sdl

Android/Java Guides

Document current as of 05/22/2019 10:40 AM.

Overview

Sdl Java Suite is a suite of libraries that are written in Java to help developers make sdl enabled apps that can communicate with sdl head units. The suite can be integrated with Android, JavaSE (embedded), and JavaEE (cloud) apps. The way developers can initially integrate Sdl libraries into their apps is slightly different from one platform to another. However, most other logic to communicate with the head unit is the same among all platforms.

Getting Started

The guides will walk you through the process of integrating Sdl into your app based on which platform your app runs on:

ANDROID

To integrate Sdl into your Android app, please follow the steps in Getting Started With Android section.

JAVASE (EMBEDDED)

To integrate Sdl into your embedded Java app, please follow the steps in Getting Started With JavaSE section.

JAVAEE (CLOUD)

To integrate Sdl into your cloud Java app, please follow the steps in Getting Started With JavaEE section.

Using Other Sdl Features

Features that are not under `Getting Started` sections are applicable to all platforms unless it is noted otherwise. For example, the Android code that developers can use to display information on the head unit's screen in Displaying Information section, can be used also in embedded and cloud apps without any change needed.

Installation

Introduction

Each SDL Android library release is published to JCenter. By adding a few lines in their app's gradle script, developers can compile with the latest SDL Android release.

To gain access to the JCenter repository, make sure your app's `build.gradle` file includes the following:

```
repositories {
    jcenter()
}
```

Gradle Build

To compile with the a release of SDL Android, include the following line in your app's `build.gradle` file,

```
dependencies {
    implementation 'com.smartdevicelink:sdl_android:{version}'
}
```

and replace `{version}` with the desired release version in format of `x.x.x`. The list of releases can be found here.

Examples

To compile release 4.8.1, use the following line:

```
dependencies {
    implementation 'com.smartdevicelink:sdl_android:4.8.1'
}
```

To compile the latest minor release of major version 4, use:

```
dependencies {
    implementation 'com.smartdevicelink:sdl_android:4.+'
}
```

Integration Basics

Getting Started on Android

In this guide, we exclusively use Android Studio. We are going to set-up a bare-bones application so you get started using SDL.

> NOTE
>
> The SDL Mobile library supports Android 2.2.x (API Level 8) or higher.

Required System Permissions

In the AndroidManifest for our sample project we need to ensure we have the following system permissions:

- Internet - Used by the mobile library to communicate with a SDL Server

- Bluetooth - Primary transport for SDL communication between the device and the vehicle's head-unit
- Access Network State - Required to check if WiFi is enabled on the device

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.BLUETOOTH"/
>
    <uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />

</manifest>
```

NOTE

If the app is targeting Android P (API Level 28) or higher, the Android Manifest file should also have the following permission to allow the app to start a foreground service:

```
<uses-permission android:name=
"android.permission.FOREGROUND_SERVICE" />
```

SmartDeviceLink Service

A SmartDeviceLink Android Service should be created to manage the lifecycle of the SDL session. The `SdlService` should build and start an instance of the `SdlManager` which will automatically connect with a headunit when available. This `SdlManager` will handle sending and receiving messages to and from SDL after connected.

Create a new service and name it appropriately, for this guide we are going to call it `SdlService`.

```
public class SdlService extends Service {
    //...
}
```

If you created the service using the Android Studio template then the service should have been added to your `AndroidManifest.xml` otherwise the service needs to be defined in the manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

        <service
        android:name=".SdlService"
        android:enabled="true"/>

    </application>

</manifest>
```

Entering the Foreground

Because of Android Oreo's requirements, it is mandatory that services enter the foreground for long running tasks. The first bit of integration is ensuring that happens in the `onCreate` method of the `SdlService` or similar. Within the service that implements the SDL lifecycle you will need to add a call to start the service in the foreground. This will include creating a notification to sit in the status bar tray. This information and icons should be relevant for what the service is doing/going to do. If you already start your service in the foreground, you can ignore this section.

```
public void onCreate() {
    super.onCreate();
    //...
    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.createNotificationChannel(...);
        Notification serviceNotification = new Notification.Builder(this, *
Notification Channel*)
            .setContentTitle(...)
            .setSmallIcon(....)
            .setLargeIcon(...)
            .setContentText(...)
            .setChannelId(channel.getId())
            .build();
        startForeground(id, serviceNotification);
    }
}
```

NOTE

The sample code checks if the OS is of Android Oreo or newer to start a foreground service. It is up to the app developer if they wish to start the notification in previous versions.

Exiting the Foreground

It's important that you don't leave you notification in the notification tray as it is very confusing to users. So in the `onDestroy` method in your service, simply call the `stopForeground` method.

```
@Override
public void onDestroy(){
    //...
    if(Build.VERSION.SDK_INT>=Build.VERSION_CODES.O){
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        if(notificationManager!=null){ //If this is the only notification on
your channel
            notificationManager.deleteNotificationChannel(* Notification
Channel*);
        }
        stopForeground(true);
    }
}
```

Implementing SDL Manager

In order to correctly connect to an SDL enabled head unit developers need to implement methods for the proper creation and disposing of an `SdlManager` in our `SdlService`.

> NOTE
>
> An instance of SdlManager cannot be reused after it is closed and properly disposed of. Instead, a new instance must be created. Only one instance of SdlManager should be in use at any given time.

```
public class SdlService extends Service {

    //The manager handles communication between the application and
SDL
    private SdlManager sdlManager = null;

    //...

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        if (sdlManager == null) {
            MultiplexTransportConfig transport = new
MultiplexTransportConfig(this, APP_ID, MultiplexTransportConfig.
FLAG_MULTI_SECURITY_OFF);

            // The app type to be used
            Vector<AppHMIType> appType = new Vector<>();
            appType.add(AppHMIType.MEDIA);

            // The manager listener helps you know when certain events
that pertain to the SDL Manager happen
            SdlManagerListener listener = new SdlManagerListener() {

                @Override
                public void onStart() {
                    // After this callback is triggered the SdlManager can be
used to interact with the connected SDL session (updating the display,
sending RPCs, etc)
                }

                @Override
                public void onDestroy() {
                    SdlService.this.stopSelf();
                }

                @Override
                public void onError(String info, Exception e) {
                }
            };

            // Create App Icon, this is set in the SdlManager builder
            SdlArtwork appIcon = new SdlArtwork(ICON_FILENAME, FileType.
GRAPHIC_PNG, R.mipmap.ic_launcher, true);

            // The manager builder sets options for your session
            SdlManager.Builder builder = new SdlManager.Builder(this,
APP_ID, APP_NAME, listener);
            builder.setAppTypes(appType);
```

```
            builder.setTransportType(transport);
            builder.setAppIcon(appIcon);
            sdlManager = builder.build();
            sdlManager.start();
        }

    }
```

The `onDestroy()` method from the `SdlManagerListener` is called whenever the manager detects some disconnect in the connection, whether initiated by the app, by SDL, or by the device's connection.

> NOTE
>
> The `sdlManager` must be shutdown properly in the `SdlService.onDestroy()` callback using the method `sdlManager.dispose()`.

Determining SDL Support

You have the ability to determine a minimum SDL protocol and a minimum SDL RPC version that your app supports. We recommend not setting these values until your app is ready for production. The OEMs you support will help you configure the correct `minimumProtocolVersion` and `minimumRPCVersion` during the application review process.

If a head unit is blocked by protocol version, your app icon will never appear on the head unit's screen. If you configure your app to block by RPC version, it will appear and then quickly disappear. So while blocking with `minimumProtocolVersion` is preferable, `minimumRPCVersion` allows you more granular control over which RPCs will be present.

```
builder.setMinimumProtocolVersion(new Version("3.0.0"));
builder.setMinimumRPCVersion(new Version("4.0.0"));
```

Listening for RPC notifications and events

We can listen for specific events using `SdlManager`'s builder `setRPCNotificationListeners`. The following example shows how to listen for HMI Status notifications. Additional listeners can be added for specific RPCs by using their corresponding `FunctionID` in place of the `ON_HMI_STATUS` in the following example and casting the `RPCNotification` object to the correct type.

EXAMPLE OF A LISTENER FOR HMI STATUS:

```
Map<FunctionID, OnRPCNotificationListener>
onRPCNotificationListenerMap = new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

SmartDeviceLink Router Service

The `SdlRouterService` will listen for a connection with an SDL enabled module. When a connection happens, it will alert all SDL enabled apps that a connection has been established and they should start their SDL services.

We must implement a local copy of the `SdlRouterService` into our project. The class doesn't need any modification, it's just important that we include it. We will extend the `com.smartdevicelink.transport.SdlRouterService` in our class named `SdlRouterService`:

> NOTE
>
> Do not include an import for `com.smartdevicelink.transport.SdlRouterService`. Otherwise, we will get an error for `'SdlRouterService' is already defined in this compilation unit`.

```
public class SdlRouterService extends  com.smartdevicelink.transport.SdlRouterService {
    //Nothing to do here
}
```

> MUST
>
> The local extension of the `com.smartdevicelink.transport.SdlRouterService` must be named `SdlRouterService`.

MUST

Make sure this local class `SdlRouterService.java` is in the same package of `SdlReceiver.java` (described below)

If you created the service using the Android Studio template then the service should have been added to your `AndroidManifest.xml` otherwise the service needs to be added in the manifest. Because we want our service to be seen by other SDL enabled apps, we need to set `android:exported="true"`. The system may issue a lint warning because of this, so we can suppress that using `tools:ignore="ExportedService"`.

MUST

The `SdlRouterService` must be placed in a separate process with the name `com.smartdevicelink.router`. If it is not in that process during it's start up it will stop itself.

Intent Filter

```
<intent-filter>
    <action android:name="com.smartdevicelink.router.service"/>
</intent-filter>
```

The new versions of the SDL Android library rely on the `com.smartdevicelink.router.service` action to query SDL enabled apps that host router services. This allows the library to determine which router service to start.

MUST

This `intent-filter` MUST be included.

Metadata

ROUTER SERVICE VERSION

```
<meta-data android:name="sdl_router_version"  android:value=
"@integer/sdl_router_service_version_value" />
```

Adding the `sdl_router_version` metadata allows the library to know the version of the router service that the app is using. This makes it simpler for the library to choose the newest router service when multiple router services are available.

CUSTOM ROUTER SERVICE

```
<meta-data android:name="sdl_custom_router" android:value="false" /
>
```

> NOTE
>
> This is only for specific OEM applications, therefore normal developers do not need to worry about this.

Some OEMs choose to implement custom router services. Setting the `sdl_custom_router` metadata value to `true` means that the app is using something custom over the default router service that is included in the SDL Android library. Do not include this `meta-data` entry unless you know what you are doing.

SmartDeviceLink Broadcast Receiver

The Android implementation of the SdlManager relies heavily on the OS's bluetooth and USB intents. When the phone is connected to SDL and the router service has sent a connection intent, the app needs to create an SdlManager, which will bind to the already connected router service. As mentioned previously, the SdlManager cannot be re-used. When a disconnect between the app and SDL occurs, the current SdlManager must be disposed of and a new one created.

The SDL Android library has a custom broadcast receiver named `SdlBroadcastReceiver` that should be used as the base for your BroadcastReceiver. It is a child class of Android's BroadcastReceiver so all normal flow and attributes will be available. Two abstract methods will be automatically populate the class, we will fill them out soon.

Create a new SdlBroadcastReceiver and name it appropriately, for this guide we are going to call it `SdlReceiver`:

```
public class SdlReceiver extends SdlBroadcastReceiver {

    @Override
    public void onSdlEnabled(Context context, Intent intent) {
        //...

    }

    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
        //...
    }
}
```

MUST

SdlBroadcastReceiver must call super if `onReceive` is overridden

```
    @Override
    public void onReceive(Context context, Intent intent) {
        super.onReceive(context, intent);
        //your code here
    }
```

If you created the BroadcastReceiver using the Android Studio template then the service should have been added to your `AndroidManifest.xml` otherwise the receiver needs to be defined in the manifest. Regardless, the manifest

needs to be edited so that the `SdlBroadcastReceiver` needs to respond to the following intents:

- android.bluetooth.device.action.ACL_CONNECTED
- sdl.router.startservice

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

        <receiver
            android:name=".SdlReceiver"
            android:exported="true"
            android:enabled="true">

            <intent-filter>
                <action android:name=
"android.bluetooth.device.action.ACL_CONNECTED" />
                <action android:name="sdl.router.startservice" />
            </intent-filter>

        </receiver>

    </application>

    <!-- Required to use the lock screen -->
    <activity android:name=
"com.smartdevicelink.managers.lockscreen.SDLLockScreenActivity"
                android:launchMode="singleTop"/>

</manifest>
```

NOTE

The intent `sdl.router.startservice` is a custom intent that will come from the SdlRouterService to tell us that we have just connected to an SDL enabled piece of hardware.

MUST

SdlBroadcastReceiver has to be exported, or it will not work correctly

Next, we want to make sure we supply our instance of the SdlBroadcastService with our local copy of the SdlRouterService. We do this by simply returning the class object in the method defineLocalSdlRouterClass:

```
public class SdlReceiver extends SdlBroadcastReceiver {
  @Override
   public void onSdlEnabled(Context context, Intent intent) {

   }

   @Override
   public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
      //Return a local copy of the SdlRouterService located in your
project
      return com.company.mySdlApplication.SdlRouterService.class;
   }
}
```

We want to start the SdlManager when an SDL connection is made via the `SdlRouterService`. We do this by taking action in the onSdlEnabled method:

MUST

Apps must start their service in the foreground as of Android Oreo (API 26).

```
public class SdlReceiver extends SdlBroadcastReceiver {

    @Override
    public void onSdlEnabled(Context context, Intent intent) {
        //Use the provided intent but set the class to the SdlService
        intent.setClass(context, SdlService.class);
        if(Build.VERSION.SDK_INT < Build.VERSION_CODES.O) {
            context.startService(intent);
        }else{
            context.startForegroundService(intent);
        }
    }

    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
        //Return a local copy of the SdlRouterService located in your
project
        return com.company.mySdlApplication.SdlRouterService.class;
    }
}
```

NOTE

The `onSdlEnabled` method will be the main start point for our SDL connection session. We define exactly what we want to happen when we find out we are connected to SDL enabled hardware.

Lock Screen Activity

An Activity entry must also be added to the manifest for the SDL lock screen. For more information about lock screens, please see the Adding the Lock Screen section.

NOTE

When using `SdlManager`, the lock screen is enabled by default via the `LockScreenManager`. Please see the link above for more information

Once added, your `AndroidManifest.xml` should be defined like below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

        <service
            android:name=
"com.company.mySdlApplication.SdlRouterService"
            android:exported="true"
            android:process="com.smartdevicelink.router"
            tools:ignore="ExportedService">
            <intent-filter>
                <action android:name="com.smartdevicelink.router.service"/
>
            </intent-filter>
            <meta-data android:name="sdl_router_version"
android:value="@integer/sdl_router_service_version_value" />
        </service>

        <!-- Required to use the lock screen -->
        <activity android:name=
"com.smartdevicelink.managers.lockscreen.SDLLockScreenActivity"
                  android:launchMode="singleTop"/>

    </application>

</manifest>
```

Main Activity

Now that the basic connection infrastructure is in place, we should add methods to start the SdlService when our application starts. In `onCreate()` in your main activity, you need to call a method that will check to see if there is currently an SDL connection made. If there is one, the onSdlEnabled method will be called and we will follow the flow we already set up. In our `MainActivity.java` we need to check for an SDL connection:

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //If we are connected to a module we want to start our SdlService
        SdlReceiver.queryForConnectedService(this);
    }
}
```

Adding the Lock Screen

In order for your SDL application to be certified with most OEMs you will be required to implement a lock screen on the mobile device. The lock screen will disable user interactions with the application on the mobile device while they are using the head-unit to control application functionality. OEMs may choose to send their logo for your app's lock screen to use; the `LockScreenManager` takes care of this automatically using the default layout.

NOTE

This guide assumes that you have an SDL Service implemented as defined in the Getting Started guide.

There is a manager called the `LockScreenManager` that is accessed through the `SdlManager` that handles much of the logic for you. If you have implemented the `SdlManager` and have defined the `SDLLockScreenActivity` in your manifest but have not defined any lock screen configuration, you are already have a working default configuration. This guide will go over specific configurations you are able to implement using the `LockScreenManager` functionality.

Lock Screen Activity

You must declare the `SDLLockScreenActivity` in your manifest. To do so, simply add the following to your app's `AndroidManifest.xml` if you have not already done so:

```
<activity android:name=
"com.smartdevicelink.managers.lockscreen.SDLLockScreenActivity"
                android:launchMode="singleTop"/>
```

MUST

This manifest entry must be added for the lock screen feature to work.

Configurations

The default configurations should work for most app developers and is simple to get up and and running. However, it is easy to perform deeper configurations to the lock screen for your app. Below are the options that are available to customize your lock screen which builds on top of the logic already implemented in the `LockScreenManager`.

There is a setter in the `SdlManager.Builder` that allows you to set a `LockScreenConfig` by calling `builder.setLockScreenConfig(lockScreenConfig)`. The following options are available to be configured with the `LockScreenConfig`.

In order to to use these features, create a `LockScreenConfig` object and set it using `SdlManager.Builder` before you build `SdlManager`.

CUSTOM BACKGROUND COLOR

In your `LockScreenConfig` object, you can set the background color to a color resource that you have defined in your `Colors.xml` file:

```
lockScreenConfig.setBackgroundColor(resourceColor); // For example, R.color.black
```

CUSTOM APP ICON

In your `LockScreenConfig` object, you can set the resource location of the drawable icon you would like displayed:

```
lockScreenConfig.setAppIcon(appIconInt); // For example, R.drawable.lockscreen_icon
```

SHOWING THE DEVICE LOGO

This sets whether or not to show the connected device's logo on the default lock screen. The logo will come from the connected hardware if set by the manufacturer. When using a Custom View, the custom layout will have to handle the logic to display the device logo or not. The default setting is false, but some OEM partners may require it.

In your `LockScreenConfig` object, you can set the boolean of whether or not you want the device logo shown, if available:

```
lockScreenConfig.showDeviceLogo(true);
```

SETTING A CUSTOM LOCK SCREEN VIEW

If you'd rather provide your own layout, it is easy to set. In your `LockScreenConfig` object, you can set the reference to the custom layout to be used for the lock screen. If this is set, the other customizations described above will be ignored:

```
lockScreenConfig.setCustomView(customViewInt);
```

DISABLING THE LOCK SCREEN MANAGER:

Please note that a lock screen will likely be required by OEMs. You can disable the `LockScreenManager`, but you will then be required to create your own implementation. This is not recommended as the `LockScreenConfig` should enable all possible settings while still adhering to most OEM requirements. However, if it is unavoidable to create one from scratch the `LockScreenManager` can be disabled via the `LockScreenConfig` as follows.

```
lockScreenConfig.setEnabled(false);
```

> NOTE
>
> When the enabled flag is set to `false` all other config options will be ignored.

Using Android Open Accessory Protocol

Incorporating AOA into an SDL enabled app allows it to create and register an SDL session over USB. This guide will assume the app has already integrated the SDL library as laid out in the previous guides. AOA connections are sent through the `SDLRouterService` to bypass an Android limitation of only one app being able to be used through the AOA intent.

Prerequisites:

- Installation guide
- Integration Basics guide

We will add or make changes to:

- Android Manifest **(of your app)**
- SdlService *(optional)*

Prerequisites

The Installation and Integration Basics guides must be completed before enabling the use of the AOA USB transport. The remainder of the guide will assume all steps of those two guides will be followed.

Android Manifest

To use the AOA protocol, you must specify so in your app's Manifest with:

```
<uses-feature android:name="android.hardware.usb.accessory"/>
```

> MUST
>
> This feature will not work without including this line!

The SDL Android library houses a USBAccessoryAttachmentActivity that you need to add between your Manifest's `<application>...</application>` tags:

```
<activity android:name=
"com.smartdevicelink.transport.USBAccessoryAttachmentActivity"
    android:launchMode="singleTop">
    <intent-filter>
        <action android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
    </intent-filter>

    <meta-data
        android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED"
        android:resource="@xml/accessory_filter" />
</activity>
```

> NOTE
>
> The accessory_filter.xml file is included with the SDL Android Library

SmartDeviceLink Service

As long as the app doesn't require high bandwidth, it shouldn't matter which transport is being connected, and will be transport to the developer. If the integration guides were followed, a multiplex transport configuration was already created and provided to the `SdlManager` like the one that follows:

```
@Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        if (sdlManager == null) {
            MultiplexTransportConfig transport = new
MultiplexTransportConfig(this, APP_ID, MultiplexTransportConfig.
FLAG_MULTI_SECURITY_OFF);

            SdlManagerListener listener = new SdlManagerListener() {
                //...
            };

            // ...

            builder.setTransportType(transport);
            sdlManager = builder.build();
            sdlManager.start();
        }
```

Using only USB / AOA

The new `MultiplexingConfig` allows for apps to be able to connect via Bluetooth and USB as primary transports. If you want your app to only use USB / AOA, then you should specifically only set that as the only allowed primary transport.

When defining your transport, also pass in a custom list that only contains the USB:

```
List<TransportType> multiplexPrimaryTransports = Arrays.asList(
TransportType.USB);

MultiplexTransportConfig transport = new MultiplexTransportConfig(this,
 appId, MultiplexTransportConfig.FLAG_MULTI_SECURITY_MED);

transport.setPrimaryTransports(multiplexPrimaryTransports);
```

Multiple Transports

Since the `SdlRouterService` now handles both bluetooth and AOA/USB connections, an app will be connected to the transport that connects first if the app includes it in their transport config. If a module supports secondary transports, the second transport to be connected of bluetooth or USB will be available as well as potentially TCP. This means even though the app might register over bluetooth, if USB or TCP are available those transports will be available for high bandwidth services. For more information please see the Multiple Transport Guide.

Multiple Transports

As of Protocol Version 5.1.0, which is supported from SDL Android 4.7 and SDL Core 5.0, a new feature was introduced called Multiple Transports. This feature allows apps to carry their SDL session over multiple transports. The first transport that the app connects to is referred to as the primary transport, and a later connected transport being a secondary. For example, apps can register

over bluetooth as a primary transport, then connect over WiFi when necessary (video/audio streaming) as a secondary transport.

SDL Android

PRIMARY TRANSPORTS

This feature coincides with our newly redesigned multiplexing transport. In SDL Android 4.7 and newer, you can connect and register apps via a multiplexed bluetooth and/or USB connection. On head units that support multiple transports, the primary transport will be used for RPC communication while the secondary will be used for high bandwidth services. Otherwise, the primary transport will be used for all applicable services for that transport type.

SUPPORTING SPECIFIC PRIMARY TRANSPORTS

Whether your app supports both bluetooth and/or USB connections are determined by what you set as acceptable primary transports. By default, both USB and bluetooth are supported and should be kept unless there is a specific reason otherwise. If you list multiple primary transports and one disconnects, if another included transport is available the app will automatically attempt to connect and register.

```
List<TransportType> multiplexPrimaryTransports = Arrays.asList(
TransportType.USB, TransportType.BLUETOOTH);
MultiplexTransportConfig mtc = new MultiplexTransportConfig(this,
APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_OFF);
mtc.setPrimaryTransports(multiplexPrimaryTransports);
```

If you only want to use bluetooth or USB, simply pass in a list with the one you want.

NOTE

For the best compatibility we suggest supporting both primary transports.

REQUIRES HIGH BANDWIDTH

Certain app types will require a high bandwidth transport to be available, which could be either primary or secondary transports. If this is the case, an app will only be registered if a high bandwidth transport is either connected or available to connect.

If this is the case for your app you can set the `setRequiresHighBandwidth` flag to `true`:

```
MultiplexTransportConfig mtc = new MultiplexTransportConfig(this,
APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_OFF);

mtc.setRequiresHighBandwidth(true);
```

HIGH BANDWIDTH APP WITH LOW BANDWIDTH SUPPORT

While some app's main integration requires high bandwidth, it is possible to support a low bandwidth integration for better visibility. As an example, a navigation app might require high bandwidth transport to stream their map view but could provide a low bandwidth integration that displays turn-by-turn directions. Another simple low bandwidth integration could simply be displaying a message that instructs the user to connect USB or WiFi to enable the app. In this case the app should set the requires high bandwidth flag to false, as it is by default.

```
MultiplexTransportConfig mtc = new MultiplexTransportConfig(this,
APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_OFF);

mtc.setRequiresHighBandwidth(false);
```

SECONDARY TRANSPORTS

Secondary transports are supported as of Protocol Version 5.1.0 , and must be enabled by the module the app is connecting. In addition to supporting bluetooth and USB, TCP is also a supported as a secondary transport.

Setting secondary transports that your app supports is similar to setting the primary transports:

```
List<TransportType> multiplexPrimaryTransports = Arrays.asList(
TransportType.USB, TransportType.BLUETOOTH);
List<TransportType> multiplexSecondaryTransports = Arrays.asList(
TransportType.TCP, TransportType.USB, TransportType.BLUETOOTH);
MultiplexTransportConfig mtc = new MultiplexTransportConfig(this,
APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_OFF);
mtc.setPrimaryTransports(multiplexPrimaryTransports);
mtc.setSecondaryTransports(multiplexSecondaryTransports);
```

By default, all three transports are set as supported secondary transports. As mentioned above, secondary transports will often be used for high bandwidth services.

Setting Security Level for Multiplexing

When connecting to Core via Multiplex Bluetooth transport, your SDL app will use a Router Service housed within your app or another SDL enabled app.

To help ensure the validility of the Router Service, you can select the security level explicity when you create your Multiplex Bluetooth transport in your app's SdlService:

```
int securityLevel = FLAG_MULTI_SECURITY_MED;

BaseTransport transport = MultiplexTransportConfig(context, appId,
securityLevel);
```

If you create the transport without specifying the security level, it will be set to `FLAG_MULTI_SECURITY_MED` by default.

Security Levels

SECURITY FLAG	MEANING
`FLAG_MULTI_SECURITY_OFF`	Multiplexing security turned off. All router services are trusted.
`FLAG_MULTI_SECURITY_LOW`	Multiplexing security will be minimal. Only trusted router services will be used. Trusted router list will be obtained from server. List will be refreshed every 20 days or during next connection session if an SDL enabled app has been installed or uninstalled.
`FLAG_MULTI_SECURITY_MED`	Multiplexing security will be on at a normal level. Only trusted router services will be used. Trusted router list will be obtained from server. List will be refreshed every 7 days or during next connection session if an SDL enabled app has been installed or uninstalled.
`FLAG_MULTI_SECURITY_HIGH`	Multiplexing security will be very strict. Only trusted router services installed from trusted app stores will be used. Trusted router list will be obtained from server. List will be refreshed every 7 days or during next connection session if an SDL enabled app has been installed or uninstalled.

Applying to the Trusted Router Service Database

For an Android application to be added to the Trusted Router Service database, the application will need to be registered on the SDL Developer Portal and

certified by the SDLC. For more information on registration, please see this guide.

Any Android application that is certified by the SDLC will be added to the Trusted Router Service database; there are no additional steps required as it is part of the certification process.

Please consult the Trusted Router Service FAQs if you have any additional questions.

Hello SDL Android

Introduction

In this guide we take you through the steps to get our sample project, Hello Sdl Android, running and connected to Sdl Core as well as showing up on the generic HMI.

First, make sure you download or clone the latest release from GitHub. It is a package within the SDL Android library.

Open the project in Android Studio. We will exclusively use Android Studio as it is the current supported platform for Android development.

Getting Started

If you are not using a Ford TDK for development, we will assume that you have SDL Core (We recommend Ubuntu 16.04) and an HMI set up prior to this point. Most people getting started with this tutorial will not have a Ford TDK, so sample outputs will be using Sdl Core and our Generic HMI.

If you don't want to set up a virtual machine for testing, we offer Manticore, which is a free service that allows you to test your apps via TCP/IP in the cloud.

NOTE

Sdl Core and an HMI or Manticore are needed to run Hello Sdl Android and to ensure that it connects

BUILD FLAVORS

Hello Sdl Android has been built with different **build flavors**.

To access the Build Variant menu to choose your flavor, click on the menu `Build` then `Select Build Variant`. A small window will appear on the bottom left of your IDE window that allows you to choose a flavor.

There are many flavors to choose from and for now we will only be concerned with the debug versions.

Versions Include:

- `multi` - Multiplexing (Bluetooth, USB, TCP (as secondary transport))
- `multi_high_bandwidth` - Multiplexing for apps that require a high bandwidth transport
- `tcp` - Transmission Control Protocol - used only for debugging purposes

We will mainly be dealing with `multi` (if using a TDK) or `tcp` (if connecting to SDL Core via a virtual machine or your localhost, or to Manticore)

Transports

CONFIGURE FOR TCP

If you aren't using a TDK or head unit, you can connect to SDL core via a virtual machine or to your localhost. To do this we will use the flavor `tcpDebug`.

For TCP to work, you will have to know the IP address of your machine that is running Sdl Core. If you don't know what it is, running `ifconfig` in a linux terminal will usually let you see it for the interface you are connected with to your network. We have to modify the IP address in Hello Sdl Android to let it know where your instance of Sdl Core is running.

In the main Java folder of Hello Sdl Android, open up `SdlService.java`

In the top of this file, locate the variable declaration for `DEV_MACHINE_IP_ADDRESS`. Change it to your Sdl Core's IP. Leave the `TCP_PORT` set to `12345`.

```
    // TCP/IP transport config
    private static final int TCP_PORT = 12345; // if using manticore,
change to assigned port
    private static final String DEV_MACHINE_IP_ADDRESS =
"192.168.1.78"; // change to your IP
```

> NOTE
>
> if you do not change the target IP address, the application will not connect to Sdl Core or show up on the HMI

CONFIGURE FOR BLUETOOTH

Right out of the box, all you need to do to run bluetooth is to select the `multi_sec_offDebug` (Multiplexing) build flavor.

CONFIGURE FOR USB (AOA)

To connect to an SDL Core instance or TDK via USB transport, select the `multi_sec_offDebug` (Multiplexing) build flavor. There is more information for USB transport under Getting Started - Using AOA Protocol.

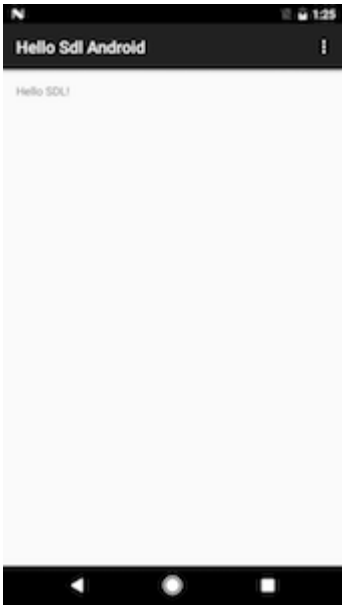
Building the Project

For TCP, you may use the built-in Android emulator or an Android phone on the same network as Sdl Core. For Bluetooth, you will need an Android phone that is paired to a TDK or head unit via Bluetooth.

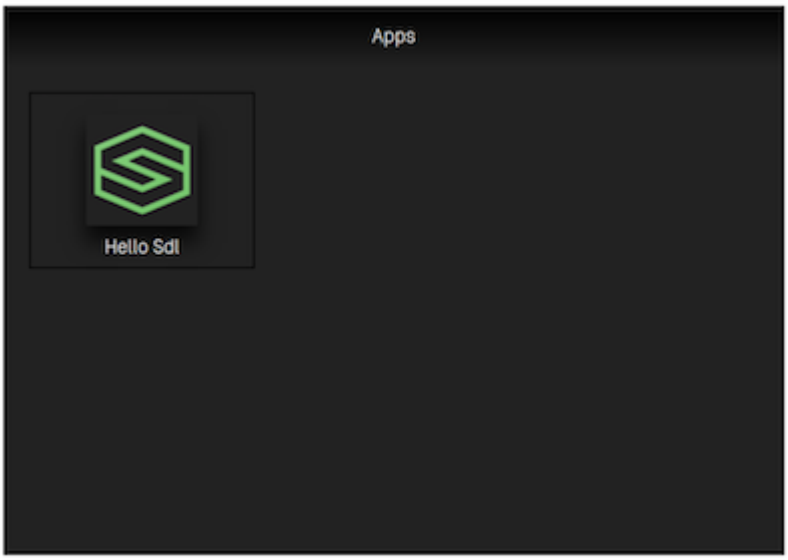
> MUST
>
> Make sure Sdl Core and the HMI are running prior to running Hello Sdl Android

Run the project in Android Studio, targeting the device you want Hello Sdl Android installed on.

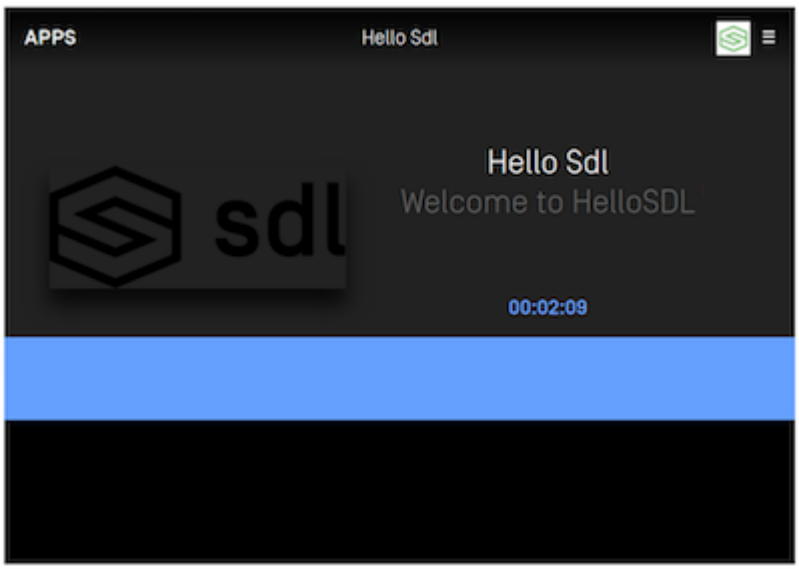
Hello Sdl Android should compile and launch on your device of choosing:



Following this, you should see an application appear on the TDK or HMI. In the case of the Generic HMI (using TCP), you will see the following:



Click on the Hello Sdl icon in the HMI.



This is the main screen of the Hello Sdl App. If you get to this point, the project is working.

On the device you are running the app on, a lock screen should now appear once the app is opened on the HMI if distracted driver notifications are set to `DD_On`:

NOTE

Lock Screens are an important part of Sdl enabled applications. The goal is to keep the driver's eyes forward and off of the device

At this point Hello Sdl Android has been compiled and is running properly! Continue reading through our guides to learn about all of the RPCs (Remote Procedure Calls) that can be made with the library.

Troubleshooting

Sometimes things don't always go as planned, and so this section exists. If your app compiles and does NOT show up on the HMI, there are a few things to check out.

TCP

1. Make sure that you have changed the IP in `SdlService.java` to match the machine running Sdl Core. Being on the same network is also important.
2. If you are sure that the IP is correct and it is still not showing up, make sure the Build Flavor that is running is `tcpDebug`.
3. If the two above dont work, make sure there is no firewall blocking the incoming port `12345` on the machine or VM running SDL Core. In the same breath, make sure your firewall allows that outgoing port.
4. There are different network configurations needed for different virtualization software (virtualbox, vmware, etc). Make sure yours is set up correctly. Or use Manticore.

