sdl

Android Guides

Document current as of 10/03/2019 11:49 AM.

Installation

In order to build your app on a SmartDeviceLink (SDL) Core, the SDL software development kit (SDK) must be installed in your app. The following steps will guide you through adding the SDL SDK to your workspace and configuring the environment.

Install SDL SDK

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 4.9.0, use the following line:

```
dependencies {
   implementation 'com.smartdevicelink:sdl_android:4.9.0'
}
```

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

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

SDK Configuration

1. Get an App Id

An app id is required for production level apps. The app id gives your app special permissions to access vehicle data. If your app does not need to access vehicle data, a dummy app id (i.e. creating a fake id like "1234") is sufficient during the development stage. However, you must get an app id before releasing the app to the public.

To obtain an app id, sign up at smartdevicelink.com.

2. Add Required System Permissions

Some permissions are required to be granted to the sdl app in order for it to work properly. In the AndroidManifest file, 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" />
```

Integration Basics

How SDL Works

SmartDeviceLink works by sending remote procedure calls (RPCs) back and forth between a smartphone application and the SDL Core. These RPCs allow you to build the user interface, detect button presses, play audio, and get vehicle data, among other things. You will use the SDL library to build your app on the SDL Core.

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.

SmartDeviceLink Service

A SmartDeviceLink 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 it is 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`. If not, then 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 your 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>

</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.

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:

```
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);
    }
}
```

Connecting to an Infotainment System

To connect to an emulator, such as Manticore or a local Ubuntu SDL Core-based emulator, make sure to use `TCPTransportConfig`. The emulator and app should be on the same network (i.e. remember to set the correct IP address and port number). The IP will most likely be the IP address of the operating system running the emulator. The port will most likely be `12345`.

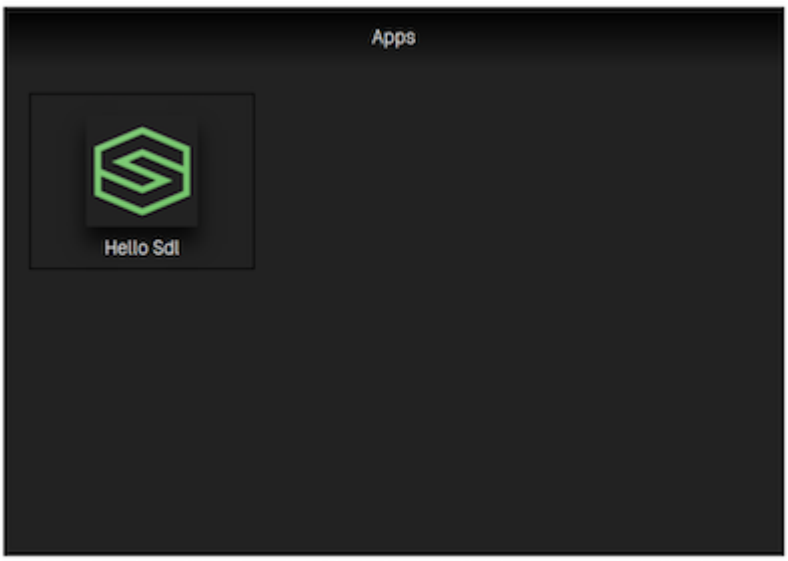
```
// Set the SdlManager.Builder transport
builder.setTransportType(new TCPTransportConfig(<IP ADDRESS>, <
PORT>, false));
```

Connecting with a Vehicle Head Unit or a Development Kit (TDK)

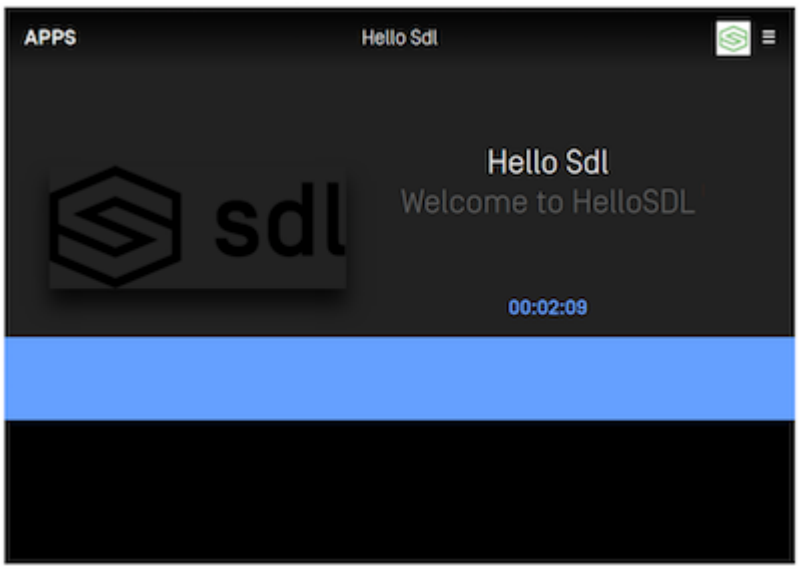
To connect your device directly to a vehicle head unit or TDK, make sure to use `MultiplexTransportConfig`. Then connect the device to the head unit or TDK using a USB cord or Bluetooth if the head unit supports it.

```
// Set the SdlManager.Builder transport
builder.setTransportType(new MultiplexTransportConfig(context, <APP ID>));
```

Run the project in Android Studio, targeting the device you want Sdl Android installed on. Sdl Android should compile and launch on your device of choosing. Following this, you should see an application appears on the TDK or HMI:



Click on the Sdl icon in the HMI.



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

Adding the Lock Screen

The lock screen is a vital part of your SDL app because it prevents the user from using the phone while the vehicle is in motion. SDL takes care of the lock screen for you. If you prefer your own look, but still want the recommended logic that SDL provides for free, you can also set your own custom lockscreen.

Configure the 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.

Using the Provided Lock Screen

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.

Customizing the Default Lock Screen

It is possible to customize the background color and app icon in the default provided lockscreen. If you choose not to set your own app icon the library will use the SDL logo.

Custom Background Color

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

Custom App Icon

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

Showing the OEM 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. 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);
```

Creating a Custom Lock Screen

If you would like to use your own lock screen instead of the one provided by the library, but still use the logic we provide, you can use a new initializer within `LockScreenConfig`:

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

Customizing the Lock Screen State

Disabling the Lock Screen

Please note that a lock screen will be required by most OEMs. You can disable the lock screen manager, but you will then be required to implement your own logic for showing and hiding the lock screen. This is not recommended as the `LockScreenConfig` adheres to most OEM lock screen requirements. However, if you must create a lock screen manager from scratch, the library's lock screen manager can be disabled via the `LockScreenConfig` as follows:

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

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
- SDK Configuration guide
- Integration Basics guide

We will add or make changes to:

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

Prerequisites

The Installation, SDK Configuration, 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 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. A multiplex transport should be used 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 transport. For example, apps can

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

Primary Transports

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 is 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 to it.

```
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.

Requiring 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 to. 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.

Adapting to the Head Unit Language

Since a head unit can support multiple languages, you may want to add support for more than one language to your SDL app. The SDL library allows you to check which language is currently used by the head unit. If desired, the app's name and the app's text-to-speech (TTS) name can be customized to reflect the head unit's current language. If your app name is not part of the current lexicon, you should tell the VR system how a native speaker will pronounce your app name by setting the TTS name using phonemes from either the Microsoft SAPI phoneme set or from the LHPLUS phoneme set.

Setting the Default Language

The initial configuration of the `SdlManager` requires a default language when setting the `Builder`. If not set, the SDL library uses American English (*EN_US*) as the default language. The connection will fail if the head unit does not support the `language` set in the `Builder`. The `RegisterAppInterface` response RPC will return `INVALID_DATA` as the reason for rejecting the request.

What if My App Does Not Support the Head Unit Language?

If your app does not support the current head unit language, you should decide on a default language to use in your app. All text should be created using this default language. Unfortunately, your VR commands will probably not work as the VR system will not recognize your users' pronunciation.

Handling a Language Change

When a user changes the language on a head unit, an `OnLanguageChange` notification will be sent from Core, causing 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 Service
- Local SDL Broadcast Receiver

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);
    }
});
```

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);
}
```

Understanding Permissions

While creating your SDL app, remember that just because your app is connected to a head unit it does not mean that the app has permission to send the RPCs you want. If your app does not have the required permissions, requests will be rejected. There are three important things to remember in regards to permissions:

1. You may not be able to send a RPC when the SDL app is closed, in the background, or obscured by an alert. Each RPC has a set of `hmiLevel`s during which it can be sent.
2. For some RPCs, like those that access vehicle data or make a phone call, you may need special permissions from the OEM to use. This permission is granted when you submit your app to the OEM for approval. Each OEM decides which RPCs it will restrict access to, so it is up you to check if you are allowed to use the RPC with the head unit.

3. Some head units may not support all RPCs.

HMI Levels

When your app is connected to the head unit you will receive notifications when the SDL app's HMI status changes. Your app can be in one of four different `hmiLevel`s:

HMI LEVEL	WHAT DOES THIS MEAN?
NONE	The user has not yet opened your app, or the app has been killed.
BACKGROUND	The user has opened your app, but is currently in another part of the head unit.
LIMITED	This level only applies to media and navigation apps (i.e. apps with an `appType` of `MEDIA` or `NAVIGATION`). The user has opened your app, but is currently in another part of the head unit. The app can receive button presses from the play, seek, tune, and preset buttons.
FULL	Your app is currently in focus on the screen.

Be careful with sending user interface related RPCs in the `NONE` and `BACKGROUND` levels; some head units may reject RPCs sent in those states. We recommended that you wait until your app's `hmiLevel` enters `FULL` to set up your app's UI.

To get more detailed information about the state of your SDL app check the current system context. The system context will let you know if a menu is open, a VR session is in progress, an alert is showing, or if the main screen is unobstructed. You can find more information about the system context below.

Monitoring the HMI Level

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);
```

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.

Checking Current Permissions of a Single RPC

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

// You can also check if a permission parameter is allowed
boolean parameterAllowed = sdlManager.getPermissionManager().
isPermissionParameterAllowed(FunctionID.GET_VEHICLE_DATA,
GetVehicleData.KEY_RPM);
```

Checking Current Permissions of a Group of RPCs

```
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
}
```

Observing Permissions

If desired, you can set a listener for a group of permissions. The listener will be called when the permissions for the group changes. If you want to be notified when the permission status of any of RPCs in the group change, set the `groupType` to `PERMISSION_GROUP_TYPE_ANY`. If you only want to be notified when all of the RPCs in the group are allowed, set the `groupType` to `PERMISSION_GROUP_TYPE_ALL_ALLOWED`.

```
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
        }
    }
});
```

Stopping Observation of Permissions

When you set up the listener, you will get an unique id back. Use this id to unsubscribe to the permissions at a later date.

```
sdlManager.getPermissionManager().removeListener(listenerId);
```

Additional HMI State Information

If you want more detail about the current state of your SDL app you can monitor the audio playback state as well as get notifications when something blocks the main screen of your app.

Audio Streaming State

The Audio Streaming State informs your app whether or not the driver will be able to hear your app's audio. It will be either `AUDIBLE`, `NOT_AUDIBLE`, or `ATTENUATED`.

You will get these notifications when an alert pops up, when you start recording the in-car audio, when voice recognition is active, when another app takes audio control, when a navigation app is giving directions, etc.

AUDIO STREAMING STATE	WHAT DOES THIS MEAN?
AUDIBLE	Any audio you are playing will be audible to the user
ATTENUATED	Some kind of audio mixing is occurring between what you are playing, if anything, and some system level audio or navigation application audio.
NOT_AUDIBLE	Your streaming audio is not audible. This could occur during a `VRSESSION` System Context.

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

System Context

The 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.

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

Setting Security Level for Multiplexing

When connecting to Core via Multiplex 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 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.

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.

Example Apps

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 HMI.

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

Open the the `sdl_java_suite/android` project using "Open an existing Android Studio 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 production head unit for development, we recommend using SDL Core and this Generic HMI for testing.

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

To know more about how to connect the app to the infotainment system, please check the Connecting to an Infotainment System guide

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)

Adaptive Interface Capabilities

Designing for Different Head Units

Since each car manufacturer has different user interface style guidelines, the number of lines of text, soft and hard buttons, and images supported will vary between different types of head units. When the app first connects to the SDL Core, a `RegisterAppInterface` response RPC will be sent by Core with the `displayCapabilities`, `buttonCapabilities`, `speechCapabilities` and other properties. You should use this information to create the user interface of your SDL app.

You may access these properties on the `SdlManager.systemCapabilityManager` instance as of library v.4.4.

System Capability Manager Properties

PARAMETERS	DESCRIPTION	NOTES
SystemCapabilityType.DISPLAY	Information about the HMI 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
SystemCapabilityType.BUTTON	A list of available buttons and whether the buttons support long, short and up-down presses.	Check ButtonCapabilities.java for more information
SystemCapabilityType.SOFTBUTTON	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
SystemCapabilityType.PRESET_BANK	If returned, the platform supports custom on-screen presets.	Check PresetBankCapabilities.java for more information
SystemCapabilityType.HMI_ZONE	Specifies HMI Zones in the vehicle. There may be a HMI available for back seat passengers as well as front seat passengers.	Check HmiZoneCapabilities.java
SystemCapabilityType.SPEECH	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
prerecordedSpeechCapabilities	Currently only available in the SDL_iOS library	currently only available in the SDL_iOS library

PARAMETERS	DESCRIPTION	NOTES
SystemCapabilityType.VOICE_RECOGNITION	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
SystemCapabilityType.AUDIO_PASSTHROUGH	Describes the sampling rate, bits per sample, and audio types available.	Check AudioPassThruCapabilities.java for more information
SystemCapabilityType.PCM_STREAMING	Describes different audio type configurations for the audio PCM stream service, e.g. {8kHz,8-bit,PCM}.	Check AudioPassThruCapabilities.java for more information
SystemCapabilityType.HMI	Returns whether or not the app can support built-in navigation and phone calls.	Check HMICapabilities.java for more information
SystemCapabilityType.APP_SERVICES	Describes the capabilities of app services including what service types are supported and the current state of services.	Check AppServicesCapabilities.java for more information
SystemCapabilityType.NAVIGATION	Describes the built-in vehicle navigation system's APIs.	Check NavigationCapability.java for more information
SystemCapabilityType.PHONE_CALL	Describes the built-in phone calling capabilities of the IVI system.	Check PhoneCapability.java for more information
SystemCapabilityType.VIDEO_STREAMING	Describes the abilities of the head unit to video stream projection applications.	Check VideoStreamingCapability.java for more information
SystemCapabilityType.REMOTE_CONTROL	Describes the abilities of an app to control built-in aspects of the IVI system.	Check RemoteControlCapabilities.java for more information

The Register App Interface Response (RAIR) RPC

The `RegisterAppInterface` response contains information about 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. The table below has a list of properties beyond those available on the system capability manager returned by the `RegisterAppInterface` response. Each property is optional, so you may not get data for all the parameters in the following table.

PARAMETERS	DESCRIPTION	NOTES
sdlMsgVersion	Specifies the version number of the SmartDeviceLink protocol that is supported by the mobile application.	Check SdlMsgVersion.java for more information
language	The currently active VR +TTS language on the module.	Check Language.java for more information
vehicleType	The make, model, year, and the trim of the vehicle.	Check VehicleType.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.	Check DiagnosticMessage.java for more information
sdlVersion	The SmartDeviceLink version.	String
systemSoftwareVersion	The software version of the system that implements the SmartDeviceLink core.	String

Image Specifics

IMAGE FILE TYPES

Images may be formatted as PNG, JPEG, or BMP. Each `sdlManager.getRegisterAppInterfaceResponse().getDisplayCapabilities().getImageFields()` property will have a list of `imageTypeSupported`.

IMAGE SIZES

If an image is uploaded that is larger than the supported size, that image will be scaled down by Core. All image sizes are available from the `sdlManager.getRegisterAppInterfaceResponse().getDisplayCapabilities().getImageFields()` property once the manager has started successfully.

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

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

Querying the RAIR Capabilities

An example of querying the Vr Capabilities:

```
sdlManager.getRegisterAppInterfaceResponse().getVrCapabilities();
```

System Capabilities

Most head units provide features that your app can use: making and receiving phone calls, an embedded navigation system, video and audio streaming, as well as supporting app services. To find out if the head unit supports a feature

as well as more information about the feature, use the `SystemCapabilityManager` to query the head unit for the desired capability. Querying for capabilities is only available on head units supporting SDL Core v4.5 or greater; if connecting to older head units, the query will return `null` even if the head unit may support the capabilty.

Querying for System Capabilities

```
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 #>
    }
});
```

Subscribing to System Capability Updates

In addition getting the current system capabilities, it is also possible to register to get updates when the head unit capabilities change. Since this information

must be queried from Core you must implement the `OnSystemCapabilityListener`. This feature is only available on head units supporting v.5.1 or greater.

SUBSCRIBE TO A CAPABILITY

```
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 #>
    }
});

GetSystemCapability getSystemCapability = new GetSystemCapability
();
getSystemCapability.setSystemCapabilityType(SystemCapabilityType.
APP_SERVICES);
getSystemCapability.setSubscribe(true);
sdlManager.sendRPC(getSystemCapability);
```

Main Screen Templates

Each head unit manufacturer supports a set of user interface templates. These templates determine the position and size of the text, images, and buttons on the screen. Once the app has connected successfully with an SDL enabled head unit, a list of supported templates is available on `((DisplayCapabilities)`

`sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.DISPLAY)).getTemplatesAvailable()`.

Change the Template

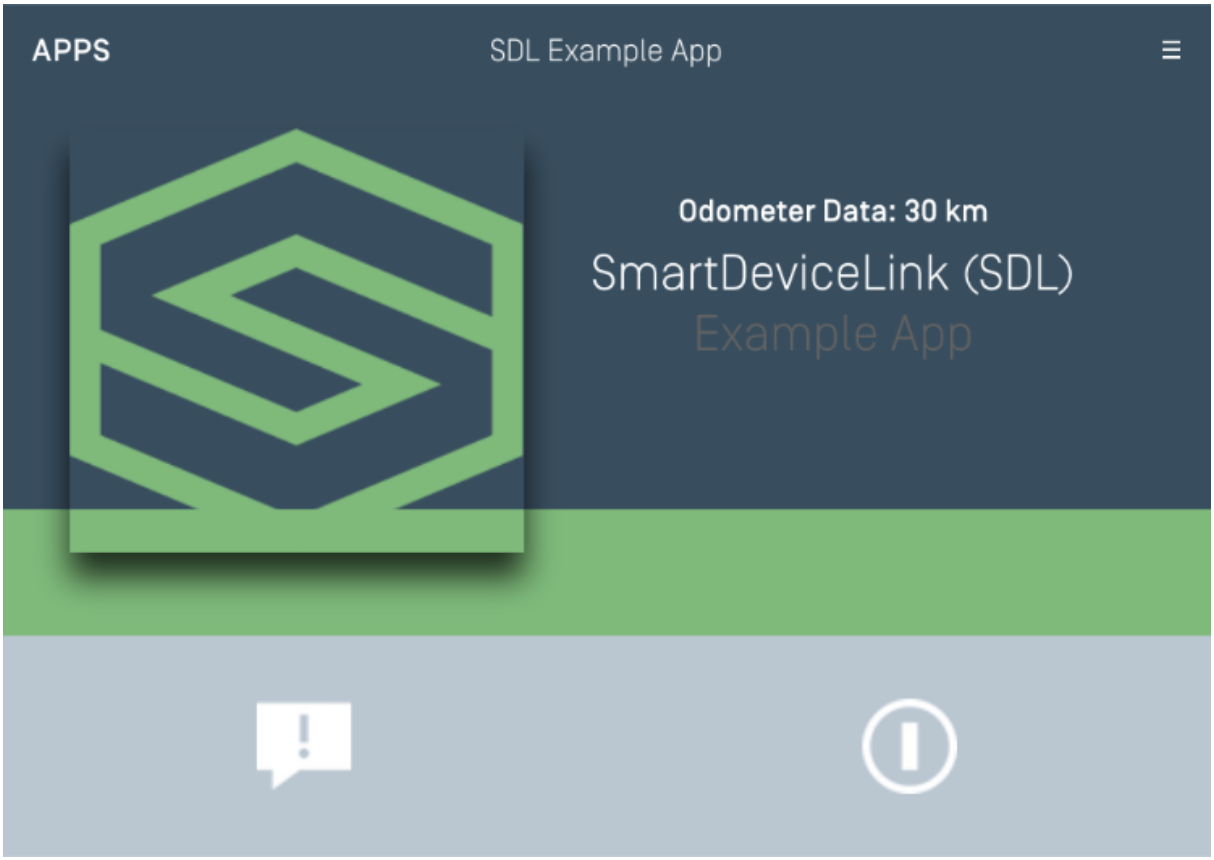
To change a template at any time, send a `SetDisplayLayout` RPC to Core.

