domain tab and click **Add New Domain**.
8. In the Add New Domain dialog box, enter an asterisk (*****) and click **Add**.

   > **Note:** Oracle Recommends that you set the domain to the specific DCS application's fully qualified domain name when you deploy your DCS application to production.

Add Live Experience JavaScript SDK to the DCS Application

To add Live Experience JavaScript SDK to your Digital Customer Service (DCS) application, you must create an lxsdk compress file and then upload this compressed file to your DCS application.

Create the lxsdk Compress File

1. On your machine, create a folder and name it **lxsdk**.
2. Extract the Live Experience JavaScript SDK in the lxsdk folder.
3. Right-click the lxsdk folder and click **Compress to ZIP file**.
   The lxsdk folder now appears as a compressed file.

Add the lxsdk Compressed Folder to Your DCS Application

Navigate to your DCS application created in Visual Builder Studio and complete these steps to add the lxsdk compressed file:

1. In the Web Apps pane, right-click the Resources folder and click **Import**.
2. In the Import Resources dialog box, drag and drop the lxsdk compressed file, and click **Import**.

You'll now have an lxsdk subfolder in the Resources folder which contains the same subdirectories and files as the lxsdk folder on your machine.

Configure a Service Connection

To use the Live Experience service, configure a service connection in your DCS application to authenticate with the service and retrieve an access token.

Here's how to configure a service connection in Visual Builder Studio:

1. In the Navigator, click the **Services** icon.
2. In the Service pane, click the **Create Service Connection** icon (**+**), and then click **Service Connection**.
3. In the Create Service Connection dialog box, select **Define by Endpoint**.
4. Configure the service connection by selecting these inputs:
   a. In the **Method** Field, select **GET**.
   b. In the **URL** field, enter **https://live.oraclecloud.com/auth/apps/api/access-token/**
   c. In the **Action Hint** field, select **Get One**.
5. Click **Next**.
6. In the Create Service Connection page, Overview tab, enter these values in the respective fields:
   - Service Name: lxAuthApi
   - Title: LX Auth API
7. Click the Server tab and enter these inputs:
   a. Select the **Allow anonymous access to the service connection infrastructure** check box.

   b. In the **Authentication for Logged-In Users** and **Authentication for Anonymous Users** fields, select **Basic** and click the **Edit** icon in the **Username** field:
      i. In the **Username** field, enter the Client ID of the Live Experience application.
      ii. In the **Password** field, enter the Client Secret of the Live Experience application.
      iii. Click **Save**.

> **Note:** This is the same Client ID and Client Secret that you noted in the *Create an Application in the Live Experience* topic.

8. Click the Request tab and enter these field values:
   a. Add these Dynamic Query Parameters:
      i. Name: nonce, Type: Number, Required (Yes)
      ii. Name: state, Type: string, Required (Yes)
   b. Add these Static Query Parameters:
      i. Name: grant_type, Value: client_credentials
9. Click the Test tab and click **Send Request**.
   If configured correctly, a confirmation response appears in the Body tab.
10. Click **Create**.

Your new service connection appears as a LXAuthAPI tab. Close this tab for now.

Add the Live Experience Variable to DCS Application

Here's how to add the Live Experience variable to your DCS application:

1. In the navigator, click **Web Apps** to open the Web Apps pane.
2. In the Web Apps pane, click **dcs** to open the DCS app.
3. Click the Variables tab.
4. Click **+Variable**.
5. In the Create Variable dialog box, make sure that **Variable** is selected.
6. In the **ID** field, enter **lxTenantName**.
7. In the **Type** field, select **String**, and click **Create**.
   The variable you created appears in the Variable list and its properties appear in the Properties pane.
8. In the Properties pane, **Default Value** field, enter your Live Experience tenant name.
9. In the **Persisted** field, select **Session**.

Add the Live Experience Widget Code to the DCS Application

To add the Live Experience widget to all pages in your application, you can add the Live Experience widget code to the shell Root Page.

1. In the Web Apps pane, expand Root Pages, and click **shell**.

2. In the shell page, click the JavaScript tab.
3. Replace `define([], function() {` with this REST dependency code:

```
define([
 'ojs/ojcore',
 'vb/helpers/rest'
], function(
 oj,
 Rest
) {
```

4. Add this code to the `PageModule prototype`:

```
PageModule.prototype.generateString = function(length) {
 const possibles = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
 const possiblesLen = possibles.length;
 let result = "";
 for (let i = 0; i < length; i++) {
 result += possibles.charAt(Math.floor(Math.random() * possiblesLen));
 }
 return result;
};

PageModule.prototype.lxStart = function(appPathVar, lxTenant, user) {

 return new Promise((resolve, reject) => {

 // get auth token from the VB service which queries LX auth server
 const getAuthToken = async (callback) => {

 // init the Service Connection using VB Rest Helper utility
 const endpointId = 'lxAuthApi/getAccessToken';
 const endpoint = Rest.get(endpointId);

 // set Dynamic Query Parameters
 endpoint.parameters({ "nonce": Math.floor(Math.random() * 999999), "state": this.generateString(9) });

 try {
 const response = await endpoint.fetch();
 callback(response.body.access_token, response.body.expires_in);
 }
 catch (error) {
 console.error(error);
 // return false up to the Action Chain
 resolve(false);
 }
 };

 var lxRequire = require.config({
 context: "lx",
 baseUrl: appPathVar + "resources/lxsdk/js",
 waitSeconds: 60,
 paths: {
 jquery: "../lib/jquery",
 text: "oracle.live.api",
 css: "oracle.live.api",
 "oracle.live.api": "oracle.live.api",
 "oracle.live.style": "oracle.live.style",
 "oracle.live.button": "oracle.live.api",
 "oracle.live.messages": "oracle.live.api",
 "oracle.live.sdk": "oracle.live.api"
 },
 shim: {
 jquery: {
 exports: ["jQuery", "$"]
 }
 }
```

```
         });

         lxRequire(["oracle.live.api"], (liveApi) => {

         getAuthToken((myAuthToken, myTokenExpiry) => {

         var email = user && user.email && user.email.length ? user.email : "anonymous@example.com.test";
         var fullName = user && user.fullName && user.fullName.length ? user.fullName : "Anonymous";

         // TODO: LX service properties
         liveApi.controller.service.userID = email;

         // TODO: update LX tenant
         liveApi.controller.service.tenantID = lxTenant;

         liveApi.controller.service.authToken = myAuthToken;

         // the callback should update the authToken property with the refreshed token
         liveApi.controller.service.authRefresh(myTokenExpiry, () => {
         getAuthToken((jwt) => { liveApi.controller.service.authToken = jwt; });
         });

         // set Engagement Scenario
         liveApi.controller.contextAttributes.set("appLocation", "Collaboration");

         // optional context attributes...
         liveApi.controller.contextAttributes.set("email", email);
         liveApi.controller.contextAttributes.set("fullName", fullName);

         liveApi.controller.addComponent();

         // return true up to the Action Chain
         resolve(true);

         }); //getAuthToken

         }); //oracle.live.api

         }); //Promise

        }; //lxStart
```

Set Up an Action Chain to Execute the Code

Set up an action chain that will run the Live Experience widget code to add the widget to the page.