BLUETOOTH

1. Make sure the build flavor `multi_sec_offDebug` is selected.
2. Ensure your phone is properly paired with the TDK
3. Make sure Bluetooth is turned on - on Both the TDK and your phone
4. Make sure apps are enabled on the TDK (in settings)

Installation

Introduction

Each SDL JavaSE library release is published to JCenter. By adding a few lines in their app's gradle script, developers can compile with the latest SDL JavaSE release.

To gain access to the JCenter repository, make sure your app's `build.gradle` file includes the following:

```
repositories {
    google()
    jcenter()
}
```

Gradle Build

To compile with the a release of SDL JavaSE, include the following line in your app's `build.gradle` file,

```
dependencies {
   implementation 'com.smartdevicelink:sdl_java_se:{version}'
}
```

and replace `{version}` with the desired release version in format of `x.x.x`.
The list of releases can be found here.

Examples

To compile release 4.8.1, use the following line:

```
dependencies {
   implementation 'com.smartdevicelink:sdl_java_se:4.8.1'
}
```

To compile the latest minor release of major version 4, use:

```
dependencies {
   implementation 'com.smartdevicelink:sdl_java_se:4.+'
}
```

Integration Basics

Getting Started on JavaSE

In this guide, we exclusively use IntelliJ. We are going to set-up a bare-bones application so you get started using SDL.

> NOTE
>
> The SDL Java library supports Java 7 and above.

SmartDeviceLink Service

A SmartDeviceLink Java Service should be created to manage the lifecycle of the SDL session. The `SdlService` should build and start an instance of the `SdlManager` which will automatically connect with a headunit when available. This `SdlManager` will handle sending and receiving messages to and from SDL after connected.

Create a new class and name it appropriately, for this guide we are going to call it `SdlService`.

```
public class SdlService {
    //...
}
```

Implementing SDL Manager

In order to correctly connect to an SDL enabled head unit developers need to implement methods for the proper creation and disposing of an `SdlManager` in our `SdlService`.

> NOTE
>
> An instance of SdlManager cannot be reused after it is closed and properly disposed of. Instead, a new instance must be created. Only one instance of SdlManager should be in use at any given time.

```
public class SdlService {

    //The manager handles communication between the application and SDL
    private SdlManager sdlManager = null;

    //...

    private void buildSdlManager(BaseTransportConfig transport) {

        if (sdlManager == null) {

            // The app type to be used
            Vector<AppHMIType> appType = new Vector<>();
            appType.add(AppHMIType.MEDIA);

            // The manager listener helps you know when certain events that pertain to the SDL Manager happen
            SdlManagerListener listener = new SdlManagerListener() {

                @Override
                public void onStart() {
                    // After this callback is triggered the SdlManager can be used to interact with the connected SDL session (updating the display, sending RPCs, etc)
                }

                @Override
                public void onDestroy() {
                }

                @Override
                public void onError(String info, Exception e) {
                }
            };

            // Create App Icon, this is set in the SdlManager builder
            SdlArtwork appIcon = new SdlArtwork(ICON_FILENAME, FileType.GRAPHIC_PNG, ICON_PATH, true);

            // The manager builder sets options for your session
            SdlManager.Builder builder = new SdlManager.Builder(APP_ID, APP_NAME, listener);
            builder.setAppTypes(appType);
            builder.setTransportType(transport);
            builder.setAppIcon(appIcon);
            sdlManager = builder.build();
            sdlManager.start();
        }
```

```
    }
}
```

> NOTE
>
> The `sdlManager` must be shutdown properly if this class is shutting down in the respective method using the method `sdlManager.dispose()`.

Determining SDL Support

You have the ability to determine a minimum SDL protocol and a minimum SDL RPC version that your app supports. We recommend not setting these values until your app is ready for production. The OEMs you support will help you configure the correct `minimumProtocolVersion` and `minimumRPCVersion` during the application review process.

If a head unit is blocked by protocol version, your app icon will never appear on the head unit's screen. If you configure your app to block by RPC version, it will appear and then quickly disappear. So while blocking with `minimumProtocolVersion` is preferable, `minimumRPCVersion` allows you more granular control over which RPCs will be present.

```
builder.setMinimumProtocolVersion(new Version("3.0.0"));
builder.setMinimumRPCVersion(new Version("4.0.0"));
```

Listening for RPC notifications and events

We can listen for specific events using `SdlManager`'s builder `setRPCNotificationListeners`. The following example shows how to listen for HMI Status notifications. Additional listeners can be added for specific RPCs by using their corresponding `FunctionID` in place of the `ON_HMI_STATUS` in the following example and casting the `RPCNotification` object to the correct type.

EXAMPLE OF A LISTENER FOR HMI STATUS:

```
Map<FunctionID, OnRPCNotificationListener>
onRPCNotificationListenerMap = new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

Main Class

Now that the basic connection infrastructure is in place, we should add methods to start the `SdlService` when our application starts. In `main(String[] args)` in your main class, you will create and start an instance of the `SdlService` class.

You will also need to fill in what port the app should listen on for an incoming web socket connection.

```
public class Main {

    Thread thread;
    SdlService sdlService;

    public static void main(String[] args) {
        Main main = new Main();
        main.startSdlService();

    }

private void startSdlService() {

        thread = new Thread(new Runnable() {

            @Override
            public void run() {
                sdlService  = new SdlService(new WebSocketServerConfig(
PORT, -1));
                sdlService.start();


            }
        });
        thread.start();
    }
}
```

Hello SDL JavaSE

Introduction

In this guide we take you through the steps to get our sample project, Hello Sdl, running and connected to Sdl Core as well as showing up on the generic HMI.

First, make sure you download or clone the latest release from GitHub. It is a project within the SDL Java Suite root directory. Then, open the Hello Sdl project in IntelliJ IDEA and wait for it to finish loading.

Getting Started

We assume that you have SDL Core (We recommend Ubuntu 16.04) and an HMI set up prior to this point. Most people getting started with this tutorial will be using Sdl Core and our Generic HMI. If you don't want to set up a virtual machine for testing, we offer Manticore, which is a free service that allows you to test your apps in the cloud.

> NOTE
>
> Sdl Core and an HMI or Manticore are needed to run Hello Sdl and to ensure that it connects.

Configuring Core

To let Sdl Core connect to your app, first you will have to know the IP address of the machine that is running the Hello Sdl app. If you don't know what it is, running `ifconfig` in the terminal will usually let you see it for the interface you are connected with to your network.

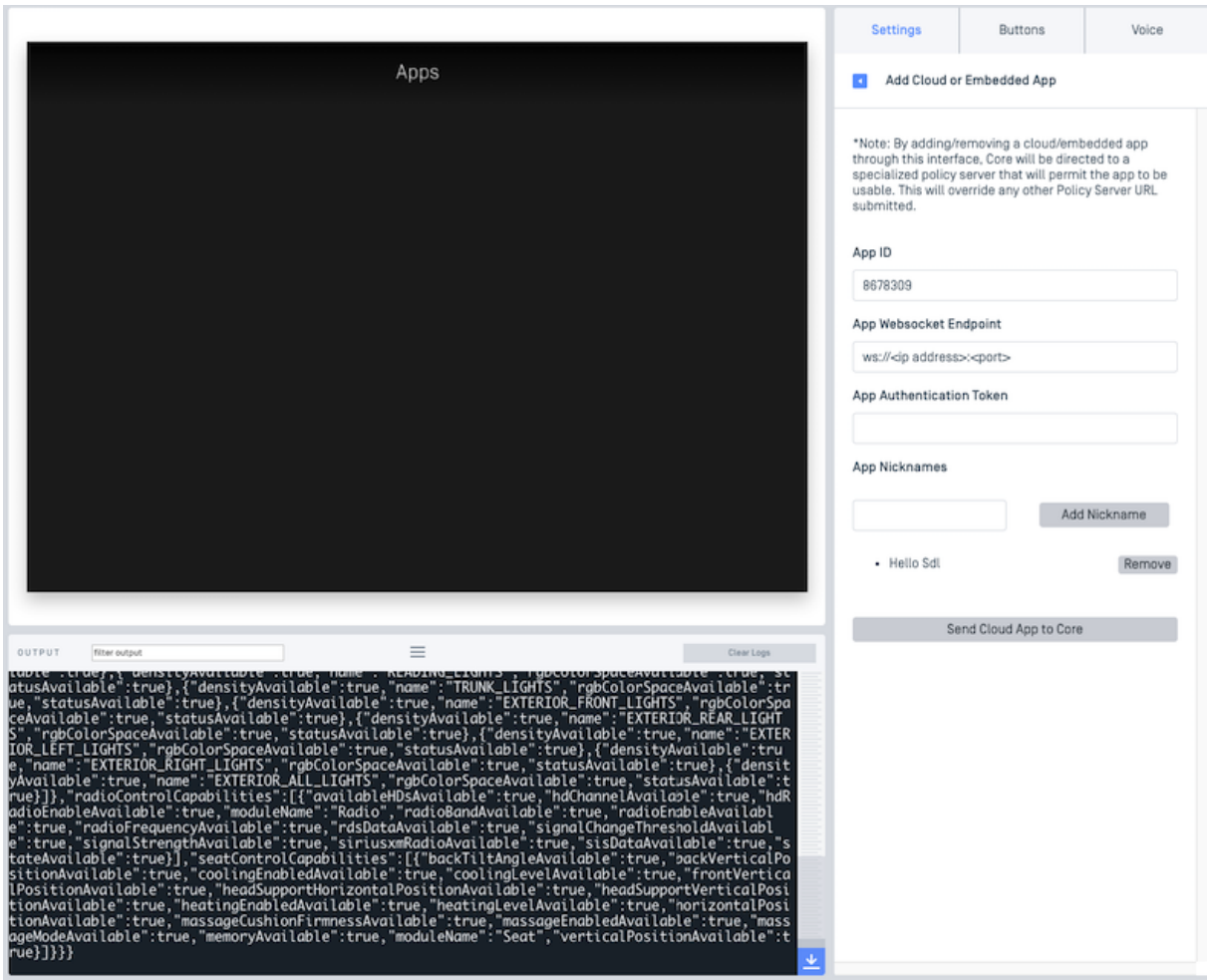
After getting the IP address, you will have to set App ID, App Websocket Endpoint, and App Nicknames in Sdl Core to let it know where your instance of Hello Sdl is running.

NOTE

The App Websocket Endpoint contains the IP Address and port as the following: `ws://<ip address>:<port>/`.

MANTICORE

If you are using Manticore, the app information can be easily set in the settings tab:



NOTE

Manticore needs to access you machine's IP address to be able to start a websocket connection with your embedded app. If you are hosting the embedded app on your local machine, you may need to do extra setup to make your machine publicly accessible. The other solution is to setup Core and HMI on your machine instead of using Manticore so Core can access your local IP address.

SDL CORE AND GENERIC HMI

If you are using Sdl Core and Generic HMI, you will have to add a policy table entry as the following one to the existing entries:

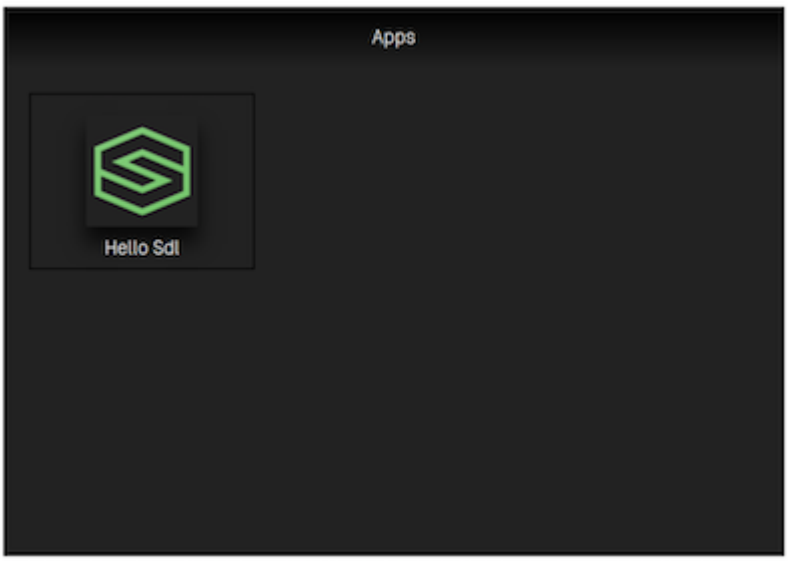
```
"8678309": {
    "keep_context": false,
    "steal_focus": false,
    "priority": "NONE",
    "default_hmi": "NONE",
    "groups": ["Base-4"],
    "RequestType": [],
    "RequestSubType": [],
    "hybrid_app_preference": "CLOUD",
    "endpoint": "ws://<ip address>:<port>",
    "enabled": true,
    "auth_token": "",
    "cloud_transport_type": "WS",
    "nicknames": ["Hello Sdl"]
}
```

Fore more information about policy tables please visit Plicy Tables Guides.

NOTE

Don't forget to replace `ws://<ip address>:<port>` with your own IP address and app port. The port that is used in Hello Sdl App is `5432`. It can be changed to a different port by modifying the number in `Main.java` class.

Following this, you should see an application appears on HMI as in the following screenshot:

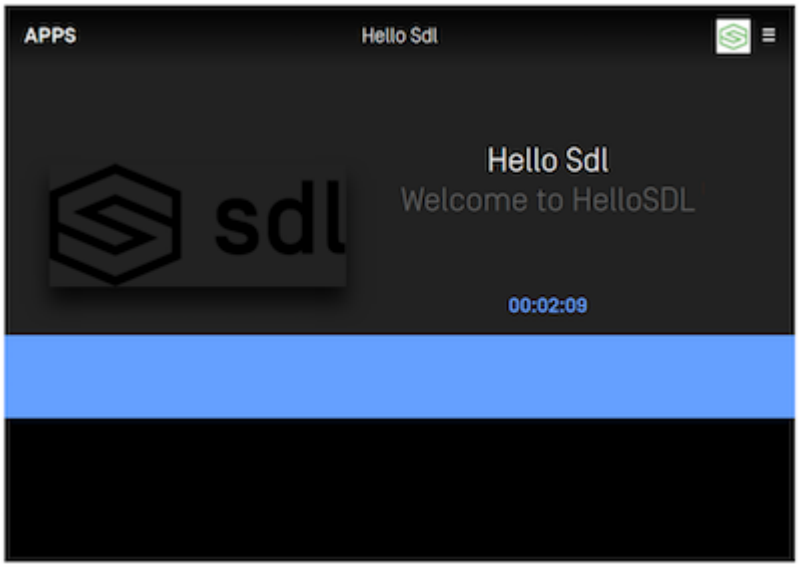


NOTE

Even though you see the app appears on HMI, you still cannot lunch the app at this point. You will have to run the Hello Sdl app from IntelliJ IDEA first as described next.

Running the App

After setting the app information in Sdl Core, you can run the project in IntelliJ IDEA. Hello Sdl should compile and launch on your your machine. After that, you can click on the Hello Sdl icon in the HMI.



This is the main screen of the Hello Sdl app. If you get to this point, the project is working.

At this point Hello Sdl has been compiled and is running properly! Continue reading through our guides to learn about all of the RPCs (Remote Procedure Calls) that can be made with the library.

Installation

Introduction

Each SDL JavaEE library release is published to Github. By building and importing the library JAR file to the project, developers can compile with the

latest SDL JavaEE release. In this guide we exclusively use IntelliJ to compile and build the project.

Building The JavaEE Library JAR

To build the library JAR from the source code, first clone the SDL Java Suite repository then, simply call:

```
gradle build
```

from within the JavaEE directory and a JAR should be generated in the build/libs folder.

Creating a New SDL Project

- Download Glassfish 5.0.0 Full Platform
- Start a new IntelliJ Project (Menu -> New -> Project)
- Select Java Enterprise as project type.
- Ensure Java EE 8 is the version used.
- Set Application Server to the directory of Glassfish 5.0.0 download.
- Check the following: (they should all be using the libraries from Glassfish)
    - Glassfish 5.0.0 - EJB
    - Glassfish 5.0.0 - WebSocket
    - Glassfish 5.0.0 - Web Application
- Give the project a name.
- Once the project is created, add the SDL Java JAR library
  (Right-click project -> Open Module Settings -> Libraries -> +)
- Go to artifacts. A problem may appear concerning the exploded war not having the library.
  Add SDL Java to the war exploded artifact (IntelliJ has an auto fix for it).
  Apply and OK.

- In the artifacts -> choose ejb exploded -> in the "output layout" tab, click on the + -> Extracted directory -> add the SDL Java JAR library
- In the artifacts -> choose war exploded -> in the "output layout" tab, click on the + -> Extracted directory -> add the SDL Java JAR library
- You can now start creating your SDL enabled application. If you already have source code to start with, you can copy it into the new project along with any jars and assets.

> NOTE
>
> Glassfish 5.0.0 only works on JDK 8 and lower.

Integration Basics

Getting Started on JavaEE

In this guide, we exclusively use IntelliJ. We are going to set-up a bare-bones application so you get started using SDL.

> NOTE
>
> The SDL Java library supports Java 7 and above.

SmartDeviceLink Service

A SmartDeviceLink Java Service should be created to manage the lifecycle of the SDL session. The `SdlService` should build and start an instance of the `SdlManager` which will automatically connect with a headunit when available. This `SdlManager` will handle sending and receiving messages to and from SDL after connected.

Create a new class and name it appropriately, for this guide we are going to call it `SdlService`.

```
public class SdlService {
   //...
}
```

Implementing SDL Manager

In order to correctly connect to an SDL enabled head unit developers need to implement methods for the proper creation and disposing of an `SdlManager` in our `SdlService`.

> NOTE
>
> An instance of SdlManager cannot be reused after it is closed and properly disposed of. Instead, a new instance must be created. Only one instance of SdlManager should be in use at any given time.

```
public class SdlService {

    //The manager handles communication between the application and SDL
    private SdlManager sdlManager = null;

    //...

    private void buildSdlManager(BaseTransportConfig transport) {

        if (sdlManager == null) {

            // The app type to be used
            Vector<AppHMIType> appType = new Vector<>();
            appType.add(AppHMIType.MEDIA);

            // The manager listener helps you know when certain events that pertain to the SDL Manager happen
            SdlManagerListener listener = new SdlManagerListener() {

                @Override
                public void onStart() {
                    // After this callback is triggered the SdlManager can be used to interact with the connected SDL session (updating the display, sending RPCs, etc)
                }

                @Override
                public void onDestroy() {
                }

                @Override
                public void onError(String info, Exception e) {
                }
            };

            // Create App Icon, this is set in the SdlManager builder
            SdlArtwork appIcon = new SdlArtwork(ICON_FILENAME, FileType.GRAPHIC_PNG, ICON_PATH, true);

            // The manager builder sets options for your session
            SdlManager.Builder builder = new SdlManager.Builder(APP_ID, APP_NAME, listener);
            builder.setAppTypes(appType);
            builder.setTransportType(transport);
            builder.setAppIcon(appIcon);
            sdlManager = builder.build();
            sdlManager.start();
        }
```

```
    }
}
```

> NOTE
>
> The `sdlManager` must be shutdown properly if this class is shutting down in the respective method using the method `sdlManager.dispose()`.

Adding EJB and Websockets

Create a new package where all the JavaEE-specific code will go.

The SDL Java library comes with a `CustomTransport` class which takes the role of sending messages between incoming sdl_core connections and your SDL application. You need to pass that class to the `SdlManager` builder to make the SDL Java library aware that you want to use your JavaEE websocket server as the transport.

Create a Java class in the new package which will be the `SDLSessionBean` class. This class utilizes the `CustomTransport` class and EJB JavaEE API which will make it the entry point of your app when a connection is made. It will open up a websocket server at `/` and create stateful beans, where the bean represents the logic of your cloud app. Every new connection to this endpoint creates a new bean containing your app logic, allowing for load balancing across all the instances of your app that were automatically created.

```
import com.smartdevicelink.transport.CustomTransport;
import javax.ejb.Stateful;
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.nio.ByteBuffer;

@ServerEndpoint("/")
@Stateful(name = "SDLSessionEJB")
public class SDLSessionBean {

    CustomTransport websocket;

    public class WebSocketEE extends CustomTransport {
        Session session;
        public WebSocketEE(String address, Session session) {
            super(address);
            this.session = session;
        }
        public void onWrite(byte[] bytes, int i, int i1) {
            try {
                session.getBasicRemote().sendBinary(ByteBuffer.wrap(bytes
));
            }
            catch (IOException e) {

            }
        }
    }

    @OnOpen
    public void onOpen (Session session) {
        websocket = new WebSocketEE("http://localhost", session) {};
        //TODO: pass your CustomTransport instance to your SDL app here
    }

    @OnMessage
    public void onMessage (ByteBuffer message, Session session) {
        websocket.onByteBufferReceived(message); //received message
from core
    }
}
```

Unfortunately, there's no way to get a client's IP address using the standard API , so localhost is passed to the CustomTransport for now as the transport address (this is only used locally in the library so it is not necessary).

The `SDLSessionBean` class's @OnOpen method is where you will start your app, and should call your entry of your application and invoke whatever is needed to start it. You need to pass the instantiated `CustomTransport` object to your application so that the connection can be passed into the `SdlManager`.

The SdlManager will need you to create a `CustomTransportConfig`, pass in the `CustomTransport` instance from the `SDLSessionBean` instance, then set the `SdlManager` Builder's transport type to that config. This will set your transport type into `CUSTOM` mode and will use your `CustomTransport` instance to handle the read and write operations.

```
// Set transport config. builder is a SdlManager.Builder
CustomTransportConfig transport = new CustomTransportConfig(
websocket);
builder.setTransportType(transport);
```

> NOTE
>
> The `SDLSessionBean` should be inside a Java package other than the default package in order for it to work properly.

ADD A NEW ARTIFACT:

- Right-click project -> Open Module Settings -> Artifacts -> + -> Web Application: Archive -> for your war: exploded artifact which should already exist
- Create Manifest. Apply + OK.
- Run Build -> Build Artifacts to get a .war file in the /out folder.

Determining SDL Support

You have the ability to determine a minimum SDL protocol and a minimum SDL RPC version that your app supports. We recommend not setting these values until your app is ready for production. The OEMs you support will help you configure the correct `minimumProtocolVersion` and `minimumRPCVersion` during the application review process.

If a head unit is blocked by protocol version, your app icon will never appear on the head unit's screen. If you configure your app to block by RPC version, it will appear and then quickly disappear. So while blocking with `minimumProtocolVersion` is preferable, `minimumRPCVersion` allows you more granular control over which RPCs will be present.

```
builder.setMinimumProtocolVersion(new Version("3.0.0"));
builder.setMinimumRPCVersion(new Version("4.0.0"));
```

Listening for RPC notifications and events

We can listen for specific events using `SdlManager`'s builder `setRPCNotificationListeners`. The following example shows how to listen for HMI Status notifications. Additional listeners can be added for specific RPCs by using their corresponding `FunctionID` in place of the `ON_HMI_STATUS` in the following example and casting the `RPCNotification` object to the correct type.

EXAMPLE OF A LISTENER FOR HMI STATUS:

```
Map<FunctionID, OnRPCNotificationListener>
onRPCNotificationListenerMap = new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

Deploying to AWS

Deploying your JavaEE App on EC2

When you want to run the project outside the IDE, take the war artifact and deploy it using a Payara server. Payara is built on top of Glassfish and is more well-maintained, and it also solves an issue with Glassfish on redeploying a JavaEE Websocket app where no connections can happen the second time.

Once you get an EC2 machine, log on it and install some libraries:

```
sudo yum install java-1.8.0-openjdk-headless.x86_64 java-1.8.0-
openjdk-devel.x86_64 -y
wget -O payara.zip  https://search.maven.org/remotecontent?
filepath=fish/payara/distributions/payara/5.184/payara-5.184.zip
jar xvf payara.zip
cd payara5/glassfish/bin/
sudo chmod 755 asadmin
```

There are two ways to start Payara. The first way will start the server without extra configuration, but it is not in a production context. asadmin's start-domain command is what runs the Payara server. Then, you need to deploy your war application. Modify the command below to point to where your war file is. When it deploys, it will be running on the root path "/" over port 8080. So, your websocket connection from SDL Core should point to the IP of the machine this is running on, port 8080, and nothing more.

```
./asadmin start-domain
./asadmin deploy --contextroot / ~/my-javaee-app.war
```

The second way is in a production context, but if you're using a small EC2 machine then it will likely fail due to out of memory errors. This is because the server will consume 2 GB of memory by default. To change this, modify the configuration file located here:

```
payara5/glassfish/domains/production/config/domain.xml
```

Find the part of the file where it says:

```
        <jvm-options>-Xmx2g</jvm-options>
        <jvm-options>-Xms2g</jvm-options>
```

“Xmx2g” means to start the server with 2 GB. Change the number to 1 to make it 1 GB, or change “g” to “m” to make it run in MB. Change both lines.

Starting in production mode:

```
./asadmin start-domain production
./asadmin deploy --contextroot / ~/bocks-ee_war.war
```

Run SDL Core and an HMI. If you're serving the HMI over nginx then nginx should be exposed on a different port than 8080, because Payara runs on 8080 by default. Be sure to modify the default policy table's app_policies object so that SDL Core is aware of where your app is:

```
"8675829": {
   "keep_context": false,
   "steal_focus": false,
   "priority": "NONE",
   "default_hmi": "NONE",
   "groups": ["Base-4"],
   "RequestType": [],
   "RequestSubType": [],
   "hybrid_app_preference": "CLOUD",
   "endpoint": "ws://$LOCAL_IP:8080/",
   "enabled": true,
   "auth_token": "no auth token",
   "cloud_transport_type": "WS",
   "nicknames": ["App1"]
},
```

Limitations and Issues

Follow the guidelines located here.

These are restrictions for what your logic should do in your EJB. Since this guide puts the whole logic of your app in an EJB, you should follow the restrictions specified above. You can still utilize other aspects of JavaEE to get around some of the limitations, but that will not be covered here.

Notable limitations include not starting or managing threads in your app, not reading from or writing to a file directly, not creating or deleting files or directories, not starting websocket connections yourself and not loading native libraries.

Memory usage increases with both redeployments and with many users connecting and disconnecting over time. The Payara server needs to be shut down to reset memory usage. This is the only post I could find online which had a similar issue.

When Payara or Glassfish is unable to handle the load, not only does your JavaEE app stop, but the server also stops.

Useful Information and Commands

Unzipping a jar file: `unzip my_jar.jar`

Packaging all items in your directory to a jar file: `jar cf my_jar.jar *`

When your app gets deployed on Payara on the default domain, it shows up in this directory:

```
payara5/glassfish/domains/domain1/applications
```

However, if the app happens to require loading assets from the same directory it originally resided in, that location changes once it is deployed, and is now located here:

```
payara5/glassfish/domains/domain1/config
```

In the PRODUCTION domain, the locations change to this:

```
payara5/glassfish/domains/production/applications
payara5/glassfish/domains/production/config
```

You can start Payara or Glassfish in different contexts, so be aware of how you started the server because it will change where you should move and modify files.

This is a load test ran using a sample app, connections closing after 5 seconds, and using a t3.small (2 GB memory)

CONNECTIONS	MEMORY %
0	24.8
300	25.2
700	25.4
1000	25.6
1500	26.1
2000	26.2
3000	26.2
4000	26.7
6000	26.8
8000	26.3
10000	27.0
15000	30.0
20000	33.5
25000	37.1
30000	39.8
40000	crashed

The test app could not handle more than 20 new connections per second.

Limit of simultaneous connections: 3562
After applying the max connections config: 7179

This page shows how to increase your max connection count for an EC2 machine
This page shows the settings to tune for a more effective Payara server

Hello SDL JavaEE

Introduction

In this guide we take you through the steps to get our sample project, Hello Sdl, running and connected to Sdl Core as well as showing up on the generic HMI.

Make sure that you follow the steps in Installation and Integration Basics sections to create a new JavaEE SDL project before continuing this section.

> NOTE
>
> The Hello Sdl JavaEE project includes samples for `SdlService` and `Main` classes that can be copied to your project.

Getting Started

We assume that you have SDL Core (We recommend Ubuntu 16.04) and an HMI set up prior to this point. Most people getting started with this tutorial will be using Sdl Core and our Generic HMI. If you don't want to set up a virtual machine for testing, we offer Manticore, which is a free service that allows you to test your apps in the cloud.

NOTE

Sdl Core and an HMI or Manticore are needed to run Hello Sdl and to ensure that it connects.

Configuring Core

To let Sdl Core connect to your app, first you will have to know the IP address of the machine that is running the Hello Sdl app. If you don't know what it is, running `ifconfig` in the terminal will usually let you see it for the interface you are connected with to your network.

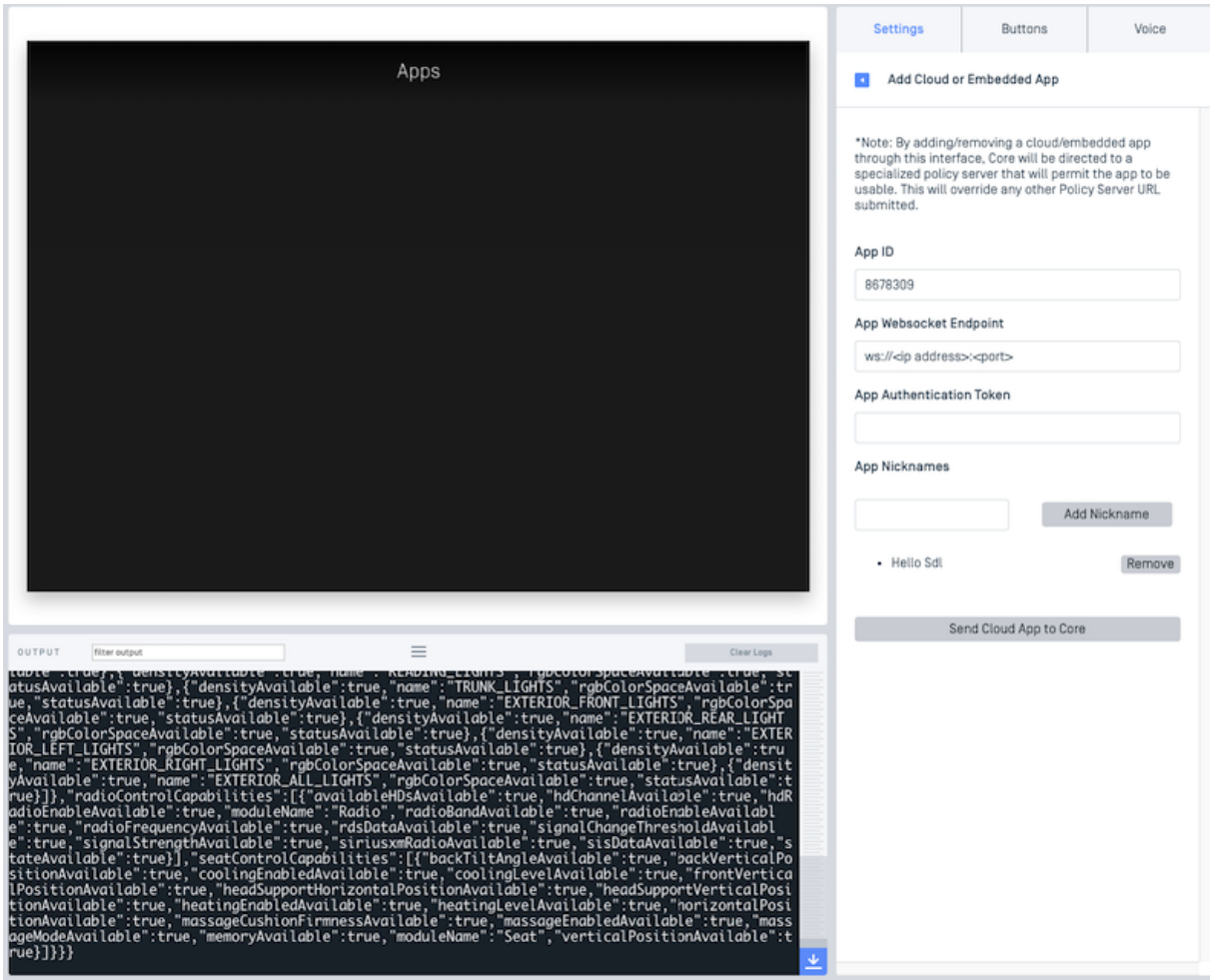
After getting the IP address, you will have to set App ID, App Websocket Endpoint, and App Nicknames in Sdl Core to let it know where your instance of Hello Sdl is running.

NOTE

The App Websocket Endpoint contains the IP Address and port as the following: `ws://<ip address>:<port>/`.

MANTICORE

If you are using Manticore, the app information can be easily set in the settings tab:



NOTE

Manticore needs to access you machine's IP address to be able to start a websocket connection with your cloud app. If you are hosting the cloud app on your local machine, you may need to do extra setup to make your machine publicly accessible. The other solution is to setup Core and HMI on your machine instead of using Manticore so Core can access your local IP address.

SDL CORE AND GENERIC HMI

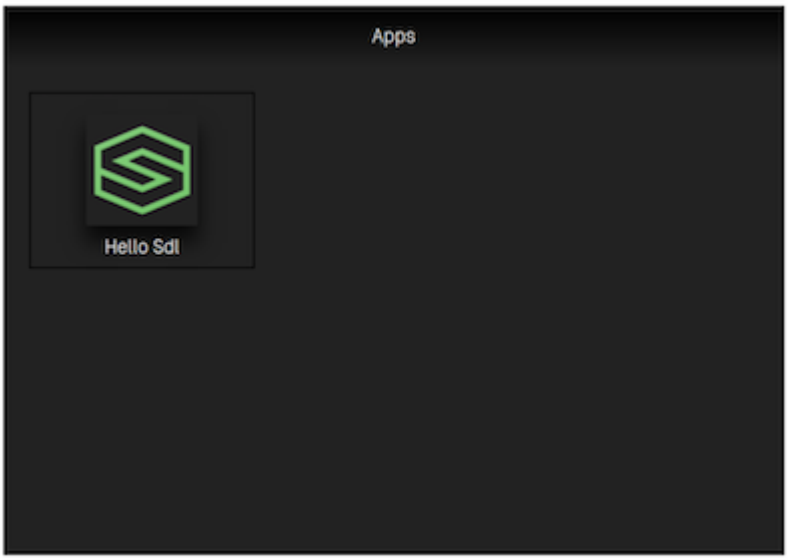
If you are using Sdl Core and Generic HMI, you will have to add a policy table entry as the following one to the existing entries:

```
"8678309": {
    "keep_context": false,
    "steal_focus": false,
    "priority": "NONE",
    "default_hmi": "NONE",
    "groups": ["Base-4"],
    "RequestType": [],
    "RequestSubType": [],
    "hybrid_app_preference": "CLOUD",
    "endpoint": "ws://<ip address>:<port>",
    "enabled": true,
    "auth_token": "",
    "cloud_transport_type": "WS",
    "nicknames": ["Hello Sdl"]
}
```

Fore more information about policy tables please visit Plicy Tables Guides.

> NOTE
>
> Don't forget to replace `ws://<ip address>:<port>` with your own IP address and app port. The port that is used in Hello Sdl App is `5432`. It can be changed to a different port by modifying the number in `Main.java` class.