```
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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(setDisplayLayoutRequest);
```

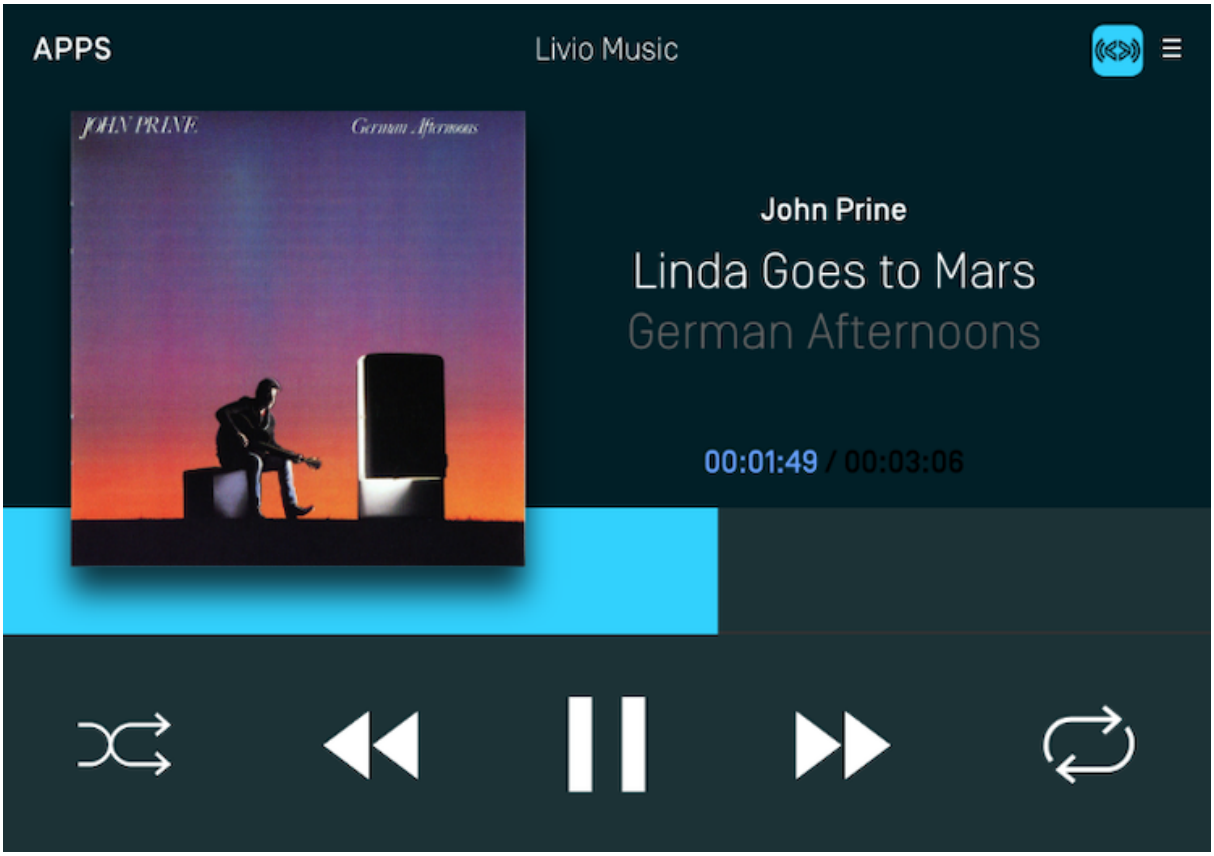
Available Templates

There are fifteen standard templates to choose from, however some head units may only support a subset of these templates. The following examples show how templates will appear on the Generic HMI and Ford's SYNC 3 HMI.

MEDIA


APPS
SDL Example App
Odometer Data: 30 km
SmartDeviceLink (SDL)
Example App

MEDIA (WITH A PROGRESS BAR)

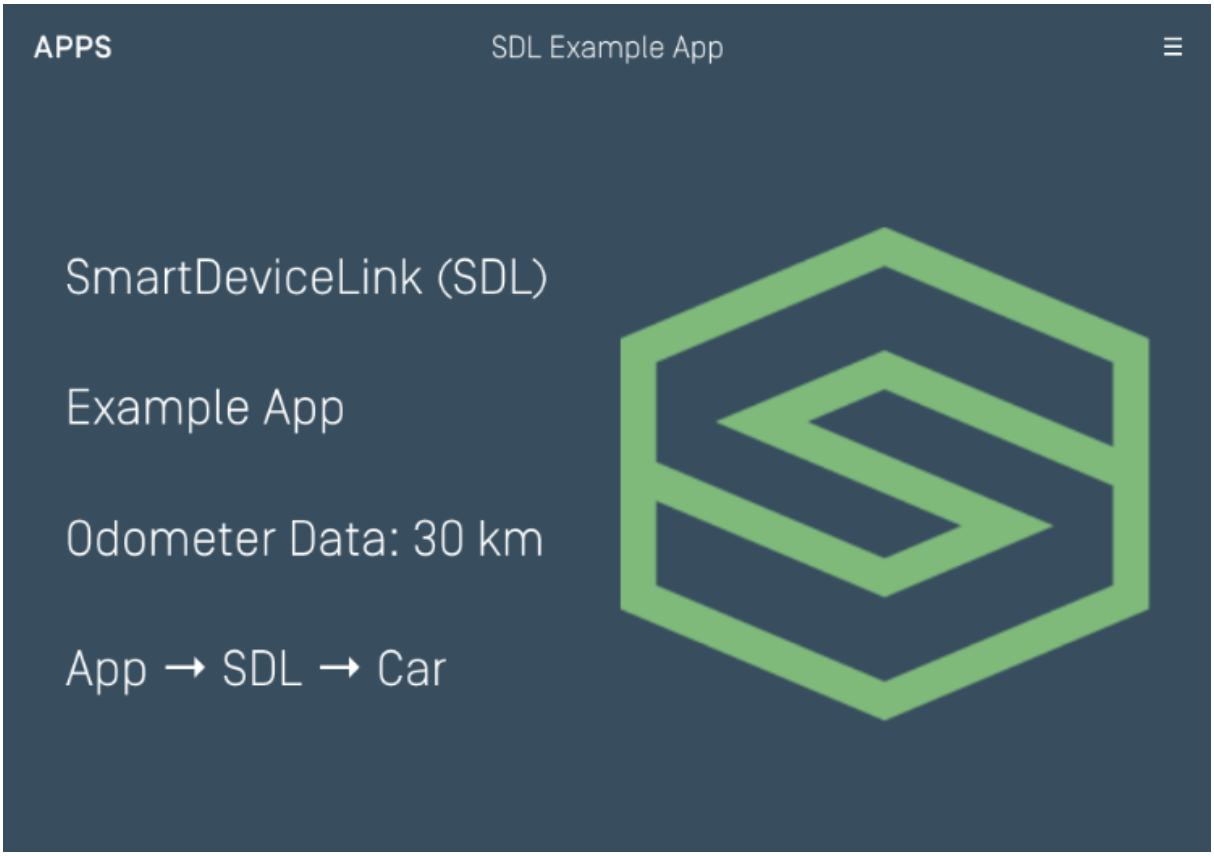


NON-MEDIA

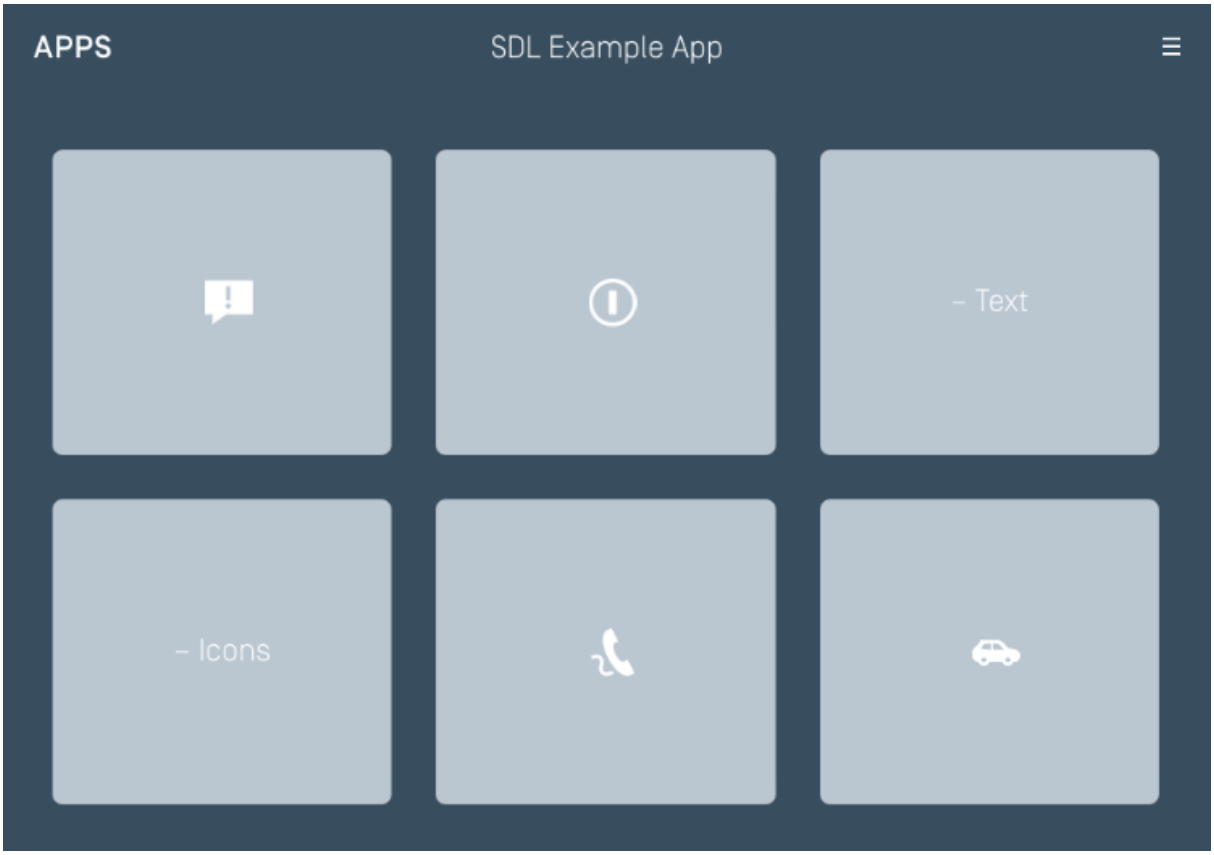

APPS
SDL Example App
SmartDeviceLink (SDL)
Odometer Data: 30 km
Example App
App → SDL → Car
- Text
- Icons

GRAPHIC WITH TEXT

TEXT WITH GRAPHIC



TILES ONLY



GRAPHIC WITH TILES

SYNC 3 - Graphic with Tiles

TILES WITH GRAPHIC

SYNC 3 - Tiles with Graphic

GRAPHIC WITH TEXT AND SOFT BUTTONS

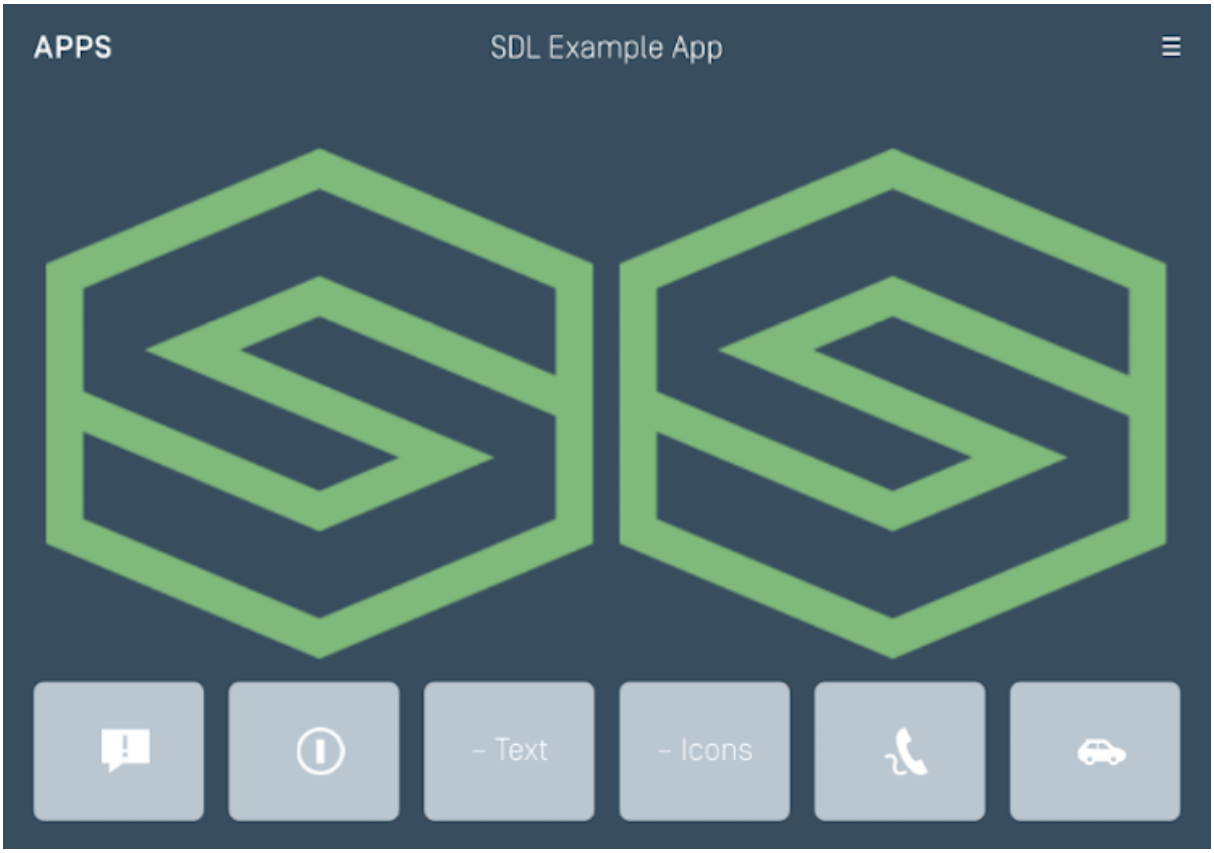
SYNC 3 - Graphic with Text and Soft Buttons

TEXT AND SOFT BUTTONS WITH GRAPHIC

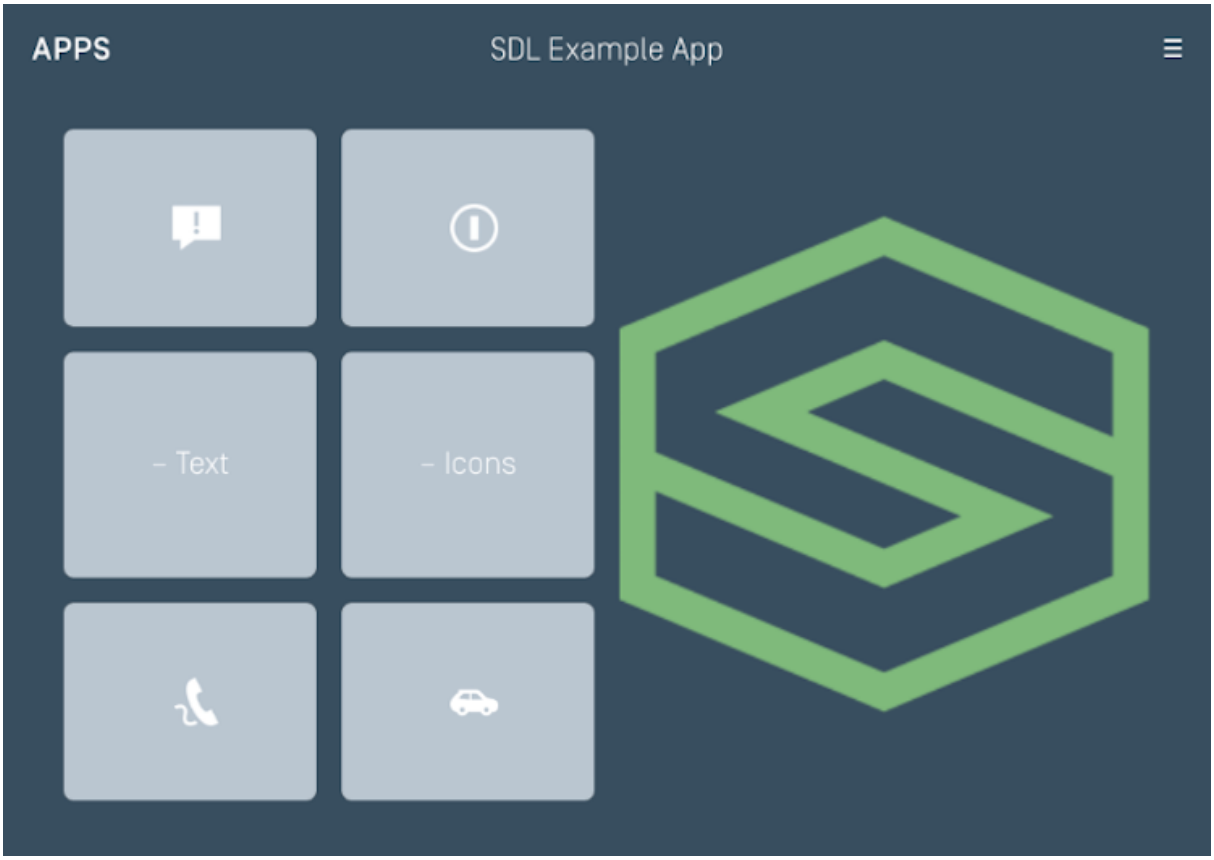
SYNC 3 Text and Softbuttons with Graphic

GRAPHIC WITH TEXT BUTTONS

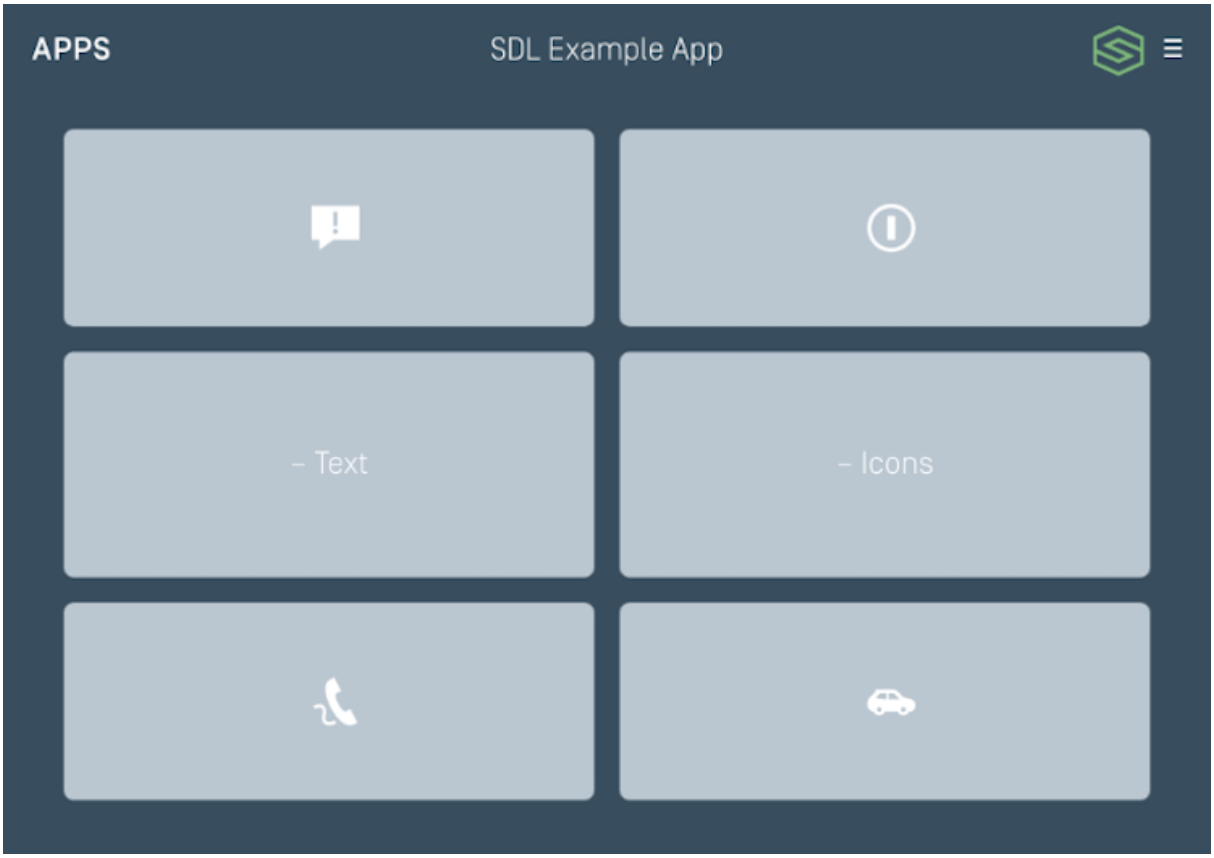
DOUBLE GRAPHIC WITH SOFT BUTTONS



TEXT BUTTONS WITH GRAPHIC



TEXT BUTTONS ONLY



LARGE GRAPHIC WITH SOFT BUTTONS

LARGE GRAPHIC ONLY

Text, Images, and Buttons

This guide covers presenting text and images on the screen as well as creating, showing, and responding to custom buttons you create.

Template Fields

The `ScreenManager` is a manager for easily creating text, images and soft buttons for your SDL app. To update the UI, simply give the manager the new UI data and sandwich the update between the manager's `beginTransaction()` and `commit()` methods.

SDLSCREENMANAGER PARAMETER NAME	DESCRIPTION
textField1	The text displayed in a single-line display, or in the upper display line of a multi-line display
textField2	The text displayed on the second display line of a multi-line display
textField3	The text displayed on the third display line of a multi-line display
textField4	The text displayed on the bottom display line of a multi-line display
mediaTrackTextField	The text displayed in the in the track field. This field is only valid for media applications
primaryGraphic	The primary image in a template that supports images
secondaryGraphic	The second image in a template that supports multiple images
textAlignment	The text justification for the text fields. The text alignment can be left, center, or right
softButtonObjects	An array of buttons. Each template supports a different number of soft buttons
textField1Type	The type of data provided in `textField1`
textField2Type	The type of data provided in `textField2`
textField3Type	The type of data provided in `textField3`
textField4Type	The type of data provided in `textField4`