1. In the shell page, click the Actions tab, then click **+ Action Chain**.
2. In the **ID** field, enter **LoadLiveExperience**, and click **Create**.

   The LoadLiveExperience action chain page opens.
3. In the Actions pane, drag **Call Function** and drop it on the **Add** icon.
4. In the Properties pane, **Function Name** field, enter **lxStart**.
5. In the **Input Parameters** field, click **Assign**.
6. In the Assign Input Parameters dialog box, drag **lxTenantName** under Variables to **lxTenant** under Target.
7. Drag **path** under System to **appPathVar** under Target.
8. Select **user parameter** in the Target list.

9. In the $user.parameter pane, select **Expression**.
10. Set the expression to `$application.user`.
11. Click **Save**.
12. In the Properties pane, **Return Type** field, select **Boolean**.
13. Click the Event Listeners tab and click **vbEnter**.
14. In the Select Action Chain dialog box, select **LoadLiveExperience**, and click **Select**.

Preview the DCS Application

Click the **Preview** icon to load the DCS application. When the page loads, you'll be able to see the Live Experience widget on the DCS application landing page. You can now make live calls to an agent.

5 Add Live Experience Features to Web Applications

Prepare Your Website for the Live Experience Widget

Set up your application environment in preparation for adding Live Experience to your website.

Set up your application environment as appropriate for your application deployment, by ensuring the you have satisfied the following requirements:

- A valid Oracle Live Experience tenant account, and access to information from the tenant Admin Console including scenario configurations.
- A copy of the Live Experience JavaScript SDK downloaded to your development machine.
- Speakers and a microphone attached to your computer.

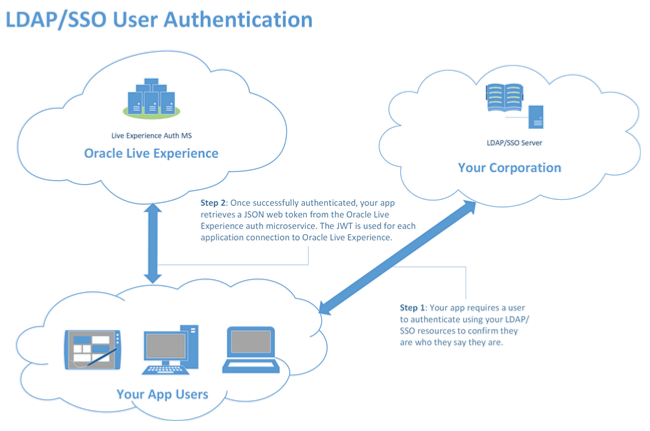
What to do next

Next, *Authenticate with Live Experience for the Web*.

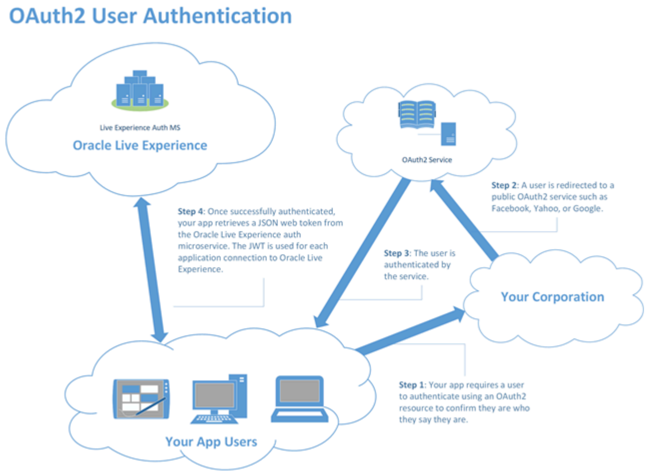
Authenticate with Live Experience for the Web

Set up User Authentication and Live Experience Authentication, which are two necessary steps for adding Live Experience to your website.

User Authentication refers to confirming that a user who's accessing your application is who they say they are. Your greater organization will determine how that authentication is handled. For instance, you may need to use OAuth authentication through a third party provider such as Google, Facebook, or Yahoo. In other cases, you may use your own organization's single sign-on facilities, some sort of LDAP system, or, during development, you may use a simple username/shared password challenge. In any case, you'll need to design an appropriate workflow along with the supporting back end code.
The following graphic shows the flow if you're using your own SSO/LDAP workflow.


LDAP/SSO User Authentication
Live Experience Auth MS
Oracle Live Experience
LDAP/SSO Server
Your Corporation
Step 2: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.
Step 1: Your app requires a user to authenticate using your LDAP/SSO resources to confirm they are who they say they are.
Your App Users

And the following graphic shows the flow if you're using a third party OAuth platform such as Facebook, Google, or


OAuth2 User Authentication
Live Experience Auth MS
Oracle Live Experience
OAuth2 Service
Step 2: A user is redirected to a public OAuth2 service such as Facebook, Yahoo, or Google.
Step 4: Once successfully authenticated, your app retrieves a JSON web token from the Oracle Live Experience auth microservice. The JWT is used for each application connection to Oracle Live Experience.
Step 3: The user is authenticated by the service.
Your Corporation
Step 1: Your app requires a user to authenticate using an OAuth2 resource to confirm they are who they say they are.
Your App Users

Yahoo.

After you authenticate a user (assuming it's required for your application), you then need to handle Live Experience Authentication. Fortunately, Live Experience authentication is much more prescribed and quite a bit simpler than User Identity Authentication. To authenticate with Live Experience, you obtain a JSON Web Token (JWT) from Live Experience which you use when opening any connection or REST interface requests. You use JavaScript XMLHttpRequests to communicate with a simple script that you'll deploy on a web server in your own domain. See *Deploy the Sample JWT Script*. While the supplied script is sufficient for development, you'll want to create something more secure for a production environment using the REST call described in *Retrieve a JWT Access Token Using the Auth REST Call*.