Following this, you should see an application appears on HMI as in the following screenshot:

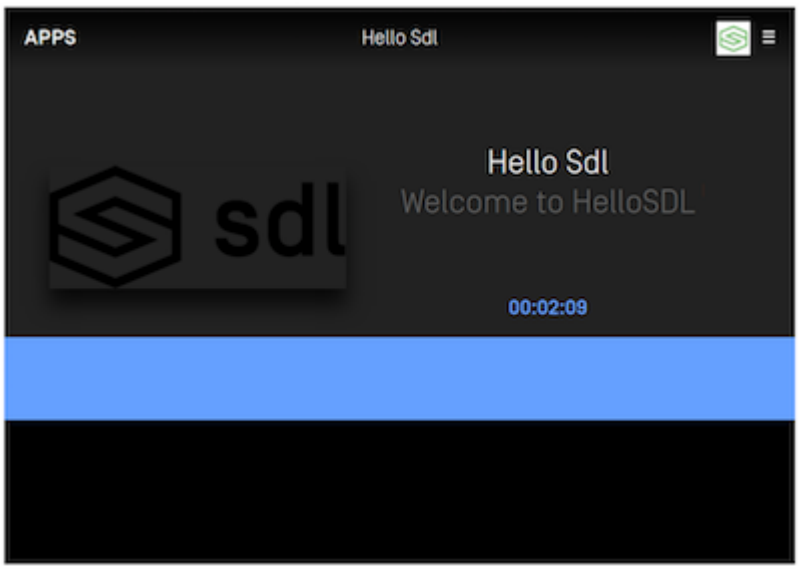


NOTE

Even though you see the app appears on HMI, you still cannot lunch the app at this point. You will have to run the Hello Sdl app from IntelliJ IDEA first as described next.

Running the App

After setting the app information in Sdl Core, you can run the project in IntelliJ IDEA. Hello Sdl should compile and launch on your your machine. After that, you can click on the Hello Sdl icon in the HMI.



This is the main screen of the Hello Sdl app. If you get to this point, the project is working.

At this point Hello Sdl has been compiled and is running properly! Continue reading through our guides to learn about all of the RPCs (Remote Procedure Calls) that can be made with the library.

Designing a User Interface

Designing for Different User Interfaces

Each car manufacturer may have different user interface style guidelines and slight variations in their templates(number of lines of text, buttons, and images supported). After the `SdlManager` has been started and is able to connect to and register on a module, the `SystemCapabilityManager` will have this capability information. The information stored in the `SystemCapabilityManager` can be used to aid in the layout and flow of your user interface.

Dynamic User Interface Capabilities

After the `SdlManager` has been successfully started the module will have sent any user interface information it has back to your app. This information includes the display type, the type of images supported, the number of text fields supported, the HMI display language, and a lot of other useful properties. This information can be accessed using the `SystemCapabilityManager`. The table below has a list of all possible properties available. Each property is optional, so you may not get information for all the parameters in the table.

PARAMETERS	DESCRIPTION	NOTES
sdlLanguage	The currently active voice-recognition and text-to-speech language on the head unit.	Check Language.java for more information
hmiDisplayLanguage	The currently active display language on the head unit.	Check Language.java for more information
displayCapabilities	Information about the head unit display. This includes information about available templates, whether or not graphics are supported, and a list of all text fields and the max number of characters allowed in each text field.	Check DisplayCapabilities.java for more information
buttonCapabilities	A list of available buttons and whether the buttons support long, short and up-down presses.	Check ButtonCapabilities.java for more information
softButtonCapabilities	A list of available soft buttons and whether the button support images. Also information about whether the button supports long, short and up-down presses.	Check SoftButtonCapabilities.java for more information
presetBankCapabilities	If returned, the platform supports custom on-screen presets.	Check PresetBankCapabilities.java for more information
speechCapabilities	Contains information about TTS capabilities on the SDL platform. Platforms may support text, SAPI phonemes, LH PLUS phonemes, pre-recorded speech, and silence.	Check SpeechCapabilities.java for more information

PARAMETERS	DESCRIPTION	NOTES
prerecordedSpeech	A list of pre-recorded sounds you can use in your app. Sounds may include a help, initial, listen, positive, or a negative jingle.	Check PrerecordedSpeech.java for more information
vrCapabilities	The voice-recognition capabilities of the connected SDL platform. The platform may be able to recognize spoken text in the current language.	Check VrCapabilities.java for more information
audioPassThruCapabilities	Describes the sampling rate, bits per sample, and audio types available.	Check AudioPassThruCapabilities.java for more information
supportedDiagModes	Specifies the white-list of supported diagnostic modes (0x00-0xFF) capable for DiagnosticMessage requests. If a mode outside this list is requested, it will be rejected.	List
hmiCapabilities	Returns whether or not the app can support built-in navigation and phone calls.	Check HMICapabilities.java for more information

Templates

Each car manufacturer supports a set of templates for the user interface. These templates determine the position and size of the text, images, and buttons on the screen. A list of supported templates is sent with `RegisterAppInterface` response and can be accessed using the `SystemCapabilityManager`.

To change a template at any time, send a `SetDisplayLayout` RPC to the SDL Core. If you want to ensure that the new template is used, wait for a response from the SDL Core before sending any more user interface RPCs.

```
SetDisplayLayout setDisplayLayoutRequest = new SetDisplayLayout();
setDisplayLayoutRequest.setDisplayLayout(PredefinedLayout.
GRAPHIC_WITH_TEXT.toString());
setDisplayLayoutRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(((SetDisplayLayoutResponse) response).getSuccess()){
            Log.i("SdlService", "Display layout set successfully.");
            // Proceed with more user interface RPCs
        }else{
            Log.i("SdlService", "Display layout request rejected.");
        }
    }
});

sdlManager.sendRPC(setDisplayLayoutRequest);
```

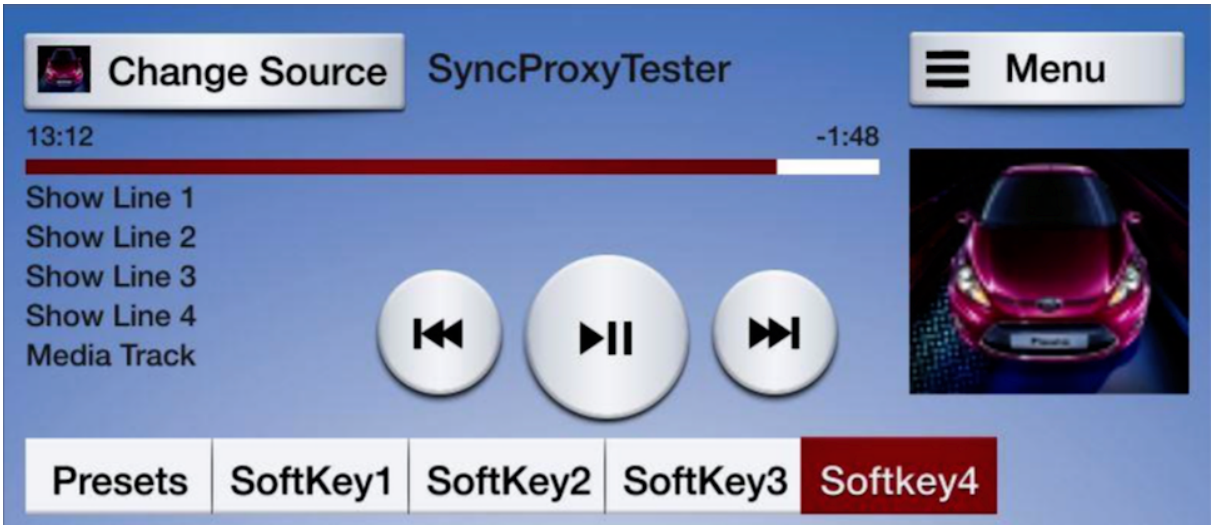
Available Templates

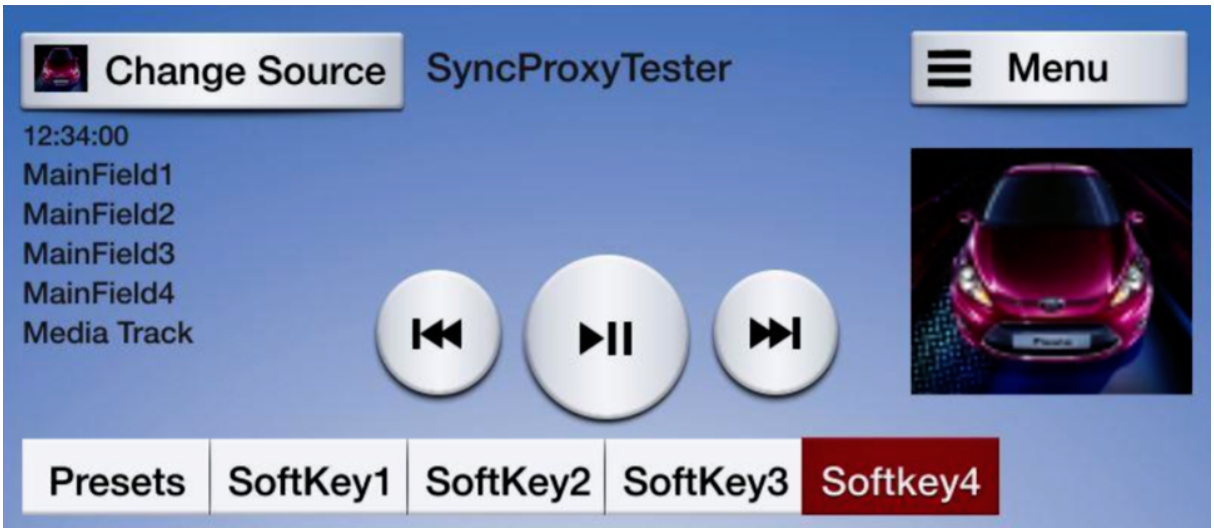
There are fifteen standard templates to choose from, however some head units may only support a subset of these templates. Please check the `DisplayCapabilities` object returned by `SystemCapabilityManager` for the supported templates. The following examples show how templates will appear on the generic head unit.

> NOTE
>
> You will automatically be assigned the media template if you set your configuration app type as `MEDIA`.

1. MEDIA - WITH AND WITHOUT PROGRESS BAR

FORD HMI





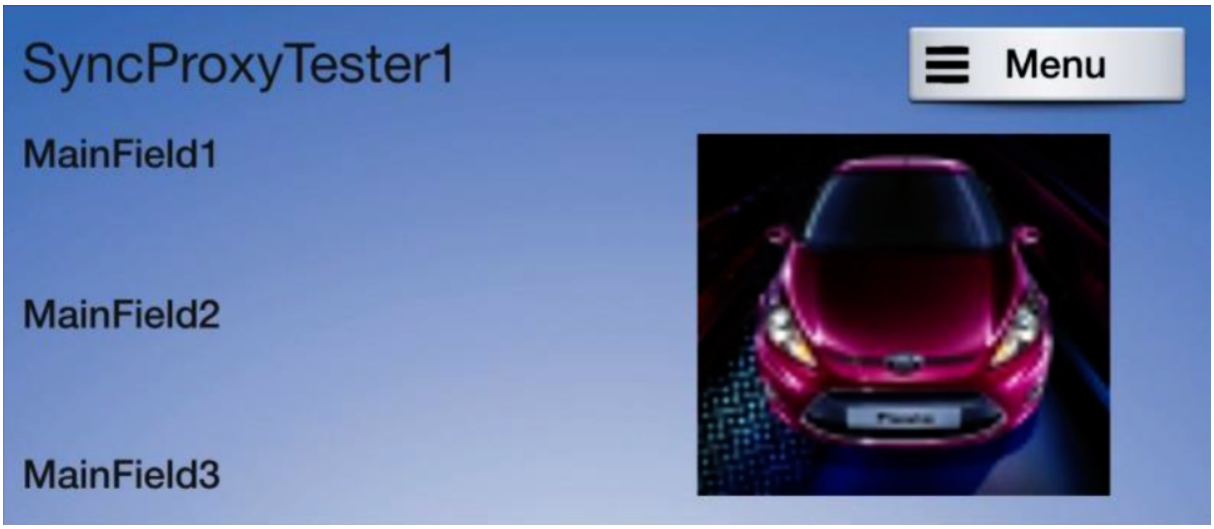
2. NON-MEDIA - WITH AND WITHOUT SOFT BUTTONS

3. GRAPHIC_WITH_TEXT

FORD HMI

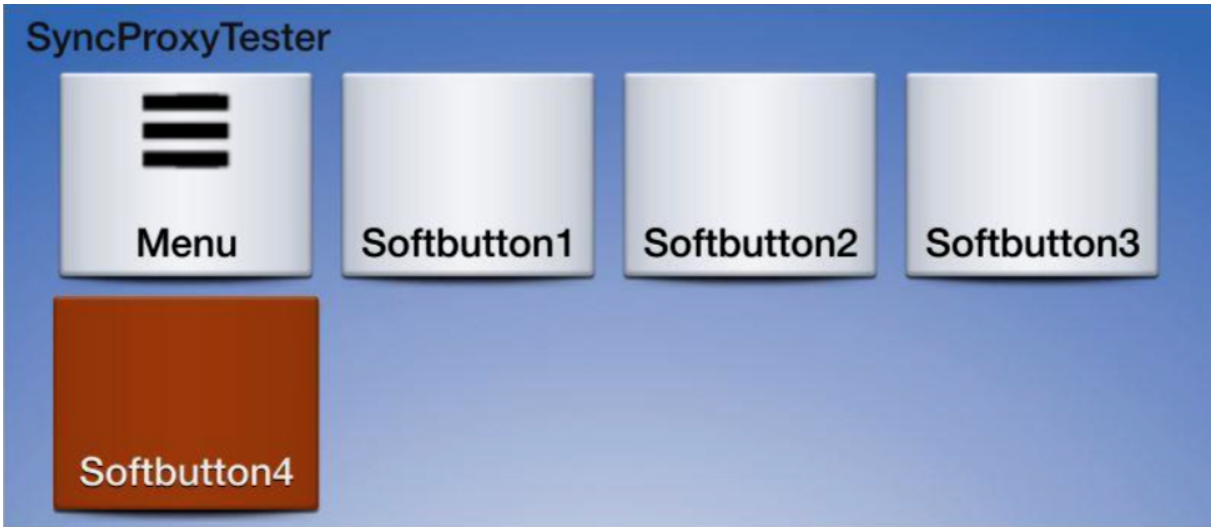
4. TEXT_WITH_GRAPHIC

FORD HMI



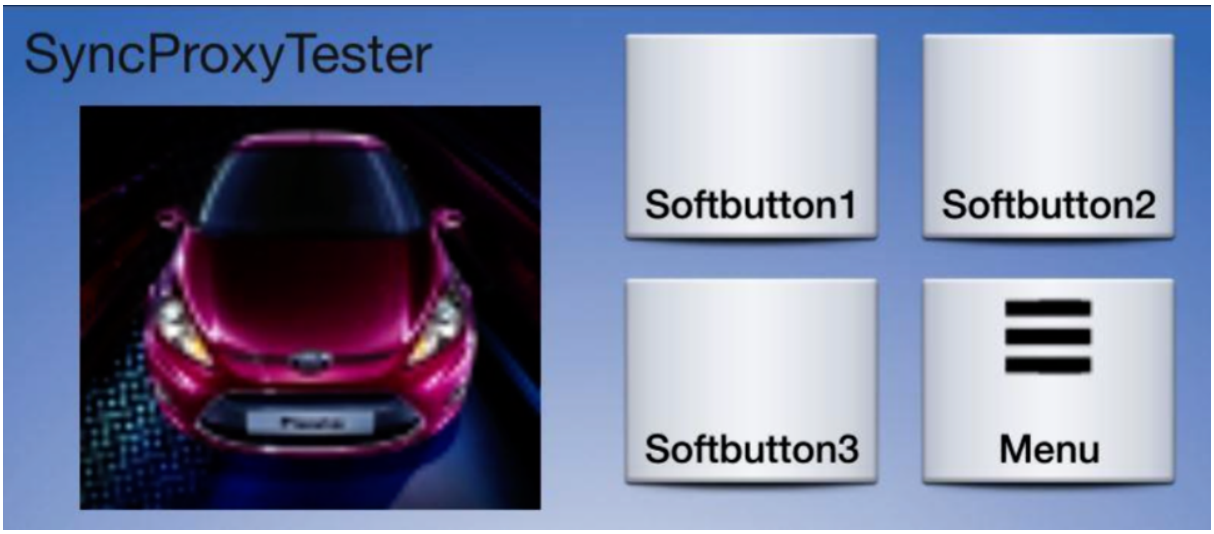
5. TILES_ONLY

FORD HMI



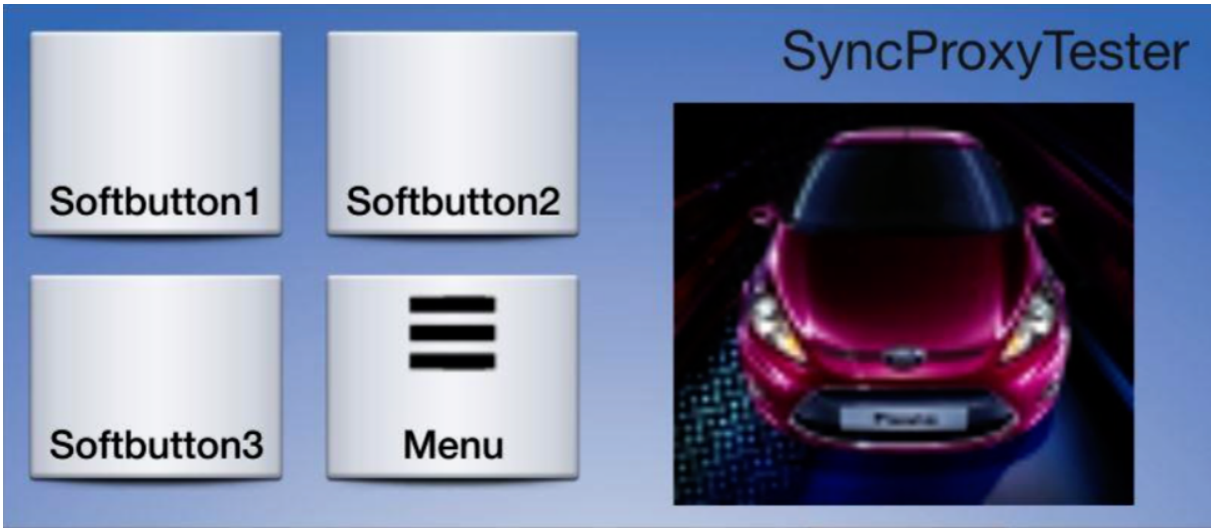
6. GRAPHIC_WITH_TILES

FORD HMI



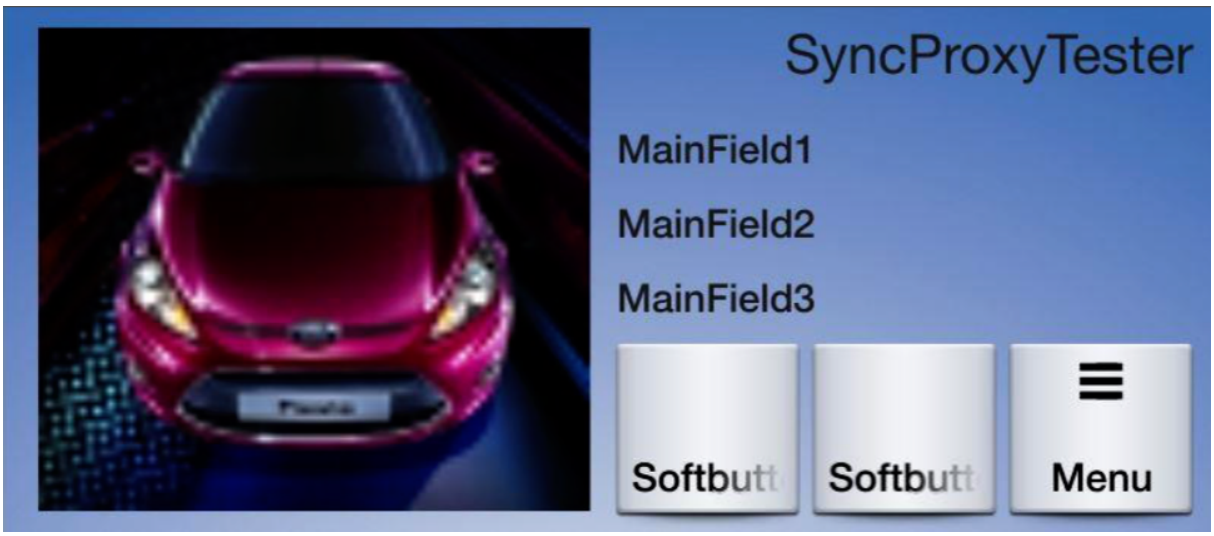
7. TILES_WITH_GRAPHIC

FORD HMI



8. GRAPHIC_WITH_TEXT_AND_SOFTBUTTONS

FORD HMI

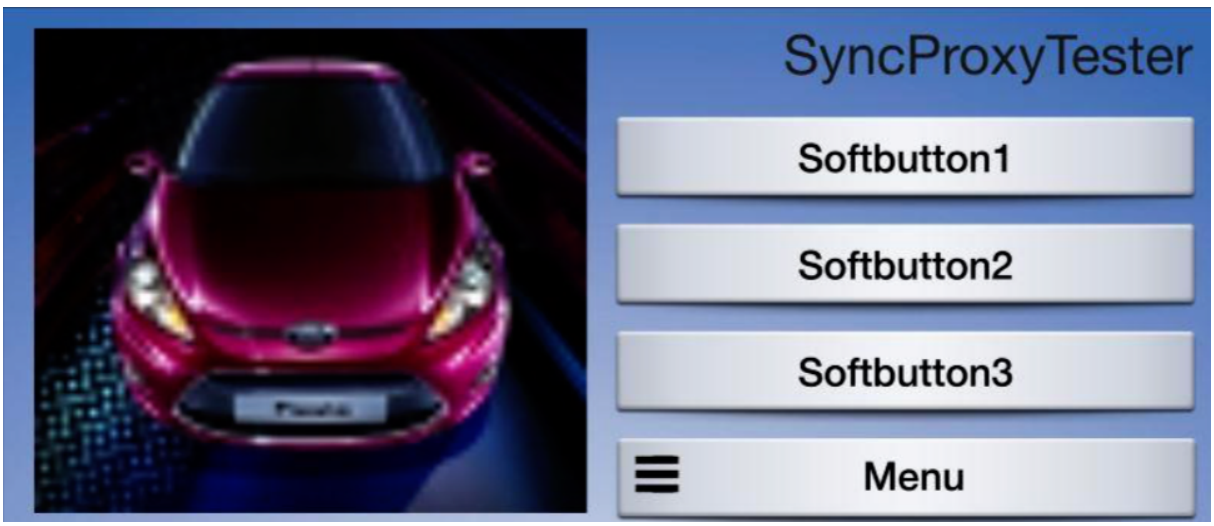


9. TEXT_AND_SOFTBUTTONS_WITH_GRAPHIC

FORD HMI

10. GRAPHIC_WITH_TEXTBUTTONS

FORD HMI

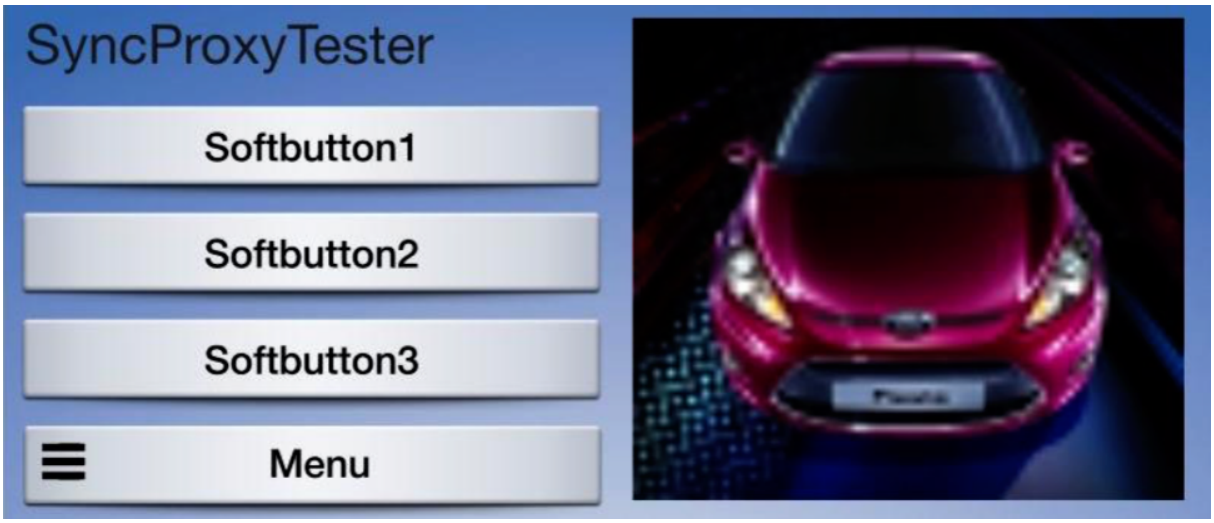


11. DOUBLE_GRAPHIC_SOFTBUTTONS

FORD HMI

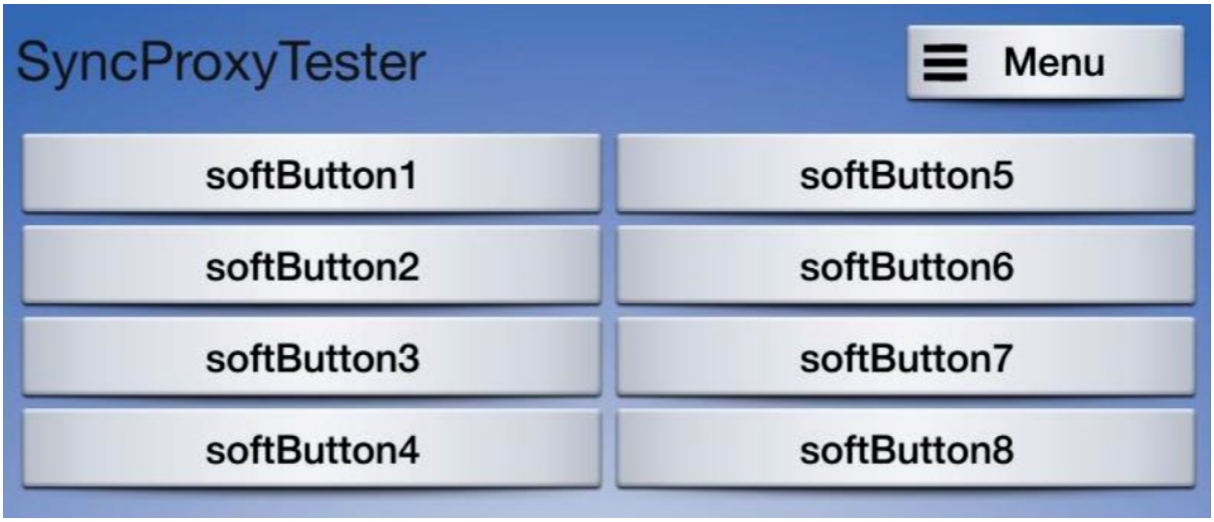
12. TEXTBUTTONS_WITH_GRAPHIC

FORD HMI



13. TEXTBUTTONS_ONLY

FORD HMI



14. LARGE_GRAPHIC_WITH_SOFTBUTTONS

FORD HMI

15. LARGE_GRAPHIC_ONLY

Text, Images, and Buttons

All text, images, and soft buttons on the HMI screen must be sent as part of a `Show` RPC. The `ScreenManager` will take care of creating and sending the `Show` request for text, images, and soft buttons so developers don't have to worry about that. Subscribe buttons need to be sent as part of a `SubscribeButton` RPC.

Text

A maximum of four lines of text can be sent to the module, however, some templates may only support 1, 2, or 3 lines of text. The `ScreenManager` well automatically handle the combining of lines based on how many lines are available and which fields the developer has set. For example, if all four lines of text are set in the `ScreenManager`, but the template only supports three lines of text, then the `ScreenManager` will hyphenate the third and fourth line and display them in one line.

```
//Start the UI updates
sdlManager.getScreenManager().beginTransaction();

sdlManager.getScreenManager().setTextField1("Hello, this is
MainField1.");
sdlManager.getScreenManager().setTextField2("Hello, this is
MainField2.");
sdlManager.getScreenManager().setTextField3("Hello, this is
MainField3.");
sdlManager.getScreenManager().setTextField4("Hello, this is
MainField4.");

//Commit the UI updates
sdlManager.getScreenManager().commit(new CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        Log.i(TAG, "ScreenManager update complete: " + success);
    }
});
```

NOTE

If you don't use `beginTransaction()` and `commit()`, `ScreenManager` will still update the text fields correctly, however, it will send a `Show` request every time a text field is set. It is always recommended to use transactions if you have a batch of `ScreenManager` updates. Transactions will let the `ScreenManager` queue the updates and send them all at once in one `Show` RPC when `commit()` is called resulting in better performance and UI stability.

Images

The position and size of images on the screen is determined by the currently set template. `ScreenManager` will handle uploading images and sending the `Show` RPC to display the images when they are ready.

> NOTE
>
> Some head units may only support certain images or possibly none at all. Please consult the `getGraphicSupported()` method in the `DisplayCapabilities` using the `SystemCapabilityManager`.

SHOW THE IMAGE ON A HEAD UNIT

To display an image in the head unit, you have to create an `SdlArtwork` object and set it using the `ScreenManager`. `SdlArtwork` supports both dynamic and static images.
To use it with dynamic images, you can use the constructor that takes multiple arguments. The `fileName` property should be set to the name that you want to use to save the file in the head unit. The `FileType` should be set to the correct type of image that is being sent, in the example it is set to `FileType.GRAPHIC_JPEG` because the image has JPEG format. The `id` is set to the Android resource id of the image that you want to use. The `persistentFile` is a boolean that represents whether you want the file to persist between sessions.

```
SdlArtwork sdlArtwork = new SdlArtwork("appImage.jpeg", FileType.GRAPHIC_JPEG, R.drawable.appImage, true);
sdlManager.getScreenManager().setPrimaryGraphic(sdlArtwork);
```

To use `SdlArtwork` with static images, you can use the constructor that takes the static icon name as the only argument as in the following sample:

```
SdlArtwork sdlArtwork = new SdlArtwork(StaticIconName.ALBUM);
sdlManager.getScreenManager().setPrimaryGraphic(sdlArtwork);
```

Soft & Subscribe Buttons

Buttons pushed by an app to the module's HMI screen are referred to as soft buttons to distinguish them from hard or preloaded buttons, which are either physical buttons on the head unit or buttons that exist on the module at all times. Don’t confuse soft buttons with subscribe buttons, which are buttons that can detect user selection on hard buttons (or built-in soft buttons).

SOFT BUTTONS

Soft buttons can be created with text, images or both text and images. The location, size, and number of soft buttons visible on the screen depends on the template. A `SoftButtonObject` can have multiple `SoftButtonState` objects; each state can have text, image, or both. Buttons can be transitioned from one state to another at runtime.

```
SoftButtonState softButtonState1 = new SoftButtonState("state1",
"state1", new SdlArtwork("state1.png", FileType.GRAPHIC_PNG, R.
drawable.state1, true));

SoftButtonState softButtonState2 = new SoftButtonState("state2",
"state2", new SdlArtwork("state2.png", FileType.GRAPHIC_PNG, R.
drawable.state2, true));

List<SoftButtonState> softButtonStates = Arrays.asList(
softButtonState1, softButtonState2);
SoftButtonObject softButtonObject = new SoftButtonObject("object",
softButtonStates, softButtonState1.getName(), null);

//We will add a listener for events in the next example here

sdlManager.getScreenManager().setSoftButtonObjects(Collections.
singletonList(softButtonObject));
```

RECEIVING SOFT BUTTONS EVENTS

Once you have created soft buttons, you will likely want to know when events happen to those buttons. These events come through two callbacks `onEvent` and `onPress`. Depending which type of event you're looking for you can use that type of callback.

```
softButtonObject.setOnEventListener(new SoftButtonObject.
OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {
        softButtonObject.transitionToNextState();
    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject,
OnButtonEvent onButtonEvent) {

    }
});
```

SUBSCRIBE BUTTONS

Subscribe buttons are used to detect changes to hard or preloaded buttons. You can subscribe to the following hard buttons:

BUTTON	TEMPLATE	BUTTON TYPE
Ok (play/pause)	media template only	soft button and hard button
Seek left	media template only	soft button and hard button
Seek right	media template only	soft button and hard button
Tune up	media template only	hard button
Tune down	media template only	hard button
Preset 0-9	any template	hard button
Search	any template	hard button
Custom	any template	hard button

Audio buttons like the OK (i.e. the `play/pause` button), seek left, seek right, tune up, and tune down buttons can only be used with a media template. The OK, seek left, and seek right buttons will also show up on the screen in a predefined location dictated by the media template on touchscreens. The app will be notified when the user selects the subscribe button on the screen or when the user manipulates the corresponding hard button.

You can subscribe to buttons using the `SubscribeButton` RPC.

```
SubscribeButton subscribeButtonRequest = new SubscribeButton();
subscribeButtonRequest.setButtonName(ButtonName.SEEKRIGHT);
sdlManager.sendRPC(subscribeButtonRequest);
```

> NOTE
>
> It is not required to manually subscribe to soft buttons. When soft buttons are added, your app will automatically be subscribed for their events.

RECEIVING SUBSCRIBE BUTTONS EVENTS

When you want to subscribe to buttons, you will be subscribing to events that happen to those buttons. These events come through two callbacks `OnButtonEvent` and `OnButtonPress`. Depending which type of event you're looking for you can use that type of callback. The `ButtonName` enum refers to which button the event happened to.

> NOTE
>
> Some templates will not show a preloaded button until an app subscribes to it. After an app subscribes to the events of that button, it will appear.