```
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);
    }
});
```

Removing Text and Images

After you have displayed text and graphics onto the screen, you may want to remove those from being displayed. In order to do so, you only need to set the screen manager property to `null`.

```
sdlManager.getScreenManager().setTextField1(null);
sdlManager.getScreenManager().setTextField2(null);
sdlManager.getScreenManager().setPrimaryGraphic(null);
```

Soft Button Objects

To create a soft button using the `ScreenManager`, you only need to create a custom name for the button and provide the text for the button's label and/or an image for the button's icon. If your button cycles between different states (e.g. a button used to set the repeat state of a song playlist can have three

states: repeat-off, repeat-one, and repeat-all) you can upload all the states on initialization. Soft Buttons can contain images, text or both.

Soft Button Layouts

SOFT BUTTON (TEXT ONLY)

```
SoftButtonState textState = new SoftButtonState("<#State Name#>",
"<#Button Label Text#>", null);
SoftButtonObject softButtonObject = new SoftButtonObject(
"softButtonObject", Collections.singletonList(textState), textState.
getName(), new SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {
    }

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

    }
});

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

SOFT BUTTON (IMAGE ONLY)

To see if soft buttons support images you should check the `getGraphicSupported()` method on `SdlManager`s `SoftButtonCapabilities` using `SystemCapabilityManager`.

```
Object softButtonCapabilities = sdlManager.
getSystemCapabilityManager().getCapability(SystemCapabilityType.
SOFTBUTTON);
List<SoftButtonCapabilities> softButtonCapabilitiesList =
SystemCapabilityManager.convertToList(softButtonCapabilities,
SoftButtonCapabilities.class);
boolean imageSupported = false;
if (softButtonCapabilities != null && !softButtonCapabilitiesList.isEmpty
() && softButtonCapabilitiesList.get(0).getImageSupported()){
    imageSupported = true;
}


if (imageSupported) {
    SoftButtonState state = new SoftButtonState("<#State Name#>",
null, imageArtwork);
    SoftButtonObject softButtonObject = new SoftButtonObject(
"softButtonObject", Collections.singletonList(state), state.getName(),
new SoftButtonObject.OnEventListener() {
        @Override
        public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {
        }

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

        }
    });

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

SOFT BUTTON (IMAGE AND TEXT)

```
SoftButtonState state = new SoftButtonState("<#State Name#>",
"<#Button Label Text#>", imageArtwork);
SoftButtonObject softButtonObject = new SoftButtonObject(
"softButtonObject", Collections.singletonList(state), state.getName(),
new SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {
    }

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

    }
});

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

Updating the Soft Button State

When the soft button state needs to be updated, simply tell the `SoftButtonObject` to transition to the next state. If your button states do not cycle in a predictable order, you can also tell the soft button the state to transition to by passing the `stateName` of the new soft button state.

```
SoftButtonState state1 = new SoftButtonState("<#State1 Name#>",
"<#Button1 Label Text#>", image1Artwork);
SoftButtonState state2 = new SoftButtonState("<#State2 Name#>",
"<#Button2 Label Text#>", image2Artwork);

SoftButtonObject softButtonObject = new SoftButtonObject(
"softButtonObject", Arrays.asList(state1, state2), state1.getName(),
new SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject,
OnButtonPress onButtonPress) {

    }

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

    }
});

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


// Transition to a new state
SoftButtonObject retrievedSoftButtonObject = sdlManager.
getScreenManager().getSoftButtonObjectByName("softButtonObject");
retrievedSoftButtonObject.transitionToNextState();
```

Deleting Soft Buttons

To delete soft buttons, simply pass the screen manager a new array of soft buttons. To delete all soft buttons, simply pass the screen manager an empty array.

```
sdlManager.getScreenManager().setSoftButtonObjects(Collections.
EMPTY_LIST);
```

Templating Images

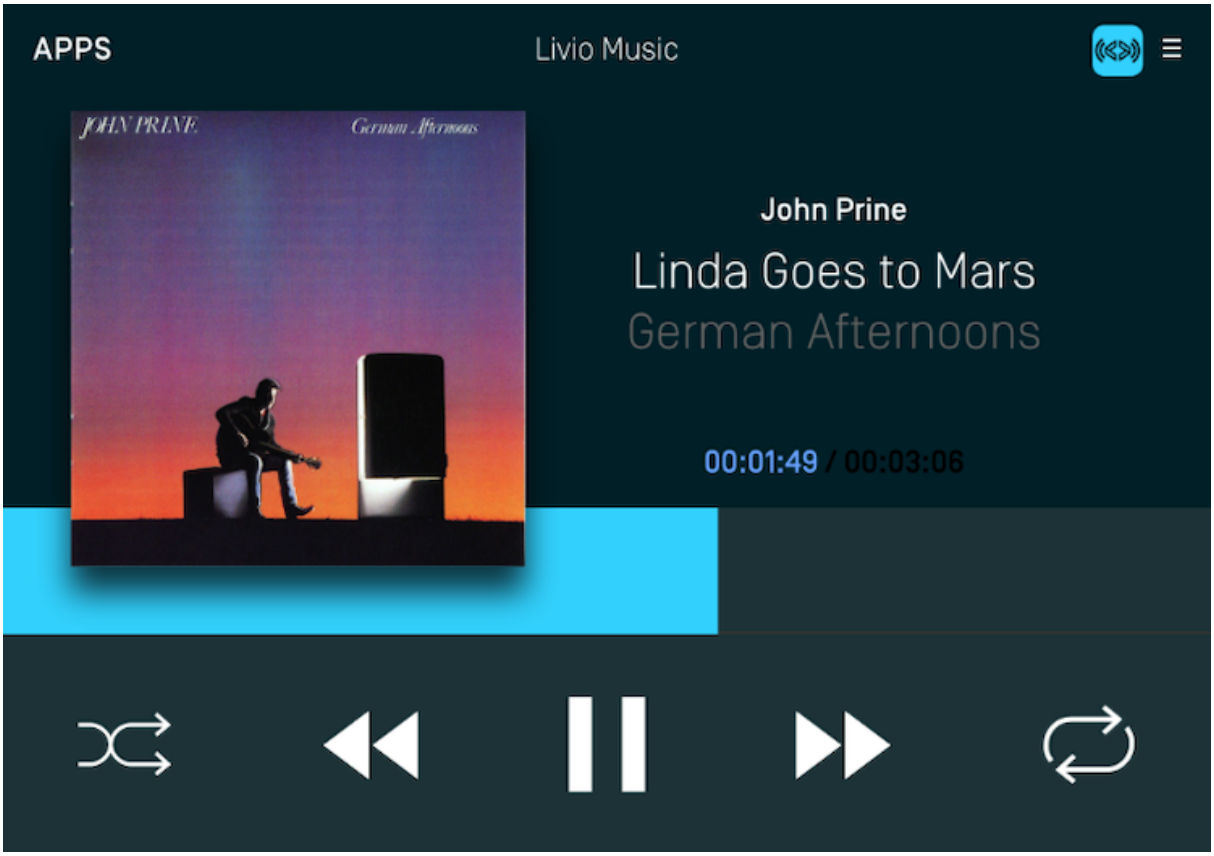
When connected to a remote system running SDL Core 5.0+, you may be able to use template images. Templated images are tinted by Core so the image is visible regardless of whether your user has set the head unit to day or night mode. For example, if a head unit is in night mode with a dark theme (see Template Coloring in the Integration Basics section for more details on how to customize theme colors), then your templated images will be displayed as white. In the day theme, the image will automatically change to black.

Soft buttons, menu icons, and primary / secondary graphics can all be templated. Images that you wish to template must be PNGs with a transparent background and only one color for the icon. Therefore, templating is only useful for things like icons and not for images that must be rendered in a specific color.

TEMPLATED IMAGES EXAMPLE

In the screenshots below, the shuffle and repeat icons have been templated. In night mode, the icons are tinted white and in day mode the icons are tinted black.

NIGHT MODE


APPS
Livio Music
JOHN PRINE
German Afternoons
John Prine
Linda Goes to Mars
German Afternoons
00:01:49 / 00:03:06

DAY MODE


```
SdlArtwork image = new SdlArtwork("ArtworkName", FileType.
GRAPHIC_PNG, R.drawable.artworkName, true);
image.setTemplateImage(true);
```

Static Icons

Static icons are pre-existing images on the remote system that you may reference and use in your own application. Static icons are fully supported by the screen manager via an `SdlArtwork` initializer.

Static icons can be used in primary and secondary graphic fields, soft button image fields, and menu icon fields.

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

Using RPCs

If you don't want to use the screen manager, you can just send raw `Show` RPC requests to Core.

Subscribing to System Buttons

Subscribe buttons are used to detect changes to hard buttons located in the car's center console or steering wheel. You can subscribe to the following hard buttons:

BUTTON	TEMPLATE	BUTTON TYPE
Play / Pause	media template only	soft button and hard button
Ok	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

Subscribe Buttons HMI

In the screenshot below, the pause, seek left and seek right icons are subscribe buttons.

NOTE

There is no way to customize a subscribe button's image or text.

Audio-Related Buttons

The play/pause, seek left, seek right, tune up, and tune down subscribe buttons can only be used in the `MEDIA` template. Depending on the manufacturer of the head unit, the subscribe button might also show up as a soft button in the media template. For example, the SYNC 3 HMI will add the ok, seek right, and seek left soft buttons to the media template when you subscribe to those buttons. You will automatically be assigned the media template if you set your app's `appType` to `MEDIA`.

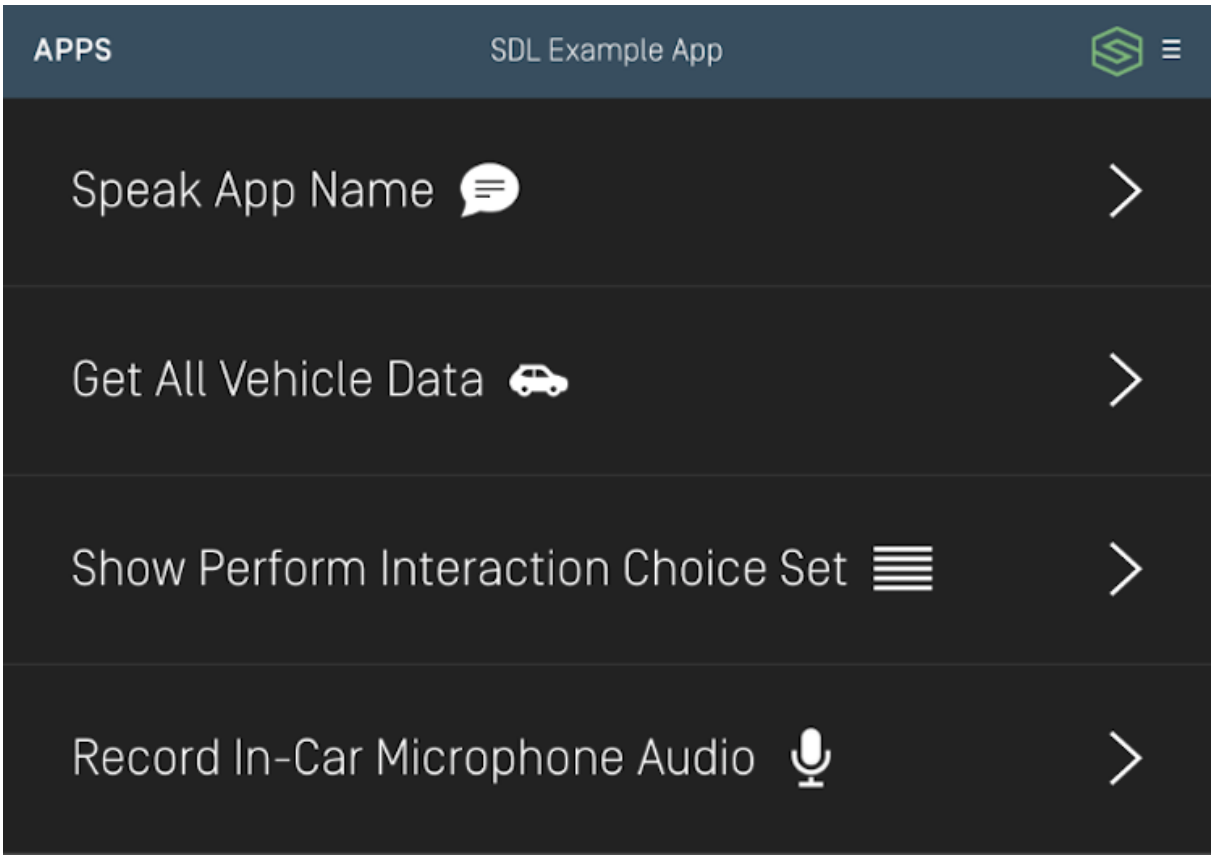
> NOTE
>
> Before library v.4.7 and SDL Core v.5.0, `Ok` and `PlayPause` were combined into `Ok`. Subscribing to `Ok` will, in v.4.7, also subscribe you to `PlayPause`. This means that for the time being, *you should not simultaneously subscribe to* `Ok` *and* `PlayPause`. In a future major version, this will change. For now, only subscribe to either `Ok` or `PlayPause` and the library will execute the right action based on the version of Core to which you are connected.

```
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 PLAY_PAUSE:
                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 PLAY_PAUSE:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
    }
});

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

Main Menu

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. You can find more information about these popups in the Popup Menus and Keyboards section.

MENU TEMPLATE



NOTE

Every template has a main menu button. The position of this button varies between templates and cannot be removed from the template. Some OEMs may format certain templates to not display the main menu button if you have no menu items (such as the navigation map view).

Adding Menu Items

The best way to create and update your menu is to the use the Screen Manager API. The screen manager contains two menu related properties: `menu`, and `voiceCommands`. Setting an array of `MenuCell`s into the `menu` property will automatically set and update your menu and submenus, while setting an array of `VoiceCommand`s into the `voiceCommands` property allows you to use "hidden" menu items that only contain voice recognition data. The user can then use the IVI system's voice engine to activate this command even though it will not be displayed within the main menu.

To find out more information on how to create `voiceCommands` see the related documentation.