To retrieve a JWT token from the script, you use an XMLHttpRequest, and store the JWT in the browser's sessionStorage so you can recall it when you actually connect to Live Experience.

1. Deploy the script as described in *Deploy the Sample JWT Script*.
2. Use an XMLHttpRequest to retrieve the JWT and store it in the browser's sessionStorage for easy retrieval:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://your-server/cgi-bin/auth.sh", true);
xhr.setRequestHeader("Accept","application/json");
xhr.onload = function () {
 // Extract the access_token field from the response
 var response = JSON.parse(xhr.responseText);
 console.log(response.access_token);
 // Save the JWT in session storage...
 sessionStorage.setItem(jwtKey, response.access_token);
};
xhr.send();
```

Results:

After you obtain and store the JWT access_token, you can use it to authenticate with Live Experience.

What to do next

Next, *Add and Configure the Live Experience Widget for your Website*.

Add and Configure the Live Experience Widget for your Website

Your customers initiate engagements with your associates by selecting the Oracle Live Experience widget.

Add and configure the widget so that it provides the exact experience you intend to deliver to your customers. We created an SDK to make this task as simple and straightforward as possible. Take your existing website or web application and extend it to include the Live Experience framework. Then you add the widget and configure the widget on as many web pages as you need.

1. Download the Live Experience web component SDK and extract it to a folder within your application (for instance, into a folder named **lx**).

   You'll reference the folder name you choose, lx in this example, when you're referencing the widget API.
2. Add a reference to **require.js** (included in the SDK) in the `<head>` element of your HTML file where you reference the folder name you just created, in this case, **lx**:

   ```
   <head>
    <script type="text/javascript"
    data-main="lx/js/api-main"
    src="lx/lib/require.js">
    </script>
   </head>
   ```

   The widget uses the **require.js** library for dependency processing. If your application already uses **require.js**, there's no need to include it again as shown in the sample above.
3. Add a **require.js** `<script>` element within your page's `<body>` element:

   ```
   <script type="text/javascript">
    require(["oracle.live.api"], function(liveApi) {
    });
   </script>
   ```
4. Retrieve the authorization JWT by making a request to an authorization module that you've defined (in this case **/my_server_path/auth.cgi**).

   You get the token and the token expiry in the response.

   ```
   <script type="text/javascript">
    require(["oracle.live.api"], function(liveApi) {
    const getAuthToken = (callback) => {
    fetch(new Request("/my_server_path/auth.cgi"))
    .then(response => { return response.json(); })
    .then(auth => { callback(auth.access_token, auth.expires_in); });
    };
    });
   </script>
   ```

   For information on deploying a simple authorization module, see *Deploy the Sample JWT Script*.
5. Create a `getAuthToken` function that takes auth token and auth token expiry arguments, which you retrieved from the auth module call above:

   ```
   <script type="text/javascript">
    require(["oracle.live.api"], function(liveApi) {
    const getAuthToken = (callback) => {
   ```

```
 fetch(new Request("/my_server_path/auth.cgi"))
 .then(response => { return response.json(); })
 .then(auth => { callback(auth.access_token, auth.expires_in); });
 };
 getAuthToken((myAuthToken, myTokenExpiry) => {
 });
 });
</script>
```

> **Note:** If the API call to acquire a valid JWT fails, and `myAuthToken` and `myTokenExpiry` are undefined, then the browser can be thrown into an indefinite refresh loop until it crashes. Make sure that `myAuthToken` and `myTokenExpiry` can't be undefined.

6. Initialize the following required context attributes:
   - `userID`: the URI for the user initiating the call
   - `tenantID`: your assigned tenant ID
   - `authToken`: the authentication JWT retrieved from your auth module

     ```
     <script type="text/javascript">
      require(["oracle.live.api"], function(liveApi) {
      const getAuthToken = (callback) => {
      fetch(new Request("/my_server_path/auth.cgi"))
      .then(response => { return response.json(); })
      .then(auth => { callback(auth.access_token, auth.expires_in); });
      };
      getAuthToken((myAuthToken, myTokenExpiry) => {
      liveApi.controller.service.userID = "bob@example.com";
      liveApi.controller.service.tenantID = "MyTenant";
      liveApi.controller.service.authToken = myAuthToken;
      });
      });
     </script>
     ```

     > **Note:** When you test the integration of the web widget into your application and make your first call, ensure the initialized `userID` isn't the same as the email address used to log into the Associate Desktop.

7. Initialize the `authRefresh` helper callback to automatically refresh the authentication JWT when required:

```
<script type="text/javascript">
 require(["oracle.live.api"], function(liveApi) {
 const getAuthToken = (callback) => {
 fetch(new Request("/my_server_path/auth.cgi"))
 .then(response => { return response.json(); })
 .then(auth => { callback(auth.access_token, auth.expires_in); });
 };
 getAuthToken((myAuthToken, myTokenExpiry) => {
 liveApi.controller.service.userID = "bob@example.com";
 liveApi.controller.service.tenantID = "MyTenant";
 liveApi.controller.service.authToken = myAuthToken;
 liveApi.controller.service.authRefresh(myTokenExpiry, () => {
 getAuthToken((jwt) => {
 liveApi.controller.service.authToken = jwt; });
 });
 });
 });
</script>
```

What to do next

Next, *Configure Context Information for Your Website*.

Configure Context Information for Your Website

The Live Experience widget lets you define a variety of context information that's relayed to the Associate Desktop and displayed to an associate when in a call.

The context information can include the caller's name and email address, phone number, and geographical location, which is then displayed for each caller in the caller information pane.
To configure a particular piece of context information, you use the `contextAttributes` set method, and specify a key indicating the name of the context attribute to set and a value indicating the value of that context attribute.

The following table lists the default context attribute key and value pairs you can configure.

Default Context Attribute Key or Value Pairs

Key	Value Example	Notes
appLocation	"Support Tab"	The appLocation value should match the name of a value configured for Application Location in the Admin Console. For information on configuring an Application Location on an engagement scenario, see Manage Engagement Scenarios.
email	"bob@example.com"	Retrieved from your application's UI.
fullName	"Bob Dobbs"	Retrieved from your application's UI.
location	"Scottsdale, AZ"	Retrieved via the device's location API.
phone	"415-555-1212"	Retrieved from your application's UI.

You can define additional context attributes using the Live Experience Admin Console. See *Configure Oracle Live Experience to Send and Receive SMS Messages*.

To configure context information, add statements as required.