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_EVENT, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress)
notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
            default:
                break;
        }
    }
});


sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_PRESS, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress)
notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
            default:
                break;
        }
    }
});
```

NOTE

The app should subscribe to button events before sending the `SubscribeButton` request to make sure that it doesn't miss any button events.

Menus

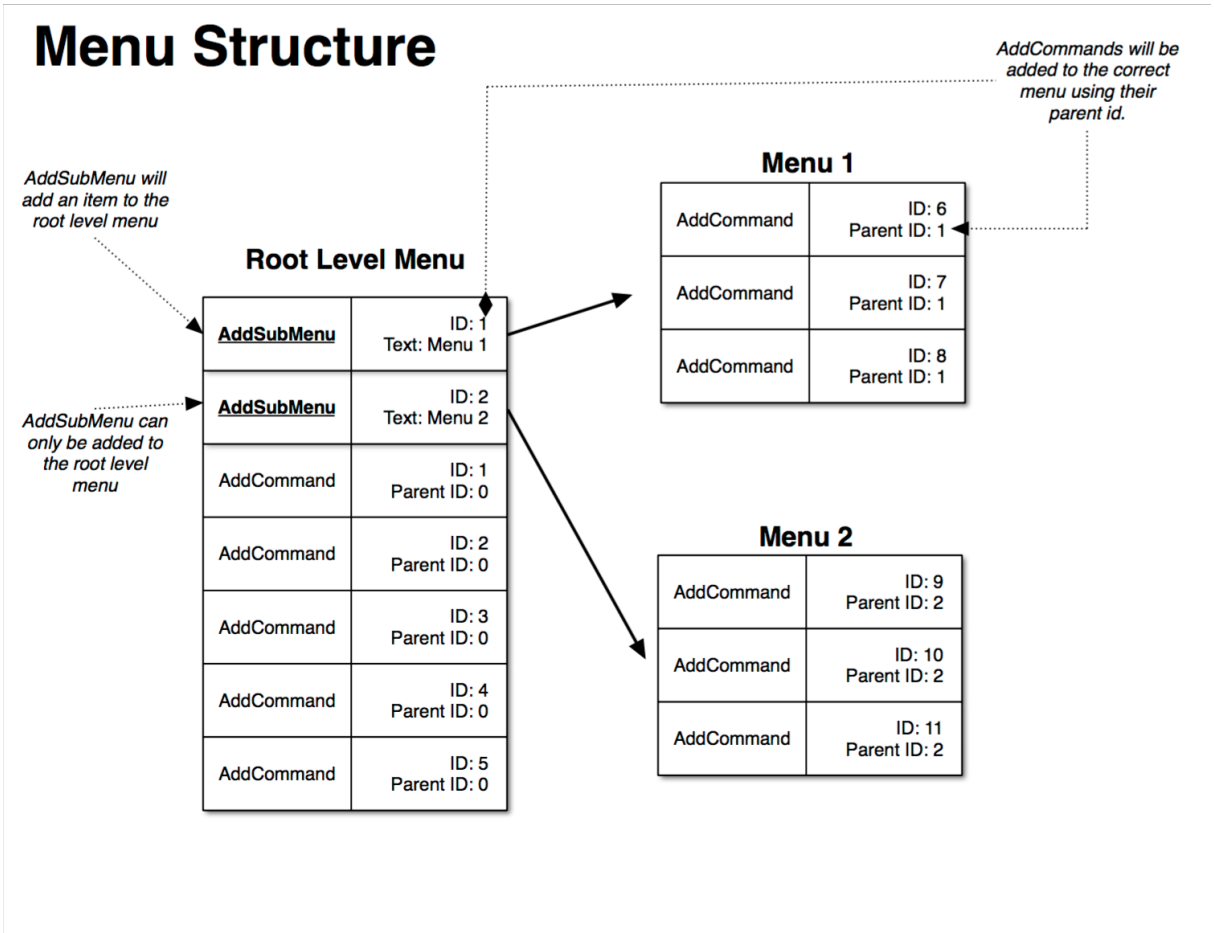
You have two different options when creating menus. One is to simply add items to the default menu available in every template. The other is to create a custom menu that pops up when needed.

Default Menu

Every template has a default menu button. The position of this button varies between templates, and can not be removed from the template. The default menu is initially empty except for an "Exit Your App Name" button. Items can be

added to the menu at the root level or to a submenu. It is important to note that a submenu can only be one level deep.

Menu Structure



ADD MENU ITEMS

The `AddCommand` RPC can be used to add items to the root menu or to a submenu. Each `AddCommand` RPC must be sent with a unique id, a voice-recognition command, and a set of menu parameters. The menu parameters include the menu name, the position of the item in the menu, and the id of the menu item's parent. If the menu item is being added to the root menu, then the

parent id is 0. If it is being added to a submenu, then the parent id is the submenu's id.

```
// Create the menu parameters
// The parent id is 0 if adding to the root menu
// If adding to a submenu, the parent id is the submenu's id
MenuParams menuParams = new MenuParams();
menuParams.setParentID(0);
menuParams.setPosition(0);
menuParams.setMenuName("Options");

AddCommand addCommand = new AddCommand();
addCommand.setCmdID(0); // Ensure this is unique
addCommand.setMenuParams(menuParams);  // Set the menu
parameters

sdlManager.sendRPC(addCommand);
```

ADD A SUBMENU

To create a submenu, first send an `AddSubMenu` RPC. When a response is received from the SDL Core, check if the submenu was added successfully. If it was, send an `AddCommand` RPC for each item in the submenu.

```
int unique_id = 313;

AddSubMenu addSubMenu = new AddSubMenu();
addSubMenu.setPosition(0);
addSubMenu.setMenuID(unique_id);
addSubMenu.setMenuName("SubMenu");
addSubMenu.setOnRPCResponseListener(new OnRPCResponseListener
() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(((AddSubMenuResponse) response).getSuccess()){
            // The submenu was created successfully, start adding the
submenu items
            // Use unique_id
        }else{
            Log.i("SdlService", "AddSubMenu request rejected.");
        }
    }
});
```

DELETE MENU ITEMS

Use the cmdID of the menu item to tell the SDL Core which item to delete using the `DeleteCommand` RPC.

```
int cmdID_to_delete = 1;

DeleteCommand deleteCommand = new DeleteCommand();
deleteCommand.setCmdID(cmdID_to_delete);

sdlManager.sendRPC(deleteCommand);
```

DELETE SUBMENUS

Use the menuID to tell the SDLCore which item to delete using the `DeleteSubMenu` RPC.

```
DeleteSubMenu deleteSubMenu = new DeleteSubMenu();
deleteSubMenu.setMenuID(submenuID_to_delete); // Replace with submenu ID to delete
```

Custom Menus

Custom menus, called **perform interactions**, are one level deep, however, you can create submenus by triggering another perform interaction when the user selects a row in a menu. Perform interactions can be set up to recognize speech, so a user can select an item in the menu by speaking their preference rather than physically selecting the item.

Perform interactions are created by sending two different RPCs. First a `CreateInteractionChoiceSet` RPC must be sent. This RPC sends a list of items that will show up in the menu. When the request has been registered successfully, then a `PerformInteraction` RPC is sent. The `PerformInteraction` RPC sends the formatting requirements, the voice-recognition commands, and a timeout command.

CREATE A SET OF CUSTOM MENU ITEMS

Each menu item choice defined in `Choice` should be assigned a unique id. The choice set in `CreateInteractionChoiceSet` should also have its own unique id.

```
CreateInteractionChoiceSet choiceSet = new
CreateInteractionChoiceSet();

Choice choice = new Choice();
choice.setChoiceID(uniqueChoiceID);
choice.setMenuName("ChoiceA");
choice.setVrCommands(Arrays.asList("ChoiceA"));

List<Choice> choiceList = new ArrayList<>();
choiceList.add(choice);

choiceSet.setChoiceSet(choiceList);
choiceSet.setInteractionChoiceSetID(uniqueIntChoiceSetID);
choiceSet.setOnRPCResponseListener(new OnRPCResponseListener() {
@Override
public void onResponse(int correlationId, RPCResponse response) {
    if(((CreateInteractionChoiceSetResponse) response).getSuccess()){
        // The request was successful, now send the
SDLPerformInteraction RPC
    }else{
        // The request was unsuccessful
    }
}
});

sdlManager.sendRPC(choiceSet);
```

FORMAT THE SET OF CUSTOM MENU ITEMS

Once the set of menu items has been sent to SDL Core, send a `PerformInteraction` RPC to get the items to show up on the HMI screen.

```
List<Integer> interactionChoiceSetIDList = new ArrayList<>();
interactionChoiceSetIDList.add(uniqueIntChoiceSetID);

PerformInteraction performInteraction = new PerformInteraction();
performInteraction.setInitialText("Initial text.");
performInteraction.setInteractionChoiceSetIDList(
interactionChoiceSetIDList);
```

INTERACTION MODE

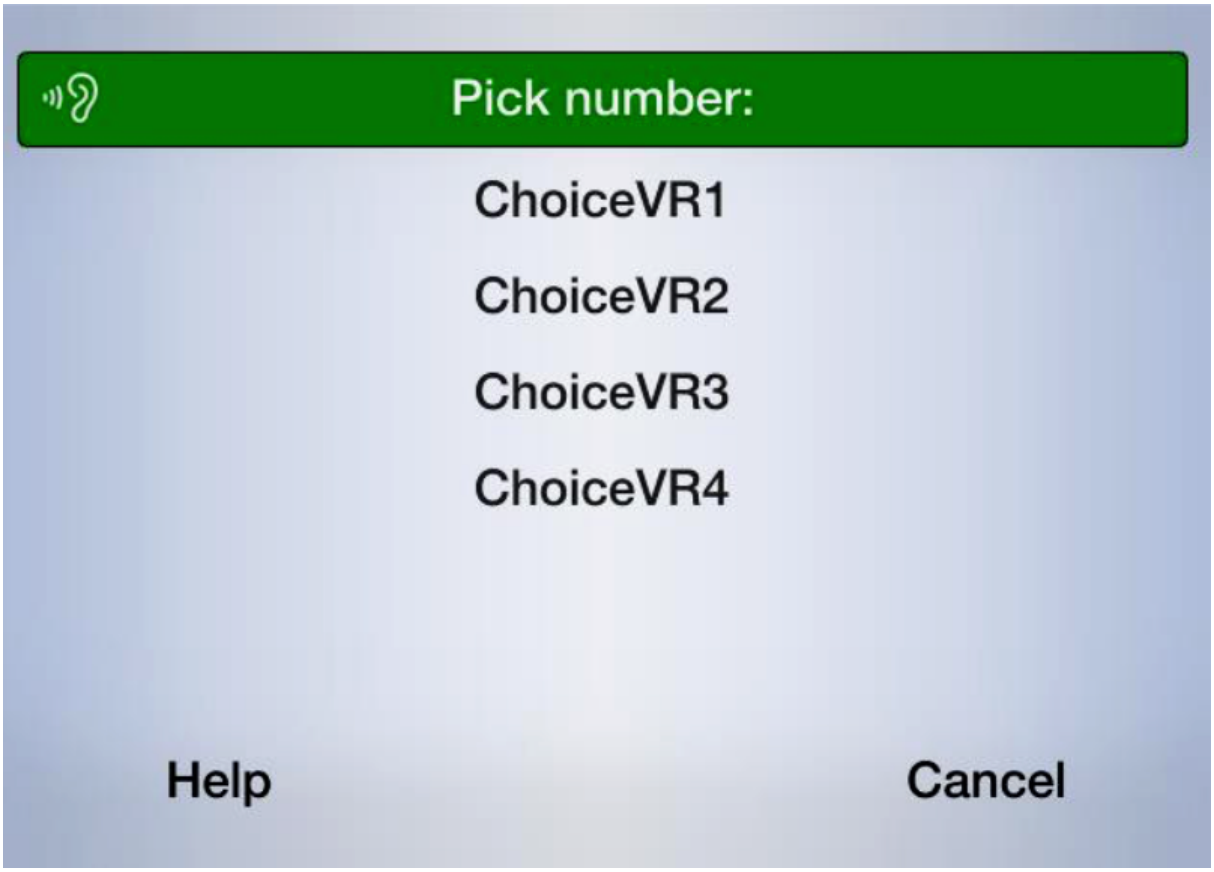
The interaction mode specifies the way the user is prompted to make a section and the way in which the user's selection is recorded.

INTERACTION MODE	DESCRIPTION
Manual only	Interactions occur only through the display
VR only	Interactions occur only through text-to-speech and voice recognition
Both	Interactions can occur both manually or through VR

```
performInteraction.setInteractionMode(InteractionMode.MANUAL_ONLY
);
```

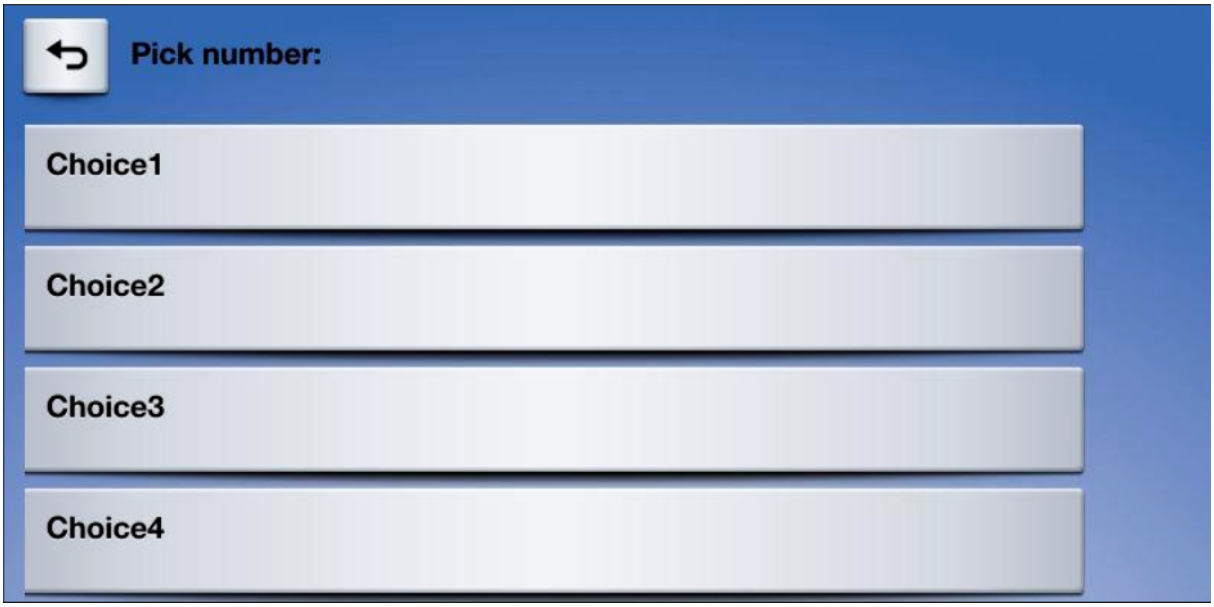
VR INTERACTION MODE

FORD HMI



MANUAL INTERACTION MODE

FORD HMI



INTERACTION LAYOUT

The items in the perform interaction can be shown as a grid of buttons (with optional images) or as a list of choices.

LAYOUT MODE	FORMATTING DESCRIPTION
Icon only	A grid of buttons with images
Icon with search	A grid of buttons with images along with a search field in the HMI
List only	A vertical list of text
List with search	A vertical list of text with a search field in the HMI
Keyboard	A keyboard shows up immediately in the HMI

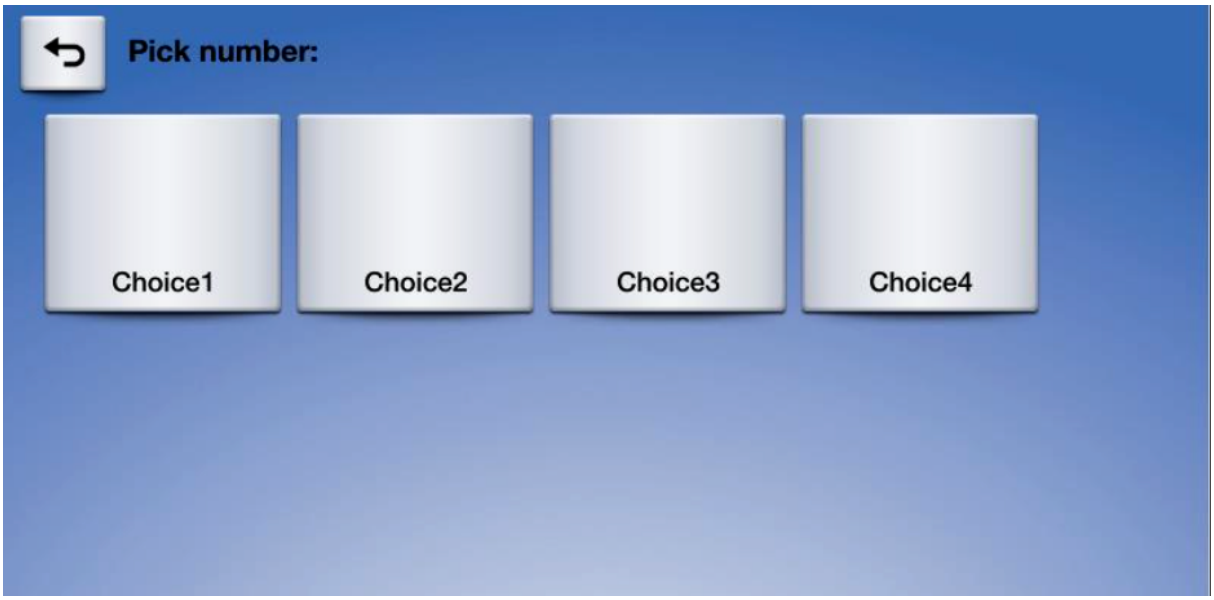
NOTE

Keyboard is currently only supported for the navigation app type.

```
performInteraction.setInteractionLayout(LayoutMode.LIST_ONLY);
```

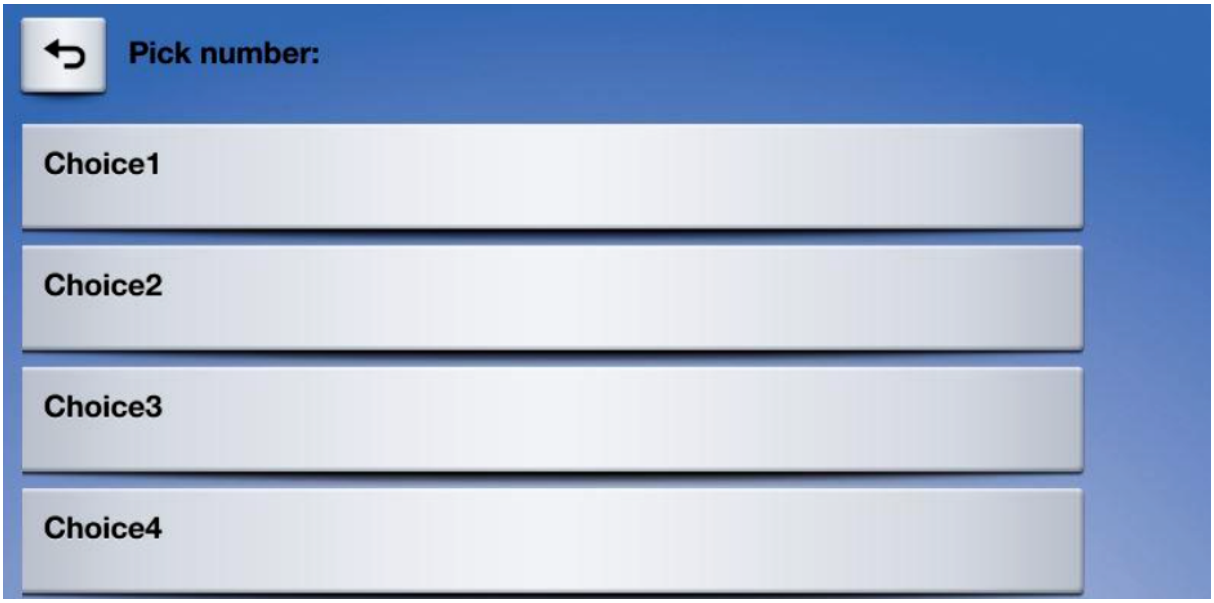
ICON ONLY INTERACTION LAYOUT

FORD HMI



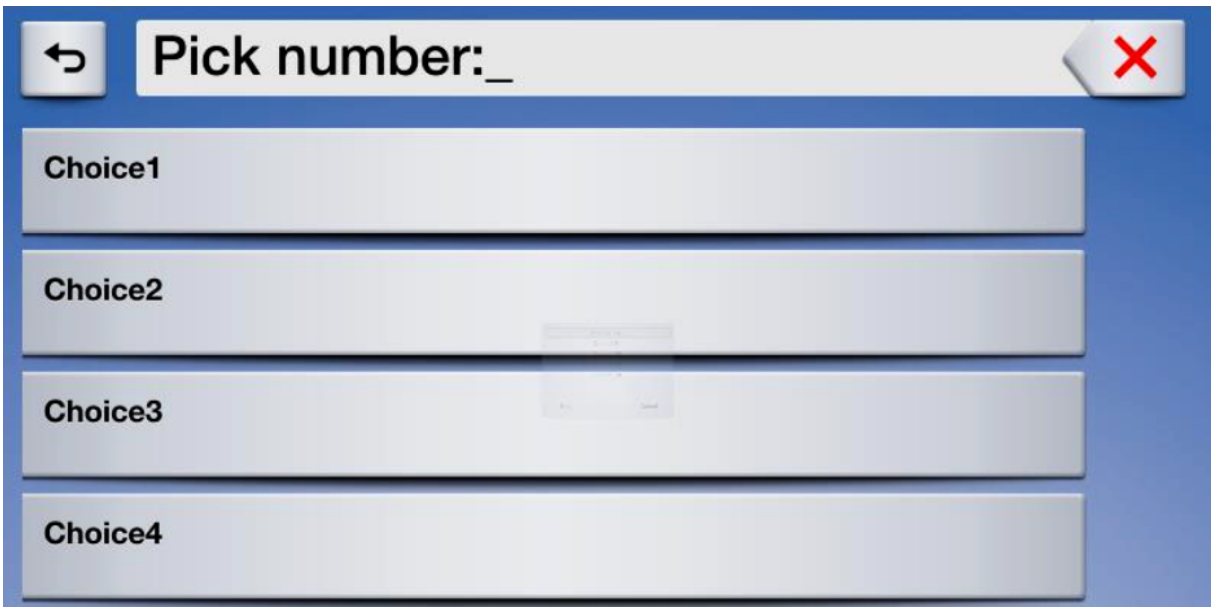
LIST ONLY INTERACTION LAYOUT

FORD HMI



LIST WITH SEARCH INTERACTION LAYOUT

FORD HMI



TEXT-TO-SPEECH (TTS)

A text-to-speech chunk is a text phrase or prerecorded sound that will be spoken by the head unit. The text parameter specifies the text to be spoken or the name of the pre-recorded sound. Use the type parameter to define the type of information in the text parameter. The `PerformInteraction` request can have a initial, timeout, and a help prompt.

```
performInteraction.setInitialPrompt(
    TTSChunkFactory.createSimpleTTSChunks("Hello, welcome."));
```

TIMEOUT

The timeout parameter defines the amount of time the menu will appear on the screen before the menu is dismissed automatically by the HMI.

```
performInteraction.setTimeout(30000); // 30 seconds
```

SEND THE REQUEST

```
performInteraction.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        PerformInteractionResponse piResponse = (
PerformInteractionResponse) response;
        if(piResponse.getSuccess()){
            // Successful request
            if(piResponse.getResultCode().equals(Result.TIMED_OUT)){
                // Interaction timed out without user input
            }else if(piResponse.getResultCode().equals(Result.SUCCESS)){
                Integer userChoice = piResponse.getChoiceID();
            }
        }else{
            // Unsuccessful request
        }
    }
});

sdlManager.sendRPC(performInteraction);
```

DELETE THE CUSTOM MENU

If the information in the menu is dynamic, then the old interaction choice set needs to be deleted with a `DeleteInteractionChoiceSet` RPC before the new information can be added to the menu. Use the interaction choice set id to delete the menu.

```
DeleteInteractionChoiceSet deleteInteractionChoiceSet = new
DeleteInteractionChoiceSet();
deleteInteractionChoiceSet.setInteractionChoiceSetID(
interactionChoiceSetID_to_delete); // Replace with interaction choice
set to delete

sdlManager.sendRPC(deleteInteractionChoiceSet);
```

Alerts

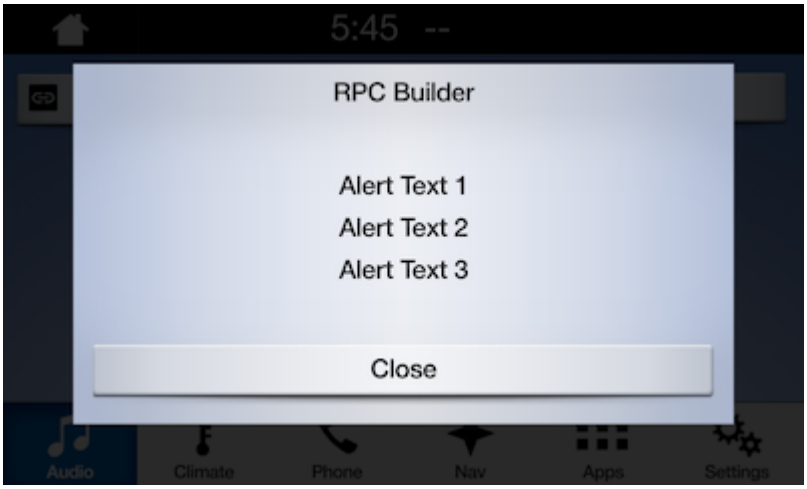
An alert is a pop-up window with some lines of text and optional soft buttons. When an alert is activated, it will abort any SDL operation that is in-progress, except the already-in-progress alert. If an alert is issued while another alert is still in progress, the newest alert will simply be ignored.

Alert UI

Depending the platform, an alert can have up to three lines of text, a progress indicator (e.g. a spinning wheel or hourglass), and up to four soft buttons.

ALERT WITHOUT SOFT BUTTONS

FORD HMI



ALERT WITH SOFT BUTTONS

FORD HMI

Alert TTS

The alert can also be formatted to speak a prompt when the alert appears on the screen. Do this by setting the `ttsChunks` parameter. To play the alert tone before the text-to-speech is spoken, set `playTone` to `true`.

Example

```
Alert alert = new Alert();
alert.setAlertText1("Alert Text 1");
alert.setAlertText2("Alert Text 2");
alert.setAlertText3("Alert Text 3");

// Maximum time alert appears before being dismissed
// Timeouts are must be between 3-10 seconds
// Timeouts may not work when soft buttons are also used in the alert
alert.setDuration(5000);

// A progress indicator (e.g. spinning wheel or hourglass)
// Not all head units support the progress indicator
alert.setProgressIndicator(true);

//Text to speech
alert.setTtsChunks(TTS_list); // TTS_list populated elsewhere

// Special tone played before the tts is spoken
alert.setPlayTone(true);

// Soft buttons
alert.setSoftButtons(softButtons); // softButtons populated elsewhere

// Send alert
sdlManager.sendRPC(alert);
```

Dismissing the Alert

The alert will persist on the screen until the timeout has elapsed, or the user dismisses the alert by selecting a button. There is no way to dismiss the alert programmatically other than to set the timeout length.

Uploading Files and Graphics

Graphics allow you to better customize what you would like to have your users see and provide a better User Interface.

When developing an application using SmartDeviceLink, two things must always be remembered when using graphics:

1. You may be connected to a head unit that does not display graphics.
2. You must upload them from your mobile device to Core before using them.

> NOTE
>
> Many of these features will be handled for you automatically by the `ScreenManager` and other managers. This guide will be for using the `FileManager` directly through `SdlManager`

Detecting if Graphics are Supported

Being able to know if graphics are supported is a very important feature of your application, as this avoids you uploading unnecessary images to the head unit. In order to see if graphics are supported, use the `getCapability()` method of a valid `SystemCapabilityManager` obtained from `sdlManager.getSystemCapabilityManager()` to find out the display capabilities of the head unit.

```
sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.DISPLAY, new OnSystemCapabilityListener(){

  @Override
  public void onCapabilityRetrieved(Object capability){
    DisplayCapabilities dispCapability = (DisplayCapabilities) capability;
  }

  @Override
  public void onError(String info){
    Log.i(TAG, "Capability could not be retrieved: "+ info);
  }
});
```

SDL File and SDL Artwork

SDL files and artwork are uploaded through the `FileManager`. This is accomplished with `SdlFile` and `SdlArtwork` objects. The `FileManager` helps streamline the file management workflow within SDL. `SdlArtwork` is an extension of `SdlFile` that pertains only to graphic specific file types, and its use case is similar. For the rest of this document, `SdlFile` will be described, but everything also applies to `SdlArtwork`.

CREATION

The first step in uploading files to the connected module is creating an instance of `SdlFile`. There are a few different constructors that can be used based on the source of the file. The following can be used to instantiate `SdlFile`:

A RESOURCE ID

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, int id
, boolean persistentFile)
```

A URI

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, Uri
uri, boolean persistentFile)
```

A BYTE ARRAY

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, byte
[] data, boolean persistentFile)
```

Uploading a File

Uploading a file with the `FileManager` is a simple process. With an instantiated `SdlManager`,
you can simply call:

```
sdlManager.getFileManager().uploadFile(sdlFile, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

UPLOADING MULTIPLE FILES

Sometimes you need to upload more than one file. We've got you covered.
Simply create a `List<SdlFile>` object, add your files, and then call:

```
sdlManager.getFileManager().uploadFiles(sdlFileList, new
MultipleFileCompletionListener() {
    @Override
    public void onComplete(Map<String, String> errors) {

    }
});
```

UPLOADING ARTWORK

As mentioned before, the behavior of `SdlFile` and `SdlArtwork` are the same. But to help separate code, we have also included `uploadArtwork` and `uploadArtworks` methods to the `FileManager` that work the same as their `SdlFile` counterparts shown above.

File Naming

The file name can only consist of letters (a-Z) and numbers (0-9), otherwise the SDL Core may fail to find the uploaded file (even if it was uploaded successfully).

File Persistance

`SdlFile` supports uploading persistent images, i.e. images that do not become deleted when your application disconnects. Persistence should be used for images relating to your UI like your app icon, and not for dynamic aspects, such as Album Artwork.

NOTE

Be aware that persistence will not work if space on the head unit is limited. Persistence is also not guaranteed.

Overwrite Stored Files

If a file being uploaded has the same name as an already uploaded file, the new file will overwrite the previous file.

Check if a File Has Already Been Uploaded

`FileManager` provides two methods that allow you to check if a file has been uploaded.

GETTING REMOTE FILES

`getRemoteFileNames()` returns a `List<String>` of the names of files that are uploaded to the head unit.

```
List<String> files = sdlManager.getFileManager().getRemoteFileNames
();
```

SEE IF A FILE IS UPLOADED

`hasUploadedFile` takes an `SdlFile` and returns a `boolean` of whether it is uploaded or not.

```
boolean isUploaded = sdlManager.getFileManager().hasUploadedFile(
sdlFile);
```

Check the Amount of File Storage

To find the amount of file storage left on the head unit, use the `ListFiles` RPC.

```
ListFiles listFiles = new ListFiles();
listFiles.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Integer spaceAvailable = ((ListFilesResponse) response).
getSpaceAvailable();
            Log.i("SdlService", "Space available on Core = " +
spaceAvailable);
        }else{
            Log.i("SdlService", "Failed to request list of uploaded files.");
        }
    }
});

sdlManager.sendRPC(listFiles);
```

Delete Stored Files

As with uploading, there are two methods that allow you to delete remote files.

FOR A SINGLE FILE

To delete a single file, call `deleteRemoteFileWithName()` and pass in the file name as a string. You can optionally pass in a `CompletionListener`.

```
sdlManager.getFileManager().deleteRemoteFileWithName("testFile",
new CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

MULTIPLE FILES

To delete multiple files, call `deleteRemoteFilesWithNames()` and pass in a list with the names of the files you want to delete. You can optionally pass in a `MultipleFileCompletionListener`.

```
sdlManager.getFileManager().deleteRemoteFilesWithNames(remoteFiles
, new MultipleFileCompletionListener() {
    @Override
    public void onComplete(Map<String, String> errors) {

    }
});
```

Image Specifics

Image File Type

Images may be formatted as PNG, JPEG, or BMP. Check the `DisplayCapabilities` object provided by `sdlManager.getSystemCapabilityManager().getCapability()` to find out what image formats the head unit supports.

Image Sizes

If an image is uploaded that is larger than the supported size, that image will be scaled down to accommodate.

IMAGE SPECIFICATIONS

IMAGE NAME	USED IN RPC	DETAILS	HEIGHT	WIDTH	TYPE
softButtonImage	Show	Will be shown on softbuttons on the base screen	70px	70px	png, jpg, bmp
choiceImage	CreateInteractionChoiceSet	Will be shown in the manual part of an performInteraction either big (ICON_ONLY) or small (LIST_ONLY)	70px	70px	png, jpg, bmp
choiceSecondaryImage	CreateInteractionChoiceSet	Will be shown on the right side of an entry in (LIST_ONLY) performInteraction	35px	35px	png, jpg, bmp
vrHelpItem	SetGlobalProperties	Will be shown during voice interaction	35px	35px	png, jpg, bmp
menuIcon	SetGlobalProperties	Will be shown on the “More...” button	35px	35px	png, jpg, bmp

IMAGE NAME	USED IN RPC	DETAILS	HEIGHT	WIDTH	TYPE
cmdIcon	AddCommand	Will be shown for commands in the "More..." menu	35px	35px	png, jpg, bmp
appIcon	SetAppIcon	Will be shown as Icon in the "Mobile Apps" menu	70px	70px	png, jpg, bmp
graphic	Show	Will be shown on the basescreen as cover art	185px	185px	png, jpg, bmp

Get Vehicle Data

Use the GetVehicleData RPC request to get vehicle data. The HMI level must be FULL, LIMITED, or BACKGROUND in order to get data.

Each vehicle manufacturer decides which data it will expose. Please check the PermissionManager to find out which data types your app currently has access to for the connected head unit.

NOTE

You may only ask for vehicle data that is available to your appName & appId combination. These will be specified by each OEM separately.

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
GPS	gps	Longitude and latitude, current time in UTC, degree of precision, altitude, heading, speed, satellite data vs dead reckoning, and supported dimensions of the GPS
Speed	speed	Speed in KPH
RPM	rpm	The number of revolutions per minute of the engine
Fuel level	fuelLevel	The fuel level in the tank (percentage)
Fuel level state	fuelLevel_State	The fuel level state: unknown, normal, low, fault, alert, or not supported
Fuel range	fuelRange	The estimate range in KM the vehicle can travel based on fuel level and consumption
Instant fuel consumption	instantFuelConsumption	The instantaneous fuel consumption in microlitres
External temperature	externalTemperature	The external temperature in degrees celsius
VIN	vin	The Vehicle Identification Number
PRNDL	prndl	The selected gear the car is in: park, reverse, neutral, drive, sport, low gear, first, second, third, fourth, fifth, sixth, seventh or eighth gear, unknown, or fault

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Tire pressure	tirePressure	Tire status of each wheel in the vehicle: normal, low, fault, alert, or not supported. Warning light status for the tire pressure: off, on, flash, or not used
Odometer	odometer	Odometer reading in km
Belt status	beltStatus	The status of each of the seat belts: no, yes, not supported, fault, or no event
Body information	bodyInformation	Door ajar status for each door. The Ignition status. The ignition stable status. The park brake active status.
Device status	deviceStatus	Contains information about the smartphone device. Is voice recognition on or off, has a bluetooth connection been established, is a call active, is the phone in roaming mode, is a text message available, the battery level, the status of the mono and stereo output channels, the signal level, the primary audio source, whether or not an emergency call is currently taking place
Driver braking	driverBraking	The status of the brake pedal: yes, no, no event, fault, not supported

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Wiper status	wiperStatus	The status of the wipers: off, automatic off, off moving, manual interaction off, manual interaction on, manual low, manual high, manual flick, wash, automatic low, automatic high, courtesy wipe, automatic adjust, stalled, no data exists
Head lamp status	headLampStatus	Status of the head lamps: whether or not the low and high beams are on or off. The ambient light sensor status: night, twilight 1, twilight 2, twilight 3, twilight 4, day, unknown, invalid
Engine torque	engineTorque	Torque value for engine (in Nm) on non-diesel variants
Engine oil life	engineOilLife	The estimated percentage of remaining oil life of the engine
Acceleration pedal position	accPedalPosition	Accelerator pedal position (percentage depressed)
Steering wheel angle	steeringWheelAngle	Current angle of the steering wheel (in degrees)
E-Call information	eCallInfo	Information about the status of an emergency call
Airbag status	airbagStatus	Status of each of the airbags in the vehicle: yes, no, no event, not supported, fault

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Emergency event	emergencyEvent	The type of emergency: frontal, side, rear, rollover, no event, not supported, fault. Fuel cutoff status: normal operation, fuel is cut off, fault. The roll over status: yes, no, no event, not supported, fault. The maximum change in velocity. Whether or not multiple emergency events have occurred
Cluster mode status	clusterModeStatus	Whether or not the power mode is active. The power mode qualification status: power mode undefined, power mode evaluation in progress, not defined, power mode ok. The car mode status: normal, factory, transport, or crash. The power mode status: key out, key recently out, key approved, post accessory, accessory, post ignition, ignition on, running, crank
My key	myKey	Information about whether or not the emergency 911 override has been activated
Turn signal	turnSignal	The status of the turn light indicator

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Electronic park brake status	electronicParkBrakeStatus	The status of the park brake as provided by Electric Park Brake (EPB) system
Cloud app vehicle id	cloudAppVehicleID	The id for the vehicle when connecting to cloud applications

Single Time Vehicle Data Retrieval

Using `GetVehicleData`, we can ask for vehicle data a single time, if needed.

```
GetVehicleData vdRequest = new GetVehicleData();
vdRequest.setPrndl(true);
vdRequest.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            PRNDL prndl = ((GetVehicleDataResponse) response).getPrndl();
            Log.i("SdlService", "PRNDL status: " + prndl.toString());
        }else{
            Log.i("SdlService", "GetVehicleData was rejected.");
        }
    }
});
sdlManager.sendRPC(vdRequest);
```

Subscribing to Vehicle Data

Subscribing to vehicle data allows you to get notified whenever we have new data available. This data should not be relied upon being received in a consistent manner. New vehicle data is available roughly every second.