```
// Create the menu cell
MenuCell cell = new MenuCell("Cell text", null, Collections.singletonList(
"cell text"), new MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        // Menu item was selected, check the `triggerSource` to know if
the user used touch or voice to activate it
        // <#Handle the Cell's Selection#>
    }
});

sdlManager.getScreenManager().setMenu(Collections.singletonList(cell
));
```

Adding Submenus

Adding a submenu is as simple as adding subcells to a `SdlMenuCell`. The submenu is automatically displayed when selected by the user. Currently menus only support one layer of subcells.

```
// Create the inner menu cell
MenuCell innerCell = new MenuCell("inner menu cell", null, Collections.
singletonList("inner menu cell"), new MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        // Menu item was selected, check the `triggerSource` to know if
the user used touch or voice to activate it
        // <#Handle the cell's selection#>
    }
});

// Create and set the submenu cell
MenuCell cell = new MenuCell("cell", null, Collections.singletonList(
innerCell));

sdlManager.getScreenManager().setMenu(Collections.singletonList(cell
));
```

Artworks

Artworks will be automatically handled when using the screen manager API. First, a "non-artwork" menu will be displayed, then, when the artworks have finished uploading, the "artwork-ified" menu will be displayed. If you are doing this manually with RPCs, you will have to upload artworks using the file manager yourself and send the correct menu when they are ready.

Deleting and Changing Menu Items

The screen manager will intelligently handle deletions for you. If you want to show new menu items, simply set a new array of menu cells. If you want to have a blank menu, set an empty array. On supported systems, the library will calculate the optimal adds / deletes to create the new menu. If the system doesn't support this sort of dynamic updating, the entire list will be removed and re-added.

If you are doing this manually, you must use the `DeleteCommand` and `DeleteSubMenu` RPCs, passing the `cmdID`s you wish to delete.

Using RPCs

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.

To create a submenu using RPCs, you must use a `AddSubMenu` RPC with a unique id. 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.

Popup Menus and Keyboards

SDL supports modal menus and keyboards. The user can respond to the list of menu options via touch, voice (if voice recognition supported by the head unit), or keyboard input.

There are several UX considerations to take into account when designing your menus. The main menu should should not be updated very often and should act as navigation for your app. Popup menus should be used to present a selection of options to your user. They can also be used to show a keyboard that lets your user perform a search or provide user input.

Presenting a Popup Menu

Presenting a popup menu is similiar to presenting a modal view to request input from your user. It is possible to chain together menus to drill down, however, it is recommended to do so judiciously. Requesting too much input from a driver while they are driving is distracting and may result in your app being rejected by OEMs.

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

Creating Cells

An `ChoiceCell` is similar to a `RecyclerView` without the ability to configure your own UI. We provide several properties on the `ChoiceCell` to set your data, but the layout itself is determined by the manufacturer of the head unit.

> NOTE
>
> On many systems, including VR commands will be *exponentially* slower than not including them. However, including them is necessary for a user to be able to respond to your prompt with their voice.

```
ChoiceCell cell = new ChoiceCell("cell1 text", Collections.singletonList(
"cell1"), null);
ChoiceCell fullCell = new ChoiceCell("cell2 text", "cell2 secondaryText",
"cell2 tertiaryText", Collections.singletonList("cell2"), image1Artwork,
image2Artwork);
```

Preloading Cells

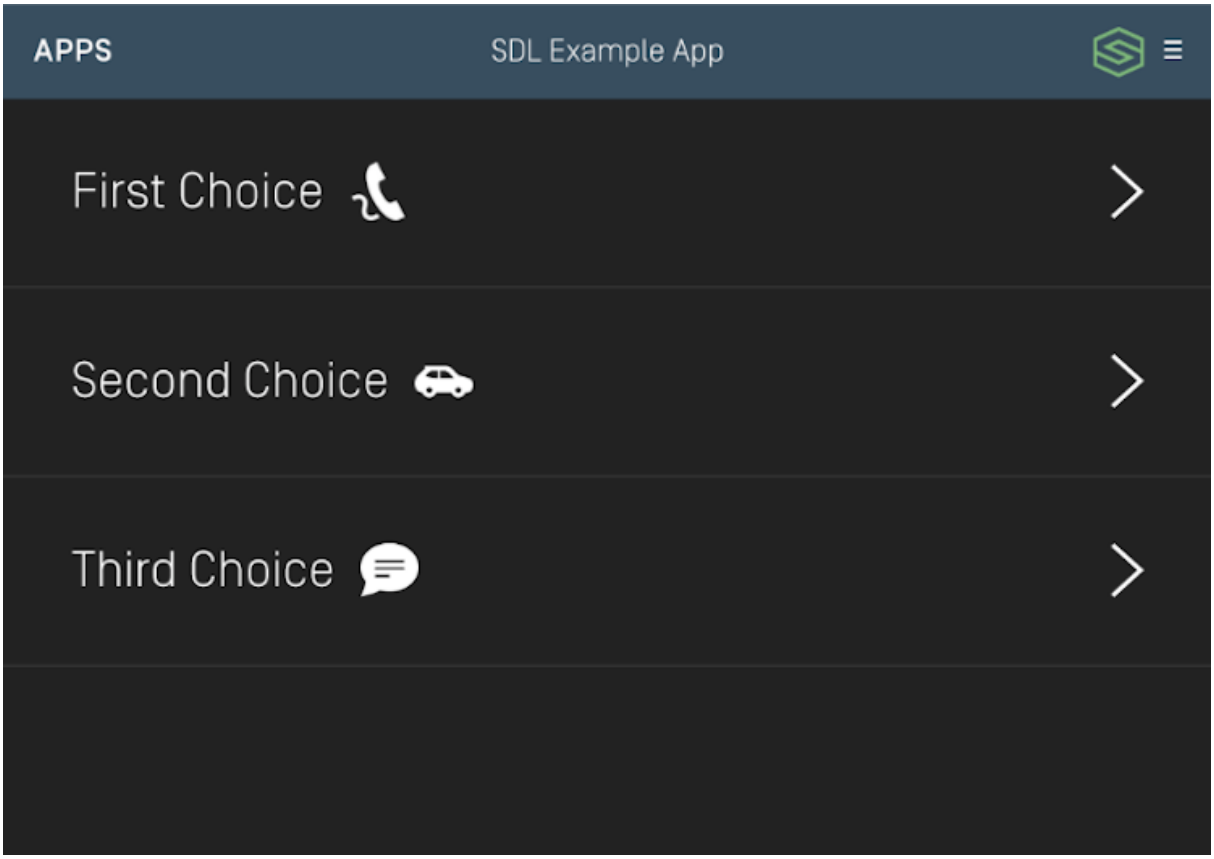
If you know the content you will show in the popup menu long before the menu is shown to the user, you can "preload" those cells in order to speed up the popup menu presentation at a later time. Once you preload a cell, you can reuse it in multiple popup menus without having to send the cell content to Core again.

```
sdlManager.getScreenManager().preloadChoices(Arrays.asList(cell,
fullCell), new CompletionListener() {
    @Override
    public void onComplete(boolean b) {
        // <#code#>
    }
});
```

Presenting a Menu

To show a popup menu to the user, you must present the menu. If some or all of the cells in the menu have not yet been preloaded, calling the `present` API will preload the cells and then present the menu once all the cells have been uploaded. Calling `present` without preloading the cells can take longer than if the cells were preloaded earlier in the app's lifecycle especially if your cell has voice commands. Subsequent menu presentations using the same cells will be faster because the library will reuse those cells (unless you have deleted them).

MENU - LIST



MENU - ICON

SYNC 3 - Icon Only Interaction Layout

NOTE

When you preload a cell, you **do not** need to maintain a reference to it. If you reuse a cell with the same properties that has already been preloaded (or previously presented), the cell will automatically be reused.

CREATING A CHOICE SET

In order to present a menu, you must bundle together a bunch of `ChoiceCell`s into an `ChoiceSet`.

> NOTE
>
> If the `ChoiceSet` contains an invalid set of `ChoiceCell`s, presenting the `ChoiceSet` will fail. This can happen, for example, if you have duplicate title text or if some, but not all choices have voice commands.

Some notes on various parameters (full documentation is available as API documentation on this website):

- Title: This is the title of the menu when presented
- Listeners: You must implement this listener interface to receive callbacks based on the user's interaction with the menu
- Layout: You may present your menu as a set of tiles (like a `GridView`) or a list (like a `RecyclerView`). If you are using tiles, it's recommended to use artworks on each item.

```
ChoiceSet choiceSet = new ChoiceSet("ChoiceSet Title", Arrays.asList(
cell, fullCell), new ChoiceSetSelectionListener() {
    @Override
    public void onChoiceSelected(ChoiceCell choiceCell, TriggerSource
triggerSource, int rowIndex) {
        // You will be passed the `cell` that was selected, the manner in
which it was selected (voice or text), and the index of the cell that was
passed.
        // <#handle selection#>
    }

    @Override
    public void onError(String error) {
        // <#handle error#>
    }
});
```

PRESENTING THE MENU WITH A MODE

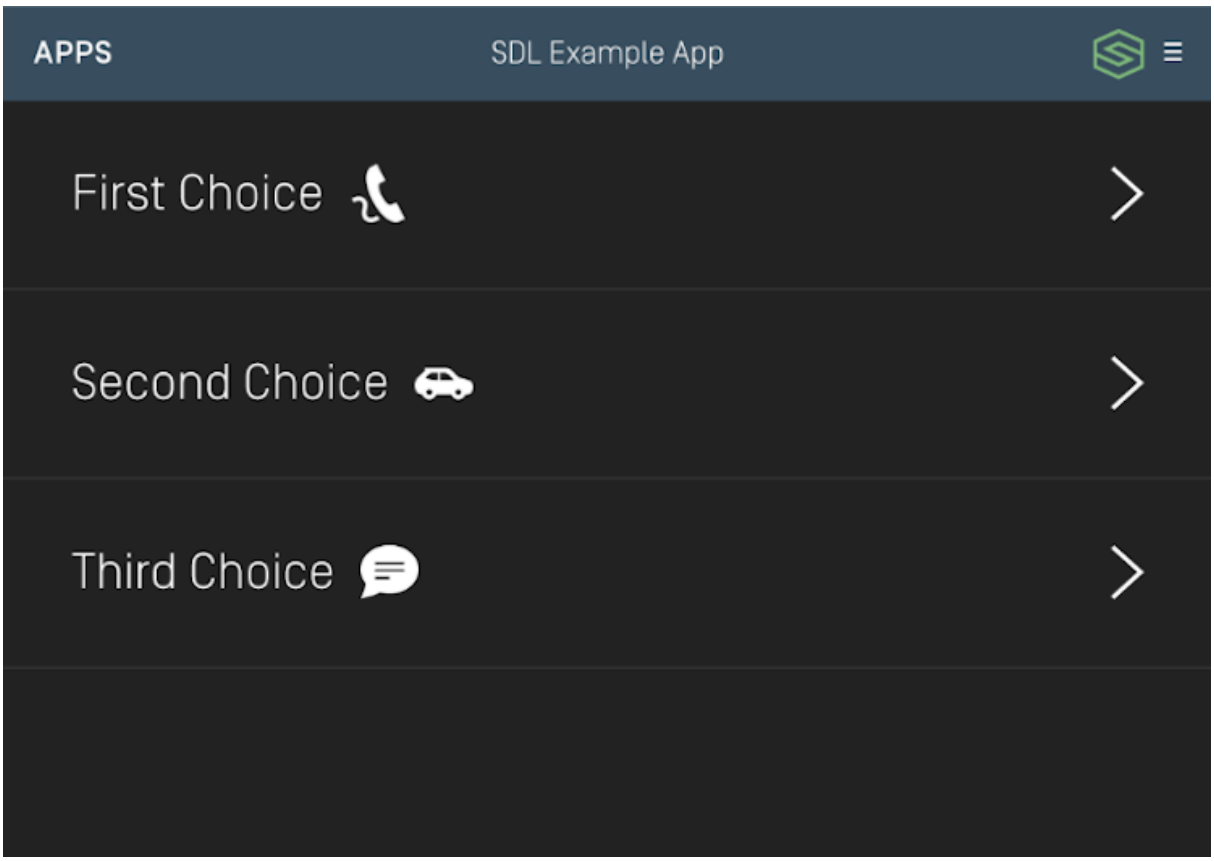
Finally, you will present the menu. When you do so, you must choose a `mode` to present it in. If you have no `vrCommands` on the choice cell you should choose `manualOnly`. If `vrCommands` are available, you may choose `voiceRecognitionOnly` or `both`.

You may want to choose this based on the trigger source leading to the menu being presented. For example, if the menu was presented via the user touching the screen, you may want to use a `mode` of `manualOnly` or `both`, but if the menu was presented via the user speaking a voice command, you may want to use a `mode` of `voiceRecognitionOnly` or `both`.

It may seem that the answer is to always use `both`. However, remember that you must provide `vrCommand`s on all cells to use `both`, which is exponentially slower than not providing `vrCommand`s (this is especially relevant for large menus, but less important for smaller ones). Also, some head units may not provide a good user experience for `both`.

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

MENU - MANUAL ONLY MODE



MENU - VOICE ONLY MODE

SYNC 3 - Menu - Voice Only

```
sdlManager.getScreenManager().presentChoiceSet(choiceSet,
InteractionMode.MANUAL_ONLY);
```

Presenting a Searchable Menu

In addition to presenting a standard menu, you can also present a "searchable" menu, that is, a menu with a keyboard input box at the top. For more information on implementing the keyboard portion of this menu, see *Presenting a Keyboard* below.

MENU WITH SEARCH

SYNC 3 - Menu with Search Interaction Layout

```
sdlManager.getScreenManager().presentSearchableChoiceSet(
choiceSet, InteractionMode.MANUAL_ONLY, keyboardListener);
```

Deleting Cells

You can discover cells that have been preloaded on `sdlManager.getScreenManager().getPreloadedChoices()`. You may then pass an array of cells to delete from the remote system. Many times this is not necessary, but if you have deleted artwork used by cells, for example, you should delete the cells as well.

```
sdlManager.getScreenManager().deleteChoices(<List of choices to
delete>);
```

Presenting a Keyboard

Presenting a keyboard or a searchable menu requires you to additionally implement the `KeyboardListener`. Note that the `initialText` in the keyboard case often acts as "placeholder text" *not* as true initial text.

> NOTE
>
> Keyboards are unavailable for use in many countries when the driver is distracted. This is often when the vehicle is moving above a certain speed, such as 5 miles per hour.

KEYBOARD SEARCH

SYNC 3 - Keyboard Search

```
sdlManager.getScreenManager().presentKeyboard("Initial text", null,
keyboardListener);
```

Implementing the Keyboard Listeners

Using the `KeyboardListener` involves implementing five methods:

```
KeyboardListener keyboardListener = new KeyboardListener() {
    @Override
    public void onUserDidSubmitInput(String inputText, KeyboardEvent
event) {
        switch (event) {
            case ENTRY_VOICE:
                // <#The user decided to start voice input, you should start
an AudioPassThru session if supported#>
                break;
            case ENTRY_SUBMITTED:
                // <#The user submitted some text with the keyboard#>
                break;
            default:
                break;
        }
    }

    @Override
    public void onKeyboardDidAbortWithReason(KeyboardEvent event) {
        switch (event) {
            case ENTRY_CANCELLED:
                // <#The user cancelled the keyboard interaction#>
                break;
            case ENTRY_ABORTED:
                // <#The system aborted the keyboard interaction#>
                break;
            default:
                break;
        }
    }

    @Override
    public void updateAutocompleteWithInput(String currentInputText,
KeyboardAutocompleteCompletionListener
keyboardAutocompleteCompletionListener) {
        // <#Check the input text and return a string with the current
autocomplete text#>
    }

    @Override
    public void updateCharacterSetWithInput(String currentInputText,
KeyboardCharacterSetCompletionListener
keyboardCharacterSetCompletionListener) {
        // <#Check the input text and return a set of characters to allow
the user to enter#>
    }

    @Override
    public void onKeyboardDidSendEvent(KeyboardEvent event, String
```

```
currentInputText) {
        // <#This is sent upon every event, such as keypresses,
cancellations, and aborting#>
    }
};
```

Using RPCs

If you don't want to use the `ScreenManager`, you can do this manually using the `Choice`, `CreateInteractionChoiceSet`, and `PerformInteraction` RPC requests. You will need to create `Choice`s, bundle them into `CreateInteractionChoiceSet`s, and then present those choice sets via a `PerformInteraction` request. As this is no longer a recommended course of action, we will leave it to you to figure out how to manually do it.

Alerts

An alert is a pop-up window showing a short message with optional 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.

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.

NOTE

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.

Alert Layouts

Alert With No Soft Buttons

NOTE

If no soft buttons are added to an alert some OEMs may add a default "cancel" or "close" button.

Alert With Soft Buttons

Creating the Alert

Text

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

Buttons

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

// Soft buttons
final int softButtonId = 123; // Set it to any unique ID
SoftButton okButton = new SoftButton(SoftButtonType.SBT_TEXT,
softButtonId);
okButton.setText("OK");

// Set the softbuttons(s) to the alert
alert.setSoftButtons(Collections.singletonList(okButton));

// This listener is only needed once, and will work for all of soft buttons
you send with your alert
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_PRESS, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPress = (OnButtonPress) notification;
        if (onButtonPress.getCustomButtonName() == softButtonId){
            Log.i(TAG, "Ok button pressed");
        }
    }
});
```

Timeouts

An optional timeout can be added that will dismiss the alert when the duration is over. Typical timeouts are between 3 and 10 seconds. If omitted a default of 5 seconds is used.

```
alert.setDuration(5000);
```

Progress Indicator

Not all OEMs support a progress indicator. If supported, the alert will show an animation that indicates that the user must wait (e.g. a spinning wheel or hourglass, etc). If omitted, no progress indicator will be shown.

```
alert.setProgressIndicator(true);
```

Text-To-Speech

An alert can also speak a prompt or play a sound file when the alert appears on the screen. This is done by setting the `ttsChunks` parameter.