```
<script type="text/javascript">
 require(["oracle.live.api"], function(liveApi) {
 const getAuthToken = (callback) => {
 fetch(new Request("/my_server_path/auth.cgi"))
 .then(response => { return response.json(); })
 .then(auth => { callback(auth.access_token, auth.expires_in); });
 };
 getAuthToken((myAuthToken, myTokenExpiry) => {
 liveApi.controller.service.userID = "bob@example.com";
 liveApi.controller.service.tenantID = "MyTenant";
 liveApi.controller.service.authToken = myAuthToken;
 liveApi.controller.service.authRefresh(myTokenExpiry, () => {
 app.getAuthToken((jwt) => {
 liveApi.controller.service.authToken = jwt; });
 });
 // Optional context attributes...
 liveApi.controller.contextAttributes.set("email", "customer@example.com");
 liveApi.controller.contextAttributes.set("fullName", "Bob Dobbs");
 liveApi.controller.contextAttributes.set("phone", "415-555-1212");
 liveApi.controller.contextAttributes.set("appLocation", "Scenario Name");
 });
 });
</script>
```

The example above sets context information for a user's name, email address, and phone number, which are displayed to associates when calls come into the Associate Desktop. The example also sets the location on the web page from where the call originates.

For more information about context configuration, see the *Live Experience JavaScript API Reference*.

What to do next

Next, *Add the Live Experience Widget to your Web Page*.

Add the Live Experience Widget to your Web Page

Display the widget in your HTML page.

To display the widget in your HTML page, call the `controller.addComponent` method after you've initialized context variables:

```
<script type="text/javascript">
 require(["oracle.live.api"], function(liveApi) {
 const getAuthToken = (callback) => {
 fetch(new Request("/my_server_path/auth.cgi"))
 .then(response => { return response.json(); })
 .then(auth => { callback(auth.access_token, auth.expires_in); });
 };
 getAuthToken((myAuthToken, myTokenExpiry) => {
 liveApi.controller.service.userID = "bob@example.com";
 liveApi.controller.service.tenantID = "MyTenant";
 liveApi.controller.service.authToken = myAuthToken;
 liveApi.controller.service.authRefresh(myTokenExpiry, () => {
 app.getAuthToken((jwt) => {
 liveApi.controller.service.authToken = jwt; });
 });
 liveApi.controller.contextAttributes.set("appLocation",
 "Scenario Name");
 liveApi.controller.addComponent();
 });
 });
</script>
```

What to do next

Next, see *Additional Configuration Options for your Web Pages*.

Additional Configuration Options for your Web Pages

In addition to context attributes, you can control other widget behaviors listed in the following table.

Additional configuration options for your web pages are listed in the following table.

Additional Configuration Options for your Web Pages

Method	Description
liveApi.controller.settings.startVideoInFullS	Choose whether the video engagement starts in full-screen mode (true), or as an overlay (false). Default value: false
liveApi.controller.service.startCallDirectly	Choose whether to start an engagement automatically when the customer opens the page containing the widget. When set to true, the widget is hidden from the customer. Default value: false
liveApi.controller.settings.hidePauseButton	Choose whether to hide the pause button from the customer on the mobile widget, preventing them from pausing the call. Default value: false
liveApi.controller.settings.hideEndCallButto	Choose whether to hide the end-call button from the customer in the mobile widget, preventing them from ending the call themselves. Default value: false
liveApi.controller.settings.hideQueueLength	Choose whether to hide the queue length indicator from the customer as they wait to be connected to an associate. Default value: false
liveApi.controller.settings.attemptFaceMatc	In the context of verifying the identity of your customers, use this setting to enable the facial recognition capability provided by Jumio. See Implementing Identity Verification in a Web or Mobile App for more info. Default value: false
liveApi.controller.service.skipRecordingPerr	Disable the recording permission request warning presented to customers before they can join a recorded call. Default value: false

For more information about context configuration, see the *Live Experience JavaScript API Reference*.

Next, see *Add Origin Domains*.

Add Origin Domains

In Oracle Live Experience, configure origin domains for each website that includes the Live Experience widget.

Adding origin domains is one of the steps required to get Live Experience up and running in your web sites and applications.
In Live Experience, one or more origin domains are required for web applications (but not for native applications). Use origin domains to specify the fully-qualified domain name of the host for the website that will host the Live Experience widget. It must match what the browser presents as the origin of a request, for example, `https://www.yourcompany.com`.

An application can have several origin domains. You can use the asterisk (*) as a wildcard character when adding origin domains so that a single entry can provide access to multiple websites. For example, `https://www.google.*` provides access to any websites that begin with `https://www.google`.

1. From the Admin Console navigation menu, select **Applications**.
2. Select your application.
3. Select the **Origin Domains** tab.
4. Add origin domains you want.

Retrieve a JWT Access Token Using the Auth REST Call

Implement your own JSON Web Token (JWT) request system.

The REST request must be sent from the server side and not the client side.

1. From the navigation menu, select **Applications**.
2. On the Applications page, select your application and then select the **Details** tab.
3. Make note of the Client ID and retrieve the Client Secret from your tenant administrator.

   > **Note:** Click Display Secret and Generate only if you haven't been provided a secret by your tenant administrator. Generating a new secret will cause any authorization configuration you have in place to be invalidated.

4. Send a REST GET request using the clientID and clientSecret from the Admin Console to one of the following Live Experience access token REST endpoints:
   - `GET https://live.oraclecloud.com/auth/apps/api/access-token`

     or, for EMEA customers
   - `GET https://emea.live.oraclecloud.com/auth/apps/api/acess-token`

     Include the following access token request header:

     ```
     Authorization: Basic encoded{clientId, clientSecret}
     ```

     Where clientId and clientSecret are the credentials you've retrieved from the Admin Console.

     The actual REST call itself will look something like this, where:
     - `grant_type` query parameter indicates the type of access token grant requested. This must be `client_credentials`.
     - `&nonce` can equal any random number from 1 to 1,000,000.
     - `&state` is returned to the client to help mitigate CSRF attacks. The value can be any number.

     ```
     https://live.oraclecloud.com/auth/apps/api/access-token?
     grant_type=client_credentials
     &nonce=360468
     &state=0
     ```

```
&scope=optional
```

> **Note:** The request is shown here with carriage returns added to promote its readability. For more information on access token requests, see *https://datatracker.ietf.org/doc/html/rfc6749#section-4.1.3*.

In response to this REST GET request, Live Experience will send you the `access_token` in a JSON-formatted response body.