First, you should add a notification listener for `OnVehicleData` notification:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_VEHICLE_DATA, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnVehicleData onVehicleDataNotification = (OnVehicleData)
notification;
        if (onVehicleDataNotification.getPrndl() != null) {
            Log.i("SdlService", "PRNDL status was updated to: " +
onVehicleDataNotification.getPrndl());
        }
    }
});
```

Then, send the Subscribe Vehicle Data Request:

```
SubscribeVehicleData subscribeRequest = new SubscribeVehicleData();
subscribeRequest.setPrndl(true);
subscribeRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Log.i("SdlService", "Successfully subscribed to vehicle data.");
        }else{
            Log.i("SdlService", "Request to subscribe to vehicle data was
rejected.");
        }
    }
});
sdlManager.sendRPC(subscribeRequest);
```

After that, the `onNotified` method should be called when there is an update to the subscribed vehicle data.

Unsubscribing from Vehicle Data

Sometimes you may not always need all of the vehicle data you are listening to. We suggest that you only are subscribing when the vehicle data is needed. To stop listening to specific vehicle data items, utilize `UnsubscribeVehicleData` .

```
UnsubscribeVehicleData unsubscribeRequest = new
UnsubscribeVehicleData();
unsubscribeRequest.setPrndl(true); // unsubscribe to PRNDL data
unsubscribeRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Log.i("SdlService", "Successfully unsubscribed to vehicle data."
);
        }else{
            Log.i("SdlService", "Request to unsubscribe to vehicle data was
rejected.");
        }
    }
});
sdlManager.sendRPC(unsubscribeRequest);
```

Knowing the In-Car UI Status

Once your app is connected to Core, most of the interaction you will be doing requires knowledge of the current In-Car UI, or HMI, Status. The HMI Status informs you of where the user is within the head unit in a general sense.

Refer to the table below of all possible HMI States:

HMI STATE	WHAT DOES THIS MEAN?
NONE	The user has not been opened your app, or it has been Exited via the "Menu" button.
BACKGROUND	The user has opened your app, but is currently in another part of the Head Unit. If you have a Media app, this means that another Media app has been selected.
LIMITED	For Media apps, this means that a user has opened your app, but is in another part of the Head Unit.
FULL	Your app is currently in focus on the screen.

Monitoring HMI Status

Monitoring HMI Status is possible through an `OnHMIStatus` notification that you can subscribe to via the `SdlManager`'s `addOnRPCNotificationListener`.

```
Map<FunctionID, OnRPCNotificationListener>
onRPCNotificationListenerMap = new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

More Detailed HMI Information

When an interaction occurs relating to your application, there is some additional pieces of information that can be observed that help figure out a more descriptive picture of what is going on with the Head Unit.

AUDIO STREAMING STATE

From the documentation, Audio Streaming State informs your app whether any currently streaming audio is audible to user (AUDIBLE) or not (NOT_AUDIBLE). A value of NOT_AUDIBLE means that either the application's audio will not be audible to the user, or that the application's audio should not be audible to the user (i.e. some other application on the mobile device may be streaming audio and the application's audio would be blended with that other audio).

You will see this come in for things such as Alert, PerformAudioPassThru, Speaks, etc.

AUDIO STREAMING STATE	WHAT DOES THIS MEAN?
AUDIBLE	Any audio you are streaming will be audible to the user.
ATTENUATED	Some kind of audio mixing is occuring between what you are streaming, if anything, and some system level sound. This can be visible is displaying an Alert with `playTone` set to `true`.
NOT_AUDIBLE	Your streaming audio is not audible. This could occur during a VRSESSSION System Context.

SYSTEM CONTEXT

System Context informs your app if there is potentially a blocking HMI component while your app is still visible. An example of this would be if your application is open, and you display an Alert. Your app will receive a System Context of ALERT while it is presented on the screen, followed by MAIN when it is dismissed.

SYSTEM CONTEXT STATE	WHAT DOES THIS MEAN?
MAIN	No user interaction is in progress that could be blocking your app's visibility.
VRSESSION	Voice Recognition is currently in progress.
MENU	A menu interaction is currently in-progress.
HMI_OBSCURED	The app's display HMI is being blocked by either a system or other app's overlay (another app's Alert, for instance).
ALERT	An alert that you have sent is currently visible (Other apps will not receive this).

Monitoring Audio Streaming State and System Context

Monitoring these two properties is quite easy using the `OnHMIStatus` notification.

```
@Override
public void onNotified(RPCNotification notification) {
    OnHMIStatus status = (OnHMIStatus) notification;
    AudioStreamingState streamingState = notification.
getAudioStreamingState();
    SystemContext systemContext = notification.getSystemContext();
}
```

Setting the Navigation Destination

Setting a Navigation Destination allows you to send a GPS location that you would like to prompt that user to navigate to using their embedded navigation. When using the `SendLocation` RPC, you will not receive a callback about how the user interacted with this location, only if it was successfully sent to Core and received. It will be handled by Core from that point on using the embedded navigation system.

> NOTE
>
> This currently is only supported for Embedded Navigation. This does not work with Mobile Navigation Apps at this time.

NOTE

`SendLocation` is an RPC that is usually restricted by OEMs. As a result, the OEM you are connecting to may limit app functionality if not approved for usage.

Determining the Result of SendLocation

`SendLocation` has 3 possible results that you should expect:

1. SUCCESS - `SendLocation` was successfully sent.
2. INVALID_DATA - The request you sent contains invalid data and was rejected.
3. DISALLOWED - Your app does not have permission to use `SendLocation`.

Detecting if SendLocation is Available

`SendLocation` is a newer RPC, so there is a possibility that not all head units will support it, especially if you are connected to a head unit that does not have an embedded navigation. To see if `SendLocation` is supported, you may look at `HmiCapabilities` that can be retrieved using `SystemCapabilityManager`.

```
HMICapabilities hmiCapabilities = (HMICapabilities) sdlManager.
getSystemCapabilityManager().getCapability(SystemCapabilityType.HMI
);
if (hmiCapabilities.isNavigationAvailable()){
    // SendLocation supported
}else{
    // SendLocation is not supported
}
```

Using SendLocation

To use `SendLocation`, you must at least include the Longitude and Latitude of the location. You can also include an address, name, description, phone number, and image.

```
SendLocation sendLocation = new SendLocation();
sendLocation.setLatitudeDegrees(42.877737);
sendLocation.setLongitudeDegrees(-97.380967);
sendLocation.setLocationName("The Center");
sendLocation.setLocationDescription("Center of the United States");

// Create Address
OasisAddress address = new OasisAddress();
address.setSubThoroughfare("900");
address.setThoroughfare("Whiting Dr");
address.setLocality("Yankton");
address.setAdministrativeArea("SD");
address.setPostalCode("57078");
address.setCountryCode("US-SD");
address.setCountryName("United States");

sendLocation.setAddress(address);

// Monitor response
sendLocation.setOnRPCResponseListener(new OnRPCResponseListener
() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();
        if(result.equals(Result.SUCCESS)){
            // SendLocation was successfully sent.
        }else if(result.equals(Result.INVALID_DATA)){
            // The request you sent contains invalid data and was rejected.
        }else if(result.equals(Result.DISALLOWED)){
            // Your app does not have permission to use SendLocation.
        }
    }
});

sdlManager.sendRPC(sendLocation);
```

Calling a Phone Number

Dialing a Phone Number allows you to send a phone number to dial on the user's phone. Regardless of platform, you must be sure that a device is connected via Bluetooth for this RPC to work. If it is not connected, you will receive a REJECTED `Result`.

> NOTE
>
> `DialNumber` is an RPC that is usually restricted by OEMs. As a result, the OEM you are connecting to may limit app functionality if not approved for usage.

Determining the Result of DialNumber

`DialNumber` has 3 possible results that you should expect:

1. SUCCESS - `DialNumber` was successfully sent, and a phone call was initiated by the user.
2. REJECTED - `DialNumber` was sent, and a phone call was cancelled by the user. Also, this could mean that there is no phone connected via Bluetooth.
3. DISALLOWED - Your app does not have permission to use `DialNumber`.

Detecting if DialNumber is Available

`DialNumber` is a newer RPC, so there is a possibility that not all head units will support it. To see if `DialNumber` is supported, you may look at the `HMICapabilities` that can be retrieved using `SystemCapabilityManager`.

```
HMICapabilities hmiCapabilities = (HMICapabilities) sdlManager.
getSystemCapabilityManager().getCapability(SystemCapabilityType.HMI
);
if(hmiCapabilities.isPhoneCallAvailable()){
    // DialNumber supported
}else{
    // DialNumber is not supported
}
```

How to Use

> NOTE
>
> For DialNumber, all characters are stripped except for `0`-`9`, `*`, `#`, `,`, `;`, and `+`

```
DialNumber dialNumber = new DialNumber();
dialNumber.setNumber("1238675309");
dialNumber.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();
        if(result.equals(Result.SUCCESS)){
            // `DialNumber` was successfully sent, and a phone call was
initiated by the user.
        }else if(result.equals(Result.REJECTED)){
            // `DialNumber` was sent, and a phone call was cancelled by
the user. Also, this could mean that there is no phone connected via
Bluetooth.
        }else if(result.equals(Result.DISALLOWED)){
            // Your app does not have permission to use DialNumber.
        }
    }
});

sdlManager.sendRPC(dialNumber);
```

Getting In-Car Audio

Capturing in-car audio allows developers to interact with users via raw audio data provided to them from the car's microphones. In order to gather the raw audio from the vehicle, we must leverage the `PerformAudioPassThru` RPC.

> NOTE
>
> PerformAudioPassThru does not support automatic speech cancellation detection, so if this feature is desired, it is up to the developer to implement.

Subscribing to AudioPassThru Notifications

Before starting audio capture, the app has to subscribe to `AudioPassThru` notification. SDL provides audio data as fast as it can gather it, and sends it to the developer in chunks. In order to retrieve this audio data, observe the `OnAudioPassThru` notification:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_AUDIO_PASS_THRU, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnAudioPassThru onAudioPassThru = (OnAudioPassThru)
notification;
        byte[] dataRcvd = onAudioPassThru.getAPTData();
        processAPTData(dataRcvd); // Do something with audio data
    }
});
```

> NOTE
>
> This audio data is only the current audio data, so the developer must be in charge of managing previously retrieved audio data.

Starting Audio Capture

To initiate audio capture, we must construct a `PerformAudioPassThru` object. The properties we will set in this object's constructor relate to how we wish to gather the audio data from the vehicle we are connected to.

```
PerformAudioPassThru performAPT = new PerformAudioPassThru();
performAPT.setAudioPassThruDisplayText1("Ask me \"What's the
weather?\"");
performAPT.setAudioPassThruDisplayText2("or \"What's 1 + 2?\"");

performAPT.setInitialPrompt(TTSChunkFactory.createSimpleTTSChunks(
"Ask me What's the weather? or What's 1 plus 2?"));
performAPT.setSamplingRate(SamplingRate._22KHZ);
performAPT.setMaxDuration(7000);
performAPT.setBitsPerSample(BitsPerSample._16_BIT);
performAPT.setAudioType(AudioType.PCM);
performAPT.setMuteAudio(false);

sdlManager.sendRPC(performAPT);
```

NOTE

`AudioPassThru` notification listener should be added before sending `PerformAudioPassThru` request or else some audio data may be missed.

FORD HMI

In order to know the currently supported audio capture capabilities of the connected head unit, please refer to the `SystemCapabilityManager`. It can retrieve the `AudioPassThruCapabilities` that the head unit supports.

> NOTE
>
> Currently, Ford's SYNC 3 vehicles only support a sampling rates of 16 khz and a bit rate of 16.

Ending Audio Capture

`AudioPassThru` is a request that works in a different way when compared to other RPCs. For most RPCs a request is followed by an immediate response that informs the developer whether or not that RPC was successful. This RPC,

however, will only send out the response when the Perform Audio Pass Thru is ended.

Audio Capture can be ended in 4 ways:

1. `AudioPassThru` has timed out.

   If the audio passthrough has proceeded longer than the requested timeout duration, Core will end this request and send a `PerformAudioPassThruResponse` with a `Result` of `SUCCESS`. You should expect to handle this audio passthrough as though it was successful.
2. `AudioPassThru` was closed due to user pressing "Cancel".

   If the audio passthrough was displayed, and the user pressed the "Cancel" button, you will receive a `PerformAudioPassThruResponse` with a `Result` of `ABORTED`. You should expect to ignore this audio pass through.
3. `AudioPassThru` was closed due to user pressing "Done".

   If the audio passthrough was displayed, and the user pressed the "Done" button, you will receive a `PerformAudioPassThruResponse` with a `Result` of `SUCCESS`. You should expect to handle this audio passthrough as though it was successful.
4. `AudioPassThru` was ended due to the developer ending the request.

   If the audio passthrough was displayed, but you have established on your own that you no longer need to capture audio data, you can send an `EndAudioPassThru` RPC.

```
EndAudioPassThru endAPT = new EndAudioPassThru();
sdlManager.sendRPC(endAPT);
```

You will receive an `EndAudioPassThruResponse` and a `PerformAudioPassThruResponse` with a `Result` of `SUCCESS`, and should expect to handle this audio passthrough as though it was successful.

Handling the Response

To process the response that we received from an ended audio capture, we monitor the `PerformAudioPassThruResponse` by adding a listener to the `PerformAudioPassThru` RPC before sending it. If the response has a successful `Result`, all of the audio data for the passthrough has been received and is ready for processing.

```
performAPT.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();

        if(result.equals(Result.SUCCESS)){
            // We can use the data
        }else{
            // Cancel any usage of the data
            Log.e("SdlService", "Audio pass thru attempt failed.");
        }
    }
});
```

Mobile Navigation

> NOTE
>
> This feature is only available on Android apps. Currently, JavaSE (embedded) and JavaEE (cloud) apps don't support that.

Mobile Navigation allows map partners to bring their applications into the car and display their maps and turn by turn easily for the user. This feature has a different behavior on the head unit than normal applications. The main differences are:

- Navigation Apps don't use base screen templates. Their main view is the video stream sent from the device
- Navigation Apps can send audio via a binary stream. This will attenuate the current audio source and should be used for navigation commands
- Navigation Apps can receive touch events from the video stream

> NOTE
>
> In order to use SDL's Mobile Navigation feature, the app must have a minimum requirement of Android 4.4 (SDK 19). This is due to using Android's provided video encoder.

Connecting an app

The basic connection is the similar for all apps. Please follow Getting Started for more information.

The first difference for a navigation app is the `appHMIType` of `NAVIGATION` that has to be set in the creation of the `SdlManager`. Navigation apps are also non-media apps.

The second difference is the requirement to call the `setSdlSecurity(List<Class<? extends SdlSecurityBase>> secList)` method from the `SdlManager.Builder` if connecting to an implementation of Core that requires secure video & audio streaming. This method requires an array of Security Managers, which will extend the `SdlSecurityBase` class. These security libraries are provided by the OEMs themselves, and will only work for that OEM. There is not a general catch-all security library.

```
SdlManager.Builder builder = new SdlManager.Builder(this, APP_ID,
APP_NAME, listener);

Vector<AppHMIType> hmiTypes = new Vector<AppHMIType>();
hmiTypes.add(AppHMIType.NAVIGATION);
builder.setAppTypes(hmiTypes);

List<? extends SdlSecurityBase> securityManagers = new ArrayList();
securityManagers.add(OEMSecurityManager1.class);
securityManagers.add(OEMSecurityManager1.class);
builder.setSdlSecurity(securityManagers);

MultiplexTransportConfig mtc = new MultiplexTransportConfig(this,
APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_OFF);
mtc.setRequiresHighBandwidth(true);
builder.setTransportType(transport);

sdlManager = builder.build();
sdlManager.start();
```

NOTE

When compiling, you must make sure to include all possible OEM's security managers that you wish to support.

After being registered, the app will start receiving callbacks. One important notification is `ON_HMI_STATUS`, which informs the app about the currently visible application on the head unit. Right after registering, the `hmiLevel` will be `NONE` or `BACKGROUND`. Streaming should commence once the `hmiLevel` has been set to `FULL` by the head unit.

Video Streaming

> NOTE
>
> This feature is only available on Android apps. Currently, JavaSE (embedded) and JavaEE (cloud) apps don't support that.

In order to stream video from an SDL app, we only need to manage a few things. For the most part, the library will handle the majority of logic needed to perform video streaming.

SDL Remote Display

The `SdlRemoteDisplay` base class provides the easiest way to start streaming using SDL. The `SdlRemoteDisplay` is extended from Android's `Presentation` class with modifications to work with other aspects of the SDL Android library.

> NOTE
>
> It is recommended that you extend this as a local class within the service that has the `SdlManager` instance.

Extending this class gives developers a familiar, native experience to handling layouts and events on screen.

```
public static class MyDisplay extends SdlRemoteDisplay{
    public MyDisplay(Context context, Display display) {
        super(context, display);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.sdl);

        final Button button1 = (Button) findViewById(R.id.button_1);

        button1.setOnTouchListener(new View.OnTouchListener() {
            @Override
            public boolean onTouch(View v, MotionEvent event) {
                Log.d(TAG, "Received motion event for button1");
            }
        });
    }
}
```

NOTE

If you are obfuscating the code in your app, make sure to exclude your class that extends `SdlRemoteDisplay`. For more information on how to do that, you can check Proguard Guidelines.

Managing the Stream

The `VideoStreamingManager` can be used to start streaming video after the `SdlManager` has successfully been started. This is performed by calling the method `startRemoteDisplayStream(Context context, final Class<? extends`

`SdlRemoteDisplay> remoteDisplay, final VideoStreamingParameters parameters, final boolean encrypted)`.

```
if (sdlManager.getVideoStreamManager() != null) {
    sdlManager.getVideoStreamManager().start(new CompletionListener
() {
        @Override
        public void onComplete(boolean success) {
            if (success) {
                sdlManager.getVideoStreamManager().
startRemoteDisplayStream(getApplicationContext(), MyDisplay.class,
null, false);
            } else {
                Log.e(TAG, "Failed to start video streaming manager");
            }
        }
    });
}
```

Ending the Stream

When the `HMIStatus` is back to `HMI_NONE` it is time to stop the stream. This is accomplished through a method `stopStreaming()`.

```
Map<FunctionID, OnRPCNotificationListener>
onRPCNotificationListenerMap = new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus status = (OnHMIStatus) notification;
        if (status != null && status.getHmiLevel() == HMILevel.HMI_NONE)
 {

            //Stop the stream
            if (sdlManager.getVideoStreamManager() != null && sdlManager
.getVideoStreamManager().isStreaming()) {
                sdlManager.getVideoStreamManager().stopStreaming();
            }

        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

Audio Streaming

> NOTE
>
> This feature is only available on Android apps. Currently, JavaSE (embedded) and JavaEE (cloud) apps don't support that.

Navigation apps are allowed to stream raw audio to be played by the head unit. The audio received this way is played immediately, and the current audio

source will be attenuated. The raw audio has to be played with the following parameters:

- **Format**: PCM
- **Sample Rate**: 16k
- **Number of Channels**: 1
- **Bits Per Second (BPS)**: 16 bits per sample / 2 bytes per sample

You can now also push `mp3` files using the `AudioStreamingManager`, which is accessed through the `SdlManager`.

> NOTE
>
> For streaming consistent audio, such as music, use a normal A2DP stream and not this method.

STREAMING AUDIO

To stream audio, we call `sdlManager.getAudioStreamManager().start()` which will start the manager. When that callback returns successful, you call `sdlManager.getAudioStreamManager().startAudioStream()`. When the callback for that is successful, you can push the audio source using `sdlManager.getAudioStreamManager().pushAudioSource()`. Below is an example of playing an `mp3` file that we have in our resource directory:

```
if (sdlManager.getAudioStreamManager() != null) {
    Log.i(TAG, "Trying to start audio streaming");
    sdlManager.getAudioStreamManager().start(new CompletionListener
() {
        @Override
        public void onComplete(boolean success) {
            if (success) {
                sdlManager.getAudioStreamManager().startAudioStream(
false, new CompletionListener() {
                    @Override
                    public void onComplete(boolean success) {
                        if (success) {
                            Resources resources = getApplicationContext().
getResources();
                            int resourceId = R.raw.exampleMp3;
                            Uri uri = new Uri.Builder()
                            .scheme(ContentResolver.
SCHEME_ANDROID_RESOURCE)
                            .authority(resources.getResourcePackageName(
resourceId))
                            .appendPath(resources.getResourceTypeName(
resourceId))
                            .appendPath(resources.getResourceEntryName(
resourceId))
                            .build();
                            sdlManager.getAudioStreamManager().
pushAudioSource(uri, new CompletionListener() {
                                @Override
                                public void onComplete(boolean success) {
                                    if (success) {
                                        Log.i(TAG, "Audio file played successfully!");
                                    } else {
                                        Log.i(TAG, "Audio file failed to play!");
                                    }
                                }
                            });
                        } else {
                            Log.d(TAG, "Audio stream failed to start!");
                        }
                    }
                });
            } else {
                Log.i(TAG, "Failed to start audio streaming manager");
            }
        }
    });
}
```

STOPPING THE AUDIO STREAM

When the stream is complete, or you receive HMI_NONE, you should stop the stream by calling:

```
sdlManager.getAudioStreamManager().stopAudioStream(new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

Supporting Haptic Input

> NOTE
>
> This feature is only available on Android apps. Currently, JavaSE (embedded) and JavaEE (cloud) apps don't support that.

SDL now supports "haptic" input, input from something other than a touch screen. This could include trackpads, click-wheels, etc. These kinds of inputs work by knowing which areas on the screen are touchable and focusing on those areas when the user moves the trackpad or click wheel. When the user selects a rect, the center of that area will be "touched".

NOTE

Currently, there are no RPCs for knowing which rect is highlighted, so your UI will have to remain static, without scrolling.

You will also need to implement touch input support (Mobile Navigation/Touch Input) in order to receive touches of the rects.

Using SDL Presentation

SDL has support for automatically detecting focusable rects within your UI and sending that data to the head unit. You will still need to tell SDL when your UI changes so that it can re-scan and detect the rects to be sent. The easiest way to use this is by taking advantage of SDL's Presentation class. This will automatically check if the capability is available and instantiate the manager for you. All you have to do is set your layout:

```
public static class MyPresentation extends SdlRemoteDisplay {

    public MyPresentation(Context context, Display display) {
        super(context, display);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.haptic_layout);
        LinearLayout videoView = (LinearLayout) findViewById(R.id.
cat_view);
        videoView.setOnTouchListener(new View.OnTouchListener() {
            @Override
            public boolean onTouch(View view, MotionEvent motionEvent) {
                // ...Update something on the ui

                MyPresentation.this.invalidate();
            }
        });
    }
}
```

This will go through your view that was passed in and then find and send the rects to the head unit for use. When your UI changes, call `invalidate()` from your class that extends `SdlRemoteDisplay`.

Sending your own Rects

It is also possible that you may want to create your own rects instead of using the automated methods in the Presentation class. It is important that if sending this data yourself that you also use the `SystemCapabilityManager` to check if you are on a head unit that supports this feature. If the capability is available, it is easy to build the area you want to become selectable:

```
public void sendHapticData() {

    Rectangle rectangle = new Rectangle();
    rectangle.setX((float) 1.0);
    rectangle.setY((float) 1.0);
    rectangle.setWidth((float) 1.0);
    rectangle.setHeight((float) 1.0);

    HapticRect hapticRect = new HapticRect();
    hapticRect.setId(123);
    hapticRect.setRect(rec);

    ArrayList<HapticRect> hapticArray = new ArrayList<HapticRect>();
    hapticArray.add(0, hr);

    SendHapticData sendHapticData = new SendHapticData();
    sendHapticData.setHapticRectData(hapticArray);

    sdlManager.sendRPC(sendHapticData);

}
```

Each SendHapticData rpc should contain the entirety of all clickable areas to be accessed via haptic controls.

Handling a Language Change

When a user changes the language on a head unit, an `OnLanguageChange` notification will be sent from Core. Then your app will disconnect. In order for your app to automatically reconnect to the head unit, there are a few changes to make in the following files:

- Local SDL Broadcast Receiver
- Local SDL Service

SDL Broadcast Receiver

When the SDL Service's connection to core is closed, we want to tell our local SDL Broadcast Receiver to restart the SDL Service. To do this, first add a public String in your app's local SDL Broadcast Receiver class that can be included as an extra in a broadcast intent.

`public static final String RECONNECT_LANG_CHANGE = "RECONNECT_LANG_CHANGE";`

Then, override the `onReceive()` method of the local SDL Broadcast Receiver to call `onSdlEnabled()` when receiving that action:

```
@Override
public void onReceive(Context context, Intent intent) {
    super.onReceive(context, intent); // Required if overriding this
method

    if (intent != null) {
        String action = intent.getAction();
        if (action != null){
            if(action.equalsIgnoreCase(TransportConstants.
START_ROUTER_SERVICE_ACTION)) {
                if (intent.getBooleanExtra(RECONNECT_LANG_CHANGE, false
)) {
                    onSdlEnabled(context, intent);
                }
            }
        }
    }
}
```

MUST

Be sure to call `super.onReceive(context, intent);` at the start of the method!

NOTE

This guide also assumes your local SDL Broadcast Receiver implements the `onSdlEnabled()` method as follows:

```
@Override
public void onSdlEnabled(Context context, Intent intent) {
    intent.setClass(context, SdlService.class);
    context.startService(intent);
}
```

SDL Service

We want to tell our local SDL Broadcast Receiver to restart the service when an `OnLanguageChange` notification is received from Core . To do so, add a notification listener as follows:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_LANGUAGE_CHANGE, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        SdlService.this.stopSelf();
        Intent intent = new Intent(TransportConstants.
START_ROUTER_SERVICE_ACTION);
        intent.putExtra(SdlReceiver.RECONNECT_LANG_CHANGE, true);
        AndroidTools.sendExplicitBroadcast(context, intent, null);
    }
});
```

System Capability Manager

The System Capability Manager is a central location to obtain capabilities about the currently connected module. Specific capabilities will be returned for a number of given keys (e.g. `NAVIGATION`, `VIDEO_STREAMING`). It also alleviates the need to individually cache results that come from the `RegisterApInterface` response or from the new `SystemCapabilityQuery`.

There are multiple capabilities that can be retrieved:

SUPPORTED CAPABILITIES
NAVIGATION
PHONE_CALL
VIDEO_STREAMING
REMOTE_CONTROL
HMI
DISPLAY
AUDIO_PASSTHROUGH
BUTTON
HMI_ZONE
PRESET_BANK
SOFTBUTTON
SPEECH
VOICE_RECOGNITION
APP_SERVICES

Querying Capabilities

Any point after receiving the first `OnHMIStatus` notification from the connected module, you can access the `SystemCapability` manager and its

data. Your instance of `SdlManager` will provide access to the `SystemCapabilityManager`.

> NOTE
>
> It is important to query capabilities before you use them. Your app may be used on a variety of head units across different manufacturers and software versions. Never assume that a capability exists.

For example (obtaining the head unit's `DISPLAY` capability):

```
sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.DISPLAY, new OnSystemCapabilityListener(){

    @Override
    public void onCapabilityRetrieved(Object capability){
        DisplayCapabilities dispCapability = (DisplayCapabilities)
capability;
    }

    @Override
    public void onError(String info){
        Log.i(TAG, "Capability could not be retrieved: "+ info);
    }
});
```

The returned capability needs to be cast into the capability type you requested. From there you can determine whether or not the head unit that the app is connected to can utilize a feature or not.

Capability Lists

These are the current responses that come back as Lists:
- `AUDIO_PASSTHROUGH`
- `BUTTON`
- `SOFTBUTTON`
- `SPEECH`
- `HMI_ZONE`
- `VOICE_RECOGNITION`

We've created a method in the `SystemCapabilityManager` to help cast these lists. Below is an example of its usage:

```
sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.BUTTON, new OnSystemCapabilityListener(){

    @Override
    public void onCapabilityRetrieved(Object capability){
        List<ButtonCapabilities> buttonCapabilityList =
SystemCapabilityManager.convertToList(capability, ButtonCapabilities.
class);
    }

    @Override
    public void onError(String info){
        Log.i(TAG, "Capability could not be retrieved: "+ info);
    }
});
```

This method prevents the developer from having to suppress a warning as well as creates a safe way to cast the object to a list.

Asynchronous vs Synchronous Queries

Some capabilities will be instantly available after the first `OnHMIStatus` notification. These are parsed from the `RegisterAppInterface` response. However, some capabilities MUST be obtained asynchronously and therefore require a callback to be obtained. If a capability can be retrieved synchronously another method can be used via the `SystemCapbilityManager` object obtained from the `SdlManager`, `sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType)`.

CAPABILITY	ASYNC REQUIRED
NAVIGATION	Yes
PHONE_CALL	Yes
VIDEO_STREAMING	Yes
REMOTE_CONTROL	Yes
HMI	No
DISPLAY	No
AUDIO_PASSTHROUGH	No
BUTTON	No
HMI_ZONE	No
PRESET_BANK	No
SOFTBUTTON	No
SPEECH	No
VOICE_RECOGNITION	No
APP_SERVICES	Yes

Permission Manager

The `PermissionManager` allows developers to easily query whether specific RPCs are allowed or not. It also allows a listener to be added for a list of RPCs so that if there are changes in their permissions, the app will be notified.

Querying Permission

Using the `PermissionManager`, you can easily know if a specific RPC is allowed or not. For example, if you want to check if the `Show` RPC is allowed you can use the `isRPCAllowed` method:

```
boolean allowed = sdlManager.getPermissionManager().isRPCAllowed(
FunctionID.SHOW);
```

> NOTE
>
> Some RPCs are allowed in specific hmi levels but not allowed in others.

Querying Permission Parameters

Some RPCs have parameters. For example, `GetVehicleData` has parameters like `speed`, `rpm`, and `airbagStatus`. The developer may need to know not

only whether `GetVehicleData` is allowed but also if a specific parameter in that RPC is allowed. For that case the `isPermissionParameterAllowed` method can be used to tell if the RPC and the parameter are both allowed:

```
boolean allowed = sdlManager.getPermissionManager().
isPermissionParameterAllowed(FunctionID.GET_VEHICLE_DATA,
GetVehicleData.KEY_RPM);
```

Querying Multiple Permissions at Once

In some cases, developers may need to know whether multiple permissions (or permission parameters) are allowed and perform a specific action based on the result. The `PermissionManager` has a convenience method that does that. For example, if the developers need to know whether `Show` and `GetVehicleData` RPCs are allowed and also make sure that `speed` and `rpm` parameters in `GetVehicleData` are allowed, they can use `getGroupStatusOfPermissions` method to do that. First, a list of `PermissionElement`s should be created. Each `PermissionElement` in the list holds the RPC that we want to check the permission for and a list of optional parameters for that permission:

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW,
null));
permissionElements.add(new PermissionElement(FunctionID.
GET_VEHICLE_DATA, Arrays.asList(GetVehicleData.KEY_RPM,
GetVehicleData.KEY_SPEED)));

int groupStatus = sdlManager.getPermissionManager().
getGroupStatusOfPermissions(permissionElements);

switch (groupStatus) {
    case PermissionManager.PERMISSION_GROUP_STATUS_ALLOWED:
        // Every permission in the group is currently allowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_DISALLOWED:
        // Every permission in the group is currently disallowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_MIXED:
        // Some permissions in the group are allowed and some disallowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_UNKNOWN:
        // The current status of the group is unknown
        break;
}
```

The previous snippet will give a quick generic status for all permissions together. However, if developers want to get a more detailed result about the status of every permission or parameter in the group, they can use `getStatusOfPermissions` method:

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW,
null));
permissionElements.add(new PermissionElement(FunctionID.
GET_VEHICLE_DATA, Arrays.asList(GetVehicleData.KEY_RPM,
GetVehicleData.KEY_AIRBAG_STATUS)));

Map<FunctionID, PermissionStatus> status = sdlManager.
getPermissionManager().getStatusOfPermissions(permissionElements);

if (status.get(FunctionID.GET_VEHICLE_DATA).getIsRPCAllowed()){
    // GetVehicleData RPC is allowed
}

if (status.get(FunctionID.GET_VEHICLE_DATA).getAllowedParameters().
get(GetVehicleData.KEY_RPM)){
    // rpm parameter in GetVehicleData RPC is allowed
}
```

Adding Permissions Change Listener

In some cases, the app may need to be notified when there is a change in some permissions. Developers can use the `PermissionManager` to add a listener that will be called when the specified permissions change. The listener can be called either when there is any change or only when all permissions become allowed. That can be determined by the `PermissionGroupType` value that is passed to the `AddListener` method:

PERMISSION GROUP TYPE	DESCRIPTION
PERMISSION_GROUP_TYPE_ALL_ALLOWED	Be notified when all of the permissions in the group are allowed, or when they all stop being allowed in some sense, that is, when they were all allowed, and now they are not.
PERMISSION_GROUP_TYPE_ANY	Be notified when any change in availability occurs among the group.