TEXT

```
alert.setTtsChunks(TTSChunkFactory.createSimpleTTSChunks("Text to Speak"));
```

SOUND FILE

The `ttsChunks` parameter can also take a file to play/speak. For more information on how to upload the file please refer to the Playing Audio Indications guide.

```
TTSChunk ttsChunk = new TTSChunk(sdlFile.getName(),
SpeechCapabilities.FILE);
alert.setTtsChunks(Collections.singletonList(ttsChunk));
```

Play Tone

To play the alert tone when the alert appears and before the text-to-speech is spoken, set `playTone` to `true`.

```
alert.setPlayTone(true);
```

Showing the Alert

```
// Handle RPC response
alert.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
      if (response.getSuccess()){
        Log.i(TAG, "Alert was dismissed successfully");
      }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
      Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(alert);
```

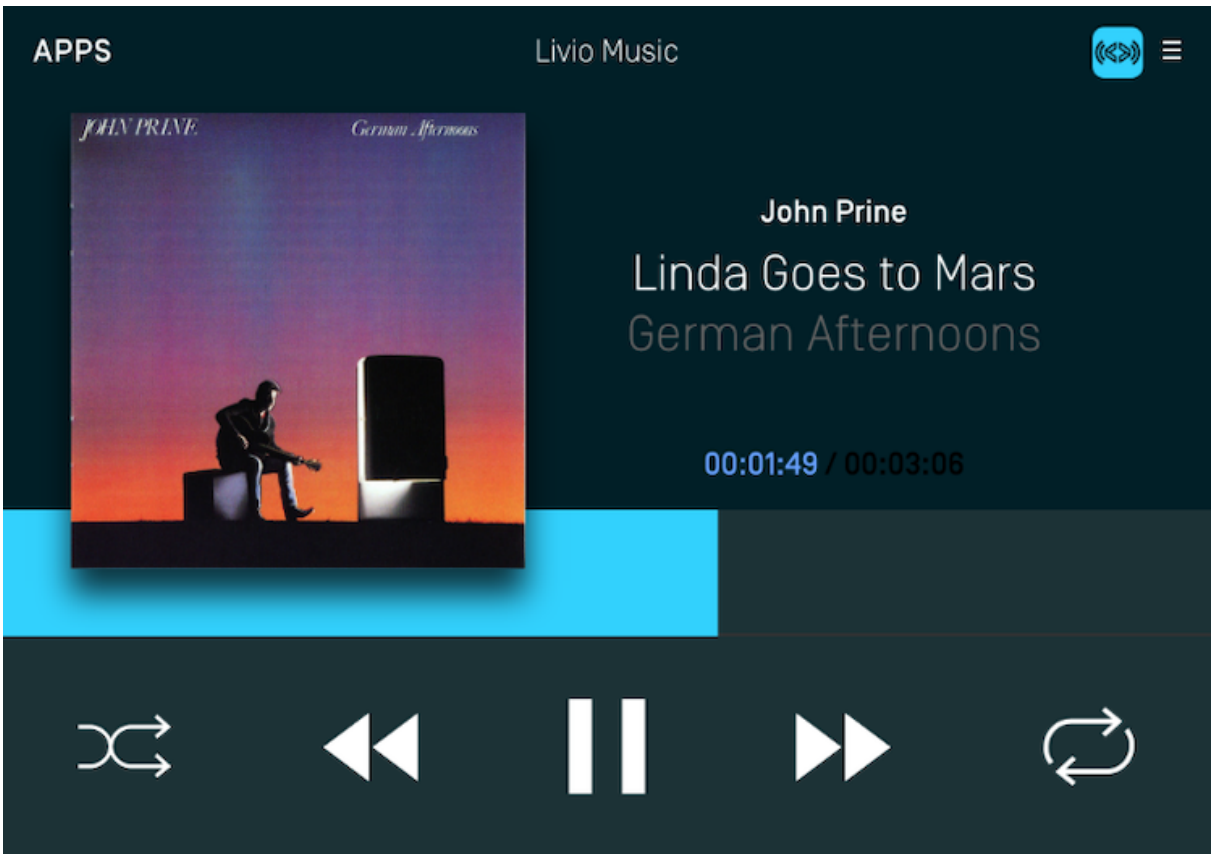
Media Clock

The media clock is used by media apps to present the current timing information of a playing media item such as a song, podcast, or audiobook.

The media clock constists of three parts: the progress bar, a current position label and a remaining time label. In addition you may want to update the play/pause button icon to reflect the current state of the audio.

NOTE

Ensure your app has an `appType` of media and you are using the media template before implementing this feature.



Counting Up

In order to count up using the timer, you will need to set a start time that is less than the end time. The "bottom end" of the media clock will always start at `0:00` and the "top end" will be the end time you specified. The start time can be

set to any position between 0 and the end time. For example, if you are starting a song at `0:30` and it ends at `4:13` the media clock timer progress bar will start at the `0:30` position and start incrementing up automatically every second until it reaches `4:13`. The current position label will start counting upwards from `0:30` and the remaining time label will start counting down from `3:43`. When the end is reached, the current time label will read `4:13`, the remaining time label will read `0:00` and the progress bar will stop moving.

The play / pause indicator parameter is used to update the play / pause button to your desired button type. This is explained below in the section "Updating the Audio Indicator"

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
countUpFromStartTimeInterval(30, 253, AudioStreamingIndicator.PAUSE
);
sdlManager.sendRPC(mediaClock);
```

Counting Down

Counting down is the opposite of counting up (I know, right?). In order to count down using the timer, you will need to set a start time that is greater than the end time. The timer bar moves from right to left and the timer will automatically count down. For example, if you're counting down from `10:00` to `0:00`, the progress bar will be at the leftmost position and start decrementing every second until it reaches `0:00`.

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
countDownFromStartTimeInterval(600, 0, AudioStreamingIndicator.
PAUSE);
sdlManager.sendRPC(mediaClock);
```

Pausing & Resuming

When pausing the timer, it will stop the timer as soon as the request is received and processed. When a resume request is sent, the timer begins again at the paused time as soon as the request is processed. You can update the start and end times using a pause command to change the timer while remaining paused.

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
pauseWithPlayPauseIndicator(AudioStreamingIndicator.PLAY);
sdlManager.sendRPC(mediaClock);
```

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
resumeWithPlayPauseIndicator(AudioStreamingIndicator.PAUSE);
sdlManager.sendRPC(mediaClock);
```

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
updatePauseWithNewStartTimeInterval(60, 240,
AudioStreamingIndicator.PLAY);
sdlManager.sendRPC(mediaClock);
```

Clearing the Timer

Clearing the timer removes it from the screen.

```
SetMediaClockTimer mediaClock = new SetMediaClockTimer().
clearWithPlayPauseIndicator(AudioStreamingIndicator.PLAY);
sdlManager.sendRPC(mediaClock);
```

Updating the Audio Indicator

The audio indicator is, essentially, the play / pause button. As of library v.4.7, when connected to an SDL v5.0+ head unit, you can tell the system what icon to display on the play / pause button to correspond with how your app works. For example, if audio is currently playing you can update the the play/pause button to show the pause icon. On older head units, the audio indicator shows an icon with both the play and pause indicators and the icon can not be updated.

For example, a radio app will probably want two button states: play and stop. A music app, in contrast, will probably want a play and pause button. If you don't send any audio indicator information, a play / pause button will be displayed.

Slider

A `Slider` creates a full screen or pop-up overlay (depending on platform) that a user can control. There are two main `Slider` layouts, one with a static footer and one with a dynamic footer.

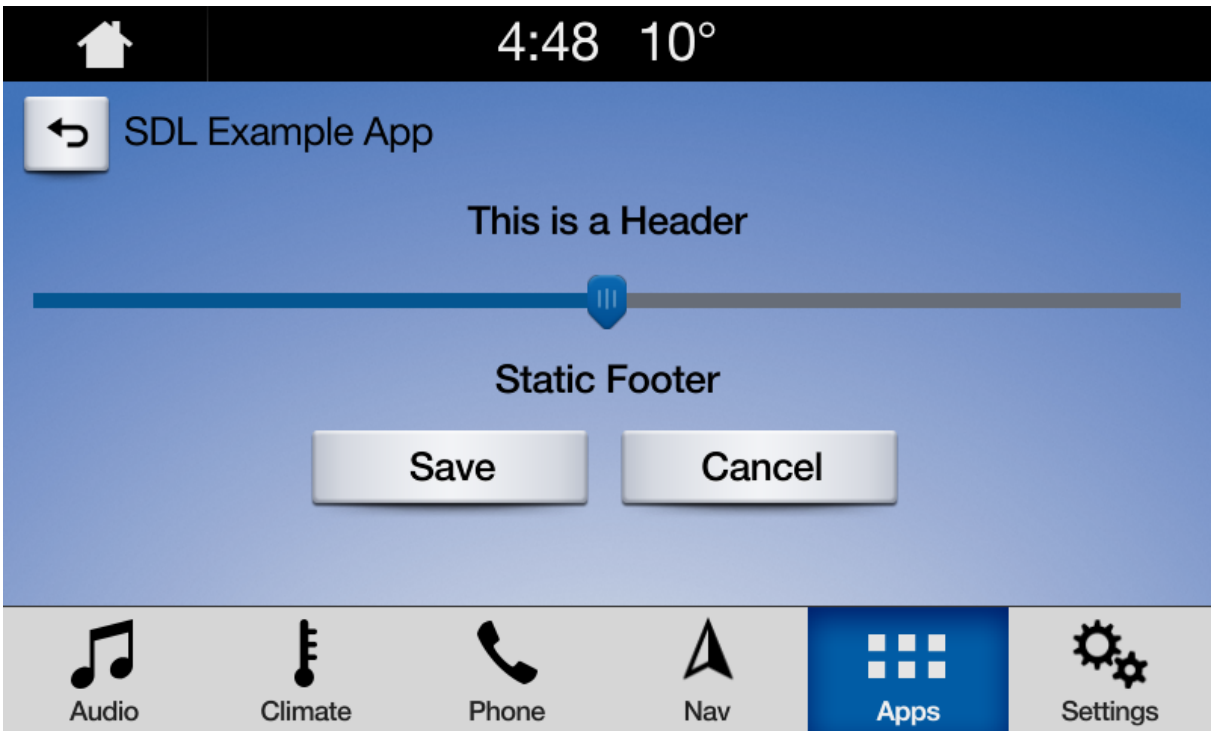
NOTE

The slider will persist on the screen until the timeout has elapsed or the user dismisses the slider by selecting a position or canceling.

Slider with Static Footer

A slider popup with a static footer displays a single, optional, footer message below the slider UI.

Slider UI



Creating the Slider

```
//Create a slider
Slider slider = new Slider(5, 1, "This is a Header");

List<String> footer = Collections.singletonList("Static Footer");
slider.setSliderFooter(footer);
slider.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        SliderResponse sliderResponse = (SliderResponse) response;
        Log.i(TAG, "Slider Position Set: "+ sliderResponse.getSliderPosition
());
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

//Send Request
sdlManager.sendRPC(slider);
```

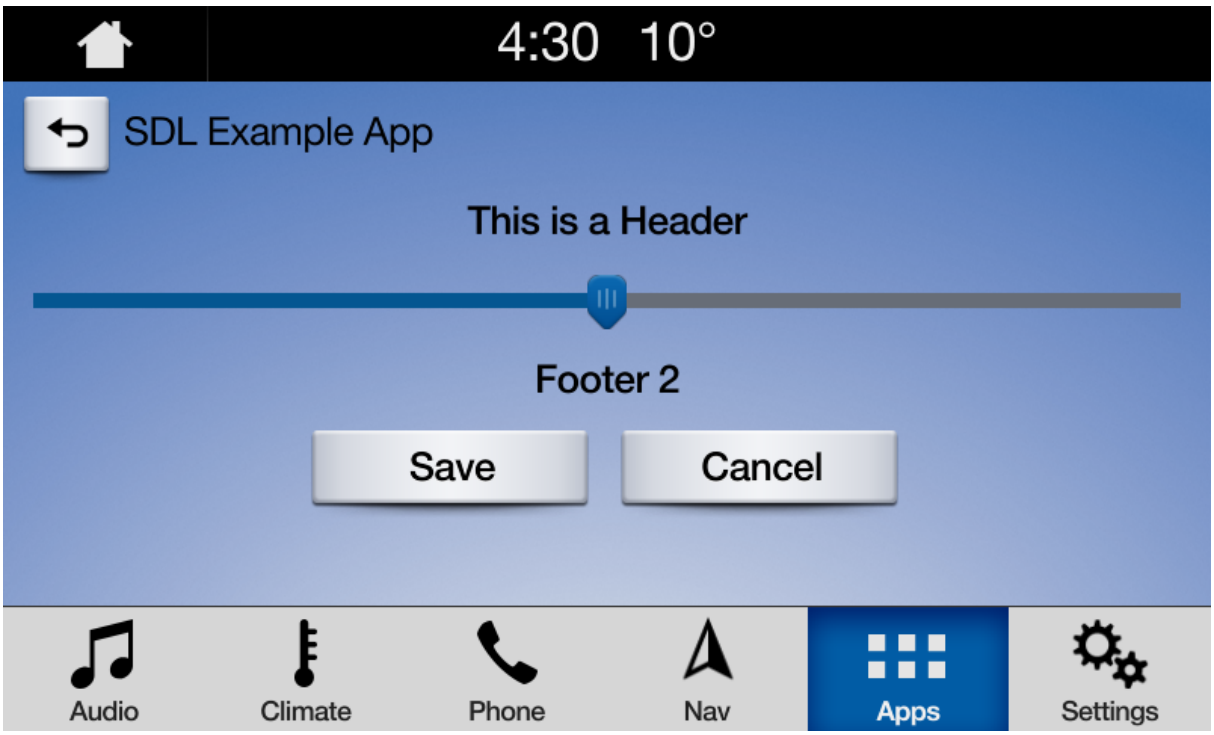
Slider with Dynamic Footer

This type of slider will have a different footer message displayed for each position of the slider. The footer is an optional paramater. The footer message displayed will be based off of the slider's current position. The footer array should be the same length as `numTicks` because each footer must correspond to a tick value. Or, you can pass `null` to have no footer at all.

Slider UI

DYNAMIC SLIDER IN POSITION 1

DYNAMIC SLIDER IN POSITION 2



Creating the Slider

```
//Create a slider
Slider slider = new Slider(3, 1, "This is a Header");

// Each footer corresponds with the slider's position
List<String> footer = Arrays.asList("Footer 1","Footer 2","Footer 3");
slider.setSliderFooter(footer);
slider.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        SliderResponse sliderResponse = (SliderResponse) response;
        Log.i(TAG, "Slider Position Set: "+ sliderResponse.getSliderPosition
());
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

//Send Request
sdlManager.sendRPC(slider);
```

Scrollable Message

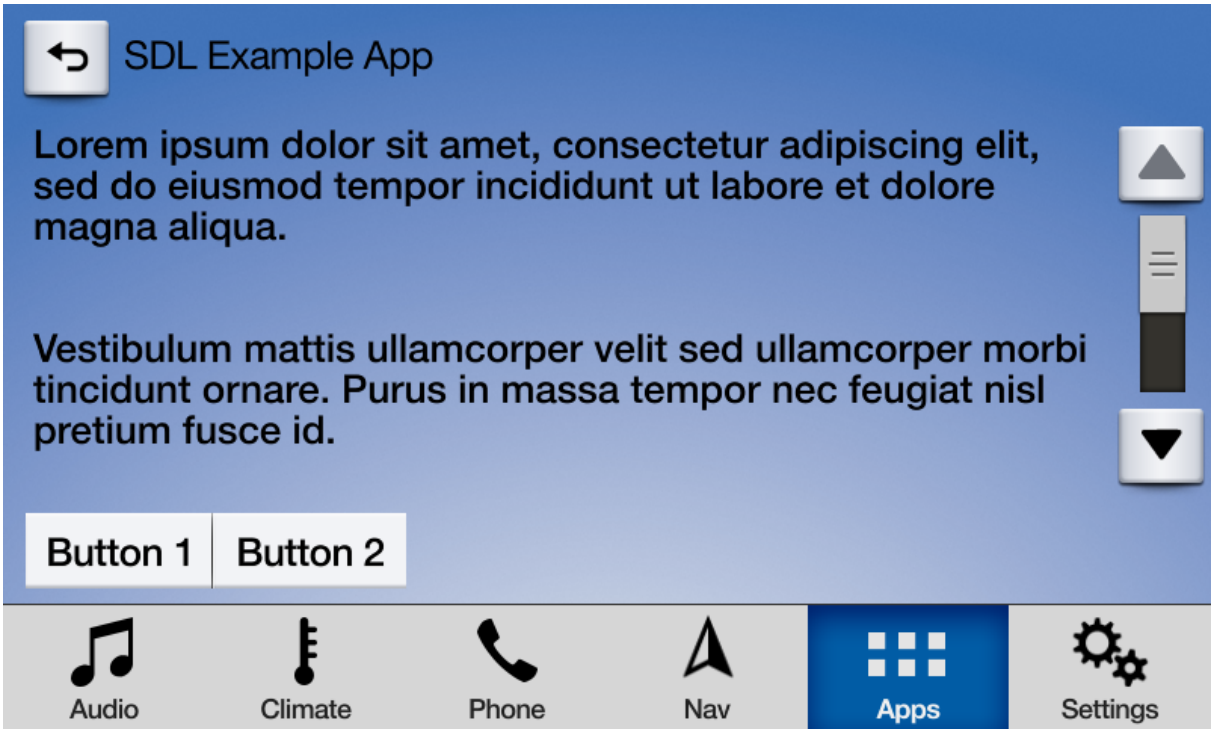
A `ScrollableMessage` creates an overlay containing a large block of formatted text that can be scrolled. `ScrollableMessage` contains a body of text, a message timeout, and up to 8 soft buttons depending on head unit. You must check the `DisplayCapabilities` to get the max number of `SoftButtons` allowed by the head unit for a `ScrollableMessage`.

You simply create a `ScrollableMessage` RPC request and send it to display the Scrollable Message.

NOTE

The message will persist on the screen until the timeout has elapsed or the user dismisses the message by selecting a soft button or cancelling (if the head unit provides cancel UI).

Scrollable Message UI



Creating the Scrollable Message