```
{
 "access_token": "eyJhbGciOiJSUzI1NiJ9.eyJhd.....",
 "expires_in": "3600",
 "id_token": "eyJhbGciOiJSUzI1NiJ9.eyJhdWQiOiJQcmVtaWVyIiwiaXNzIjoiZGVuM....",
 "state": "0",
 "token_type": "Bearer"
}
```

The following table describes the key and value pairs returned in the response body.

Key and Value Pair Description

Key	Example Value	Description
access_token	"abc123zyx987..."	The access token required to authenticate with Oracle Live Experience. This is also referred to as a JWT.
expires_in	"1200"	The expiry time in seconds. The default is 1200 seconds (20 minutes).
id_token	"zza3443kslle..."	An ID token. This is not used.
state	"0"	The request state. This will always be 0 unless an error occurs.
token_type	"Bearer"	The type of access token. This will always be Bearer.

5. You can then retrieve the `access_token` value within your application using the examples described in the following:
    - JavaScript: *Authenticate with Live Experience for the Web*
    - Swift (iOS): *Authenticate with Live Experience for iOS*
    - Java (Android): *Authenticate with Live Experience for Android*

The Live Experience REST API Overview

Learn how to establish a secure API connection with Oracle Live Experience, and discover all the programmatic tasks you can complete using a REST API client.

Obtaining a secure API connection is required before you can successfully make any Live Experience API calls. You should already be familiar with the concept of using REST APIs.

You need a REST API client. You can use any client you like. If you choose to access Live Experience REST operations in code (such as Java, Python, PHP, etc.), the translation should be straightforward if you are familiar with process.

The *Oracle Live Experience REST API Reference* describes the details of each API operation.

The Live Experience REST API allows you to perform the following actions:

- Create tenant applications (using the application endpoint)
- Audit events (using the audit endpoint)
- Obtain an authentication token, necessary before doing any other REST API calls (using the auth endpoint)
- Retrieve or delete engagements (using the engagements endpoint)
- Retrieve, modify, create, and delete engagement scenarios (using the engagement-scenarios endpoint)
- Perform certain integration steps for adding Live Experience to Service Cloud (using the integration endpoint)
- Create a meeting and retrieve a meeting (using the meetings endpoint)
- Retrieve metrics (using the metrics endpoint)
- Perform certain integration steps for configuring SAML authentication (using the saml endpoint)
- Retrieve, modify, create, and delete users (using the users endpoint)

As a Live Experience distribution partner, you can use the REST API to handle the following additional tasks:

- Create, modify, and delete your tenants (using the dp endpoint)

Establish a Secure REST API Connection

Before you can work with any of the Live Experience REST operations, you need to establish a secure connection with the Live Experience REST API.

Retrieve a JSON Web Token (JWT) using the Live Experience auth endpoint. Use the JWT to authorize your subsequent REST operations.
A JWT automatically expires after 20 minutes, after which you need to obtain a new one. As long as you have a valid JWT, you can send REST operations without needing to re-authenticate.

Oracle recommends that you save this request to obtain a JWT so you can easily access it as required.

1. From the Live Experience Admin Console, get the Client ID for your application.
2. From your tenant administrator, get the Client Secret for your application.
3. Concatenate the Client ID and the Client Secret, separated by a colon. For example, `1u4fejhn9pi1gnikha4e:64e740ae-77e7-41a3-97a7-712ba7a9b9f0`.
4. Encode the resulting string as base64 (using UTF-8 as the character encoding).

   You can use any encoding utility you like (for example, `https://www.base64decode.org`). Here's an example of the text string: `MXU0ZmVqaG45cGkxZ25pa2hhNGU6NjRlNzQwYWUtNzdlNy00MWEzLTk3YTctNzEyYmE3YTliOWYw`.
5. In your REST client, select a GET operation and specify one of these auth endpoints, replacing `Tenant_Name` with your own tenant name:
   - Non EMEA users: `https://api.live.oraclecloud.com/v1/auth/Tenant_Name`

- EMEA users: `https://api.emea.live.oraclecloud.com/v1/auth/Tenant_Name`

6. Add these values to the keys of the GET request:
   - **grant_type**: `password`
   - **domain_name**: Insert your Live Experience domain URL, either live.oraclecloud.com or emea.live.oraclecloud.com.
   - **credentials**: Enter your base64 encode string you created earlier.
7. Add these Headers to your request:
   - Accept: application/json
   - Origin: Insert a Live Experience domain URL. Don't include the protocol prefix.
8. Configure authorization by setting the Type to **Basic Auth**, then provide your IDCS credentials.
9. Send the request to generate result. The operation returns a JSON value that looks like this:

```
{
 "access_token": "eyJhbGciOiJSUOpDBdG9... ...MmlTDEb4e0TQQK3yIpJEkJrRieA",
 "token_type": "Bearer",
 "expires_in": "1200"
}
```

   The key and values are:
   - `access_token`: The JWT (much longer than the example).
   - `token_type`: This is always bearer and can be ignored.
   - `expires_in`: The amount of time until the JWT token expires in seconds.

Results:

You'll use the value for `access_token` in all your other requests. Configure authentication in the REST client by setting the Type to **Bearer** and enter the `access_token` value for Token.
Your JWT expires in 1200 seconds, as indicated by the `expires_in` key, after which you need to generate a new one. Oracle recommends that you save the request so you can easily access it as required.

Establish a Secure REST API Connection for Oracle Identity Cloud Service (IDCS) Authentication

You need to establish a secure connection with the Live Experience REST API before you can work with any of the Live Experience REST operations. Retrieve a JSON Web Token (JWT) using the Live Experience auth endpoint and then use the JWT to authorize your subsequent REST operations.

A JWT automatically expires after 20 minutes, after which you need to get a new one. As long as you have a valid JWT, you can send REST operations without needing to re-authenticate. Oracle recommends that you save this request to get a JWT so you can easily access it as required.

Note: You need to enable IDCS authentication to establish a secure REST API connection. For details, see *Enable Oracle Identity Cloud Service (IDCS) Authentication*.

Here's how to establish a secure REST API connection:

1. From your tenant administrator, get these details for your IDCS Confidential Application:
   - IDCS Stripe Base URL
   - Client ID and Client Secret
2. Concatenate the Client ID and the Client Secret, separated by a colon. For example:

   ```
   1u4fejhn9pi1gnikha4e:64e740ae-77e7-41a3-97a7-712ba7a9b9f0
   ```
3. Encode the resulting string as base64 (using UTF-8 as the character encoding). You can use any encoding utility you like (for example, *https://www.base64decode.org*). Here's an example of the text string:

   ```
   MXU0ZmVqaG45cGkxZ25pa2hhNGU6NjRlNzQwYWUtNzdlNy00MWEzLTk3YTctNzEyYmE3YTliOWYw.
   ```
4. In your REST client, select a POST operation and specify this auth endpoint, replacing <IDCS stripe base URL> with your IDCS base URL:

   ```
   <IDCS stripe base URL>/oauth2/v1/token
   ```
5. Add these values to the HTTP parameters of the POST request:
   - **grant_type:** `client_credentials`
   - **scope:** `api/`
6. Configure authorization by setting the Type to **Basic Auth**, then provide your IDCS credentials.

   Note: The IDCS credentials are the Client ID and Client Secret that you got from the tenant administrator.
7. Send the request to generate result. The operation results a HTTP value that looks like this:

   ```
   Authorization: Basic <Base64 encoded ClientID:Client Secret>
   ```
8. Send the request. The operation returns a JSON value with access_token value set to the IDCS signed JWT.

   Here's a request sample in curl format:

   ```
   curl -i -H 'Authorization: Basic
   OTljOGNjZGJhYjRjNDQ0YTlmNDUxZDZmYjc0Y2I2Y2I6YTEzM2U3ZWEtZWZjNS00YjY3LWJmMGUtOTBhNmQyZDNiZjM2'
   -H 'Content-Type: application/x-www-form-urlencoded;charset=UTF-8'
   --request POST https://idcs-a05be02e3e9042cbbba8f3581d17bf9a.identity.c9dev2.oc9qadev.com/oauth2/v1/
   token
   -d 'grant_type=client_credentials&scope=api/'
   ```
9. In your REST client, select a GET operation and specify one of these auth endpoints, replacing `Tenant_Name` with your own tenant name:
   - Non EMEA users: `https://api.live.oraclecloud.com/v1/auth/Tenant_Name`
   - EMEA users: `https://api.emea.live.oraclecloud.com/v1/auth/Tenant_Name`
10. Add these values to the keys of the GET request:
    - **grant_type:** `authorization_code`
    - **code:** `idcs_jwt`
11. Add these Headers to your request:
    - **Accept:** `application/json`
    - **Origin:** Insert a Live Experience domain URL. Don't include the protocol prefix.
12. To configure authorization, set the Type to **Bearer**, then provide the IDCS JWT.
13. Send the request to generate result. The operation results a HTTP value that looks like this:

    ```
    Authorization: Bearer <IDCS JWT>
    ```
14. Send the request.

Here's a request sample in curl format:

```
curl -k -X GET 'https://api.live.oraclecloud.com/v1/auth/TestTenantHere?
grant_type=authorization_code&code=idcs_jwt'
--header 'Accept: */*' --header 'Authorization: Bearer eyJ...SVbIgQ'
```

The operation returns a JSON value that looks like this:

```
Copy
{
 "access_token": "eyJhbGciOiJSUOpDBdG9... ...MmlTDEb4e0TQQK3yIpJEkJrRieA",
 "token_type": "Bearer",
 "expires_in": "1200"
}
```

The key and values are:

- `access_token`: The JWT (much longer than the example).
- `token_type`: This is always bearer and can be ignored.
- `expires_in`: The amount of time until the JWT token expires in seconds.

Results:

You'll use the value for `access_token` in all your other requests. Configure authentication in the REST client by setting the Type to **Bearer** and enter the `access_token` value for Token.

Your JWT expires in 1200 seconds, as indicated by the `expires_in` key, after which you need to generate a new one. Oracle recommends that you save the request so you can easily access it as required.

Deploy the Sample JWT Script

We provide a simple Bourne shell script that you can deploy on an available web server in your domain.

The script returns a JSON Web Token (JWT) when queried from your client application by reading your Client ID and Client Secret from a plain text file and then using the command line utility curl to retrieve a JWT.
While the supplied script is sufficient for development purposes, you'll want to create something more secure for a production environment using the REST operation described in *Retrieve a JWT Access Token Using the Auth REST Call*.

1. From the Admin Console navigation menu, select **Applications**.
2. On the Applications page, select your application.
3. Select the **Details** tab.
4. Make note of the Client ID and retrieve the Client Secret from your tenant administrator.

   Note: Click Display Secret and Generate only if you haven't been provided a secret by your tenant administrator. Generating a new secret will cause any authorization configuration you have in place to be invalidated.

5. Copy the following source into a file named **auth.sh** and save the file:

```
#!/bin/bash
# Copyright (c) 2017 Oracle. All rights reserved.
# This material is the confidential property of Oracle Corporation or its
# licensors and may be used, reproduced, stored or transmitted only in
# accordance with a valid Oracle license or sublicense agreement.
# Live Experience Sample Auth Module
#
```

```
# This shell script allows a Javascript application to retrieve a JWT token
# from Live Experience when provided with a valid client ID and secret.
# The client-credentials should be written to a text file in the format:
# <ID>:<SECRET>
# e.g. using the command: echo "ID:SECRET" >secret.txt
# This line specifies the path to the client-credentials file,
# if you move the file update this line to the new location
SECRET_PATH="./secret.txt"
# make sure curl command can be found
PATH=/bin:/usr/bin:/usr/sbin:$PATH
# build up auth server URL (allow replacing the server on the command line for testing)
if [ $# -eq 1 ]; then
 AUTH_SERVER="$1"
else
 AUTH_SERVER="https://live.oraclecloud.com"
 ## EMEA customers use:
 ## AUTH_SERVER="https://emea.live.oraclecloud.com"
fi
AUTH_PATH="/auth/apps/api/access-token"
AUTH_ARGS="?grant_type=client_credentials&state=0&scope=optional&nonce=${RANDOM}"
AUTH_SECRET=`cat ${SECRET_PATH}`
# add curl arguments to temporary file to avoid including on curl command file
tmpdir=$(mktemp -d "${TMPDIR:-/tmp/}.XXXXXXXXXXXX")
cat >${tmpdir}/args.txt <<EOF
--insecure
--silent
--show-error
url = "${AUTH_SERVER}${AUTH_PATH}${AUTH_ARGS}"
user = "${AUTH_SECRET}"
EOF
# set the content type
echo "Content-type: application/json"
echo ""
# retrieve the JWT token
curl --disable --config ${tmpdir}/args.txt
echo ""
# remove the temporary directory
rm -rf ${tmpdir}
exit 0
```

6. Copy **auth.sh** and to the **cgi-bin** directory on a web server you've deployed.
   The script requires access to the curl utility.
7. Optionally, rename the file **auth.cgi**.
8. Make sure the file is flagged executable and owned by a secure user such as www:

```
chmod +x auth.sh
chown www:www auth.sh
```

9. Create a plain text file named **secret.txt** and add a single line of the format `ID:SECRET`, where `ID` is the Client ID and `SECRET` is the Client Secret: `echo "1f611f7kjloqj3j5i98s:eyJhbGciOiJSUzI1NiJ9.eyJhd..." > secret.txt`.
   If you change the name or path of secret.txt, you'll need to change the location in auth.sh on the line that starts with `SECRET=`.
   An example response from the script is a JSON formatted string that will look something like:
   `{"access_token":"abc123zyx987","expires_in":"1200","id_token":"abc123zyx987","state":"0","token_type":"Bearer"}`.
   The following table describes the key and value pairs returned by the script.

   Key and Value Pairs Returned by the Script

Key	Example Value	Description
access_token	"abc123zyx987..."	Access token required to authenticate with Live Experience. Also interchangeably referred to as a JWT.
expires_in	"1200"	Expiry time in seconds. Default is 1200 seconds (20 minutes).
id_token	"zza3443kslle..."	An ID token. Not used.
state	"0"	Request state. Will always be 0 unless an error occurs.
token_type	"Bearer"	Type of access token. Will always be Bearer.

10. After the script is deployed, see the following examples to retrieve the JWT:
    - JavaScript: *Authenticate with Live Experience for the Web*
    - Swift (iOS): *Authenticate with Live Experience for iOS*
    - Java (Android):*Authenticate with Live Experience for Android*

    For information about the API to retrieve a JWT for your own JWT request system implementation, see the *Oracle Live Experience REST API Reference*.

Handling Cross-Origin Resource Sharing

If you're hosting a Live Experience application in a third party container system such as Google App Engine, you may not be able to directly host the sample JWT Bourne shell script.

To retrieve a JWT token from the sample script, you'll need to deploy a mechanism to handle Cross-origin Resource Sharing (CORS). The CORS mechanism will enable you to pull a JWT from the JWT script within your container service.

The example below shows a simple Python script that you can deploy in your container service to handle CORS issues. You'll need to note the following:

- If you've renamed the sample JWT script to `auth.cgi`, update the appropriate lines referencing `auth.sh`.
- Likewise on the final line, set the IP address of the hosting system and the port on which the python script is listening (default: 8000).

- You can then run the sample JWT script from the **/cgi-bin** of the web service.

Note: Use the CORS script for development purpose only.

```
import bottle
from bottle import response
import subprocess
class EnableCors(object):
 name = 'enable_cors'
 api = 2
 def apply(self, fn, context):
 def _enable_cors(*args, **kwargs):
 # set CORS headers
 response.headers['Access-Control-Allow-Origin'] = '*'
 response.headers['Access-Control-Allow-Methods'] = 'GET, POST, PUT, OPTIONS'
 response.headers['Access-Control-Allow-Headers'] = 'Origin, Accept,
 Content-Type,
 X-Requested-With,
 X-CSRF-Token'
 if bottle.request.method != 'OPTIONS':
 # actual request; reply with the actual response
 return fn(*args, **kwargs)
 return _enable_cors
app = bottle.app()
@app.route('/auth.sh', method=['OPTIONS', 'GET'])
def auth():
 response.set_header('Content-Type', 'application/json')
 completed = subprocess.run(["/bin/bash","./auth.sh"], stdout=subprocess.PIPE)
 print('auth.sh:', completed.returncode)
 print('Have {} bytes in stdout:\n{}'.format(len(completed.stdout), completed.stdout.decode('utf-8')))
 return completed.stdout.decode('utf-8')
app.install(EnableCors())
app.run(host="0.0.0.0", port=8000)
```

Developer Deep Dive: Live Experience Authentication

The first step in integrating Oracle Live Experience into your app is gaining access to the service.

Live Experience SDK uses a JSON Web Token (JWT) as the access key. It's up to the developer to request a JWT and keep it up to date.

A REST call is used to retrieve a JWT access token. See the *Oracle Live Experience REST API Reference* for more information.

While this REST call can be made from your app directly, it requires your tenant's authentication (client_id/client_secret) credentials. If these credentials change, you'll need to update the app on all devices it has been deployed to. A better approach is to have a web application make the REST call on the device's behalf. This gives you a central point of control and allows integration with an existing enterprise authentication or SSO platform.

For an example implementation of centralized authentication in iOS, see *Authenticate with Live Experience for iOS*.

See the following for more details about the JWT access token, waiting for authentication, how to handle JWT expiration, and steps to troubleshoot authentication problems in your application:

- *The JWT Access Token*

- *Waiting for Authentication*
- *Handling Access Token Expiration*
- *Authentication Troubleshooting*

The JWT Access Token

The REST call returns the JWT embedded inside a JSON object.

The tokens are shortened to save space:

```
{
 "access_token": "eyJhbG...",
 "expires_in": "1200",
 "id_token": "eyJhbG...",
 "state": "0",
 "token_type": "Bearer"
}
```

We are interested in the first two key/value pairs, `access_token` and `expires_in`. The JWT itself is stored in `access_token` and `expires_in` tells us how many seconds until this JWT is no longer valid.

We will need to parse the JSON to get these two items. For example, let's say the above JSON was in the variable `restData`.

In JavaScript you'd parse the JSON like this:

```
var obj = JSON.parse(restData);
var token = obj.access_token;
var expString = obj.expires_in;
var expires = Number(expString);
```

In Swift on iOS you would parse the JSON like this:

```
let json = JSONSerialization.jsonObject(with: restData, options: []) as? [String: Any]
let token = json?["access_token"] as? String
let expString = json?["expires_in"] as? String
let expires = Double(expString)
```

Note: `expires_in` is provided as a string so we need to convert it to a number.

If you look at the JWT inside `access_token` it may appear to be random text, but it actually has a defined structure. The data is Base64Url encoded and includes an RSASHA256 signature. The payload of a Live Experience JWT looks like this:

```
{
 "aud": "LiveExperienceDemo1",
 "iss": "auth162442238d0m",
 "exp": 1523983737,
 "server_version": "18.3.1",
 "iat": 1523982537,
 "nonce": "26741",
 "tenant_role": [
 "guest"
 ],
 "username": "1fntfqg6k45nh4o8t3rt"
}
```

You don't need to do anything with what's inside the JWT. We just pass it as is to the Live Experience SDK. For example, in iOS: `Controller.shared.service.authToken = token`.

Waiting for Authentication

Since access tokens are requested over a network connection the request takes some time to complete.

If you attempt to add the Live Experience widget before you have a valid token, the widget will fail to appear in your app. Therefore you need to keep track of the authentication process and wait for it to complete before placing your widget. Network requests happen in a background thread so your app will not wait and continue to run during this time. You will need some kind of notification, callback, or completion handler to let you know when the token has been fetched.

Here is an example of using a completion handler in Swift on iOS to wait for authentication before placing the widget:

```
fetchToken() {
 Controller.shared.addComponent(viewController: view)
}
```

Handling Access Token Expiration

The JWT access token is only valid for a finite period of time.

Using an expired JWT will cause operations to fail. As you saw above, we are told how long a token is valid through `expires_in`. This value is normally 1200 seconds or 20 minutes. Expired tokens are not refreshed. We just fetch a new token and pass it to the Live Experience SDK.

You need to track the JWT expiration yourself. One way is to set a timer in your app to fetch a new token after 1200 seconds. Here is a timer in Swift:

```
Timer.scheduledTimer(timeInterval: expires, target: self, selector: #selector(fetchToken), userInfo: nil,
 repeats: false)
```

Another is to compute an expiration time by adding 1200 seconds to the current system time when the token was retrieved. So, if the time when you fetch the token was 10:00 the expiration time would be 10:20. An example of this in Swift looks like: `let expireTime = Date().addingTimeInterval(expires)`.

You then compare the current time to the expiration time to see if the token has expired.

Twenty minutes is a long time and you may launch your app several times during that period. Storing your token in persistent storage allows it to survive after the app exits. Here is an example of storing a token and its expiration time in persistent storage with Swift:

```
UserDefaults.standard.set(token, forKey: "LXtoken")
UserDefaults.standard.set(expireTime, forKey: "tokenExpireTime")
```

Authentication Troubleshooting

As you develop your Live Experience app you will run into bugs related to authentication and token handling.

The following information will be useful as you debug your code.

- REST Authentication Failure

  If your credentials (`client_id` and `client_secret`) are invalid, the REST API will issue a 401 HTTP response code with the following payload: `{"description":"Incorrect client credential","error":"invalid_client"}`

  Verify your credentials match those in the Live Experience tenant Admin Console.
- Authentication Succeeds but Call Times Out

  If your application-side client uses the same logon credentials as for Associate Desktop, authentication will succeed and you can make calls from your app to Live Experience, but calls will remain stuck in the "Connecting" state and the call will time out.

  In brief, calls will fail to connect because you can't call yourself.

  To resolve this issue, in the SDK, ensure that `CommunicationFragment.service.serUserID("emailAddress")` is set to a different email address than the one you use to authenticate with Live Experience.
- Network Issues in the SDK

  Once you have passed a token to the SDK it is used for the SDK's own network requests. If these network requests fail the Live Experience SDK provides no notification to your application. The only way to see if a failure has occurred is to monitor the console log of your application for HTTP response errors. Here are some common failure scenarios and how they appear in the console.

  > **Note:** These examples use the iOS console. The JavaScript and Android consoles produce similar error messages.

  - Scenario Does Not Exist

    Live Experience allows you to define engagement scenarios for your application. You select the engagement scenario you want to use by setting the *`appLocation`* context attribute in the SDK. If you choose a scenario that has not been defined in the Admin Console, the widget will not appear. When this happens, look for HTTP status code 404 in the console:

    ```
    OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):46 |
    Error when fetching scenario. HTTP Response Status code: 404
    OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):51 |
    Http response status is not correct
    Optional(<NSHTTPURLResponse: 0x6040000306a0> { URL:
    https://live.oraclecloud.com/tenant/api/tenants/Demo1/match-meta-scenario/?version=18.3.2 }
    { Status Code: 404, Headers {
    OracleLive | ERROR | Controller.swift:updateScenario(completionHandler:):697
    Could not get scenario Collaboratio: Error Domain=OracleLive Code=101 "(null)"
    OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):46 |
    Error when fetching scenario. HTTP Response Status code: 404
    OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):51 |
    Http response status is not correct
    Optional(<NSHTTPURLResponse: 0x604000033420> { URL:
    https://live.oraclecloud.com/tenant/api/tenants/Demo1/match-meta-scenario/?version=18.3.2 }
    { Status Code: 404, Headers {
    OracleLive | ERROR | Controller.swift:updateScenario(completionHandler:):697 |
    Could not get scenario Collaboratio: Error Domain=OracleLive Code=101 "(null)"
    ```
  - Wrong Tenant Specified

    You specify your Live Experience tenant by setting the `Controller.shared.service.tenantID` attribute. If the access token you retrieved is for a different tenant, HTTP status 500 will appear:

    ```
    OracleLive | ERROR | AvailabilityClient.swift:getAvailability():44 |
    ```

```
Error when fetching agent availability. Status code: 500
OracleLive | ERROR | AvailabilityClient.swift:getAvailability():49 |
Http response status is not correct
Optional(<NSHTTPURLResponse: 0x60000003d540>
{ URL:
https://live.oraclecloud.com/tenant/api/tenants/Demo1/application/?version=18.3.2
}
{ Status Code: 500, Headers {
```

- Invalid Token

  If you pass a token that has already expired to the Live Experience SDK, you will see HTTP status 401 in the console:

  ```
  OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):46 |
  Error when fetching scenario. HTTP Response Status code: 401
  OracleLive | ERROR | ScenarioClient.swift:setupScenario(completionHandler:):51 |
  Http response status is not correct Optional(<NSHTTPURLResponse: 0x600000430300> {
  URL:
  https://live.oraclecloud.com/tenant/api/tenants/Demo1/match-meta-scenario/?version=18.3.2 }
  { Status Code: 401, Headers {
  ```

- Call Attempt After a Token Has Expired

  If your token expires after the widget is already displayed in your app, nothing will happen. If you click on the call button, however, you will see the following console error message:

  ```
  OracleLive | ERROR | CallQueueClient.swift:pushContext():184 | Response 401, fail
  ```

  You will also see a system message notification in your application UI.