For example, to setup a listener that will be called when there is any update to `Show` or `GetVehicleData` permissions or `rpm`, `airbagStatus` parameter permissions in the `GetVehicleData` RPC, you can use the following code snippet:

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW,
null));
permissionElements.add(new PermissionElement(FunctionID.
GET_VEHICLE_DATA, Arrays.asList(GetVehicleData.KEY_RPM,
GetVehicleData.KEY_AIRBAG_STATUS)));


UUID listenerId = sdlManager.getPermissionManager().addListener(
permissionElements, PermissionManager.
PERMISSION_GROUP_TYPE_ANY, new OnPermissionChangeListener() {
    @Override
    public void onPermissionsChange(@NonNull Map<FunctionID,
PermissionStatus> allowedPermissions, @NonNull int
permissionGroupStatus) {
        if (allowedPermissions.get(FunctionID.GET_VEHICLE_DATA).
getIsRPCAllowed()) {
            // GetVehicleData RPC is allowed
        }

        if (allowedPermissions.get(FunctionID.GET_VEHICLE_DATA).
getAllowedParameters().get(GetVehicleData.KEY_RPM)){
            // rpm parameter in GetVehicleData RPC is allowed
        }
    }
});
```

NOTE

Don't forget to remove the listener using the `removeListener` method when you are done with it.

Remote Control

Remote Control provides a framework to allow apps to control certain safe modules within a vehicle.

NOTE

Not all vehicles have this functionality. Even if they support remote control, you will likely need to request permission from the vehicle manufacturer to use it.

WHY IS THIS HELPFUL?

Consider the following scenarios:

- A radio application wants to use the in-vehicle radio tuner. It needs the functionality to select the radio band (AM/FM/XM/HD/DAB), tune the radio

frequency or change the radio station, as well as obtain general radio information for decision making.

- A climate control application needs to turn on the AC, control the air circulation mode, change the fan speed and set the desired cabin temperature.
- A user profile application wants to remember users' favorite settings and apply it later automatically when the users get into the same/another vehicle.

Currently, the Remote Control feature supports these modules:

SUPPORTED RC MODULES
Climate
Radio
Seat
Audio
Light
HMI Settings

The following table lists what control items are in each control module.

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
Climate	Current Cabin Temperature		Get/ Notification	read only, value range depends on OEM
	Desired Cabin Temperature		Get/Set/ Notification	value range depends on OEM
	AC Setting	on, off	Get/Set/ Notification	
	AC MAX Setting	on, off	Get/Set/ Notification	
	Air Recirculation Setting	on, off	Get/Set/ Notification	
	Auto AC Mode Setting	on, off	Get/Set/ Notification	
	Defrost Zone Setting	front, rear, all, none	Get/Set/ Notification	
	Dual Mode Setting	on, off	Get/Set/ Notification	
	Fan Speed Setting	0%-100%	Get/Set/ Notification	
	Ventilation Mode Setting	upper, lower, both, none	Get/Set/ Notification	
Radio	Radio Enabled	true,false	Get/Set/ Notification	read only, all other radio control items need radio enabled to work
	Radio Band	AM,FM,XM	Get/Set/ Notification	

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Radio Frequency		Get/Set/ Notification	value range depends on band
	Radio RDS Data		Get/ Notification	read only
	Available HD Channel	1-3	Get/ Notification	read only
	Current HD Channel	1-3	Get/Set/ Notification	
	Radio Signal Strength		Get/ Notification	read only
	Signal Change Threshold		Get/ Notification	read only
	Radio State	Acquiring, acquired, multicast, not_found	Get/ Notification	read only
Seat	Seat Heating Enabled	true, false	Get/Set/ Notification	Indicates whether heating is enabled for a seat
	Seat Cooling Enabled	true, false	Get/Set/ Notification	Indicates whether cooling is enabled for a seat
	Seat Heating level	0-100%	Get/Set/ Notification	Level of the seat heating
	Seat Cooling level	0-100%	Get/Set/ Notification	Level of the seat cooling

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Seat Horizontal Positon	0-100%	Get/Set/ Notification	Adjust a seat forward/ backward, 0 means the nearest position to the steering wheel, 100% means the furthest position from the steering wheel
	Seat Vertical Position	0-100%	Get/Set/ Notification	Adjust seat height (up or down) in case there is only one actuator for seat height, 0 means the lowest position, 100% means the highest position

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Seat-Front Vertical Position	0-100%	Get/Set/ Notification	Adjust seat front height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position
	Seat-Back Vertical Position	0-100%	Get/Set/ Notification	Adjust seat back height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Seat Back Tilt Angle	0-100%	Get/Set/ Notification	Backrest recline, 0 means the angle that back top is nearest to the steering wheel, 100% means the angle that back top is furthest from the steering wheel
	Head Support Horizontal Positon	0-100%	Get/Set/ Notification	Adjust head support forward/ backward, 0 means the nearest position to the front, 100% means the furthest position from the front

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Head Support Vertical Position	0-100%	Get/Set/ Notification	Adjust head support height (up or down), 0 means the lowest position, 100% means the highest position
	Seat Massaging Enabled	true, false	Get/Set/ Notification	Indicates whether massage is enabled for a seat
	Massage Mode	List of Struct {MassageZone, MassageMode}	Get/Set/ Notification	list of massage mode of each zone
	Massage Cushion Firmness	List of Struct {Cushion, 0-100%}	Get/Set/ Notification	list of firmness of each massage cushion
	Seat memory	Struct{ id, label, action (SAVE/ RESTORE/ NONE)}	Get/Set/ Notification	seat memory
Audio	Audio volume	0%-100%	Get/Set/ Notification	The audio source volume level

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Audio Source	MOBILE_APP, RADIO_TUNER, CD, BLUETOOTH, USB, etc. see PrimaryAudioSource	Get/Set/ Notification	defines one of the available audio sources
	keep Context	true, false	Set only	control whether HMI shall keep current application context or switch to default media UI/ APP associated with the audio source
	Equilizer Settings	Struct {Channel ID as integer, Channel setting as 0%-100%}	Get/Set/ Notification	Defines the list of supported channels (band) and their current/ desired settings on HMI
Light	Light Status	ON, OFF	Get/Set/ Notification	turn on/off a single light or all lights in a group

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Light Density	float 0.0-1.0	Get/Set/ Notification	change the density/ dim a single light or all lights in a group
	Light Color	RGB color	Get/Set/ Notification	change the color scheme of a single light or all lights in a group
HMI Settings	Display Mode	DAY, NIGHT, AUTO	Get/Set/ Notification	Current display mode of the HMI display
	Distance Unit	MILES, KILOMETERS	Get/Set/ Notification	Distance Unit used in the HMI (for maps/ tracking distances)
	Temperature Unit	FAHRENHEIT, CELSIUS	Get/Set/ Notification	Temperature Unit used in the HMI (for temperature measuring systems)

Remote Control can also allow mobile applications to send simulated button press events for the following common buttons in the vehicle.

The system shall list all available buttons for Remote Control in the `RemoteControlCapabilities`. The capability object will have a List of `ButtonCapabilities` that can be obtained using `getButtonCapabilities()`.

RC MODULE	CONTROL BUTTON
Climate	AC
	AC MAX
	RECIRCULATE
	FAN UP
	FAN DOWN
	TEMPERATURE UP
	TEMPERATURE DOWN
	DEFROST
	DEFROST REAR
	DEFROST MAX
	UPPER VENT
	LOWER VENT
Radio	VOLUME UP
	VOLUME DOWN
	EJECT
	SOURCE
	SHUFFLE
	REPEAT

Integration

> NOTE
>
> For Remote Control to work, the head unit must support SDL Core Version 4.4 or newer. Also your app's appHMIType should be set to `REMOTE_CONTROL`.

SYSTEM CAPABILITY

> MUST
>
> Prior to using any Remote Control RPCs, you must check that the head unit has the Remote Control capability. As you may encounter head units that do *not* support it, this check is important.

To check for this capability, use the following call:

```
// First you can check to see if the capability is supported on the module
if (sdlManager.getSystemCapabilityManager().isCapabilitySupported(
SystemCapabilityType.REMOTE_CONTROL)){
    // Since the module does support this capability we can query it for
more information
    sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.REMOTE_CONTROL, new
OnSystemCapabilityListener(){

        @Override
        public void onCapabilityRetrieved(Object capability){
            RemoteControlCapabilities remoteControlCapabilities = (
RemoteControlCapabilities) capability;
            // Now it is possible to get details on how this capability
            // is supported using the remoteControlCapabilities object
        }

        @Override
        public void onError(String info){
            Log.i(TAG, "Capability could not be retrieved: "+ info);
        }
    });
}
```

GETTING DATA

It is possible to retrieve current data relating to these Remote Control modules. The data could be used to store the settings prior to setting them, saving user preferences, etc. Following the check on the system's capability to support Remote Control, we can actually retrieve the data. The following is an example of getting data about the `RADIO` module. It also subscribes to updates to radio data, which will be discussed later on in this guide.

```
GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData();
interiorVehicleData.setModuleType(ModuleType.RADIO);
interiorVehicleData.setSubscribe(true);
interiorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        GetInteriorVehicleData getResponse = (GetInteriorVehicleData)
response;
        //This can now be used to retrieve data
    }
});

sdlManager.sendRPC(interiorVehicleData);
```

SETTING DATA

Of course, the ability to set these modules is the point of Remote Control. Setting data is similar to getting it. Below is an example of setting `ClimateControlData`.

```
Temperature temp = new Temperature();
temp.setUnit(TemperatureUnit.FAHRENHEIT);
temp.setValue((float) 74.1);

ClimateControlData climateControlData = new ClimateControlData();
climateControlData.setAcEnable(true);
climateControlData.setAcMaxEnable(true);
climateControlData.setAutoModeEnable(false);
climateControlData.setCirculateAirEnable(true);
climateControlData.setCurrentTemperature(temp);
climateControlData.setDefrostZone(DefrostZone.FRONT);
climateControlData.setDualModeEnable(true);
climateControlData.setFanSpeed(2);
climateControlData.setVentilationMode(VentilationMode.BOTH);
climateControlData.setDesiredTemperature(temp);

ModuleData moduleData = new ModuleData();
moduleData.setModuleType(ModuleType.CLIMATE);
moduleData.setClimateControlData(climateControlData);

SetInteriorVehicleData setInteriorVehicleData = new
SetInteriorVehicleData();
setInteriorVehicleData.setModuleData(moduleData);

sdlManager.sendRPC(setInteriorVehicleData);
```

It is likely that you will not need to set all the data as it is in the example, so if there are settings you don't wish to modify, then you don't have to.

BUTTON PRESSES

Another unique feature of Remote Control is the ability to send simulated button presses to the associated modules, imitating a button press on the hardware itself.

Simply specify the module, the button, and the type of press you would like:

```
ButtonPress buttonPress = new ButtonPress();
buttonPress.setModuleType(ModuleType.RADIO);
buttonPress.setButtonName(ButtonName.EJECT);
buttonPress.setButtonPressMode(ButtonPressMode.SHORT);

sdlManager.sendRPC(buttonPress);
```

SUBSCRIBING TO CHANGES

It is also possible to subscribe to changes in data associated with supported modules.

To do so, during your `GET` request for data, simply add in `setSubscribe(Boolean)`. To unsubscribe, send the request again with the boolean set to `False`. A code sample for setting the subscription is in the `GET` example above.

The response to a subscription will come in a form of a notification. You can receive this notification by adding a notification listener for `OnInteriorVehicleData`.

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_INTERIOR_VEHICLE_DATA, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnInteriorVehicleData onInteriorVehicleData = (
OnInteriorVehicleData) notification;
        //Perform action based on notification
    }
});

//Then send the GetInteriorVehicleData with subscription set to true
GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData();
interiorVehicleData.setModuleType(ModuleType.RADIO);
interiorVehicleData.setSubscribe(true);

sdlManager.sendRPC(interiorVehicleData);
```

NOTE

The notification listener should be added before sending the `GetInteriorVehicleData` request.

Sending Multiple RPCs

Batch Sending RPCs

There are two ways to send multiple requests to the head unit: concurrently and sequentially. Which method you should use depends on the type of RPCs being sent. Concurrently sent requests might finish in a random order and should only be used when none of the requests in the group depend on the response of another, such as when uploading a group of artworks. Sequentially sent requests only send the next request in the group when a response has been received for the previously sent RPC. Requests should be sent sequentially when you need to know the result of a previous request before sending the next, like when sending the several different requests needed to create a menu.

Both methods have optional listeners that are specific to them, the `OnMultipleRequestListener`. This listener will provide additional information than the normal `OnRPCResponseListener`. Its use is shown below.

Send Requests

`sendRPCs` allows you to easily send an ArrayList of `RPCRequests` easily to the head unit. When you send multiple RPCs concurrently there is no guarantee of the order in which the RPCs will be sent or in which order Core will return responses. The method also comes with its own listener, `OnMultipleRequestListener` that will provide you with updates as the sending progresses, errors that may arise, and let you know when the sending is finished. Below is a sample call:

```
List<RPCRequest> rpcs = new ArrayList<>();

// rpc 1
SubscribeButton subscribeButtonRequestLeft = new SubscribeButton();
subscribeButtonRequestLeft.setButtonName(ButtonName.SEEKLEFT);
rpcs.add(subscribeButtonRequestLeft);

// rpc 2
SubscribeButton subscribeButtonRequestRight = new SubscribeButton
();
subscribeButtonRequestRight.setButtonName(ButtonName.SEEKRIGHT
);
rpcs.add(subscribeButtonRequestRight);


sdlManager.sendRPCs(rpcs, new OnMultipleRequestListener() {
    @Override
    public void onUpdate(int remainingRequests) {

    }

    @Override
    public void onFinished() {

    }

    @Override
    public void onResponse(int correlationId, RPCResponse response) {

    }

    @Override
    public void onError(int correlationId, RPCResponse response) {

    }
});
```

Send Sequential Requests

As you may have guessed, this method is called similarly to `sendRPCs` but sends the requests synchronously, guaranteeing order. It is important to note that you want to build your array with the items that you want to send first,

first. This is particularly useful for RPCs that are dependent upon other ones, such as a `performInteraction` needing a `createInteractionChoiceSet`'s id.

This method call is exactly the same as above, except for the method name being `sendSequentialRPCs`. For your convenience, the listener is also the same and performs similarly.

```
List<RPCRequest> rpcs = new ArrayList<>();

// rpc 1
SubscribeButton subscribeButtonRequestLeft = new SubscribeButton();
subscribeButtonRequestLeft.setButtonName(ButtonName.SEEKLEFT);
rpcs.add(subscribeButtonRequestLeft);

// rpc 2
SubscribeButton subscribeButtonRequestRight = new SubscribeButton
();
subscribeButtonRequestRight.setButtonName(ButtonName.SEEKRIGHT
);
rpcs.add(subscribeButtonRequestRight);

sdlManager.sendSequentialRPCs(rpcs, new OnMultipleRequestListener()
 {
    @Override
    public void onUpdate(int remainingRequests) {

    }

    @Override
    public void onFinished() {

    }

    @Override
    public void onResponse(int correlationId, RPCResponse response) {

    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {

    }
});
```

Creating a Cloud App Store

A new feature of SDL 5.1 and SDL Java Suite 4.8 allows OEMs to offer an app store that lets users browse and install remote cloud apps. If the cloud app requires users to login with their credentials, the app store can use an authentication token to automatically login users after their first session.

> NOTE
>
> An OEM app store can be a mobile app or a cloud app.

User Authentication

App stores can handle user authentication for the installed cloud apps. For example, users can log in after installing a cloud app using the app store. After that, the app store will save an authentication token for the cloud app in the local policy table. Then, the cloud app can retrieve the authentication token from the local policy table and use it to authenticate a user with the application. If desired, an optional parameter, `CloudAppVehicleID`, can be used to identify the head unit.

Setting and Getting Cloud App Properties

An OEM's app store can manage the properties of a specific cloud app by setting and getting its `CloudAppProperties`. This table summarizes the properties that are included in `CloudAppProperties`:

PARAMETER NAME	DESCRIPTION
appID	appID for the cloud app
nicknames	List of possible names for the cloud app. The cloud app will not be allowed to connect if its name is not contained in this list
enabled	If true, cloud app will be displayed on HMI
authToken	Used to authenticate the user, if the app requires user authentication
cloudTransportType	Specifies the connection type Core should use. Currently Core supports WS and WSS , but an OEM can implement their own transport adapter to handle different values
hybridAppPreference	Specifies the user preference to use the cloud app version, mobile app version, or whichever connects first when both are available
endpoint	Remote endpoint for websocket connections

NOTE

Only trusted app stores are allowed to set or get `CloudAppProperties` for other cloud apps.

SETTING CLOUD APP PROPERTIES

App stores can set properties for a cloud app by sending a `SetCloudAppProperties` request to Core to store them in the local policy table. For example, in this

piece of code, the app store can set the `authToken` to associate a user with a cloud app after the user logs in to the app by using the app store:

```
CloudAppProperties cloudAppProperties = new CloudAppProperties(
"<appId>");
cloudAppProperties.setAuthToken("<auth token>");
SetCloudAppProperties setCloudAppProperties = new
SetCloudAppProperties(cloudAppProperties);
setCloudAppProperties.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()) {
            Log.i("SdlService", "Request was successful.");
        } else {
            Log.i("SdlService", "Request was rejected.");
        }
    }
});
sdlManager.sendRPC(setCloudAppProperties);
```

GETTING CLOUD APP PROPERTIES

To retrieve cloud properties for a specific cloud app from local policy table, app stores can send `GetCloudAppProperties` and specify the `appId` for that cloud app as in this example:

```
GetCloudAppProperties getCloudAppProperties = new
GetCloudAppProperties("<appId>");
getCloudAppProperties.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()) {
            Log.i("SdlService", "Request was successful.");
            GetCloudAppPropertiesResponse
getCloudAppPropertiesResponse = (GetCloudAppPropertiesResponse)
response;
            CloudAppProperties cloudAppProperties =
getCloudAppPropertiesResponse.getCloudAppProperties();
            // Use cloudAppProperties
        } else {
            Log.i("SdlService", "Request was rejected.");
        }
    }
});
sdlManager.sendRPC(getCloudAppProperties);
```

Getting the Cloud App Icon

Cloud app developers don't need to add any code to download the app icon. The cloud app icon will be automatically downloaded from the url provided by the policy table and sent to Core to be later displayed on the HMI.

Getting the Authentication Token

When users install cloud apps from an OEM's app store, they may be asked to login to that cloud app using the app store. After logging in, app store can save the `authToken` in the local policy table to be used later by the cloud app for user authentication.
A cloud app can retrieve its `authToken` from local policy table after starting the RPC service. The `authToken` can be used later by the app to authenticate the user:

```
String authToken = sdlManager.getAuthToken();
```

Getting CloudAppVehicleID (Optional)

The `CloudAppVehicleID` is an optional parameter used by cloud apps to identify a head unit. The content of `CloudAppVehicleID` is up to the OEM's implementation. Possible values could be the VIN or a hashed VIN.
The `CloudAppVehicleID` value can be retrieved as part of the `GetVehicleData` RPC. To find out more about how to retrieve `CloudAppVehicleID`, check out the Getting Vehicle Data section.

Proguard Guidelines

SmartDeviceLink and its dependent libraries are open source and not intended to be obfuscated. When using Proguard in an app that integrates SmartDeviceLink, it is necessary to follow these guidelines.

Required Proguard Rules

Apps that are code shrinking a release build with Proguard typically have a section resembling this snippet in their `build.gradle`:

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                    'proguard-rules.pro'
        }
    }
    ...
}
```

Developers using Proguard in this manner should be sure to include the following lines in their `proguard-rules.pro` file:

```
-keep class com.smartdevicelink.** { *; }
-keep class com.livio.** { *; }
# Video streaming apps must add the following line
-keep class ** extends com.smartdevicelink.streaming.video.
SdlRemoteDisplay { *; }
```

> NOTE
>
> Failure to include these Proguard rules may result in a failed build or cause issues during runtime.

Creating an App Service

App services is a powerful feature enabling both a new kind of vehicle-to-app communication and app-to-app communication via SDL.

App services are used to publish navigation, weather and media data (such as temperature, navigation waypoints, or the current playlist name). This data can then be used by both the vehicle head unit and, if the publisher of the app service desires, other SDL apps.

Vehicle head units may use these services in various ways. One app service for each type will be the "active" service to the module. For media, for example, this will be the media app that the user is currently using or listening to. For navigation, it would be a navigation app that the user is using to navigate. For weather, it may be the last used weather app, or a user-selected default. The system may then use that service's data to perform various actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

An SDL app can also subscribe to a published app service. Once subscribed, the app will be sent the new data when the app service publisher updates its data. To find out more about how to subscribe to an app service check out the Using App Services section. Subscribed apps can also send certain RPCs and generic URI-based actions (see the section Supporting App Actions, below) to your service.

Currently, there is no high-level API support for publishing an app service, so you will have to use raw RPCs for all app service related APIs.

Using an app service is covered in another guide.

App Service Types

Apps are able to declare that they provide an app service by publishing an app service manifest. Three types of app services are currently available, and more will be made available over time. The currently available types are: Media, Navigation, and Weather. An app may publish multiple services (one each for different service types), if desired.

Publishing an App Service

Publishing a service is a several step process. First, create your app service manifest. Second, publish your app service using your manifest. Third, publish your service data using `OnAppServiceData`. Fourth, respond to `GetAppServiceData` requests. Fifth, you should support RPCs related to your service. Last, optionally, you can support URI based app actions.

1. Creating an App Service Manifest

The first step to publishing an app service is to create an `AppServiceManifest` object. There is a set of generic parameters you will need to fill out as well as service type specific parameters based on the app service type you are creating.

JAVA

```
AppServiceManifest manifest = new AppServiceManifest(
AppServiceType.MEDIA.toString());
manifest.setServiceName("My Media App"); // Must be unique across
app services.
manifest.setServiceIcon(new Image("Service Icon Name", ImageType.
DYNAMIC)); // Previously uploaded service icon. This could be the same
as your app icon.
manifest.setAllowAppConsumers(true); // Whether or not other apps
can view your data in addition to the head unit. If set to `NO` only the
head unit will have access to this data.
manifest.setRpcSpecVersion(new SdlMsgVersion(5,0)); // An *optional*
parameter that limits the RPC spec versions you can understand to the
provided version *or below*.
manifest.setHandledRpcs(List<FunctionID>); // If you add function ids
to this *optional* parameter, you can support newer RPCs on older
head units (that don't support those RPCs natively) when those RPCs
are sent from other connected applications.
manifest.setMediaServiceManifest(<#Code#>); // Covered Below
```

CREATING A MEDIA SERVICE MANIFEST

Currently, there's no information you have to provide in your media service manifest! You'll just have to create an empty media service manifest and set it into your general app service manifest.

JAVA

```
MediaServiceManifest mediaManifest = new MediaServiceManifest();
manifest.setMediaServiceManifest(mediaManifest);
```

CREATING A NAVIGATION SERVICE MANIFEST

You will need to create a navigation manifest if you want to publish a navigation service. You will declare whether or not your navigation app will accept waypoints. That is, if your app will support receiving *multiple* points of navigation (e.g. go to this McDonalds, then this Walmart, then home).

JAVA

```
NavigationServiceManifest navigationManifest = new
NavigationServiceManifest();
navigationManifest.setAcceptsWayPoints(true);
manifest.setNavigationServiceManifest(navigationManifest);
```

CREATING A WEATHER SERVICE MANIFEST

You will need to create a weather service manifest if you want to publish a weather service. You will declare the types of data your service provides in its `WeatherServiceData`.

JAVA

```
WeatherServiceManifest weatherManifest = new
WeatherServiceManifest();
weatherManifest.setCurrentForecastSupported(true);
weatherManifest.setMaxMultidayForecastAmount(10);
weatherManifest.setMaxHourlyForecastAmount(24);
weatherManifest.setMaxMinutelyForecastAmount(60);
weatherManifest.setWeatherForLocationSupported(true);
manifest.setWeatherServiceManifest(weatherManifest);
```

2. Publish Your Service

Once you have created your service manifest, publishing your app service is simple.

JAVA

```
PublishAppService publishServiceRequest = new PublishAppService();
publishServiceRequest.setAppServiceManifest(manifest);
publishServiceRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Error Handling#>
    }
});
sdlManager.sendRPC(publishServiceRequest);
```

Once you have your publish app service response, you will need to store the information provided in its `appServiceRecord` property. You will need the information later when you want to update your service's data.

WATCHING FOR APP RECORD UPDATES

As noted in the introduction to this guide, one service for each type may become the "active" service. If your service is the active service, your `AppServiceRecord` parameter `serviceActive` will be updated to note that you are now the active service.

After the initial app record is passed to you in the `PublishAppServiceResponse`, you will need to be notified of changes in order to observe whether or not you have become the active service. To do so, you will have to observe the new `SystemCapabilityType.APP_SERVICES` using `GetSystemCapability` and `OnSystemCapabilityUpdated`.

For more information, see the Using App Services guide and see the "Getting and Subscribing to Services" section.

3. Update Your Service's Data

After your service is published, it's time to update your service data. First, you must send an `onAppServiceData` RPC notification with your updated service data. RPC notifications are different than RPC requests in that they will not receive a response from the connected head unit.

> NOTE
>
> You should only update your service's data when you are the active service; service consumers will only be able to see your data when you are the active service.

First, you will have to create an `MediaServiceData`, `NavigationServiceData` or `WeatherServiceData` object with your service's data. Then, add that service-specific data object to an `AppServiceData` object. Finally, create an `OnAppServiceData` notification, append your `AppServiceData` object, and send it.

MEDIA SERVICE DATA

JAVA

```
MediaServiceData mediaData = new MediaServiceData();
mediaData.setMediaTitle("Some media title");
mediaData.setMediaArtist("Some media artist");
mediaData.setMediaAlbum("Some album");
mediaData.setPlaylistName("Some playlist");
mediaData.setIsExplicit(true);
mediaData.setTrackPlaybackProgress(45);
mediaData.setQueuePlaybackDuration(90);
mediaData.setTrackPlaybackProgress(45);
mediaData.setQueuePlaybackDuration(150);
mediaData.setQueueCurrentTrackNumber(2);
mediaData.setQueueTotalTrackCount(3);

AppServiceData appData = new AppServiceData();
appData.setServiceID(myServiceId);
appData.setServiceType(AppServiceType.MEDIA.toString());
appData.setMediaServiceData(mediaData);

OnAppServiceData onAppData = new OnAppServiceData();
onAppData.setServiceData(appData);

sdlManager.sendRPC(onAppData);
```

NAVIGATION SERVICE DATA

JAVA

```
final SdlArtwork navInstructionArt = new SdlArtwork("turn", FileType.
GRAPHIC_PNG, R.drawable.turn, true);

sdlManager.getFileManager().uploadFile(navInstructionArt, new
CompletionListener() { // We have to send the image to the system
before it's used in the app service.
    @Override
    public void onComplete(boolean success) {
        if (success){
            Coordinate coordinate = new Coordinate(42f,43f);

            LocationDetails locationDetails = new LocationDetails();
            locationDetails.setCoordinate(coordinate);

            NavigationInstruction navigationInstruction = new
NavigationInstruction(locationDetails, NavigationAction.TURN);
            navigationInstruction.setImage(navInstructionArt.getImageRPC
());

            DateTime dateTime = new DateTime();
            dateTime.setHour(2);
            dateTime.setMinute(3);
            dateTime.setSecond(4);

            NavigationServiceData navigationData = new
NavigationServiceData(dateTime);
            navigationData.setInstructions(Collections.singletonList(
navigationInstruction));

            AppServiceData appData = new AppServiceData();
            appData.setServiceID(myServiceId);
            appData.setServiceType(AppServiceType.NAVIGATION.toString
());
            appData.setNavigationServiceData(navigationData);

            OnAppServiceData onAppData = new OnAppServiceData();
            onAppData.setServiceData(appData);

            sdlManager.sendRPC(onAppData);
        }
    }
});
```

WEATHER SERVICE DATA

JAVA

```
final SdlArtwork weatherImage = new SdlArtwork("sun", FileType.
GRAPHIC_PNG, R.drawable.sun, true);

sdlManager.getFileManager().uploadFile(weatherImage, new
CompletionListener() { // We have to send the image to the system
before it's used in the app service.
    @Override
    public void onComplete(boolean success) {
        if (success) {

            WeatherData weatherData = new WeatherData();
            weatherData.setWeatherIcon(weatherImage.getImageRPC());

            Coordinate coordinate = new Coordinate(42f, 43f);

            LocationDetails locationDetails = new LocationDetails();
            locationDetails.setCoordinate(coordinate);

            WeatherServiceData weatherServiceData = new
WeatherServiceData(locationDetails);

            AppServiceData appData = new AppServiceData();
            appData.setServiceID(myServiceId);
            appData.setServiceType(AppServiceType.WEATHER.toString());
            appData.setWeatherServiceData(weatherServiceData);

            OnAppServiceData onAppData = new OnAppServiceData();
            onAppData.setServiceData(appData);

            sdlManager.sendRPC(onAppData);
        }
    }
});
```

4. Handling App Service Subscribers

If you choose to make your app service available to other apps, you will have to handle requests to get your app service data when a consumer requests it directly.

Handling app service subscribers is a two step process. First, you must register for notifications from the subscriber. Then, when you get a request, you will either have to send a response to the subscriber with the app service data or if you have no data to send, send a reponse with a relevant failure result code.

LISTENING FOR REQUESTS

First, you will need to register for updates for when a `GetAppServiceDataRequest` is received by your application.

JAVA

```
// Get App Service Data Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.
GET_APP_SERVICE_DATA, new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        <#Handle Request#>
    }
});
```

SENDING A RESPONSE TO SUBSCRIBERS

Second, you need to respond to the request when you receive it with your app service data. This means that you will need to store your current service data

after your most recent update using `OnAppServiceData` (see the section Updating Your Service Data).

JAVA

```
// Get App Service Data Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.
GET_APP_SERVICE_DATA, new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        GetAppServiceData getAppServiceData = (GetAppServiceData)
request;

        GetAppServiceDataResponse response = new
GetAppServiceDataResponse();
        response.setSuccess(true);
        response.setCorrelationID(getAppServiceData.getCorrelationID());
        response.setResultCode(Result.SUCCESS);
        response.setInfo("<#Use to provide more information about an
error#>");
        response.setServiceData(<#Your App Service Data#>);

        sdlManager.sendRPC(response);
    }
});
```

Supporting Service RPCs and Actions

5. Service RPCs

Certain RPCs are related to certain services. The chart below shows the current relationships:

MEDIA	NAVIGATION	WEATHER
ButtonPress (OK)	SendLocation	
ButtonPress (SEEKLEFT)	GetWayPoints	
ButtonPress (SEEKRIGHT)	SubscribeWayPoints	
ButtonPress (TUNEUP)	OnWayPointChange	
ButtonPress (TUNEDOWN)		
ButtonPress (SHUFFLE)		
ButtonPress (REPEAT)		

When you are the active service for your service's type (e.g. media), and you have declared that you support these RPCs in your manifest (see section 1. Creating an App Service Manifest), then these RPCs will be automatically routed to your app. You will have to set up listeners to be aware that they have arrived, and you will then need to respond to those requests.

JAVA

```
AppServiceManifest manifest = new AppServiceManifest(
AppServiceType.MEDIA.toString());
...
manifest.setHandledRpcs(Collections.singletonList(FunctionID.
BUTTON_PRESS.getId()));
```

JAVA

```
sdlManager.addOnRPCRequestListener(FunctionID.BUTTON_PRESS, new
 OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        ButtonPress buttonPress = (ButtonPress) request;

        ButtonPressResponse response = new ButtonPressResponse();
        response.setSuccess(true);
        response.setResultCode(Result.SUCCESS);
        response.setCorrelationID(buttonPress.getCorrelationID());
        response.setInfo("<#Use to provide more information about an
error#>");
        sdlManager.sendRPC(response);
    }
});
```

6. Service Actions

App actions are the ability for app consumers to use the SDL services system to send URIs to app providers in order to activate actions on the provider. Service actions are *schema-less*, i.e. there is no way to define the appropriate URIs through SDL. If you already provide actions through your app and want to expose them to SDL, or if you wish to start providing them, you will have to document your available actions elsewhere (such as your website).

In order to support actions through SDL services, you will need to observe and respond to the `PerformAppServiceInteraction` RPC request.

JAVA

```java
// Perform App Services Interaction Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.
PERFORM_APP_SERVICES_INTERACTION, new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        PerformAppServiceInteraction performAppServiceInteraction = (
PerformAppServiceInteraction) request;

        // If you have multiple services, this will let you know which of
your services is being addressed
        serviceID = performAppServiceInteraction.getServiceID();

        // The URI sent by the consumer. This must be something you
understand
        String serviceURI = performAppServiceInteraction.getServiceUri();

        // A result you want to send to the consumer app.
        PerformAppServiceInteractionResponse response = new
PerformAppServiceInteractionResponse();
        response.setServiceSpecificResult("Some Result");
        response.setCorrelationID(performAppServiceInteraction.
getCorrelationID());
        response.setInfo("<#Use to provide more information about an
error#>");
        response.setSuccess(true);
        response.setResultCode(Result.SUCCESS);
        sdlManager.sendRPC(response);
    }
});
```

Using App Services

App services is a powerful feature enabling both a new kind of vehicle-to-app communication and app-to-app communication via SDL.

App services are used to publish navigation, weather and media data (such as temperature, navigation waypoints, or the current playlist name). This data can

then be used by both the vehicle head unit and, if the publisher of the app service desires, other SDL apps. Creating an app service is covered in another guide.

Vehicle head units may use these services in various ways. One app service for each type will be the "active" service to the module. For media, for example, this will be the media app that the user is currently using or listening to. For navigation, it would be a navigation app that the user is using to navigate. For weather, it may be the last used weather app, or a user-selected default. The system may then use that service's data to perform various actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

An SDL app can also subscribe to a published app service. Once subscribed, the app will be sent the new data when the app service publisher updates its data. This guide will cover subscribing to a service. Subscribed apps can also send certain RPCs and generic URI-based actions (see the section Supporting App Actions, below) to your service.

Currently, there is no high-level API support for using an app service, so you will have to use raw RPCs for all app service related APIs.

Getting and Subscribing to Services

Once your app has connected to the head unit, you will first want to be notified of all available services and updates to the metadata of all services on the head unit. Second, you will narrow down your app to subscribe to an individual app

service and subscribe to its data. Third, you may want to interact with that service through RPCs, or fourth, through service actions.

1. Getting and Subscribing to Available Services

To get information on all services published on the system, as well as on changes to published services, you will use the `SystemCapabilityManager` to get the information. Because this information is initially available asynchronosly, we have to attach an `OnSystemCapabilityListener` to the `getCapability` request.

JAVA

```
// Grab the capability once
sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.APP_SERVICES, new OnSystemCapabilityListener
() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (
AppServicesCapabilities) capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});

...

// Subscribe to updates
sdlManager.getSystemCapabilityManager().
addOnSystemCapabilityListener(SystemCapabilityType.APP_SERVICES,
new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (
AppServicesCapabilities) capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});
```

CHECKING THE APP SERVICE CAPABILITY

Once you've retrieved the initial list of app service capabilities (in the `GetSystemCapability` response), or an updated list of app service capabilities (from the `OnSystemCapabilityUpdated` notification), you may want to inspect the data

to find what you are looking for. Below is example code with comments explaining what each part of the app service capability is used for.

JAVA

```
// This array contains all currently available app services on the system
List<AppServiceCapability> appServices = servicesCapabilities.
getAppServices();

if (appServices!= null && appServices.size() > 0) {
    for (AppServiceCapability anAppServiceCapability : appServices) {
        // This will tell you why a service is in the list of updates
        ServiceUpdateReason updateReason = anAppServiceCapability.
getUpdateReason();

        // The app service record will give you access to a service's
generated id, which can be used to address the service directly (see
below), it's manifest, used to see what data it supports, whether or not
the service is published (it always will be here), and whether or not the
service is the active service for its service type (only one service can
be active for each type)
        AppServiceRecord serviceRecord = anAppServiceCapability.
getUpdatedAppServiceRecord();
    }
}
```

2. Getting and Subscribing to a Service Type's Data

Once you have information about all of the services available, you may want to view or subscribe to a service type's data. To do so, you will use the `GetAppServiceData` RPC.

Note that you will currently only be able to get data for the *active* service of the service type. You can attempt to make another service the active service by using the `PerformAppServiceInteraction` RPC, discussed below in "Sending an Action to a Service Provider."

JAVA

```java
// Get service data once
GetAppServiceData getAppServiceData = new GetAppServiceData(
AppServiceType.MEDIA.toString());

// Subscribe to future updates if you want them
getAppServiceData.setSubscribe(true);

getAppServiceData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response != null){
            GetAppServiceDataResponse serviceResponse = (
GetAppServiceDataResponse) response;
            MediaServiceData mediaServiceData = serviceResponse.
getServiceData().getMediaServiceData();
        }
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <# Handle Error #>
    }
});
sdlManager.sendRPC(getAppServiceData);

...