```
// Create Message To Display
String scrollableMessageText = "Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua.Vestibulum mattis ullamcorper velit sed
ullamcorper morbi tincidunt ornare. Purus in massa tempor nec feugiat
nisl pretium fusce id. Pharetra convallis posuere morbi leo urna
molestie at elementum eu. Dictum sit amet justo donec enim diam.";

// Create SoftButtons
SoftButton softButton1 = new SoftButton(SoftButtonType.SBT_TEXT, 0);
softButton1.setText("Button 1");

SoftButton softButton2 = new SoftButton(SoftButtonType.SBT_TEXT, 1);
softButton2.setText("Button 2");

// Create SoftButton Array
List<SoftButton> softButtonList = Arrays.asList(softButton1,
softButton2);

// Create ScrollableMessage Object
ScrollableMessage scrollableMessage = new ScrollableMessage();
scrollableMessage.setScrollableMessageBody(scrollableMessageText);
scrollableMessage.setTimeout(50000);
scrollableMessage.setSoftButtons(softButtonList);

// Send the scrollable message
sdlManager.sendRPC(scrollableMessage);
```

To listen for `OnButtonPress` events for `SoftButton`s, we need to add a listener that listens for their Id's:

```
sdlManager.addOnRPCNotificationListener(FunctionID.
ON_BUTTON_PRESS, new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPress = (OnButtonPress) notification;
        switch (onButtonPress.getCustomButtonName()){
            case 0:
                Log.i(TAG, "Button 1 Pressed");
                break;
            case 1:
                Log.i(TAG, "Button 2 Pressed");
                break;
        }
    }
});
```

Playing Spoken Feedback

Since your user will be driving while interacting with your SDL app, speech phrases can provide important feedback to your user. At any time during your app's lifecycle you can send a speech phrase using the `Speak` request and the head unit's text-to-speech (TTS) engine will produce synthesized speech from your provided text.

When using the `Speak` RPC, you will receive a response from the head unit once the operation has completed. From the response you will be able to tell if the speech was completed, interrupted, rejected or aborted. It is important to keep in mind that a speech request can interrupt another on-going speech

request. If you want to chain speech requests you must wait for for the current speech request to finish before sending the next speech request.

Creating the Speak Request

The speech request you send can simply be a text phrase, which will be played back in accordance with the user's current language settings, or it can consist of phoneme specifications to direct SDL's TTS engine to speak a language-independent, speech-sculpted phrase. It is also possible to play a pre-recorded sound file (such as an MP3) using the speech request. For more information on how to play a sound file please refer to Playing Audio Indications.

Getting Head Unit Speech Capabilities

To get the head unit's supported speech capabilities, check the `sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.SPEECH)` after successfully connecting to the head unit. Below is a list of commonly supported speech capabilities.

SPEECH CAPABILITY	DESCRIPTION
Text	Text phrases
SAPI Phonemes	Microsoft speech synthesis API
File	A pre-recorded sound file

Text Phrase

```
TTSChunk ttsChunk = new TTSChunk("hello", SpeechCapabilities.TEXT);
List<TTSChunk> ttsChunkList = Collections.singletonList(ttsChunk);
Speak speak = new Speak(ttsChunkList);
```

SAPI Phonemes Phrase

```
TTSChunk ttsChunk = new TTSChunk("h eh - l ow 1",
SpeechCapabilities.SAPI_PHONEMES);
List<TTSChunk> ttsChunkList = Collections.singletonList(ttsChunk);
Speak speak = new Speak(ttsChunkList);
```

Sending the Speak Request

```
speak.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        SpeakResponse speakResponse = (SpeakResponse) response;
        if (!speakResponse.getSuccess()){
            switch (speakResponse.getResultCode()){
                case DISALLOWED:
                    Log.i(TAG, "The app does not have permission to use the
speech request");
                    break;
                case REJECTED:
                    Log.i(TAG, "The request was rejected because a higher
priority request is in progress");
                    break;
                case ABORTED:
                    Log.i(TAG, "The request was aborted by another higher
priority request");
                    break;
                default:
                    Log.i(TAG, "Some other error occurred");
            }
            return;
        }
        Log.i(TAG, "Speech was successfully spoken");
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.i(TAG, "onError: " + info);
    }
});
sdlManager.sendRPC(speak);
```

Playing Audio Indications

As of library v.4.7 and SDL Core v.5.0+, you can pass an uploaded audio file's name to `TTSChunk`, allowing any API that takes a text-to-speech parameter to pass and play your audio file. A sports app, for example, could play a distinctive audio chime to notify the user of a score update alongside an `Alert` request.

Uploading the Audio File

The first step is to make sure the audio file is available on the remote system. To upload the file use the `FileManager`.

```
SdlFile audioFile = new SdlFile("Audio file name", FileType.AUDIO_MP3,
Uri.parse("File Location"), true);
sdlManager.getFileManager().uploadFile(audioFile, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

For more information about uploading files, see the Uploading Files guide.

Using the Audio File

Now that the file is uploaded to the remote system, it can be used in various RPCs, such as `Speak`, `Alert`, and `AlertManeuver`. To use the audio file in an alert, you simply need to construct a `TTSChunk` referring to the file's name.

```
Alert alert = new Alert();
alert.setAlertText1("Alert Text 1");
alert.setAlertText2("Alert Text 2");
alert.setDuration(5000);
alert.setTtsChunks(Arrays.asList(new TTSChunk("Audio file name",
SpeechCapabilities.FILE)));
```

Setting Up Voice Commands

Voice commands are global commands available anywhere on the head unit to users of your app. Once the user has opened your SDL app (i.e. your SDL app has left the HMI state of `NONE`) they have access to the voice commands you have setup. Your app will be notified when a voice command has been triggered even if the SDL app has been backgrounded.

> NOTE
>
> The head unit manufacturer will determine how these voice commands are triggered and some head units will not support voice commands.

You have the ability to create voice command shortcuts to your Main Menu cells which we highly recommended that you implement. Global voice commands should be created for functions that you wish to make available as voice commands that are **not** available as menu cells. We recommend creating

global voice commands for common actions such as the actions performed by your Soft Buttons.

Creating Voice Commands

To create voice commands, you simply create and set `VoiceCommand` objects to the `voiceCommands` array on the screen manager.

```
VoiceCommand voiceCommand = new VoiceCommand(Collections.
singletonList("Command One"), new VoiceCommandSelectionListener()
{
    @Override
    public void onVoiceCommandSelected() {
        // <#Handle the VoiceCommand's Selection#>
    }
});

sdlManager.getScreenManager().setVoiceCommands(Collections.
singletonList(voiceCommand));
```

Using RPCs

If you wish to do this without the aid of the screen manager, you can create `AddCommand` objects without the `menuParams` parameter to create global voice commands.

Getting Microphone Audio

Capturing in-car audio allows developers to interact with users by requesting raw audio data provided to them from the car's microphones. In order to gather

the raw audio from the vehicle, you must leverage the `PerformAudioPassThru` RPC.

SDL does not support automatic speech cancellation detection, so if this feature is desired, it is up to the developer to implement. The user may press an "OK" or "Cancel" button, the dialog may timeout, or you may close the dialog with `EndAudioPassThru`.

> NOTE
>
> SDL does not support an open microphone. However, SDL is working on wake-word support in the future. You may implement a voice command and start an audio pass thru session when that voice command occurs.

Starting Audio Capture

To initiate audio capture, first construct a `PerformAudioPassThru` request. You must use a sampling rate, bit rate, and audio type supported by the head unit. To get the head unit's supported audio capture capabilities, check the `sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.AUDIO_PASSTHROUGH)` after successfully connecting to the head unit.

AUDIO PASS THRU CAPABILITY	PARAMETER NAME	DESCRIPTION
Sampling Rate	samplingRate	The sampling rate
Bits Per Sample	bitsPerSample	The sample depth in bits
Audio Type	audioType	The audio type

```
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);
```

Gathering Audio Data

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, the developer must observe the `OnAudioPassThru` notification.

NOTE

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

```
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
    }
});
```

FORMAT OF AUDIO DATA

The format of audio data is described as follows:
- It does not include a header (such as a RIFF header) at the beginning.
- The audio sample is in linear PCM format.
- The audio data includes only one channel (i.e. monaural).

- For bit rates of 8 bits, the audio samples are unsigned. For bit rates of 16 bits, the audio samples are signed and are in little endian.

Ending Audio Capture

`PerformAudioPassThru` is a request that works in a different way than other RPCs. For most RPCs, a request is followed by an immediate response, with whether that RPC was successful or not. This RPC, however, will only send out the response when the audio pass thru has ended.

Audio capture can be ended in 4 ways:

1. Audio pass thru has timed out.
    - If the audio pass thru has proceeded longer than the requested timeout duration, Core will end this request with a `resultCode` of `SUCCESS`. You should handle the audio pass thru though it was successful.
2. Audio pass thru was closed due to user pressing "Cancel".
    - If the audio pass thru was displayed, and the user pressed the "Cancel" button, you will receive a `resultCode` of `ABORTED`. You should ignore the audio pass thru.
3. Audio pass thru was closed due to user pressing "Done".
    - If the audio pass thru was displayed and the user pressed the "Done" button, you will receive a `resultCode` of `SUCCESS`. You should handle the audio pass thru as though it was successful.
4. Audio pass thru was ended due to the developer ending the request.
    - If the audio pass thru was displayed, but you have established on your own that you no longer need to capture audio data, you can send an `EndAudioPassThru` RPC. You will receive a `resultCode` of `SUCCESS`, and should handle the audio pass thru as though it was successful.

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

Handling the Response

To process the response received from an ended audio capture, 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.");
        }
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
```

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 subscribing to several hard buttons. 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 listener that is specific to them, the `OnMultipleRequestListener`. This listener will provide more information than the normal `OnRPCResponseListener`.

Sending Concurrent Requests

When you send multiple RPCs concurrently, it will not wait for the response of the previous RPC before sending the next one. Therefore, there is no guarantee that responses will be returned in order, and you will not be able to use information sent in a previous RPC for a later RPC.

```
SubscribeButton subscribeButtonLeft = new SubscribeButton(
ButtonName.SEEKLEFT);
SubscribeButton subscribeButtonRight = new SubscribeButton(
ButtonName.SEEKRIGHT);
sdlManager.sendRPCs(Arrays.asList(subscribeButtonLeft,
subscribeButtonLeft), new OnMultipleRequestListener() {
    @Override
    public void onUpdate(int remainingRequests) {

    }

    @Override
    public void onFinished() {

    }

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

    }

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

    }
});
```

Sending Sequential Requests

Requests sent sequentially are sent in a set order. The next request is only sent when a response has been received for the previously sent request.

The code example below shows how to create a perform interaction choice set. When creating a perform interaction choice set, the `PerformInteraction` RPC can only be sent after the `CreateInteractionChoiceSet` RPC has been registered by Core, which is why the requests must be sent sequentially.

```
int choiceId = 111, choiceSetId = 222;
Choice choice = new Choice(choiceId, "Choice title");
CreateInteractionChoiceSet createInteractionChoiceSet = new
CreateInteractionChoiceSet(choiceSetId, Collections.singletonList(
choice));
PerformInteraction performInteraction = new PerformInteraction("Initial
Text", InteractionMode.MANUAL_ONLY, Collections.singletonList(
choiceSetId));
sdlManager.sendSequentialRPCs(Arrays.asList(
createInteractionChoiceSet, performInteraction), new
OnMultipleRequestListener() {
    @Override
    public void onUpdate(int i) {

    }

    @Override
    public void onFinished() {

    }

    @Override
    public void onError(int i, Result result, String s) {

    }

    @Override
    public void onResponse(int i, RPCResponse rpcResponse) {

    }
});
```

Retrieving Vehicle Data

You can use the `GetVehicleData` and `SubscribeVehicleData` RPC requests to get vehicle data. Each vehicle manufacturer decides which data it will expose and to whom they will expose it. Please check the response from Core to find out which data you will have permission to access. Additionally, be aware that

the user may have the ability to disable vehicle data access through the settings menu of their head unit. It may be possible to access vehicle data when the `hmiLevel` is `NONE` (i.e. the user has not opened your SDL app) but you will have to request this permission from the vehicle manufacturer.

> NOTE
>
> You will only have access to vehicle data that is allowed to your `appName` and `appId` combination. Permissions will be granted by each OEM separately. See Understanding Permissions for more details.

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Acceleration Pedal Position	accPedalPosition	Accelerator pedal position (percentage depressed)
Airbag Status	airbagStatus	Status of each of the airbags in the vehicle: yes, no, no event, not supported, fault
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.
Cloud App Vehicle Id	cloudAppVehicleID	The id for the vehicle when connecting to cloud applications
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

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
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
E-Call Infomation	eCallInfo	Information about the status of an emergency call
Electronic Parking Brake Status	electronicParkingBrakeStatus	The status of the electronic parking brake. Available states: closed, transition, open, drive active, 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
Engine Oil Life	engineOilLife	The estimated percentage (0% - 100%) of remaining oil life of the engine
Engine Torque	engineTorque	Torque value for engine (in Nm) on non-diesel variants
External Temperature	externalTemperature	The external temperature in degrees celsius
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

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
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
Instant Fuel Consumption	instantFuelConsumption	The instantaneous fuel consumption in microlitres
My Key	myKey	Information about whether or not the emergency 911 override has been activated
Odometer	odometer	Odometer reading in km
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
Speed	speed	Speed in KPH
Steering Wheel Angle	steeringWheelAngle	Current angle of the steering wheel (in degrees)

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
Turn Signal	turnSignal	The status of the turn signal. Available states: off, left, right, both
RPM	rpm	The number of revolutions per minute of the engine
VIN	vin	The Vehicle Identification Number
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

One-Time Vehicle Data Retrieval

To get vehicle data a single time, use the `GetVehicleData` RPC.

```
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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(vdRequest);
```

Subscribing to Vehicle Data

Subscribing to vehicle data allows you to get notifications whenever new data is available. You should not rely upon getting this data in a consistent manner. New vehicle data is available roughly every second, but this is totally dependent on which head unit you are connected to.

First, you should add a notification listener for the `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());
        }
    }
});
```

Second, send the `SubscribeVehicleData` 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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(subscribeRequest);
```

Third, the `onNotified` method will be called when there is an update to the subscribed vehicle data.

Unsubscribing from Vehicle Data

We suggest that you only subscribe to vehicle data as needed. To stop listening to specific vehicle data use the `UnsubscribeVehicleData` RPC.

```
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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(unsubscribeRequest);
```

Remote Control Vehicle Features

The remote control framework allows apps to control certain modules, such as climate, radio, seat, lights, etc., within a vehicle.

> NOTE
>
> Not all head units support this feature. If using this feature in your app you will most likely need to request permission from the vehicle manufacturer.

Why Use Remote Control?

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.

Remote Control Modules

Currently, the remote control feature supports these modules:

REMOTE CONTROL 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 Button Presses

The remote control framework also allows mobile applications to send simulated button press events for the following common buttons in the vehicle.

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

For remote control to work, the head unit must support SDL Core v.4.4 or newer. Also your app's `appHMIType` must be set to `REMOTE_CONTROL`.

Checking Permissions

Prior to using any remote control RPCs, you must check that the head unit has the remote control capability. As you will encounter head units that do *not* support it, this check is important. To check for this capability, use the following call:

If you do have permission to use the remote control feature, the capability object will have a list of `ButtonCapabilities` that can be obtained via the `buttonCapabilities` property.

```
// 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

Once you know you have permission to use the remote control feature, you can retrieve the data. The following code is an example of how to get data from the radio module. The example also subscribes to updates to radio data, which will be discussed later on in this guide.

```
GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData(ModuleType.RADIO);
interiorVehicleData.setSubscribe(true);
interiorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        GetInteriorVehicleDataResponse getResponse = (
GetInteriorVehicleDataResponse) response;
        // This can now be used to retrieve data
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(interiorVehicleData);
```

Setting Data

Of course, the ability to set these modules is the point of the remote control framework. Setting data is similar to getting it. Below is an example of setting climate control data. It is likely that you will not need to set all the data as in the code example. If there are settings you don't wish to modify you can skip setting them.

```
Temperature temp = new Temperature(TemperatureUnit.FAHRENHEIT,
74.1f);

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(ModuleType.CLIMATE);
moduleData.setClimateControlData(climateControlData);

SetInteriorVehicleData setInteriorVehicleData = new
SetInteriorVehicleData(moduleData);
sdlManager.sendRPC(setInteriorVehicleData);
```

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(ModuleType.RADIO,
ButtonName.EJECT, 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 request for data, simply set `subscribe` to `true`. To unsubscribe, send the request again with `subscribe` set to `false`. 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`.

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

```
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(ModuleType.RADIO);
interiorVehicleData.setSubscribe(true);
sdlManager.sendRPC(interiorVehicleData);
```

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 `false` 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 setup listeners for 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 response with a relevant failure result code.

LISTENING FOR REQUESTS

First, you will need to setup a listener for a `GetAppServiceDataRequest`.

```
// 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).

```
// 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 asynchronously, 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);
```

Calling a Phone Number

The `DialNumber` RPC allows you make a phone call via the user's phone. Regardless of platform (Android or iOS), you must be sure that a device is

connected via Bluetooth (even if using USB) for this RPC to work. If the phone is not connected via Bluetooth, you will receive a result of `REJECTED` from Core.

> 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.

Checking if Dial Number is Available

`DialNumber` is a newer RPC, so there is a possibility that not all head units will support it. To find out if the RPC is supported by the head unit, check the system capability manager's `hmiCapabilities.isPhoneCallAvailable()` property after the manager has been started successfully.

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

Sending a DialNumber Request

> NOTE
>
> `DialNumber` strips all characters 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.
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(dialNumber);
```

DialNumber Result

`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.

Setting the Navigation Destination

The `SendLocation` RPC gives you the ability to send a GPS location to the active navigation app on the head unit.

When using the `SendLocation` RPC, you will not have access to any information about how the user interacted with this location, only if the request was successfully sent. The request will be handled by Core from that point on using the active navigation system.

Checking If Your App Has Permission to Use Send Location

The `SendLocation` RPC is restricted by most vehicle manufacturers. As a result, the head unit you are connecting to will reject the request if you do not have the correct permissions. Please check the Understanding Permissions section for more information on how to check permissions for an RPC.

Checking if Head Unit Supports Send Location

Since there is a possibility that some head units will not support the send location feature, you should check head unit support before attempting to send the request. You should also update your app's UI based on whether or not you can use `SendLocation`.

If using library v.4.4 and connecting to SDL Core v.4.5 or newer, you can use the `SystemCapabilityManager` to check the navigation capability returned by Core as shown in the code sample below.

If connecting to older versions of Core (or using older versions of the library), you will have to check the `sdlManager.getRegisterAppInterfaceResponse().getHmiCapabilities().isNavigationAvailable();` after the SDL app has started successfully to see if an embedded navigation system is available. If it is, then you can assume that `SendLocation` will work.

```
sdlManager.getSystemCapabilityManager().getCapability(
SystemCapabilityType.NAVIGATION, new OnSystemCapabilityListener()
{
    @Override
    public void onCapabilityRetrieved(Object capability) {
        NavigationCapability navCapability = (NavigationCapability)
capability;
        Boolean isNavigationSupported = navCapability.
getSendLocationEnabled();
    }

    @Override
    public void onError(String info) {
        HMICapabilities hmiCapabilities = (HMICapabilities) sdlManager.
getSystemCapabilityManager().getCapability(SystemCapabilityType.HMI
);
        Boolean isNavigationSupported = hmiCapabilities.
isNavigationAvailable();
    }
});
```

Using Send Location

To use the `SendLocation` request, you must at minimum include the longitude and latitude of the location.

```
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.
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(sendLocation);
```

Checking the Result of Send Location

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

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

Uploading Files

In almost all cases, you will not need to handle uploading images because the screen manager API will do that for you. There are some situations, such as VR help-lists and turn-by-turn directions, that are not currently covered by the screen manager so you will have manually upload the image yourself in those cases. For more information about uploading images, see the Uploading Images guide.

Uploading an MP3 Using the File Manager

The `FileManager` uploads files and keeps track of all the uploaded files names during a session. To send data with the file manager you need to create either a `SdlFile` or `SdlArtwork` object. Both `SdlFile`s and `SdlArtwork`s can be created with a `Uri`, `byte[]`, or `resourceId`.

```
byte[] mp3Data = Get the file data;
SdlFile audioFile = new SdlFile("File Name", FileType.AUDIO_MP3,
mp3Data, true);
sdlManager.sendRPC(audioFile);
```

Batching File Uploads

If you want to upload a group of files, you can use the `FileManager` batch upload methods. Once all of the uploads have completed you will be notified if any of the uploads failed.

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

    }
});
```

File Persistance

`SdlFile` and its subclass `SdlArtwork` support uploading persistant files, i.e. files that are not deleted when the car turns off. Persistance should be used for files that will be used every time the user opens the app. If the file is only displayed for short time the file should not be persistent because it will take up unnecessary space on the head unit. You can check the persistence via:

```
Boolean isPersistent = file.isPersistent();
```

NOTE

Be aware that persistance will not work if space on the head unit is limited. The `FileManager` will always handle uploading images if they are non-existent.

Checking the Amount of File Storage Left

To find the amount of file storage left for your app 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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(listFiles);
```

Checking if a File Has Already Been Uploaded

You can check out if an image has already been uploaded to the head unit via the `FileManager`'s `remoteFileNames` property.

```
Boolean fileIsOnHeadUnit = sdlManager.getFileManager().
getRemoteFileNames().contains("Name Uploaded As")
```

Deleting Stored Files

Use the file manager’s delete request to delete a file associated with a file name.

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

    }
});
```

Batch Deleting Files

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

    }
});
```

Uploading Images

> NOTE
>
> If you are looking to upload images for use in template graphics, soft buttons, or the menu, you can use the ScreenManager. Other situations, such as VR help lists and turn by turn directions, are not currently covered by the `ScreenManager`.

You should be aware of these four things when using images in your SDL app:

1. You may be connected to a head unit that does not have the ability to display images.
2. You must upload images from your mobile device to the head unit before using them in a template.
3. Persistent images are stored on a head unit between sessions. Ephemeral images are destroyed when a session ends (i.e. when the user turns off their vehicle).
4. Images can not be uploaded when the app's `hmiLevel` is `NONE`. For more information about permissions, please review Understanding Permissions.

To learn how to use images once they are uploaded, please see Text, Images, and Buttons.

Checking if Graphics are Supported

Before uploading images to a head unit you should first check if the head unit supports graphics. If not, you should avoid uploading unnecessary image data.

To check 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;
    boolean graphicsSupported = dispCapability.getGraphicSupported
();
  }

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

Uploading an Image Using SDL FileManager

The `FileManager` uploads files and keeps track of all the uploaded files names during a session. To send data with the `FileManager`, you need to create either a `SdlFile` or `SdlArtwork` object. Both `SdlFile`s and `SdlArtwork`s can be created with a `Uri`, `byte[]`, or `resourceId`.

```
SdlArtwork artwork = new SdlArtwork("image_name", FileType.
GRAPHIC_PNG, <image byte[]>, false);
sdlManager.getFileManager().uploadFile(artwork, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        if (success){
            <#Image Upload Successful#>
        }
    }
});
```

Batch File Uploads, Persistence, etc.

Similar to other files, artworks can be persistent, batched, overwrite, etc. See Uploading Files for more information.

Creating an OEM Cloud App Store

A new feature of SDL Core v5.1 and Sdl Java Suite v.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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
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.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
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 Retrieving Vehicle Data section.

Introduction

Mobile navigation allows map partners to easily display their maps as well as present visual and audio turn-by-turn prompts on the head unit.

Navigation apps have 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.

Configuring a Navigation App

The basic connection setup is similar for all apps. Please follow the Integration Basics guide for more information.

In order to create a navigation app an `appHMIType` of `NAVIGATION` must be set in the `SdlManager`'s `Builder`.

The second difference is the ability 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 and audio streaming. This method requires an array of security libraries, which will extend the `SdlSecurityBase` class. These security libraries are provided by the OEMs themselves, and will only work for that OEM. There is no 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);

// Add security managers if Core requires secure video & audio
streaming
List<? extends SdlSecurityBase> securityManagers = new ArrayList();
builder.setSdlSecurity(Arrays.asList(OEMSecurityManager1.class,
OEMSecurityManager2.class));

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

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

MUST

When compiling your app for production, make sure to include all possible OEM security managers that you wish to support.

Keyboard Input

To present a keyboard (such as for searching for navigation destinations), you should use the `ScreenManager`'s keyboard presentation feature. For more information, see the Popup Menus and Keyboards guide.

Video Streaming

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.stream);

        Button button = findViewById(R.id.button);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Log.i(TAG, "Button Clicked");
            }
        });
    }
}
```

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)` .

```
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.stream);

        String videoUri = "android.resource://" + context.getPackageName
() + "/" + R.raw.sdl;
        VideoView videoView = findViewById(R.id.videoView);
        videoView.setVideoURI(Uri.parse(videoUri));
        videoView.start();
    }
}
```

```
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

A navigation app can stream raw audio to the head unit. This audio data is played immediately. If audio is already playing, the current audio source will be attenuated and your audio will play. Raw audio must 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

To stream audio from a SDL app, use the `AudioStreamingManager` class. A reference to this class is available from the `SdlManager`s `audioStreamManager` property.

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().pushResource()`. 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) {
                            sdlManager.getAudioStreamManager().pushResource(
R.raw.exampleMp3, 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");
            }
        }
    });
}
```

USING A BUFFER

You can also send `ByteBuffer`s to the `AudioStreamManager` to be played. To use it, replace the `pushResource` call in the example above to the `pushBuffer` call shown below:

```
sdlManager.getAudioStreamManager().pushBuffer(byteBuffer, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        if (success) {
            Log.i(TAG, "Buffer played successfully!");
        } else {
            Log.i(TAG, "Buffer failed to play!");
        }
    }
});
```

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) {
        // do something once the stream is stopped
    }
});
```

Supporting Haptic Input

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 views on the screen are touchable and focusing / highlighting on those areas when the user moves the trackpad or click wheel. When the user selects within a view, the center of that area will be "touched".

> NOTE
>
> Currently, there are no RPCs for knowing which view is highlighted, so your UI will have to remain static (i.e. you should not create a scrolling menu in your SDL app).

You will also need to implement touch input support in order to receive touches on the views.

Automatic Focusable Rects

SDL has support for automatically detecting focusable views 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 views 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`.

Manual Focusable 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.

Displaying Turn Directions

While your app is navigating the user, you will also want to send turn by turn directions. This is useful for if your app is in the background or if the user is in the middle of a phone call, and gives the system additional information about the next maneuver the user must make.
When your navigation app is guiding the user to a specific destination, you can provide the user with visual and audio turn-by-turn prompts. These prompts will be presented even when your SDL app is backgrounded or a phone call is ongoing.
While your app is navigating the user, you will also want to send turn by turn

directions. This is useful if your app is in the background or if the user is in the middle of a phone call, and gives the system additional information about the next maneuver the user must make.

To create a turn-by-turn direction that provides both a visual and audio cues, a combination of the `ShowConstantTBT` and `AlertManeuver` RPCs must should be sent to the head unit.

> NOTE
>
> If the connected device has received a phone call in the vehicle, the `AlertManeuver` is the only way for your app to inform the user of the next turn.

Visual Turn Directions

The visual data is sent using the `ShowConstantTBT` RPC. The main properties that should be set are `navigationText1`, `navigationText2`, and `turnIcon`. A best practice for navigation apps is to use the `navigationText1` as the direction to give (i.e. turn right) and `navigationText2` to provide the distance to that direction (i.e. 3 mi.).

Audio Turn Directions

The audio data is sent using the `AlertManeuver` RPC. When sent, the head unit will speak the text you provide (e.g. In 3 miles turn right).

Sending Both Audio and Visual Turn Directions

```
Image turnIcon =  <#Create Image#>

ShowConstantTbt turnByTurn = new ShowConstantTbt();
turnByTurn.setNavigationText1("Turn Right");
turnByTurn.setNavigationText2("3 mi");
turnByTurn.setTurnIcon(turnIcon);
turnByTurn.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (!response.getSuccess()){
            Log.e(TAG, "onResponse: Error sending TBT");
            return;
        }

        AlertManeuver alertManeuver = new AlertManeuver();
        alertManeuver.setTtsChunks(TTSChunkFactory.
createSimpleTTSChunks("In 3 miles turn right"));
        alertManeuver.setOnRPCResponseListener(new
OnRPCResponseListener() {
            @Override
            public void onResponse(int correlationId, RPCResponse response
) {
                if (!response.getSuccess()){
                    Log.e(TAG, "onResponse: Error sending AlertManeuver");
                }
            }

            @Override
            public void onError(int correlationId, Result resultCode, String
info){
                Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
            }
        });
        sdlManager.sendRPC(alertManeuver);
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(turnByTurn);
```

Remember when sending a `Image`, that the image must first be uploaded to the head unit with the `FileManager`.

Clearing the Turn Directions

To clear a navigation direction from the screen, send a `ShowConstantTbt` with the `maneuverComplete` property set to true. This will also clear the accompanying `AlertManeuver`.

```
ShowConstantTbt turnByTurn = new ShowConstantTbt();
turnByTurn.setManeuverComplete(true);
turnByTurn.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (!response.getSuccess()){
            Log.e(TAG, "onResponse: Error sending TBT");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(turnByTurn);
```

Configuring SDL Logging

Sdl Java Suite has a built-in logging framework that is designed to make debugging easier. It provides many of the features common to other 3rd party logging frameworks for java and can be used by your own app as well. We

recommend that your app's integration with SDL provide logging using this framework rather than any other 3rd party framework your app may be using or `System.out.print`. This will consolidate all SDL related logs into a common format and to a common destination.

Enabling the DebugTool

To make sure that log messages are displayed, you should enable the SDL Debug Tool:

```
DebugTool.enableDebugTool();
```

If you don't want the messages to be logged, you can disable the Debug Tool anytime:

```
DebugTool.disableDebugTool();
```

> NOTE
>
> If you use SDL Debug Tool to log messages without enabling the DebugTool nothing wrong will happen. It will simply not display the log messages. This gives the develop control on whether the logs should be displayed or not.

Logging messages

The SDL debug tool can be used to log messages with different log levels. The log level defines how serious the log message is. This table summarizes when to use each log level:

LOG LEVEL	WHEN TO USE
Info	Use this to post useful information to the log
Warning	Use this when you suspect something shady is going on
Error	Use this when bad stuff happens

To log an info message:

```
DebugTool.logInfo("info message goes here");
```

To log a warning message:

```
DebugTool.logWarning("warning message goes here");
```

To log an error message:

```
DebugTool.logError("error message goes here");
```

If you want to log error message with exception, you can add the exception as a second parameter to the `logError` method:

```
DebugTool.logError("error message goes here", new SdlException("Sdl connection failed", SdlExceptionCause.SDL_CONNECTION_FAILED));
```

Filtering logs

The log level defines which logs will be logged to the target outputs. For example, if you set the log level filter in `Logcat` to `Warning`, all error, and warning logs will be logged, but info level logs will not be logged.

LOGLEVEL	VISIBLE LOGS
Error	error
Warning	error and warning
Info	error, warning, and info

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);
        }
    }
```

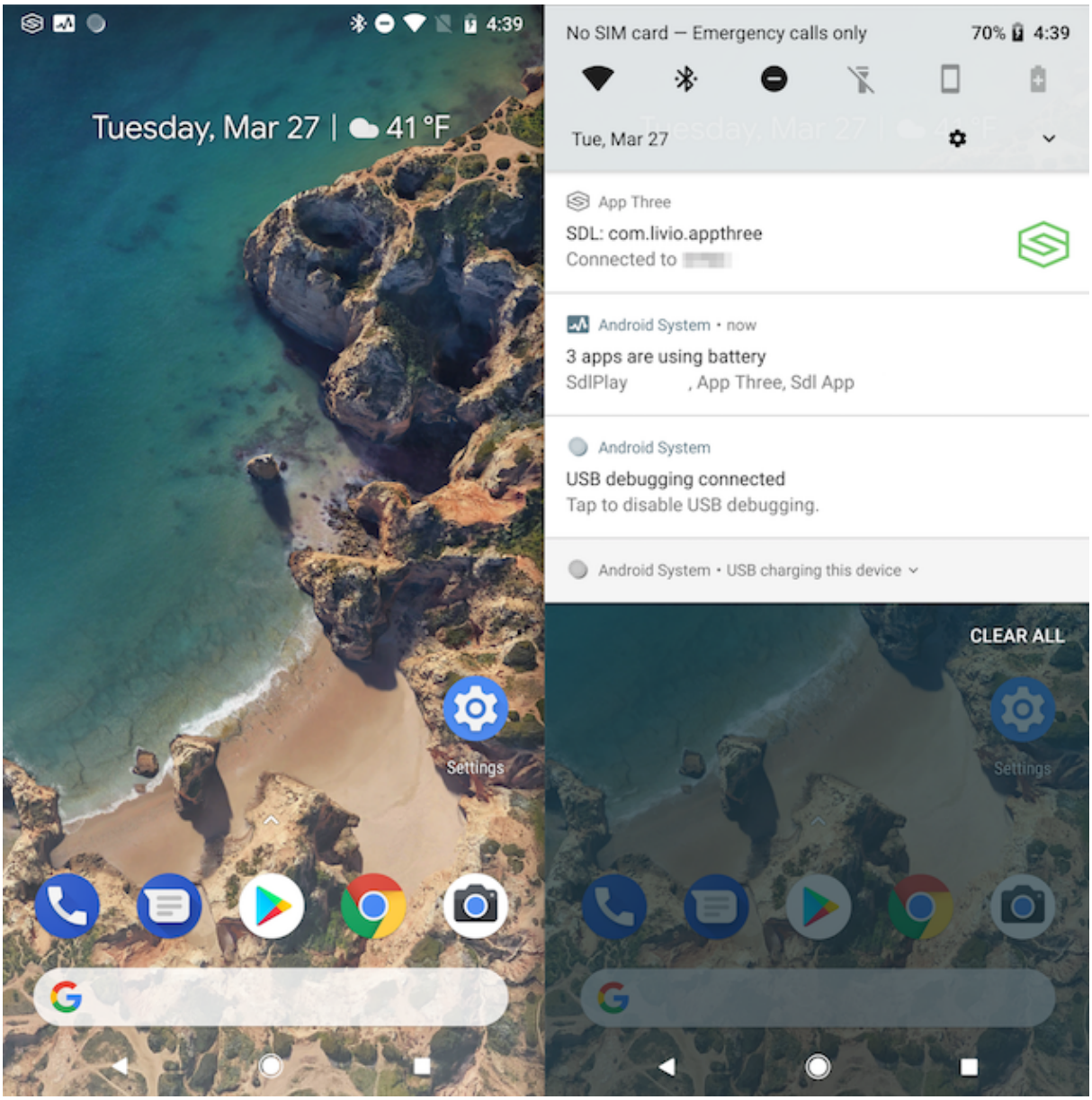
THE NOT SO CORRECT WAY

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

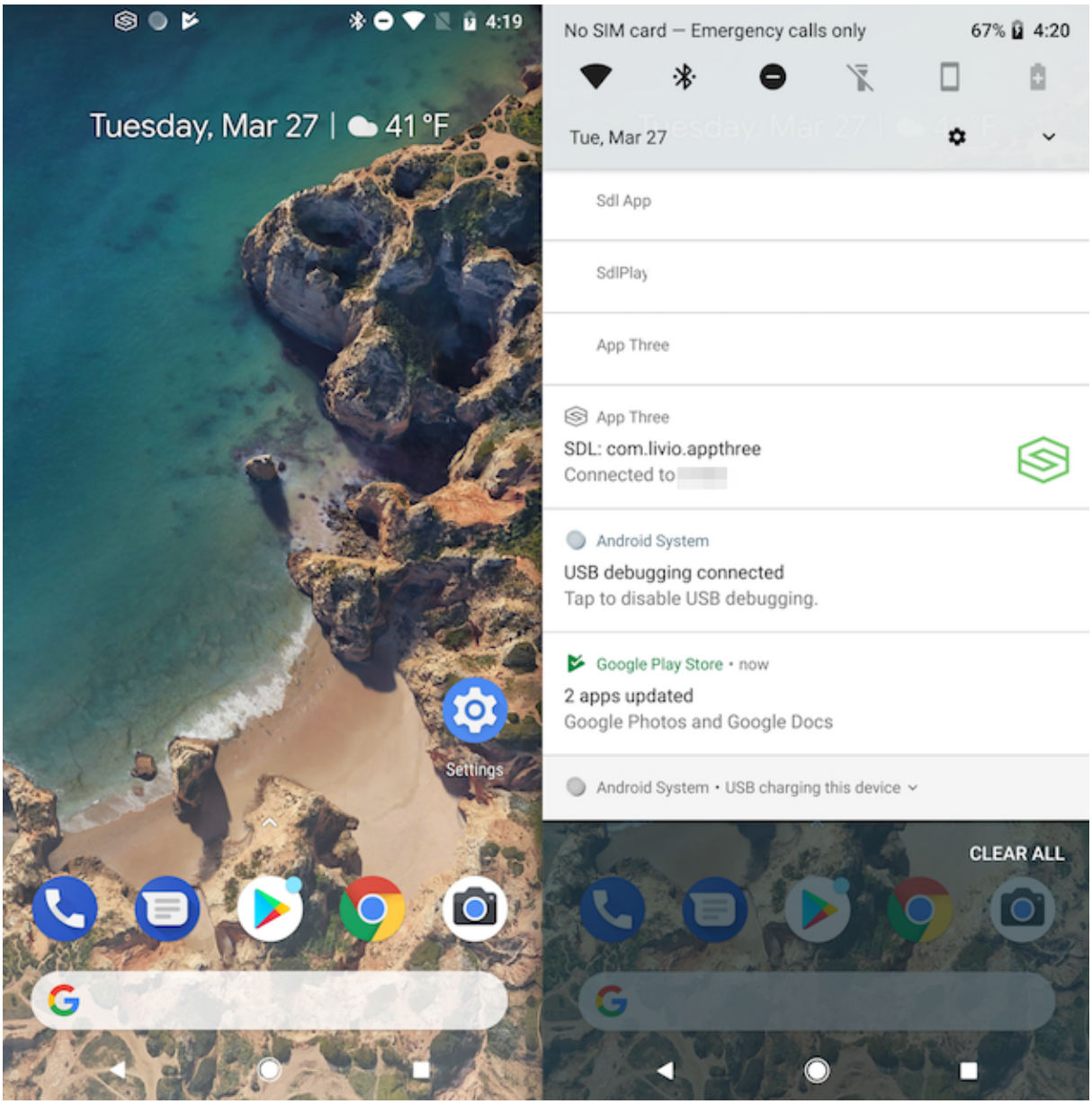
It is possible to create a somewhat invisible notification. This will appear to just be blank space in the notification tray. With adding minimal content to the notification when the user pulls down the tray it will have a very small footprint on the screen. However, this is completely disingenuous to 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);
```

Upgrading to 4.9

Overview

This guide is to help developers get setup with the SDL Java library version 4.9. It is assumed that the developer is already updated to at least version 4.7 or 4.8 of the library.