// Unsubscribe from updates
GetAppServiceData unsubscribeServiceData = new GetAppServiceData
(AppServiceType.MEDIA.toString());
unsubscribeServiceData.setSubscribe(false);
sdlManager.sendRPC(unsubscribeServiceData);
```

Interacting with a Service Provider

Once you have a service's data, you may want to interact with a service provider by sending RPCs or actions.

3. Sending RPCs to a Service Provider

Only certain RPCs are available to be passed to the service provider based on their service type. See the Creating an App Service guide (under the "Supporting Service RPCs and Actions" section) for a chart detailing which RPCs work with which service types. The RPC can only be sent to the active service of a specific service type, not to any inactive service.

Sending an RPC works exactly the same as if you were sending the RPC to the head unit system. The head unit will simply route your RPC to the appropriate app automatically.

> NOTE
>
> Your app may need special permissions to use the RPCs that route to app service providers.

JAVA

```
ButtonPress buttonPress = new ButtonPress();
buttonPress.setButtonPressMode(ButtonPressMode.SHORT);
buttonPress.setButtonName(ButtonName.OK);
buttonPress.setModuleType(ModuleType.AUDIO);
buttonPress.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle the Error#>
    }
});
sdlManager.sendRPC(buttonPress);
```

4. Sending an Action to a Service Provider

Actions are generic URI-based strings sent to any app service (active or not). You can also use actions to request to the system that they make the service the active service for that service type. Service actions are *schema-less*, i.e. there is no way to define the appropriate URIs through SDL. The service provider must document their list of available actions elsewhere (such as their website).

JAVA

```
PerformAppServiceInteraction performAppServiceInteraction = new
PerformAppServiceInteraction("sdlexample://x-callback-url/showText?x-
source=MyApp&text=My%20Custom%20String","<#Previously
Retrieved ServiceID#>","<#Your App Id#>");
performAppServiceInteraction.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle the Error#>
    }
});
sdlManager.sendRPC(performAppServiceInteraction);
```

Updating to 4.4 (Upgrading To Multiplexing)

This guide is to help developers get setup with the SDL Android library 4.4. Upgrading apps to utilize the multiplexing transport flow will require us to do a few steps. This guide will assume the SDL library is already integrated into the app.

We will make changes to:

- SdlService
- SdlRouterService **(new)**
- SdlBroadcastReceiver
- MainActivity

SmartDeviceLink Service

The SmartDeviceLink proxy object instantiation needs to change to the new constructor. We also need to check for a boolean extra supplied through the intent that started the service.

The old instantiation should look similar to this:

```
proxy = new SdlProxyALM(this, APP_NAME, true, APP_ID);
```

The new constructor should look like this

```
public class SdlService extends Service implements IProxyListenerALM
{

  //...

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        boolean forceConnect = intent !=null && intent.getBooleanExtra(
TransportConstants.FORCE_TRANSPORT_CONNECTED, false);
        if (proxy == null) {
            try {
                //Create a new proxy using Bluetooth transport
                //The listener, app name,
                //whether or not it is a media app and the applicationId are
supplied.
                proxy = new SdlProxyALM(this.getBaseContext(),this,
APP_NAME, true, APP_ID);
            } catch (SdlException e) {
                //There was an error creating the proxy
                if (proxy == null) {
                //Stop the SdlService
                    stopSelf();
                }
            }
        }else if(forceConnect){
            proxy.forceOnConnected();
        }

        //use START_STICKY because we want the SDLService to be
explicitly started and stopped as needed.
        return START_STICKY;
    }
```

Notice we now gather the extra boolean from the intent and add to our if-else statement. If the proxy is not null, we need to check if the supplied boolean extra is true and if so, take action.

```
    if (proxy == null) {
      //...
    }else if(forceConnect){
        proxy.forceOnConnected();
    }
```

SmartDeviceLink Router Service (New)

The SdlRouterService will listen for a bluetooth connection with an SDL enabled module. When a connection happens, it will alert all SDL enabled apps that a connection has been established and they should start their SDL services.

We must implement a local copy of the SdlRouterService into our project. The class doesn't need any modification, it's just important that we include it. We will extend the `com.smartdevicelink.transport.SdlRouterService` in our class named `SdlRouterService`:

> NOTE
>
> Do not include an import for `com.smartdevicelink.transport.SdlRouterService`. Otherwise, we will get an error for `'SdlRouterService' is already defined in this compilation unit`.

```
public class SdlRouterService extends  com.smartdevicelink.transport.SdlRouterService {
//Nothing to do here
}
```

MUST

The local extension of the `com.smartdevicelink.transport.SdlRouterService` must be named `SdlRouterService`.

MUST

Make sure this local class (SdlRouterService.java) is in the same package of SdlReceiver.java (described below)

If you created the service using the Android Studio template then the service should have been added to your `AndroidManifest.xml` otherwise the service needs to be added in the manifest. Because we want our service to be seen by other SDL enabled apps, we need to set `android:exported="true"`. The system may issue a lint warning because of this, so we can suppress that using `tools:ignore="ExportedService"`. Once added, it should be defined like below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

    ...

        <service
            android:name=
"com.company.mySdlApplication.SdlRouterService"
            android:exported="true"
            android:process="com.smartdevicelink.router"
            tools:ignore="ExportedService">
        </service>

    </application>

    ...

</manifest>
```

MUST

The `SdlRouterService` must be placed in a separate process with the name `com.smartdevicelink.router`. If it is not in that process during it's start up it will stop itself.

SmartDeviceLink Broadcast Receiver

The SmartDeviceLink Android Library now includes a base BroadcastReceiver that needs to be used. It's called `SdlBroadcastReceiver`. Our old

BroadcastReceiver will just need to extend this class instead of the Android BroadcastReceiver. Two abstract methods will be automatically populate the class, we will fill them out soon.

```
public class SdlReceiver extends SdlBroadcastReceiver {

    @Override
    public void onSdlEnabled(Context context, Intent intent) {...}

    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {...}

}
```

Next, we want to make sure we supply our instance of the SdlBroadcastService with our local copy of the SdlRouterService. We do this by simply returning the class object in the method defineLocalSdlRouterClass:

```
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
        //Return a local copy of the SdlRouterService located in your
project
        return com.company.mySdlApplication.SdlRouterService.class;
    }
```

We want to start the SDL Proxy when an SDL connection is made via the `SdlRouterService`. This is likely code included on the `onReceive` method call previously. We do this by taking action in the onSdlEnabled method:

NOTE

The actual package definition for the SdlRouterService might be different. Just make sure to return your local copy and not the class object from the library itself.

```
public class SdlReceiver extends SdlBroadcastReceiver {

   @Override
    public void onSdlEnabled(Context context, Intent intent) {
       //Use the provided intent but set the class to the SdlService
       intent.setClass(context, SdlService.class);
       context.startService(intent);

    }


    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
       //Return a local copy of the SdlRouterService located in your
project.
       return com.company.mySdlApplication.SdlRouterService.class;
    }
}
```

NOTE

The `onSdlEnabled` method will be the main start point for our SDL connection session. We define exactly what we want to happen when we find out we are connected to SDL enabled hardware.

MUST

SdlBroadcastReceiver must call super if `onReceive` is overridden

```
@Override
public void onReceive(Context context, Intent intent) {
    super.onReceive(context, intent);
    //your code here
}
```

Now we need to add two extra intent actions to or our intent filter for the SdlBroadcastReceiver:

- android.bluetooth.adapter.action.STATE_CHANGED
- **sdl.router.startservice**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

        ...

        <receiver
            android:name=".SdlReceiver"
            android:exported="true"
            android:enabled="true">

            <intent-filter>
                <action android:name=
"android.bluetooth.device.action.ACL_CONNECTED" />
                <action android:name=
"android.bluetooth.device.action.ACL_DISCONNECTED"/>
                <action android:name=
"android.bluetooth.adapter.action.STATE_CHANGED"/>
                <action android:name=
"android.media.AUDIO_BECOMING_NOISY" />
                <action android:name="sdl.router.startservice" />
            </intent-filter>

        </receiver>

    </application>

...

</manifest>
```

MUST

SdlBroadcastReceiver has to be exported, or it will not work correctly

Main Activity

Our previous MainActivity class probably looked similar to this:

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Start the SDLService
        Intent sdlServiceIntent = new Intent(this, SdlService.class);
        startService(sdlServiceIntent);
    }
}
```

However now instead of starting the service every time we launch the application we can do a query that will let us know if we are connected to SDL enabled hardware or not. If we are, the `onSdlEnabled` method in our SdlBroadcastReceiver will be called and the proper flow should start. We do this by removing the intent creation and startService call and instead replace them with a single call to `SdlReceiver.queryForConnectedService(Context)`.

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //If we are connected to a module we want to start our SdlService
        SdlReceiver.queryForConnectedService(this);
    }
}
```

Updating from 4.4 to 4.5

This guide is to help developers get setup with the SDL Android library 4.5. It is assumed that the developer is already updated to 4.4 of the library. There are a few very important changes that we need to make to the integration to keep things working well. The first is a few new additions to the AndroidManifest.xml and the `SdlRouterService` entry. Next, we have to prepare for Android Oreo's push towards foreground services.

We will make changes to:

- AndroidManifest.xml
- SdlService
- SdlBroadcastReceiver

AndroidManifest.xml Updates

Assuming the manifest was up to date with version 4.4 requirements we need to add an intent-filter and a meta-data item. The entire entry should look as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

    ...

        <service
            android:name=
"com.company.mySdlApplication.SdlRouterService"
            android:exported="true"
            android:process="com.smartdevicelink.router"
            tools:ignore="ExportedService">
            <intent-filter>
                <action android:name="com.smartdevicelink.router.service"/
>
            </intent-filter>
            <meta-data android:name="sdl_router_version"
android:value="@integer/sdl_router_service_version_value" />
        </service>

    </application>

    ...

</manifest>
```

Intent Filter

```
<intent-filter>
    <action android:name="com.smartdevicelink.router.service"/>
</intent-filter>
```

The new versions of the SDL Android library rely on the `com.smartdevicelink.router.service` action to query SDL enabled apps that host router services. This allows the library to determine which router service to start.

MUST

This `intent-filter` MUST be included.

Metadata

ROUTER SERVICE VERSION

```
<meta-data android:name="sdl_router_version"  android:value=
"@integer/sdl_router_service_version_value" />
```

Adding the `sdl_router_version` metadata allows the library to know the version of the router service that the app is using. This makes it simpler for the library to choose the newest router service when multiple router services are available.

CUSTOM ROUTER SERVICE

```
<meta-data android:name="sdl_custom_router" android:value="false" /
>
```

NOTE

This is only for specific OEM applications, therefore normal developers do not need to worry about this.

Some OEMs choose to implement custom router services. Setting the `sdl_custom_router` metadata value to `true` means that the app is using something custom over the default router service that is included in the SDL Android library. Do not include this `meta-data` entry unless you know what you are doing.

Android Oreo's Push To Foreground Services

Previous versions of Android allowed our SDL app partners to start their SDL services in the background and attach themselves to the foregrounded SDL router service. Android Oreo (API 26) has changed that. Due to new OS limitations, apps must start their SDL service in the foreground.

What do I need to do?

There are a few changes to make, one in the `SdlBroadcastReceiver` and the other in the `SdlService` (or which service the proxy is implemented).

SDLBROADCASTRECEIVER

PREVIOUS VERSION

```
@Override
public void onSdlEnabled(Context context, Intent intent) {
    Log.d(TAG, "SDL Enabled");
    intent.setClass(context, SdlService.class);
    context.startService(intent);
}
```

SAMPLE UPDATE

```
@Override
public void onSdlEnabled(Context context, Intent intent) {
    Log.d(TAG, "SDL Enabled");
    intent.setClass(context, SdlService.class);
    if(Build.VERSION.SDK_INT < Build.VERSION_CODES.O) {
        context.startService(intent);
    }else{
        context.startForegroundService(intent);
    }

}
```

This means the app will start the SDL service in the background if we are on a device that uses Android N or earlier. If the app is running on Android Oreo or newer, the service will make a promise to the OS that the service will move into the foreground. If the service doesn't explicitly move into the foreground an exception will be thrown.

SDLSERVICE (OR SIMILAR)

Within the `SdlService` class or similar you will need to add a call to start the service in the foreground. This will include creating a notification to sit in the

status bar tray. This information and icons should be relevant for what the service is doing/going to do. If you already start your service in the foreground, you can ignore this section.

```
public void onCreate() {
    super.onCreate();
    ...

    NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
     notificationManager.createNotificationChannel(...);
     Notification serviceNotification = new Notification.Builder(this, *
Notification Channel*)
          .setContentTitle(...)
          .setSmallIcon(....)
          .setLargeIcon(...)
          .setContentText(...)
          .setChannelId(channel.getId())
          .build();
     startForeground(id, serviceNotification);
}
```

EXITING THE FOREGROUND

It's important that you don't leave you notification in the notification tray as it is very confusing to users. So in the `onDestroy` method in your service, simply call the `stopForeground` method.

```
@Override
public void onDestroy(){
    //...
    if(Build.VERSION.SDK_INT>=Build.VERSION_CODES.O){
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        if(notificationManager!=null){ //If this is the only notification on
your channel
            notificationManager.deleteNotificationChannel(* Notification
Channel*);
        }
        stopForeground(true);
    }
}
```

Notification Suggestions

We realize that pushing a notification to the notification tray is not ideal for any apps, but with Android's push for more transparency to users it's important that we don't try to workaround that. Android is getting stricter with their guidelines and could potentially prevent apps from being released if they are found to be not adhering to these rules.

THE CORRECT WAY

The right way to handle the new foreground service requirement is to simply push a full fledged notification to the notification tray.

How to do it

```
    @Override
    public void onCreate() {
        super.onCreate();
        ...
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationManager notificationManager = (NotificationManager
) getSystemService(Context.NOTIFICATION_SERVICE);
            NotificationChannel channel = new NotificationChannel(
"MyApp", "SdlService", NotificationManager.IMPORTANCE_DEFAULT);
            notificationManager.createNotificationChannel(channel);
            Notification serviceNotification = new Notification.Builder(this,
channel.getId())
                    .setContentTitle("MyApp is connected through SDL")
                    .setSmallIcon(R.drawable.ic_launcher_foreground)
                    .build();
            startForeground(id, serviceNotification);
        }
    }
```

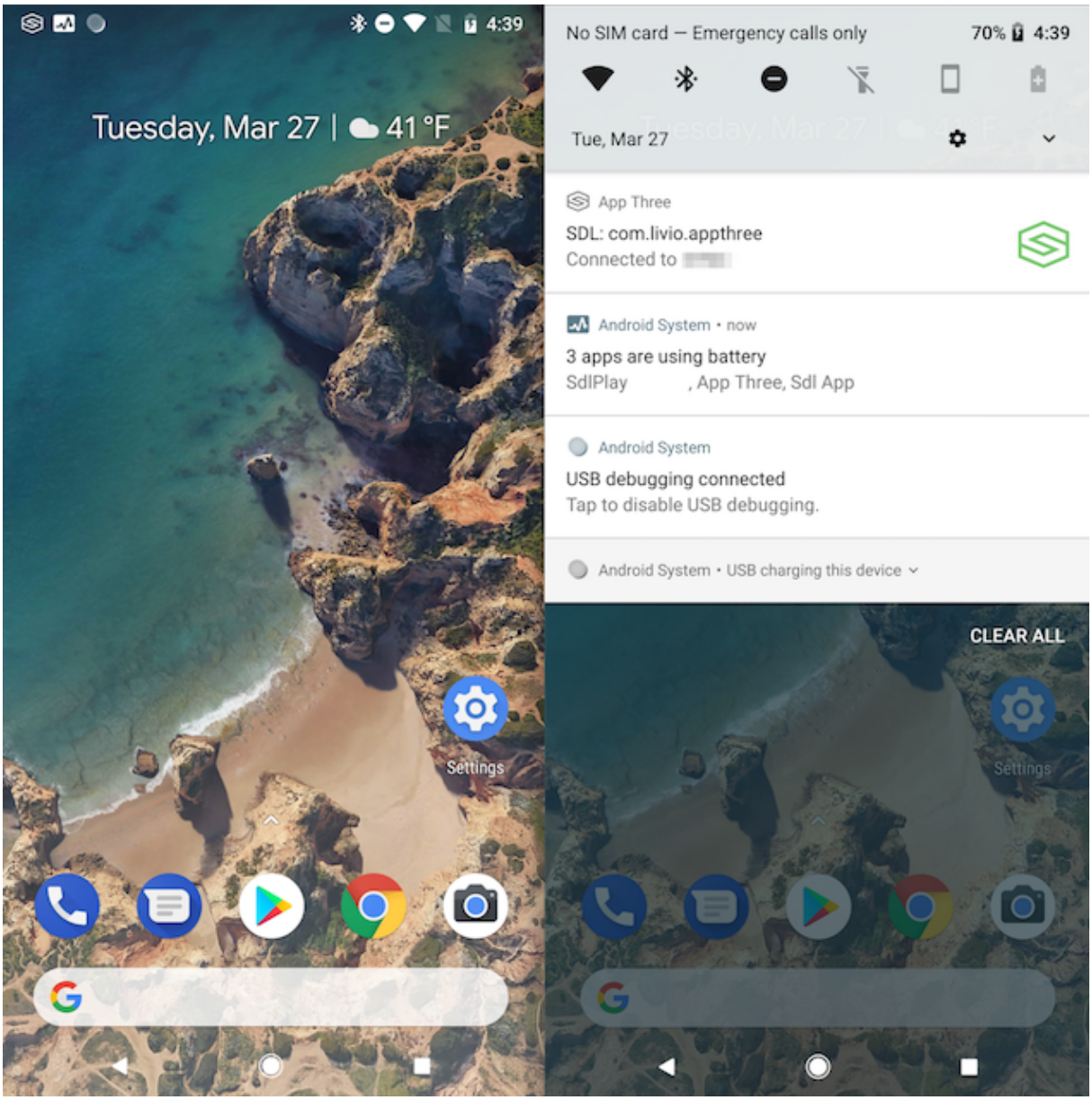
THE NOT SO CORRECT WAY

Currently Android Oreo allows a notification to be used that has not declared a notification channel. This results in the notification icon not actually appearing on its own. Instead it is grouped together into the notification channel that reads "# apps are using battery" from the Android System. This is likely to prevent breaking changes from apps that have not updated their integration to Android Oreo, however, we fully anticipate this to be changed in the future so it is not recommended.

How to do it

```
    @Override
    public void onCreate() {
        super.onCreate();
        ...
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            Notification serviceNotification = new Notification.Builder(this,
"NoChannel")
                    .setContentTitle("MyApp is connected through SDL")
                    .setSmallIcon(R.drawable.ic_launcher_foreground)
                    .build();
            startForeground(id, serviceNotification);
        }
    }
```

How it looks



THE ABSOLUTELY NOT CORRECT WAY

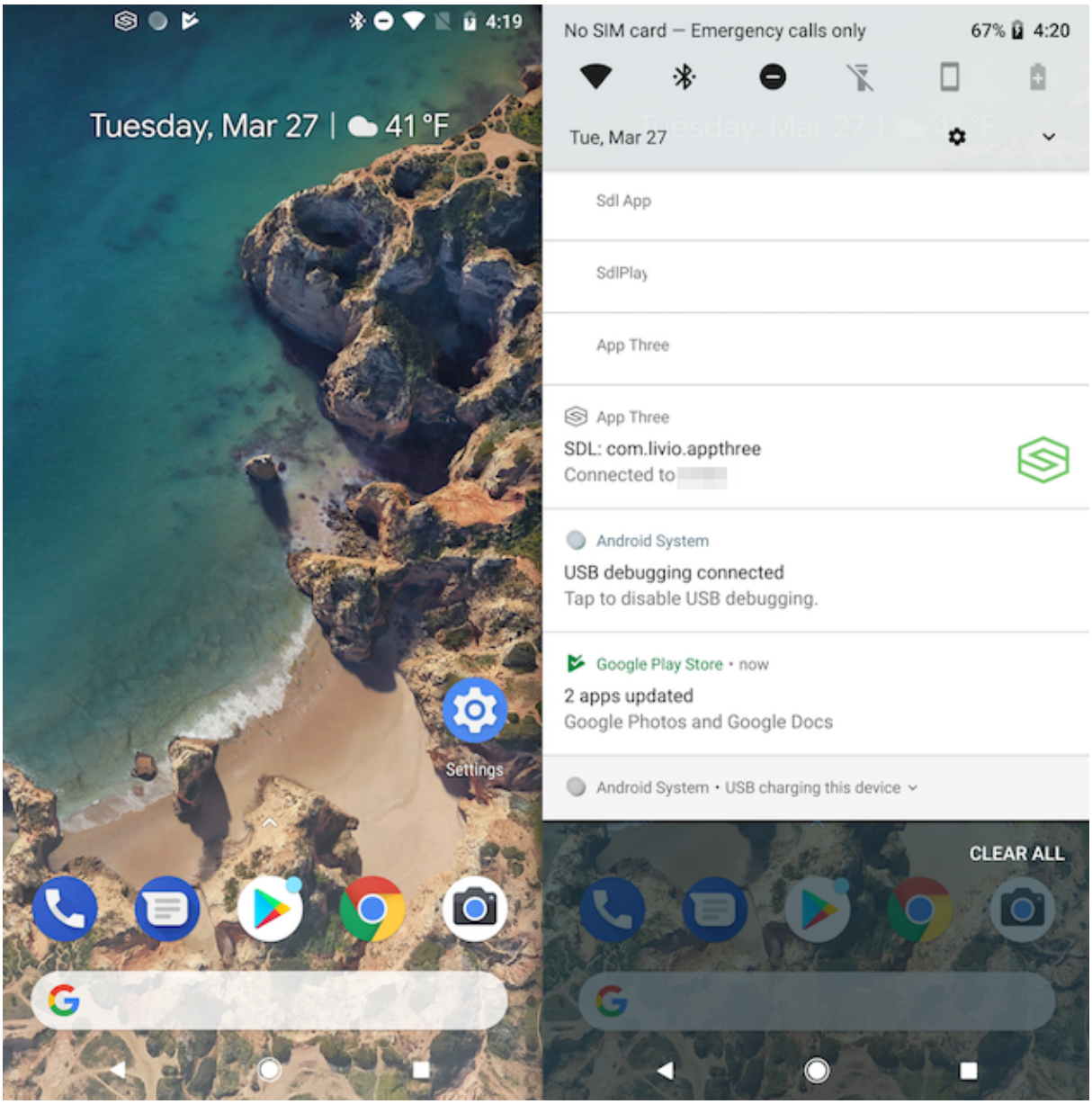
It is possible to create a somewhat invisible notification. This will appear to just be blank space in the notification tray. With adding minimal content to the notification when the user pulls down the tray it will have a very small footprint on the screen. However, this is completely disingenuous to the user and should not be considered a solution. Android will most likely see this as bad behavior

and could prevent you from releasing your app or even remove your app from the play store with a ban included. Don't do this.

How to do it

```
    @Override
    public void onCreate() {
        super.onCreate();
        ...
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationManager notificationManager = (NotificationManager
) getSystemService(Context.NOTIFICATION_SERVICE);
            NotificationChannel channel = new NotificationChannel(
"MyApp", "SdlService", NotificationManager.IMPORTANCE_DEFAULT);
            notificationManager.createNotificationChannel(channel);
            Notification serviceNotification = new Notification.Builder(this,
channel.getId())
                    .setSmallIcon(R.drawable.sdl_tray_invis)
                    .build();
            startForeground(id, serviceNotification);
        }
    }
```

How it looks



Updating from 4.5 to 4.6

This guide is to help developers get setup with the SDL Android library 4.6. It is assumed that the developer is already updated to 4.5 of the library. There are a few important changes that we need to make to the integration to keep things working well. The first is removing some of the BroadcastReceiver's intent

filters in `AndroidManifest.xml` that are now unneccessary. Secondly, the gradle integration of our library should now use `implementation` instead of `compile`. Lastly, the `RPCRequestFactory` class has been deprecated and constructors with mandatory parameters have been added for each RPC class.

We will make changes to:

- AndroidManifest.xml
- build.gradle
- any usage of RPCRequestFactory

AndroidManifest.xml Updates

Assuming the manifest was up to date with version 4.5, we can now remove some of the intent-filters (`ACL_DISCONNECTED`, `STATE_CHANGED`, `AUDIO_BECOMING_NOISY`) for your app's BroadcastReceiver. The BroadcastReceiver section of the manifest should look as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.company.mySdlApplication">

    <application>

        ...

        <receiver
            android:name=".SdlReceiver"
            android:exported="true"
            android:enabled="true">

            <intent-filter>
                <action android:name=
"android.bluetooth.device.action.ACL_CONNECTED" />
                <action android:name="sdl.router.startservice" />
            </intent-filter>

        </receiver>

    </application>

...

</manifest>
```

Gradle Update

The previous way of including the libary via `compile` should now use `implementation`. The dependencies section of your app's `build.gradle` file should now appear as:

```
dependencies {
    implementation 'com.smartdevicelink:sdl_android:4.+'
}
```

Deprecation of RPCRequestFactory

The RPCRequestFactory has been deprecated in 4.6. To build RPC requests, developers should use the constructors in the desired RPC request class. For example, instead of using `RPCRequestFactory.buildAddCommand(...)` to build an `AddCommand` request, try the following:

```
AddCommand addCommand = new AddCommand(100);
addCommand.setMenuParams(new MenuParams("Skip"));
proxy.sendRPCRequest(addCommand);
```

Updating from 4.6 to 4.7

Overview

This guide is to help developers get setup with the SDL Android library version 4.7. It is assumed that the developer is already updated to 4.6 of the library. This version includes the addition of the SdlManagers and a re-working of the transports which greatly enhances the use of the `SdlRouterService`, along with adding the functionality for secondary transports on supporting versions of SDL Core.

In this guide we will be focusing on the transitioning from the proxy, which implemented `SdlProxyALM` into using the `SdlManager` system, which

includes specialized sub-managers that you can interact with through the `SdlManager`. We will follow the naming convention of the guides, highlighting the previous way of implementing SDL and showing the new ways of implementing it.

> NOTE
>
> Moving from the `SdlProxyALM` implementation to the `SdlManager` API will require you to manually subscribe to the notifications and responses that you wish to receive instead of all of the notifications and responses being passed through the `IProxyListenerALM` interface.

Integration Basics

The `SdlService` class will contain a great deal of changes as it acts as the main bridge to SDL functionality. There are going to be two main differences with how this class was set up in 4.6 versus 4.7.

Removal of IProxyListenerALM

Previously, your `SdlService` had to implement the `IProxyListenerALM` interface. This often added many unnecessary lines of code to the class due to the need to override all of its functions. The need to do this has been removed in 4.7 with the inclusion of the `SdlManager` APIs. Developers now only have to add the listeners they need.

4.6:

```
public class SdlService extends Service implements IProxyListenerALM
{

    // The proxy handles communication between the application and
SDL
    private SdlProxyALM proxy = null;

    //...

    @Override
    public void someListener(){}
    //...
}
```

4.7: THE REQUIREMENT TO IMPLEMENT `IPROXYLISTENERALM` IS REMOVED:

```
public class SdlService extends Service {

    // The SdlManager exposes the APIs needed to communicate
between the application and SDL
    private SdlManager sdlManager = null;

    //...
}
```

After removing `IProxyListenerALM` from the `SdlService`, all of its previously overridden functions will need to be removed. If your app used any of these callback methods, it will help to document which ones they were, as you will need to add in the listeners that you need using the `SdlManager`'s `addOnRPCNotificationListener`.

NOTE

When you start using the managers, you have to make sure that your app subscribes to notifications before sending the corresponding RPC requests and subscriptions or else some notifications may be missed.

Creation of SdlManager

As we no longer want to directly instantiate `SdlProxyALM`, we need to instantiate the `SdlManager` instead. This is best done using the `SdlManager.Builder` class using your application's details and configurations. In order to receive life cycle events from the `SdlManager`, an `SdlManagerListener` must be provided. The new code should resemble the following:

```
public class SdlService extends Service {

    //The manager handles communication between the application and
SDL
    private SdlManager sdlManager = null;

    //...

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        if (sdlManager == null) {
            MultiplexTransportConfig transport = new
MultiplexTransportConfig(this, APP_ID, MultiplexTransportConfig.
FLAG_MULTI_SECURITY_OFF);

            // The app type to be used
            Vector<AppHMIType> appType = new Vector<>();
            appType.add(AppHMIType.MEDIA);

            // The manager listener helps you know when certain events
that pertain to the SDL Manager happen
            SdlManagerListener listener = new SdlManagerListener() {

                @Override
                public void onStart() {
                    // RPC listeners and other functionality can be called once
this callback is triggered.
                }

                @Override
                public void onDestroy() {
                    SdlService.this.stopSelf();
                }

                @Override
                public void onError(String info, Exception e) {
                }
            };

            // Create App Icon, this is set in the SdlManager builder
            SdlArtwork appIcon = new SdlArtwork(ICON_FILENAME, FileType.
GRAPHIC_PNG, R.mipmap.ic_launcher, true);

            // The manager builder sets options for your session
            SdlManager.Builder builder = new SdlManager.Builder(this,
APP_ID, APP_NAME, listener);
            builder.setAppTypes(appType);
            builder.setTransportType(transport);
```

```
            builder.setAppIcon(appIcon);
            sdlManager = builder.build();
            sdlManager.start();
        }

    //...

}
```

Once you receive the `onStart` callback from `SdlManager`, you can add in your listeners and start adding UI elements. There will be more about adding the UI elements later. The last example in this section will be about adding specific listeners. Because we removed the `IProxyListenerALM` implementation, you will have to set listeners for the needs of your app.

Listening for RPC notifications and events

We can listen for specific events using `SdlManager`'s `addOnRPCNotificationListener`. These listeners can be added either in the `onStart()` callback of the `SdlManagerListener` or after it has been triggered. The following example shows how to listen for HMI Status notifications. Additional listeners can be added for specific RPCs by using their corresponding `FunctionID` in place of the `ON_HMI_STATUS` in the following example and casting the `RPCNotification` object to the correct type.

EXAMPLE OF A LISTENER FOR HMI STATUS:

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_HMI_STATUS,
new OnRPCNotificationListener() {
        @Override
        public void onNotified(RPCNotification notification) {
            OnHMIStatus status = (OnHMIStatus) notification;
            if (status.getHmiLevel() == HMILevel.HMI_FULL && ((
OnHMIStatus) notification).getFirstRun()) {
                // first time in HMI Full
            }
        }
    });
```

Sending RPCs

There are new method names and locations that mimic previous functionality for sending RPCs. These methods are located in the `SdlManager` and have the new names of `sendRPC`, `sendRPCs`, and `sendSequentialRPCs`.

4.6:

```
// single RPC
proxy.sendRPCRequest(request);

// muliple RPCs, non-sequential
proxy.sendRequests(rpcs, new OnMultipleRequestListener() {
    //...
});

// multiple RPCs, sequential
proxy.sendSequentialRequests(rpcs, new OnMultipleRequestListener() {
    //...
});
```

In 4.7, we use the `SdlManager` to send the requests.

4.7:

```
// single RPC
sdlManager.sendRPC(request);

// muliple RPCs, non-sequential
sdlManager.sendRPCs(rpcs, new OnMultipleRequestListener() {
    //...
});

// multiple RPCs, sequential
sdlManager.sendSequentialRPCs(rpcs, new OnMultipleRequestListener()
 {
    //...
});
```

Using AOA Protocol

If your app uses USB to connect to SDL, this update provides a very useful enhancement. AOA connections now work with the `SdlRouterService`. This means that multiple USB apps can be connected to the head unit at once.

SDLBROADCASTRECEIVER

Since the AOA transport will now use the multiplexing feature, it is important that your app correctly adds funcitonality for the `SdlRouterService`. This starts in the `SdlBroadcastReciever`.

4.6:

```
public class SdlReceiver extends com.smartdevicelink.
SdlBroadcastReceiver {

    @Override
    public void onSdlEnabled(Context context, Intent intent) {
        //Use the provided intent but set the class to your SdlService
        intent.setClass(context, SdlService.class);
        context.startService(intent);
    }

    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
         return null;
    }

}
```

4.7:

```
public class SdlReceiver extends com.smartdevicelink.
SdlBroadcastReceiver {

    @Override
    public void onSdlEnabled(Context context, Intent intent) {
        //Use the provided intent but set the class to your SdlService
        intent.setClass(context, SdlService.class);
        context.startService(intent);
    }

    @Override
    public Class<? extends SdlRouterService> defineLocalSdlRouterClass
() {
        // define your local router service. For example:
        return com.sdl.hellosdlandroid.SdlRouterService.class;
    }

}
```

SDLROUTERSERVICE

The `SdlRouterService` will listen for a connection with an SDL enabled module. When a connection happens, it will alert all SDL enabled apps that a connection has been established and they should start their SDL services.

4.6:

(No implementation required).

4.7:

We must implement a local copy of the `SdlRouterService` into our project. The class doesn't need any modification, it's just important that we include it. We will extend the `com.smartdevicelink.transport.SdlRouterService` in our class named `SdlRouterService`:

NOTE

Do not include an import for `com.smartdevicelink.transport.SdlRouterService`. Otherwise, we will get an error for `'SdlRouterService' is already defined in this compilation unit`.

```
public class SdlRouterService extends  com.smartdevicelink.transport.SdlRouterService {
//Nothing to do here
}
```

MUST

The local extension of the `com.smartdevicelink.transport.SdlRouterService` must be named `SdlRouterService`.

MUST

Make sure this local class (SdlRouterService.java) is in the same package of SdlReceiver.java

SDLSERVICE

4.6:

```
transport = new USBTransportConfig(getBaseContext(), (UsbAccessory)
 intent.getParcelableExtra(UsbManager.EXTRA_ACCESSORY), false, false
);
```

4.7:

```
MultiplexTransportConfig transport = new MultiplexTransportConfig(this,
 APP_ID, MultiplexTransportConfig.FLAG_MULTI_SECURITY_MED);
```

ADDITIONAL CONFIGURATIONS:

If your app requires high bandwidth transport, you can now specify that:

```
transport.setRequiresHighBandwidth(true);
```

NOTE

If your app only works when a high bandwidth transport is available, you should set `setRequiresHighBandwidth` to `true`. You cannot be certain that all core implementations support multiple transports. You could also set `TransportType.USB` as your only supported primary transport

Since the `SdlRouterService` now works with multiple transports, you can set your own configuration, for example:

```
static final List<TransportType> multiplexPrimaryTransports = Arrays.
asList(TransportType.USB, TransportType.BLUETOOTH);
static final List<TransportType> multiplexSecondaryTransports = Arrays
.asList(TransportType.TCP, TransportType.USB, TransportType.
BLUETOOTH);

//...

transport.setPrimaryTransports(multiplexPrimaryTransports);
transport.setSecondaryTransports(multiplexSecondaryTransports);
```

NOTE

Multiple transports only work on supported versions of SDL Core.