The full release notes are published here.

The main differences between the previous release and this are mainly additive, including 3 new managers which we will describe briefly. Additionally, we have fixed an issue where symlinks were not working on Windows machines by creating a gradle task that builds them for you. Additionally, we have added the ability to pass a buffer to the AudioStreamManager to play raw data.

Voice Command Manager

The voice command manager is accessed via the `ScreenManager`. It allows for an easy way to create global voice commands for your application. These are not supposed to be a replacement for menu voice commands, but rather an easy way to trigger main events in your application, similar to something you might use a `SoftButton` for. These commands, once sent, will be available on the system as voice commands for the duration of the session.

An example is as follows:

```
List<String> list1 = Collections.singletonList("Command One");
List<String> list2 = Collections.singletonList("Command two");

VoiceCommand voiceCommand1 = new VoiceCommand(list1, new
VoiceCommandSelectionListener() {
    @Override
    public void onVoiceCommandSelected() {
        Log.i(TAG, "Voice Command 1 triggered");
    }
});

VoiceCommand voiceCommand2 = new VoiceCommand(list2, new
VoiceCommandSelectionListener() {
    @Override
    public void onVoiceCommandSelected() {
        Log.i(TAG, "Voice Command 2 triggered");
    }
});

sdlManager.getScreenManager().setVoiceCommands(Arrays.asList(
voiceCommand1,voiceCommand2));
```

Menu Manager

Menus have now become simpler with the `MenuManager`, which is accessed via the `ScreenManager`. The cells, called `MenuCell`'s contain 2 constructors. One is for a cell itself, and the other is a cell that contains a sub-menu. Note that currently SmartDeviceLink (SDL) only supports sub-menus to the depth of 1.

`MenuCell`s contain a `MenuSelectionListener` which informs you that the cell has been triggered, so that you might perform an action based on the cell selected. Note that you can add images and voice commands to menu cells.

NOTE

When submitting a list of Menu cells, or adding a list of sub cells to a menu cell, the order in which the cells will appear from top to bottom will be the order in which they are in the list.

Example use:

```
// SUB MENU CELLS FOR MAIN MENU CELL 2

// Sub cells are just normal cells
MenuCell subCell1 = new MenuCell("SubCell 1",null, null, new
MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Sub cell 1 triggered. Source: "+ trigger.toString());
    }
});

MenuCell subCell2 = new MenuCell("SubCell 2",null, null, new
MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Sub cell 2 triggered. Source: "+ trigger.toString());
    }
});

// THE MAIN MENU CELLS

// normal cell
MenuCell mainCell1 = new MenuCell("Test Cell 1 (speak)", null, null,
new MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Test cell 1 triggered. Source: "+ trigger.toString());
    }
});

// sub menu parent cell
MenuCell mainCell2 = new MenuCell("Test Cell 3 (sub menu)", null,
Arrays.asList(subCell1,subCell2));

// Send the entire menu off to be created
sdlManager.getScreenManager().setMenu(Arrays.asList(mainCell1,
mainCell2));
```

Choice Set Manager

Previously it required a lot of code to use `PerformInteraction`s with SDL. To alleviate some of this pain, we have introduced the Choice Set Manager which

is accessible via the `ScreenManager`. Because the Choice Set Manager covers so many items, we will do a brief overview here. You may continue to the Popup Menus and Keyboards section for more detailed information.

There are 2 main use cases for using this manager, one is to display a choice set, and the other is to display a keyboard.

Choice Set

Displaying a choice set is achieved by creating some `ChoiceCell`s. If you know what your choices will be, we recommend using the `preloadChoices` method. This will ensure your `ChoiceSet` is ready to be displayed when you want to display it, and your user is not kept waiting. You can preload cells as follows:

```
// create some choice cells
ChoiceCell cell1 = new ChoiceCell("Item 1");
ChoiceCell cell2 = new ChoiceCell("Item 2");
ChoiceCell cell3 = new ChoiceCell("Item 3");

// create the array of choice cells
choiceCellList = Arrays.asList(cell1,cell2,cell3);

// pre-load the cells on the head unit
sdlManager.getScreenManager().preloadChoices(choiceCellList, null);
```

> NOTE
>
> You will want to reference this array of cells when presenting your choice set later (even if you add more cells). This is why we are setting this list to a variable for now.

Once you are ready to present the Choice Set, you can do so by:

```
ChoiceSet choiceSet = new ChoiceSet("Choose an Item from the list",
choiceCellList, new ChoiceSetSelectionListener() {
    @Override
    public void onChoiceSelected(ChoiceCell choiceCell, TriggerSource
triggerSource, int rowIndex) {
        // do something with the selection
    }

    @Override
    public void onError(String error) {
        Log.e(TAG, "There was an error showing the perform interaction: "
+ error);
    }
});
sdlManager.getScreenManager().presentChoiceSet(choiceSet,
InteractionMode.MANUAL_ONLY);
```

Displaying A Keyboard

There is now also an easy way to display a keyboard, and listen for key events. You simply need a `KeyboardListener` object.

```
KeyboardListener keyboardListener = new KeyboardListener() {
    @Override
    public void onUserDidSubmitInput(String inputText, KeyboardEvent
event) {

    }

    @Override
    public void onKeyboardDidAbortWithReason(KeyboardEvent event) {

    }

    @Override
    public void updateAutocompleteWithInput(String currentInputText,
KeyboardAutocompleteCompletionListener
keyboardAutocompleteCompletionListener) {

    }

    @Override
    public void updateCharacterSetWithInput(String currentInputText,
KeyboardCharacterSetCompletionListener
keyboardCharacterSetCompletionListener) {

    }

    @Override
    public void onKeyboardDidSendEvent(KeyboardEvent event, String
currentInputText) {

    }
};
```

You can note that two of the methods contain a `KeyboardAutocompleteCompletionListener` and a `KeyboardCharacterSetCompletionListener`. These listeners allow you to show auto completion text and to modify the available keys, respectively, on supported head units.

To actually display the keyboard, call:

```
sdlManager.getScreenManager().presentKeyboard("initialText", null,
keyboardListener);
```

The `null` parameter in this example is a `KeyboardProperties` object that you can optionally pass in to modify the keyboard for this request.

Audio Stream Buffer

We now have the option to send `ByteBuffer`s to the `AudioStreamManager` to be played.

```
sdlManager.getAudioStreamManager().pushBuffer(byteBuffer, new
CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        // do something once the buffer is played
    }
});
```

Symlinks in Windows

With the creation of the Java Suite, we had the need to share base files between the Android and the JavaSE and JavaEE projects. To allow the Android project to read these base files, we created symlinks to allow the files to be seen from within the project. However, symlinks work differently on Mac / Linux machines than they do on Windows.

To fix this, we created a gradle task to create the Windows symlinks. Simply call:

```
gradle buildWindowSymLinks
```

from Android Studio's terminal.

> NOTE
>
> You will need administrator privileges and Python installed to execute this task.

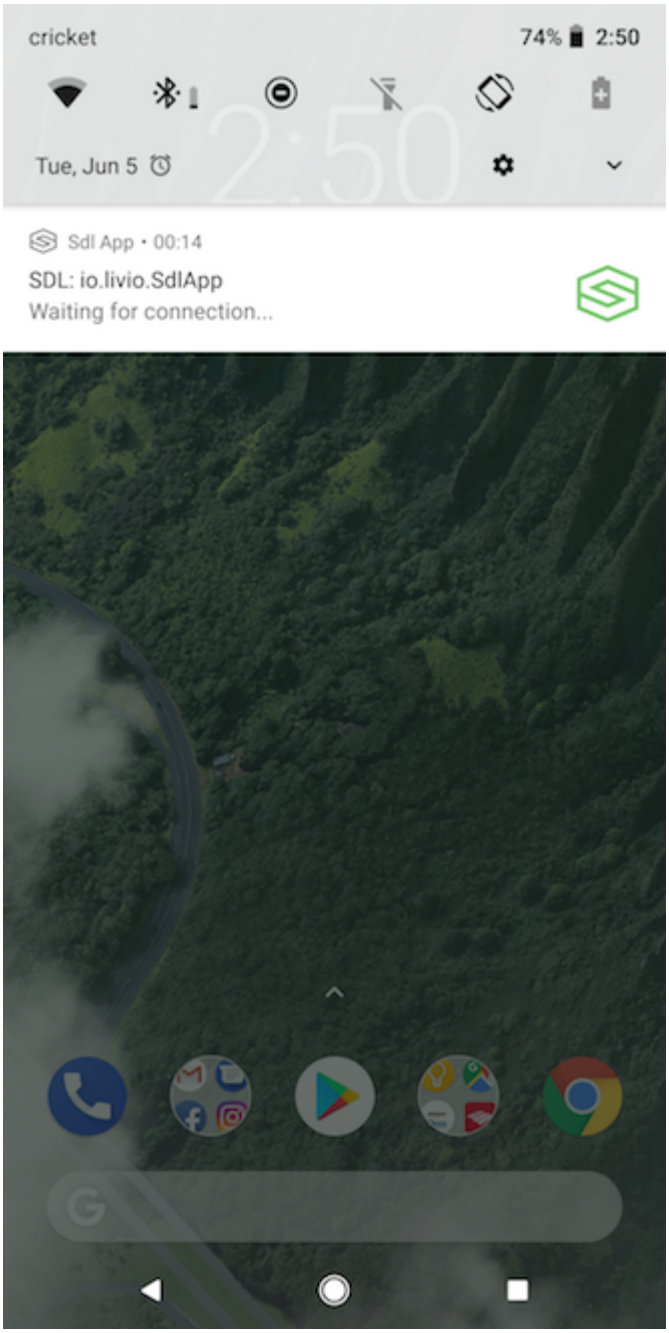
What is SDL?

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.

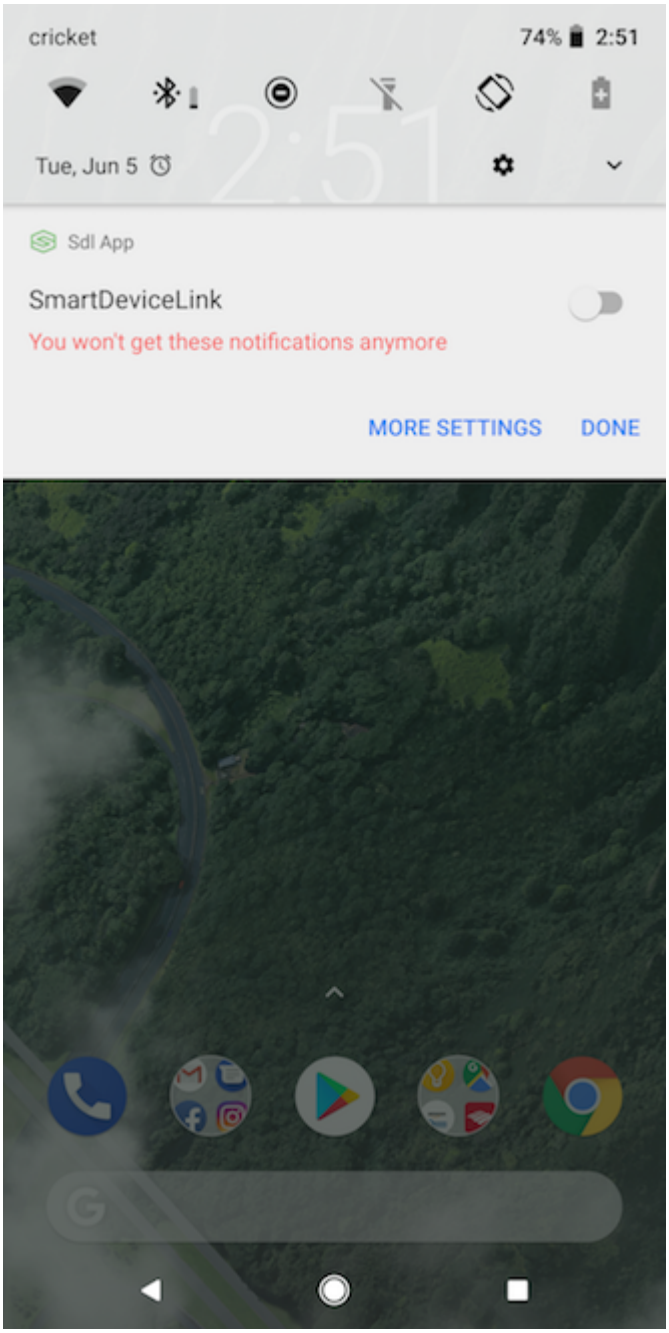
Why do you see SDL notifications?

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 (SDL) 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 SmartDeviceLink Consortium (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.

Multiple Processes

The SmartDeviceLink Android library uses multiple processes and there are some items that should be understood about why that is necessary and precautions to take while handling that situation.

Why does the router service run in its own process?

The router service is designed to live outside the normal lifecycle of the app integrating the SDL framework. The different process allows a level of security to cut off access to the hosting application's data because Android allocates a different memory space for the router service process to run. It also allows the router service to not interfere with the hosting application's runtime; this means if the router service unexpectedly stops or crashes, it will not take down the hosting app. This relationship also works in the opposite direction, which is important to maintain a good user experience when apps are connected through a router service.

Content providers and multiple processes

Android content providers have a unique lifecycle that does not work in the expected flow. Content providers are actually started before the `Application` class and following `Activities`, `Services`, etc. Some libraries use this to know when their code/module can initialize and always be ready for the entire lifecycle of the application. This is found with many Google libraries (Firebase, Jetpack, etc).

The issue is that, by default, content providers are only attached to and initialized for the main process. This means, when the main process starts the content provider will be started, but if a different process other than the main process is started the content provider will not be started. So if the app has its first start from a component that is designed to run in a different process, the content provider won't be ready by the time those components start up; this includes the `Application` instance for that process.

Why is this a problem?

The issue occurs when there is code in a developer's custom `Application` class that assumes the content provider or module using the content provider lifecycle has already been initialized, but an instance of that child `Application` class is created for a process outside of the main process.

For example:

```
public class MyApplication extends Application {


    @Override
    public void onCreate() {
        super.onCreate();

         ModuleUsingContentProviderForInit.doSomething();

    }
}
```

If an instance of this extended `Application` class is created outside the main process before the main process has started, this application will crash with a runtime exception. This can happen when components that use a different process are started directly instead of the app itself being launched by the user directly. The SDL library does this to provide a seamless connection for apps to head units without the requirement of user interaction.

Workaround

Depending on the module that uses a content provider for initialization, it could be possible to start/initialize it from the `onCreate` method of the extended `Application` class. It should be noted that the module would then need to be set up for a multiple process environment, which is not always the case.

If the module can't be initialized in this way, then the `Application` child class will need to keep a flag that prevents code from executing that would cause errors.

For SDL the solution can be as follows:

```
public class MyApplication extends Application {

    private static final String ROUTER_SERVICE_PROCESS =
"com.smartdevicelink.router";

    boolean isSdlProcessFlag = false;

    @Override
    public void onCreate() {
        super.onCreate();

        isSdlProcessFlag = isSdlProcess()
        if (isSdlProcessFlag) {
          //This application instance is running in the SDL process
            return;
        }

        ModuleUsingContentProviderForInit.doSomething();

    }

    /**
    *
    * @return if this process is the SDL router service process
    */
    private boolean isSdlProcess(){
        int myPid = android.os.Process.myPid();
        ActivityManager am = (ActivityManager)this.getSystemService(
ACTIVITY_SERVICE);
        if (am == null || am.getRunningAppProcesses() == null) {
            return false;
        }

        for (ActivityManager.RunningAppProcessInfo processInfo : am.
getRunningAppProcesses()) {
            if (processInfo != null && processInfo.pid == myPid) {
                return ROUTER_SERVICE_PROCESS.equals(processInfo.
processName);
            }
        }
        return false;
    }
}
```

NOTE

If other callback methods in your `Application` class are used, they must also check this flag to prevent unintended behavior.

The use of this flag can help prevent errors when extending the `Application` class that assume it always has the main process started first. This solution could be modified to change the flag to monitor if this is the main process or not very easily.

Custom Application classes instance for each process

While the documentation on this is a little scarce, the Android OS creates a new instance of the supplied `Application` class for each process that is started in your app. This means your custom `Application` class needs to be ready to run on different processes. The previous example is a good sample that can prevent code from executing in your custom class that is only intended to run on the main process.