ANDROIDMANIFEST

4.6

```
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />

    <uses-feature android:name="android.hardware.usb.accessory"/>

    <service
        android:name=".SdlService"
       android:enabled="true"/>


    <receiver
        android:name=".SdlReceiver"
       android:enabled="true"
       android:exported="true"
       tools:ignore="ExportedReceiver">
       <intent-filter>
        <action android:name=
"com.smartdevicelink.USB_ACCESSORY_ATTACHED"/> <!--For AOA -->
          <action android:name="sdl.router.startservice" />
        </intent-filter>
    </receiver>

    <activity android:name=
"com.smartdevicelink.transport.USBAccessoryAttachmentActivity"
        android:launchMode="singleTop">
       <intent-filter>
           <action android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
       </intent-filter>

       <meta-data
           android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED"
         android:resource="@xml/accessory_filter" />
    </activity>
```

4.7

```
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.BLUETOOTH"/
>
    <uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name=
"android.permission.FOREGROUND_SERVICE" />

    <uses-feature android:name="android.hardware.usb.accessory"/>


        <service
        android:name=".SdlService"
        android:enabled="true"/>

    <service
        android:name="com.company.mySdlApplication.SdlRouterService"
        android:exported="true"
        android:process="com.smartdevicelink.router"
       tools:ignore="ExportedService">
       <intent-filter>
        <action android:name="com.smartdevicelink.router.service"/>
       </intent-filter>
       <meta-data android:name="sdl_router_version"  android:value=
"@integer/sdl_router_service_version_value" />
    </service>
    <receiver
        android:name=".SdlReceiver"
       android:enabled="true"
       android:exported="true"
       tools:ignore="ExportedReceiver">
       <intent-filter>
        <action android:name=
"com.smartdevicelink.USB_ACCESSORY_ATTACHED"/> <!--For AOA -->
         <action android:name=
"android.bluetooth.device.action.ACL_CONNECTED" />
         <action android:name="sdl.router.startservice" />
        </intent-filter>
    </receiver>

    <activity android:name=
"com.smartdevicelink.transport.USBAccessoryAttachmentActivity"
        android:launchMode="singleTop">
       <intent-filter>
           <action android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
       </intent-filter>

       <meta-data
```

```
            android:name=
"android.hardware.usb.action.USB_ACCESSORY_ATTACHED"
            android:resource="@xml/accessory_filter" />
    </activity>
```

Lock Screen

There has been a major overhaul for lock screens in 4.7. Complicated lock screen setups are no longer required due to the addition of the `LockScreenManager`. Instead of going over the previous lock screen tutorial and then writing another one I will give brief instructions on how to either continue using your lock screen implementation, or upgrading to the new managed system. This review is brief, it is recommended that you look at the full lock screen guide

USING YOUR CURRENT IMPLEMENTATION

If you would like to keep your current lock screen implementation, but would like to use the `SdlManager` for its other functionalities, you must disable the `LockScreenManager`. (This is not recommended as the new `LockScreenManager` takes care of a lot of boiler plate code and reduces possible errors)

DISABLING THE LOCK SCREEN MANAGER:

To disable, create a `LockScreenConfig` object and set it in the `SdlManager.Builder` in your `SdlService.java` class.

```
lockScreenConfig.setEnabled(false);
//...
builder.setLockScreenConfig(lockScreenConfig);
```

USING THE NEW LOCKSCREENMANAGER

If you want SDL to handle the lock screen logic for you, it is simple. You will remove the classes that currently handle your lock screen, and set the variables you want for your new lock screen as defined in the lock screen guide. This simple addition is handled during the instantiation of the the `SdlManager` within `SdlService.java`.

LOCK SCREEN ACTIVITY

You must declare the `SDLLockScreenActivity` in your manifest. To do so, simply add the following to your app's `AndroidManifest.xml` if you have not already done so:

```
<activity android:name=
"com.smartdevicelink.managers.lockscreen.SDLLockScreenActivity"
                android:launchMode="singleTop"/>
```

> **MUST**
>
> This manifest entry must be added for the lock screen feature to work.

CONFIGURATIONS

The default configurations should work for most app developers and is simple to get up and and running. However, it is easy to perform deeper configurations to the lock screen for your app. Below are the options that are available to customize your lock screen which builds on top of the logic already implemented in the `LockScreenManager`.

There is a setter in the `SdlManager.Builder` that allows you to set a `LockScreenConfig` by calling `builder.setLockScreenConfig(lockScreenConfig)`. The following options are available to be configured with the `LockScreenConfig`.

In order to to use these features, create a `LockScreenConfig` object and set it using `SdlManager.Builder` before you build `SdlManager`.

Custom Background Color

In your `LockScreenConfig` object, you can set the background color to a color resource that you have defined in your `Colors.xml` file:

```
lockScreenConfig.setBackgroundColor(resourceColor); // For example, R.color.black
```

Custom App Icon

In your `LockScreenConfig` object, you can set the resource location of the drawable icon you would like displayed:

```
lockScreenConfig.setAppIcon(appIconInt); // For example, R.drawable.lockscreen_icon
```

Showing The Device Logo

This sets whether or not to show the connected device's logo on the default lock screen. The logo will come from the connected hardware if set by the manufacturer. When using a Custom View, the custom layout will have to handle the logic to display the device logo or not. The default setting is false, but some OEM partners may require it.

In your `LockScreenConfig` object, you can set the boolean of whether or not you want the device logo shown, if available:

```
lockScreenConfig.showDeviceLogo(true);
```

Setting A Custom Lock Screen View

If you'd rather provide your own layout, it is easy to set. In your `LockScreenConfig` object, you can set the reference to the custom layout to be used for the lock screen. If this is set, the other customizations described above will be ignored:

```
lockScreenConfig.setCustomView(customViewInt);
```

Displaying Information

Setting text:

Previously, to set text fields, the developer had to create a `Show` RPC, set the text fields, and then send the PRC. It was also the developer's responsibility to make sure that they set only the lines of text that are supported by the template. In 4.7, the `ScreenManager` can be used and handles such logic internally. If a specific text field is not supported, it will be automatically hyphenated with other texts to make sure that everything is displayed correctly.

4.6:

```
Show show = new Show();
show.setMainField1("Hello, this is MainField1.");
show.setMainField2("Hello, this is MainField2.");
show.setMainField3("Hello, this is MainField3.");
show.setMainField4("Hello, this is MainField4.");
show.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (((ShowResponse) response).getSuccess()) {
            Log.i("SdlService", "Successfully showed.");
        } else {
            Log.i("SdlService", "Show request was rejected.");
        }
    }
});
proxy.sendRPCRequest(show);
```

4.7:

```
sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setTextField1("Hello, this is
MainField1.");
sdlManager.getScreenManager().setTextField2("Hello, this is
MainField2.");
sdlManager.getScreenManager().setTextField3("Hello, this is
MainField3.");
sdlManager.getScreenManager().setTextField4("Hello, this is
MainField4.");
sdlManager.getScreenManager().commit(new CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        Log.i(TAG, "ScreenManager update complete: " + success);

    }
});
```

Setting images:

Previously, to set an image, the developer had to upload the image using the `PutFile` RPC. When it is uploaded, a `Show` RPC was then created and sent to display the image. In 4.7, the `ScreenManager` handles uploading the image and sending the RPCs internally.

4.6:

```
Image image = new Image();
image.setImageType(ImageType.DYNAMIC);
image.setValue("appImage.jpeg"); // a previously uploaded filename
using PutFile RPC

Show show = new Show();
show.setGraphic(image);
show.setCorrelationID(CorrelationIdGenerator.generateId());
show.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (((ShowResponse) response).getSuccess()) {
            Log.i("SdlService", "Successfully showed.");
        } else {
            Log.i("SdlService", "Show request was rejected.");
        }
    }
});
proxy.sendRPCRequest(show);
```

4.7:

```
SdlArtwork sdlArtwork = new SdlArtwork("appImage.jpeg", FileType.
GRAPHIC_JPEG, R.drawable.appImage, true);
sdlManager.getScreenManager().setPrimaryGraphic(sdlArtwork);
```

Using soft buttons:

Previously, to add a soft button with an image the developer had to upload the image by sending a `PutFile` RPC, and after the image is uploaded, creating a `SoftButton` object, then creating a `Show` RPC. They would then need to set the button in the RPC, and then send the request. In 4.7, the `ScreenManager` takes care of sending the RPCs. The developer just has to create `softButtonObject`, add a state to it, then use the `ScreenManager` to set soft button objects.

4.6:

```
Image cancelImage = new Image();
cancelImage.setImageType(ImageType.DYNAMIC);
cancelImage.setValue("cancel.jpeg"); // a previously uploaded filename using PutFile RPC

List<SoftButton> softButtons = new ArrayList<>();

SoftButton cancelButton = new SoftButton();
cancelButton.setType(SoftButtonType.SBT_IMAGE);
cancelButton.setImage(cancelImage);
cancelButton.setSoftButtonID(1);

softButtons.add(cancelButton);

Show show = new Show();
show.setSoftButtons(softButtons);
proxy.sendRPCRequest(show);
```

4.7:

```
SoftButtonState softButtonState = new SoftButtonState("state1",
"cancel", new SdlArtwork("cancel.jpeg", FileType.GRAPHIC_JPEG, R.
drawable.cancel, true));
SoftButtonObject softButtonObject = new SoftButtonObject("object",
Collections.singletonList(softButtonState), softButtonState.getName(),
null);
sdlManager.getScreenManager().setSoftButtonObjects(Collections.
singletonList(softButtonObject));
```

Receiving button events on previous versions of SDL had to be done using `onOnButtonEvent` and `onOnButtonPress` callbacks from the `IProxyListenerALM` interface. The id had to be checked to know the exact button that received the event. In 4.7, it is much cleaner: a listener can be added to the `SoftButtonObject`, so the developer can easily tell when and which soft button received the event.

4.6:

```
@Override
public void onOnButtonEvent(OnButtonEvent notification) {
  Log.i(TAG, "onOnButtonEvent: ");

  if (notification.getButtonName() == CUSTOM_BUTTON){
      int ID = notification.getCustomButtonName();
      Log.i(TAG, "Button event received for button " + ID);
   }
}

@Override
public void onOnButtonPress(OnButtonPress notification) {
   Log.i(TAG, "onOnButtonPress: ");

   if (notification.getButtonName() == CUSTOM_BUTTON){
      int ID = notification.getCustomButtonName();
      Log.i(TAG, "Button press received for button " + ID);
   }
}
```

4.7:

```
softButtonObject.setOnEventListener(new SoftButtonObject.
OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {
        Log.i(TAG, "OnButtonPress: ");
    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject,
OnButtonEvent onButtonEvent) {
        Log.i(TAG, "OnButtonEvent: ");
    }
});
```

Receiving Subscribe Buttons Events

Previously, your `SdlService` had to implement `IProxyListenerALM` interface which means your `SdlService` class had to override all of the `IProxyListenerALM` callback methods including `OnButtonEvent` and `OnButtonPress`.

4.6

```
@Override
public void onOnHMIStatus(OnHMIStatus notification) {
    if(notification.getHmiLevel() == HMILevel.HMI_FULL && notification.
getFirstRun()) {
        SubscribeButton subscribeButtonRequest = new SubscribeButton
();
        subscribeButtonRequest.setButtonName(ButtonName.SEEKRIGHT
);
        proxy.sendRPCRequest(subscribeButtonRequest);
    }
}

@@Override
public void onOnButtonEvent(OnButtonEvent notification) {
    switch(notification.getButtonName()){
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
}

@Override
public void onOnButtonPress(OnButtonPress notification) {
        switch(notification.getButtonName()){
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
}
```

In 4.7 and the new manager APIs, in order to receive the `OnButtonEvent` and `OnButtonPress` notifications, your app must add a `OnRPCNotificationListener` using the `SdlManager`'s method `addOnRPCNotificationListener`. This will subscribe the app to any notifications of the provided type, in this case `ON_BUTTON_EVENT` and `ON_BUTTON_PRESS`. The listener should be added before sending the corresponding RPC request/subscription or else some notifications may be missed.

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_EVENT, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress)
notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
    }
});

sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_PRESS, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress)
notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
    }
});

SubscribeButton subscribeButtonRequest = new SubscribeButton();
subscribeButtonRequest.setButtonName(ButtonName.SEEKRIGHT);
sdlManager.sendRPC(subscribeButtonRequest);
```

Changing The Template:

Previously, developers had to pass a string that represents the name of the template to `SetDisplayLayout`. In 4.7, a new `PredefinedLayout` enum is introduced to hold all possible values for the templates.

4.6:

```
SetDisplayLayout setDisplayLayoutRequest = new SetDisplayLayout();
setDisplayLayoutRequest.setDisplayLayout("GRAPHIC_WITH_TEXT");
try{
    proxy.sendRPCRequest(setDisplayLayoutRequest);
}catch (SdlException e){
    e.printStackTrace();
}
```

4.7:

```
SetDisplayLayout setDisplayLayoutRequest = new SetDisplayLayout();
setDisplayLayoutRequest.setDisplayLayout(PredefinedLayout.
GRAPHIC_WITH_TEXT.toString());

sdlManager.sendRPC(setDisplayLayoutRequest);
```

Uploading Files and Graphics

SDL Android 4.7 introduces the `FileManager`, which is accessible through the `SdlManager`. Previous methods of uploading files and performing their functions still work, but now there are a set of convenience methods that do a lot of the boilerplate work for you.

Check out the Uploading Files and Graphics guide for code examples and detailed explanations.

SDL File and SDL Artwork

New to version 4.7 of the SDL Android library are `SdlFile` and `SdlArtwork` objects. These have been created in parallel with the `FileManager` to help streamline SDL workflow. `SdlArtwork` is an extension of `SdlFile` that pertains only to graphic specific file types, and its use case is similar. For the rest of this document, `SdlFile` will be described, but everything also applies to `SdlArtwork`.

CREATION

One of the hardest parts about getting a file into SDL was the boilerplate code needed to convert the file into a byte array that was used by the head unit. Now, you can instantiate a `SdlFile` with:

A RESOURCE ID

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, int id
, boolean persistentFile)
```

A URI

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, Uri
uri, boolean persistentFile)
```

And last but not least

A BYTE ARRAY

```
new SdlFile(@NonNull String fileName, @NonNull FileType fileType, byte
[] data, boolean persistentFile)
```

without the need to implement the methods needed to do the conversion of data yourself.

Uploading a File

Uploading a file with the `FileManager` is a simple process. With an instantiated `SdlManager`,
you can simply call:

```
sdlManager.getFileManager().uploadFile(sdlFile, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

Getting Vehicle Data and Subscribing to Notifications

Previously, your `SdlService` had to implement `IProxyListenerALM` interface which means your `SdlService` class had to override all of the `IProxyListenerALM` callback methods including `onOnVehicleData`.

4.6:

```
@Override
public void onOnHMIStatus(OnHMIStatus notification) {
    if(notification.getHmiLevel() == HMILevel.HMI_FULL && notification.
getFirstRun()) {
        SubscribeVehicleData subscribeRequest = new
SubscribeVehicleData();
        subscribeRequest.setPrndl(true);
        subscribeRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
            @Override
            public void onResponse(int correlationId, RPCResponse response
) {
                if(response.getSuccess()){
                    Log.i("SdlService", "Successfully subscribed to vehicle
data.");
                }else{
                    Log.i("SdlService", "Request to subscribe to vehicle data
was rejected.");
                }
            }
        });
        try {
            proxy.sendRPCRequest(subscribeRequest);
        } catch (SdlException e) {
            e.printStackTrace();
        }
    }
}

@Override
public void onOnVehicleData(OnVehicleData notification) {
    PRNDL prndl = notification.getPrndl();
    Log.i("SdlService", "PRNDL status was updated to: " prndl.toString());
}
```

In 4.7 and the new manager APIs, in order to receive the `OnVehicleData` notifications, your app must add a `OnRPCNotificationListener` using the `SdlManager`'s method `addOnRPCNotificationListener`. This will subscribe the app to any notifications of the provided type, in this case `ON_VEHICLE_DATA`. The listener should be added before sending the corresponding RPC request/subscription or else some notifications may be missed.

4.7:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_VEHICLE_DATA, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnVehicleData onVehicleDataNotification = (OnVehicleData)
notification;
        if (onVehicleDataNotification.getPrndl() != null) {
            Log.i("SdlService", "PRNDL status was updated to: " +
onVehicleDataNotification.getPrndl());
        }
    }
});


SubscribeVehicleData subscribeRequest = new SubscribeVehicleData();
subscribeRequest.setPrndl(true);
subscribeRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Log.i("SdlService", "Successfully subscribed to vehicle data.");
        }else{
            Log.i("SdlService", "Request to subscribe to vehicle data was
rejected.");
        }
    }
});
sdlManager.sendRPC(subscribeRequest);
```

Getting In-Car Audio

Subscribing to AudioPassThru Notifications

Previously, your `SdlService` had to implement `IProxyListenerALM` interface which means your `SdlService` class had to override all of the `IProxyListenerALM` callback methods including `onOnAudioPassThru`.

4.6:

```
@Override
public void onOnHMIStatus(OnHMIStatus notification) {
    if(notification.getHmiLevel() == HMILevel.HMI_FULL && notification.
getFirstRun()) {
        PerformAudioPassThru performAPT = new PerformAudioPassThru
();
        performAPT.setAudioPassThruDisplayText1("Ask me \"What's the
weather?\"");
        performAPT.setAudioPassThruDisplayText2("or \"What's 1 + 2?\"");
        performAPT.setInitialPrompt(TTSChunkFactory.
createSimpleTTSChunks("Ask me What's the weather? or What's 1 plus
2?"));
        performAPT.setSamplingRate(SamplingRate._22KHZ);
        performAPT.setMaxDuration(7000);
        performAPT.setBitsPerSample(BitsPerSample._16_BIT);
        performAPT.setAudioType(AudioType.PCM);
        performAPT.setMuteAudio(false);
        proxy.sendRPCRequest(performAPT);
    }
}

@Override
public void onOnAudioPassThru(OnAudioPassThru notification) {
    byte[] dataRcvd = notification.getAPTData();
    processAPTData(dataRcvd); // Do something with audio data
}
```

In 4.7 and the new manager APIs, in order to receive the `OnAudioPassThru` notifications, your app must add a `OnRPCNotificationListener` using the `SdlManager`'s method `addOnRPCNotificationListener`. This will subscribe the app to any notifications of the provided type, in this case `ON_AUDIO_PASS_THRU`. The listener should be added before sending the corresponding RPC request/subscription or else some notifications may be missed.

4.7:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_AUDIO_PASS_THRU, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnAudioPassThru onAudioPassThru = (OnAudioPassThru)
notification;
        byte[] dataRcvd = onAudioPassThru.getAPTData();
        processAPTData(dataRcvd); // Do something with audio data
    }
});

PerformAudioPassThru performAPT = new PerformAudioPassThru();
performAPT.setAudioPassThruDisplayText1("Ask me \"What's the
weather?\"");
performAPT.setAudioPassThruDisplayText2("or \"What's 1 + 2?\"");
performAPT.setInitialPrompt(TTSChunkFactory.createSimpleTTSChunks(
"Ask me What's the weather? or What's 1 plus 2?"));
performAPT.setSamplingRate(SamplingRate._22KHZ);
performAPT.setMaxDuration(7000);
performAPT.setBitsPerSample(BitsPerSample._16_BIT);
performAPT.setAudioType(AudioType.PCM);
performAPT.setMuteAudio(false);
sdlManager.sendRPC(performAPT);
```

Mobile Navigation

Video Streaming:

Previously, developers had to make sure that the app was in HMI_FULL before starting the video stream, In 4.7, after the `SdlManager` has called its `onStart` method, the developer can start video streaming in `VideoStreamingManager.start()`'s `CompletionListener`. The `VideoStreamingManager` will take care of starting the video when the app becomes ready.

4.6:

```
    if(notification.getHmiLevel().equals(HMILevel.HMI_FULL)){
        if (notification.getFirstRun()) {
            proxy.startRemoteDisplayStream(getApplicationContext(),
MyDisplay.class, null, false);
        }
    }

}
```

4.7:

```
sdlManager.getVideoStreamManager().start(new CompletionListener()
{
    @Override
    public void onComplete(boolean success) {
        if (success) {
            sdlManager.getVideoStreamManager().
startRemoteDisplayStream(getApplicationContext(), MyDisplay.class,
null, false);
        }
    }
});
```

Audio Streaming

With the addition of the `AudioStreamingManager`, which is accessed through `SdlManager`, you can now use `mp3` files in addition to `raw`. The `AudioStreamingManager` also handles `AudioStreamingCapabilities` for you, so your stream will use the correct capabilities for the connected head unit. We suggest that for any audio streaming that this is now used. Below is the difference in streaming from 4.6 to 4.7

4.6

```
 private void startAudioStream(){

        final InputStream is = getResources().openRawResource(R.raw.
audio_file);

        AudioStreamingParams audioParams = new
AudioStreamingParams(44100, 1);
        listener = proxy.startAudioStream(false, AudioStreamingCodec.
LPCM, audioParams);
        if (listener != null){
            try {
                listener.sendAudio(readToByteBuffer(is), -1);

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    private void stopAudioStream(){
        proxy.endAudioStream();
    }

    static ByteBuffer readToByteBuffer(InputStream inStream) throws
IOException {
        byte[] buffer = new byte[8000];
        ByteArrayOutputStream outStream = new ByteArrayOutputStream
(8000);
        int read;
        while (true) {
            read = inStream.read(buffer);
            if (read == -1)
                break;
            outStream.write(buffer, 0, read);
        }
        ByteBuffer byteData = ByteBuffer.wrap(outStream.toByteArray());
        return byteData;
    }
```

4.7

```
if (sdlManager.getAudioStreamManager() != null) {
    Log.i(TAG, "Trying to start audio streaming");
    sdlManager.getAudioStreamManager().start(new CompletionListener
() {
        @Override
        public void onComplete(boolean success) {
            if (success) {
                sdlManager.getAudioStreamManager().startAudioStream(
false, new CompletionListener() {
                    @Override
                    public void onComplete(boolean success) {
                        if (success) {
                            Resources resources = getApplicationContext().
getResources();
                            int resourceId = R.raw.audio_file;
                            Uri uri = new Uri.Builder()
                            .scheme(ContentResolver.
SCHEME_ANDROID_RESOURCE)
                            .authority(resources.getResourcePackageName(
resourceId))
                            .appendPath(resources.getResourceTypeName(
resourceId))
                            .appendPath(resources.getResourceEntryName(
resourceId))
                            .build();
                            sdlManager.getAudioStreamManager().
pushAudioSource(uri, new CompletionListener() {
                                @Override
                                public void onComplete(boolean success) {
                                    if (success) {
                                        Log.i(TAG, "Audio file played successfully!");
                                    } else {
                                        Log.i(TAG, "Audio file failed to play!");
                                    }
                                }
                            });
                        } else {
                            Log.d(TAG, "Audio stream failed to start!");
                        }
                    }
                });
            } else {
                Log.i(TAG, "Failed to start audio streaming manager");
            }
        }
    });
}
```

Checking Permissions:

Previously, it was not easy to check if specific permission had changed. Developers had to keep checking `onOnHMIStatus` and `onOnPermissionsChange` callbacks and manually check the responses to see if the permission is allowed. In 4.7, the `PermissionManager` implements all of this logic internally. It keeps a cached copy of the callback responses whenever an update is received. So developer can call `isRPCAllowed()` any time to know if a permission is allowed. It also makes it very simple to add a listener.

4.6:

```
@Override
public void onOnHMIStatus(OnHMIStatus notification) {
    hmiLevel = notification.getHmiLevel();
    if (checkShowPermission(FunctionID.SHOW.toString(), hmiLevel,
permissionItems)){
        // Show RPC is allowed
    }
}

@Override
public void onOnPermissionsChange(OnPermissionsChange notification)
 {
    permissionItems = notification.getPermissionItem();
    if (checkShowPermission(FunctionID.SHOW.toString(), hmiLevel,
permissionItems)){
        // Show RPC is allowed
    }
}

private boolean checkShowPermission(String rpcName, HMILevel
hmiLevel, List<PermissionItem> permissionItems){
    PermissionItem permissionItem = null;
    for (PermissionItem item : permissionItems) {
        if (rpcName.equals(item.getRpcName())){
            permissionItem = item;
            break;
        }
    }
    if (hmiLevel == null || permissionItem == null || permissionItem.
getHMIPermissions() == null || permissionItem.getHMIPermissions().
getAllowed() == null){
            return false;
    } else if (permissionItem.getHMIPermissions().getUserDisallowed() !=
 null){
        return permissionItem.getHMIPermissions().getAllowed().contains(
hmiLevel) && !permissionItem.getHMIPermissions().getUserDisallowed
().contains(hmiLevel);
    } else {
        return permissionItem.getHMIPermissions().getAllowed().contains(
hmiLevel);
    }
}
```

4.7:

To check if a permission is allowed:

```
boolean allowed = sdlManager.getPermissionManager().isRPCAllowed(
FunctionID.SHOW);
```

To setup a permission listener:

```
List<PermissionElement> permissionElements = Collections.
singletonList(new PermissionElement(FunctionID.SHOW, null));
UUID listenerId = sdlManager.getPermissionManager().addListener(
permissionElements, PermissionManager.
PERMISSION_GROUP_TYPE_ANY, new OnPermissionChangeListener() {
    @Override
    public void onPermissionsChange(@NonNull Map<FunctionID,
PermissionStatus> allowedPermissions, @NonNull int
permissionGroupStatus) {
        if (allowedPermissions.get(FunctionID.SHOW).getIsRPCAllowed()) {
            // Show RPC is allowed
        }
    }
});
```

For more information about `PermissionManager`, you can check this page.

Handling a Language Change

Previously, to let your app reconnect after the user changes the head unit language, your app had to send an intent in the `onProxyClosed` callback. That intent should be received by `SdlReceiver` to start the `SdlService`. The `SdlReceiver` part did not change so we will only cover the changes in sending the intent which was done in previous versions as the following:

```
@Override
public void onProxyClosed(String info, Exception e,
SdlDisconnectedReason reason) {
    stopSelf();
    if(reason.equals(SdlDisconnectedReason.LANGUAGE_CHANGE)){
        Intent intent = new Intent(TransportConstants.
START_ROUTER_SERVICE_ACTION);
        intent.putExtra(SdlReceiver.RECONNECT_LANG_CHANGE, true);
        sendBroadcast(intent);
    }
}
```

In 4.7, the app has to send the intent in a `ON_LANGUAGE_CHANGE` notification listener as the following:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_LANGUAGE_CHANGE, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        SdlService.this.stopSelf();
        Intent intent = new Intent(TransportConstants.
START_ROUTER_SERVICE_ACTION);
        intent.putExtra(SdlReceiver.RECONNECT_LANG_CHANGE, true);
        AndroidTools.sendExplicitBroadcast(context, intent, null);
    }
});
```

Fore more information about handling language changes please visit this page

Remote Control

Subscribing to OnInteriorVehicleData Notifications

Previously, your `SdlService` had to implement `IProxyListenerALM` interface which means your `SdlService` class had to override all of the `IProxyListenerALM` callback methods including `onOnInteriorVehicleData`.

4.6:

```
@Override
public void onOnHMIStatus(OnHMIStatus notification) {
    if(notification.getHmiLevel() == HMILevel.HMI_FULL && notification.
getFirstRun()) {
        GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData();
        interiorVehicleData.setModuleType(ModuleType.RADIO);
        interiorVehicleData.setSubscribe(true);
        interiorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
            @Override
            public void onResponse(int correlationId, RPCResponse
response) {
                GetInteriorVehicleData getResponse = (
GetInteriorVehicleData) response;
                //This can now be used to retrieve data
            }
        });
        proxy.sendRPCRequest(interiorVehicleData);
    }
}

@Override
public void onOnInteriorVehicleData(OnInteriorVehicleData response) {
    //Perform action based on notification
}
```

In 4.7 and the new manager APIs, in order to receive the `OnInteriorVehicleData` notifications, your app must add a `OnRPCNotificationListener` using the `SdlManager`'s method `addOnRPCNotificationListener`. This will subscribe the app to any notifications of the provided type, in this case `ON_INTERIOR_VEHICLE_DATA`. The listener should be added before sending the corresponding RPC request/subscription or else some notifications may be missed.

4.7:

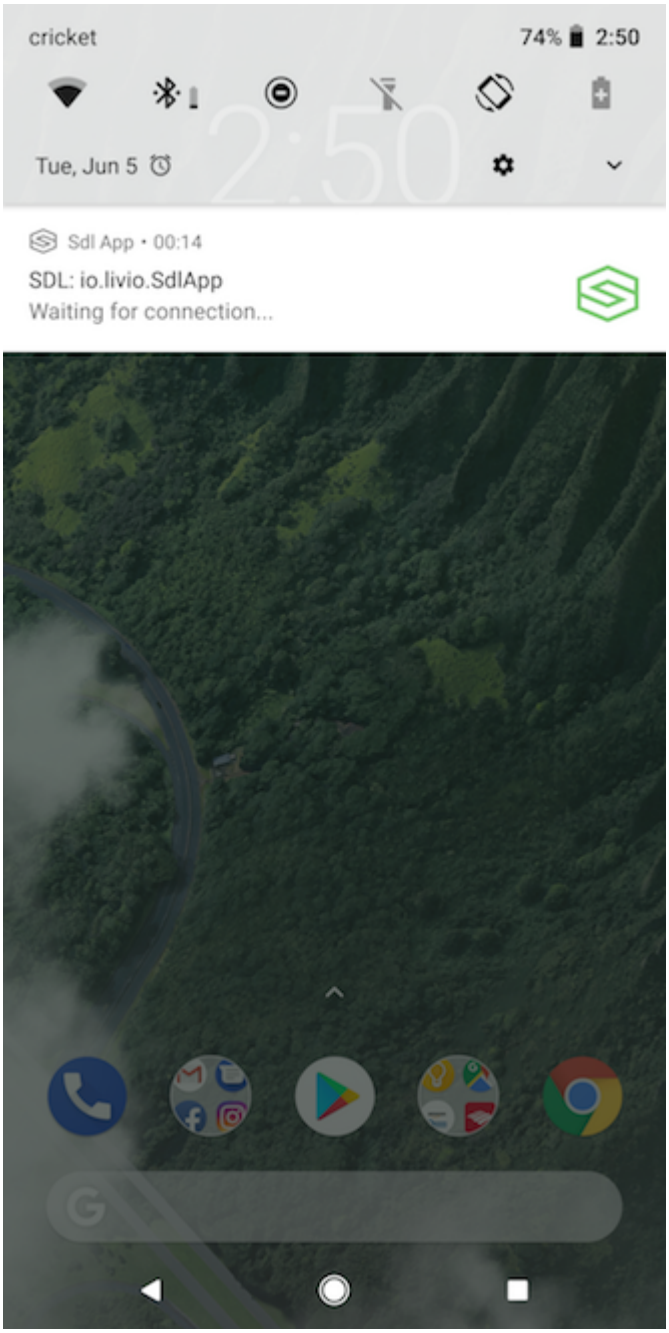
```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_INTERIOR_VEHICLE_DATA, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnInteriorVehicleData onInteriorVehicleData = (
OnInteriorVehicleData) notification;
        //Perform action based on notification
    }
});

GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData();
interiorVehicleData.setModuleType(ModuleType.RADIO);
interiorVehicleData.setSubscribe(true);
interiorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        GetInteriorVehicleData getResponse = (GetInteriorVehicleData)
response;
        //This can now be used to retrieve data
    }
});
sdlManager.sendRPC(interiorVehicleData);
```

What is SDL?

SmartDeviceLink (SDL) connects in-vehicle infotainment systems to smartphone apps. SDL allows automakers to provide highly integrated connected experiences and allows users to operate smartphone apps through the in-vehicle infotainment screen and, if equipped, voice recognition system.

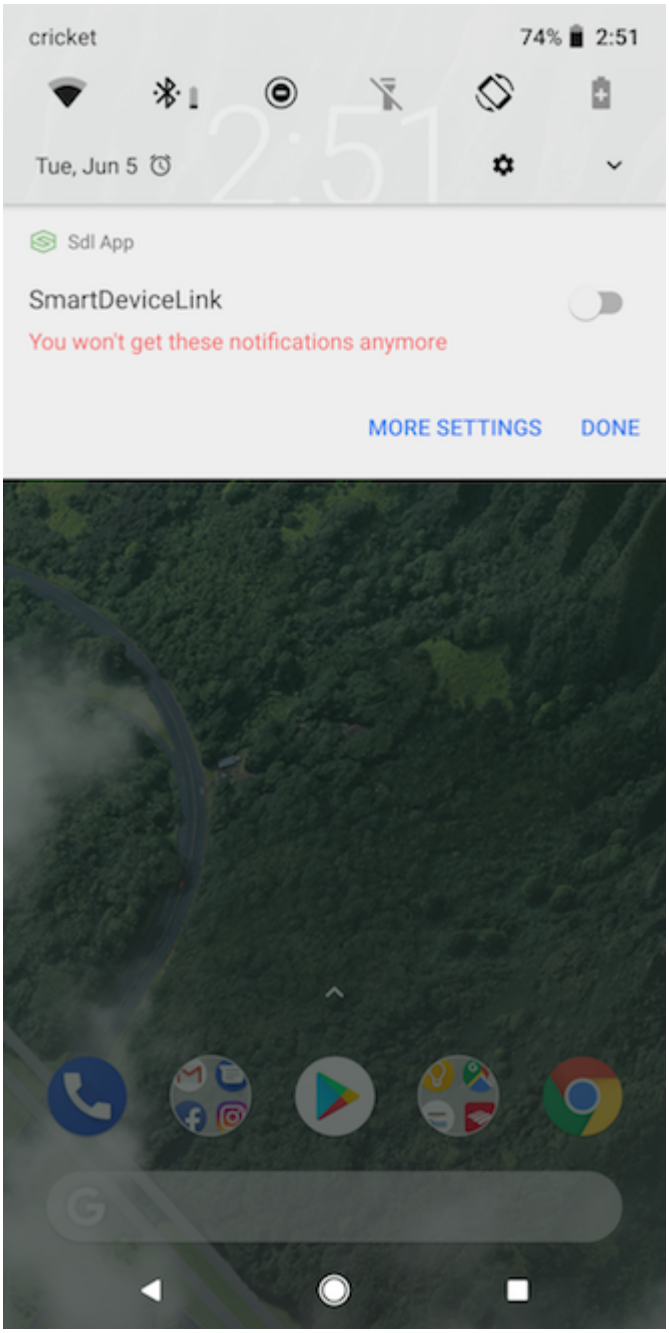
Why do you see SDL notifications?

If you see a notification similar to the one in the screenshot below, that means you are using an app that has an SDL integration that allows it to push content to cars that support SDL. However, if your car doesn’t support SDL, you can simply hide the notification.


cricket
74% 2:50
Tue, Jun 5
Sdl App • 00:14
SDL: io.livio.SdlApp
Waiting for connection...

How do you hide the notifications?

If you would like to hide the notification, you can simply long click on the notification and disable it as shown in the following screenshot.


cricket
74% 2:51
Tue, Jun 5
Sdl App
SmartDeviceLink
You won't get these notifications anymore
MORE SETTINGS
DONE

What is the Android Router Service?

The Android OS has limitations around the availability of certain transports (Bluetooth RFCOMM channels, single app AOA/USB permissions). Therefore, SmartDeviceLink introduced a service that operates as a router, using a single transport pipe and extending it to many different bound apps. The router service is part of the required integration to become SDL enabled and can be hosted by any of the SDL enabled apps on a phone. Some OEMs might choose to have their own companion app that always hosts a router service for their specific hardware.

What is a Trusted Router Service?

Since information is being shared through the Android router service it is important that the app hosting the router service can be trusted. This is done through a certification process and a back-end server that maintains a database of apps that can act as a Trusted Router Service. The SDLC will verify the integration of SDL apps to ensure there is no malicious activity. If the app is

certified, it will be added to the Trusted Router Service database and be able to act as a Trusted Router Service.

How do I add my app to the SDL Trusted Router Service database?

For an Android application to be added to the Trusted Router Service database, the application will need to be registered on the SDL Developer Portal and certified by the SDLC. For more information on registration, please see this guide. Any Android application that is certified by the SDLC will be added to the Trusted Router Service database; there are no additional steps required as it is part of the certification process.

How do I know if an app is hosting a Trusted Router Service?

Each app will retrieve and cache a list of Trusted Router Services from the back-end server. Based on that app's security levels, they will perform checks against the currently running router service, and if trusted it will bind to the Trusted Router Service. If not, the app will attempt to use its own local transport.