SAMSUNG

Getting started with Samsung KNOX SDK development

SRIN B2B Innovation Lab

b2b-tech-support-sea@samsung.com

SAMSUNG

Table of Contents

Introduction

Samsung KNOX SDK is a collection of API (Application Programming Interface) that enables enterprises to design applications which can manage their employee's mobile devices. Applications developed with KNOX SDK API can reduce security threats and risks from lost or stolen devices that contain sensitive corporate data. [1]

Samsung KNOX SDK includes various SDKs, including Standard SDK, Premium SDK, Customization SDK, SSO SDK, VPN SDK, ISV SDK, Samsung EDU SDK for Android operating system, Standard SDK for Tizen operating system, Cloud SDK, and various KNOX based services such as KNOX Enabled App and KNOX Customization Service.

Before you can use the Samsung Knox SDK in your app, you need to obtain a Samsung Knox license (SKL). Samsung Knox uses a license manager server to identify and authenticate apps that can take control of devices. You use your SKL key in your app for authentication purposes. You SKL key is passed to every device that you manage with your app. Should your SKL key become compromised, Samsung can revoke it, and any app that uses the compromised key can't take control of a device.

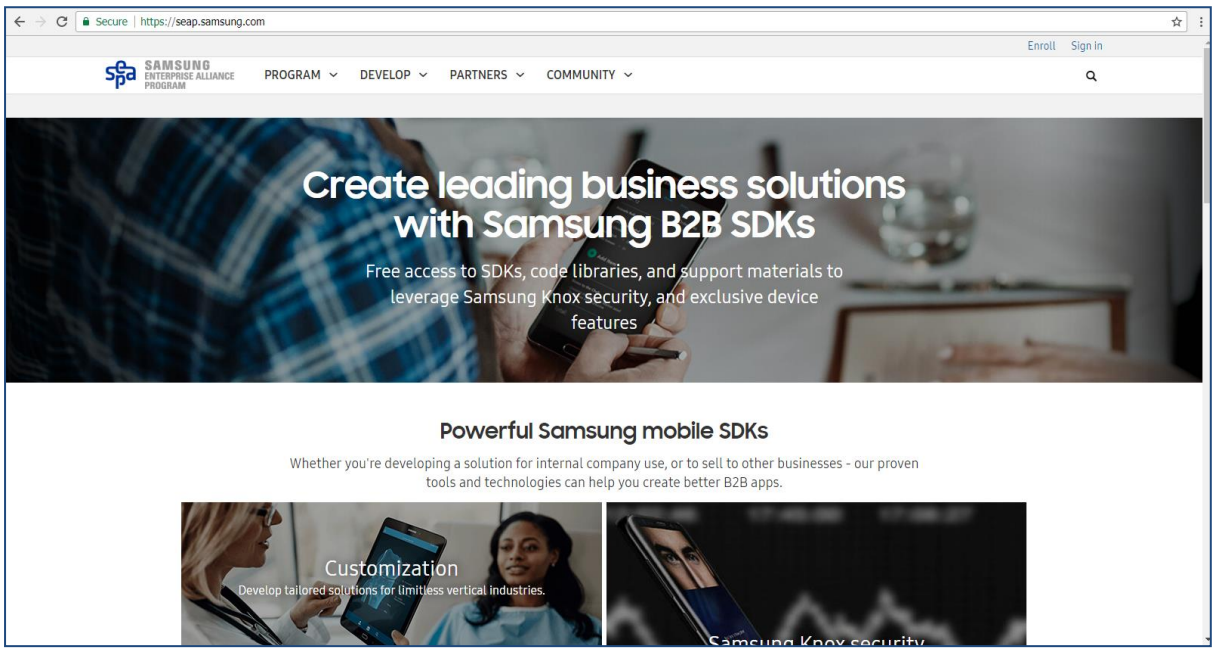
NOTE – The SKL key replaces the ELM and ISV keys that were used in the legacy Knox SDKs. If you are still using these legacy SDKs, for example, on older devices, see About license keys. Currently, for paid premium features, you still need to use Production ELM and KLM licenses. An SKL license will **not work.** This should change eventually after the SKL license is out of beta. You can generate Production ELM and KLM licenses from the SEAP Portal.

This **Introduction to Samsung KNOX Standard SDK** guide is dedicated to first time developers who are interested to develop enterprise applications using KNOX SDK.
This guide book covers handful of courses including preparing to develop for KNOX SDK, request ELM license , up to developme nt using KNOX Standard SDK for Android operating system on Samsung mobile devices.

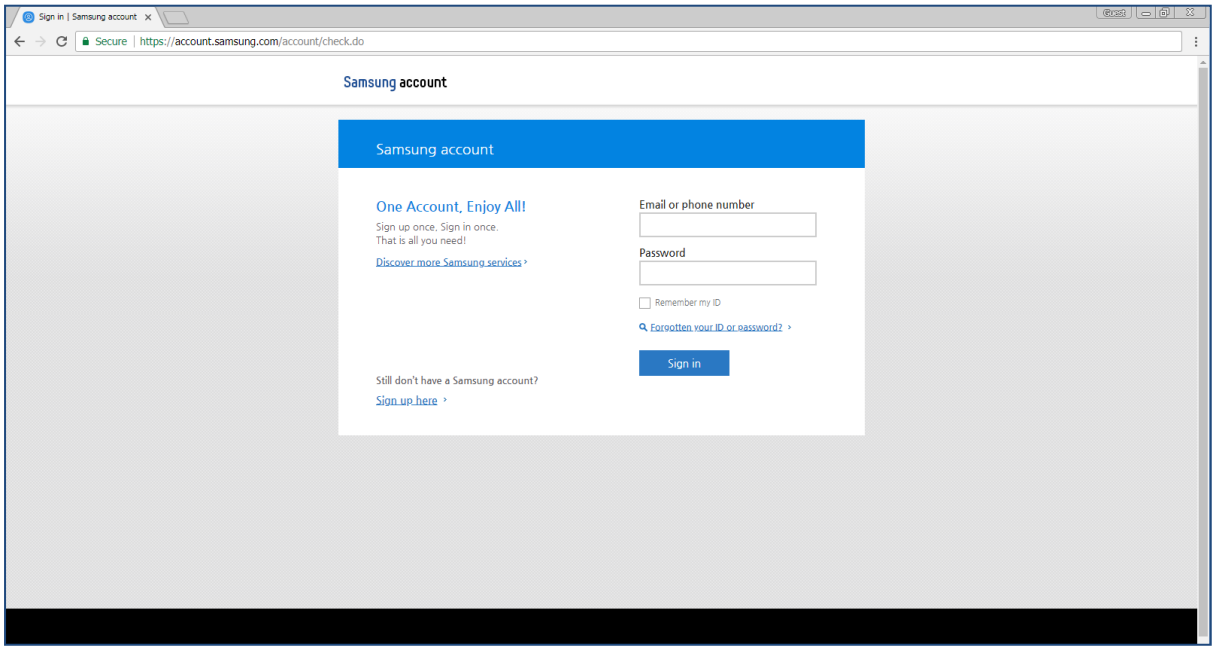
Get KNOX SDK

To obtain KNOX SDK and licenses to be used for developing applications using KNOX SDK, users need to be registered as B2B partner in the SEAP (Samsung Enterprise Alliance Program) portal.

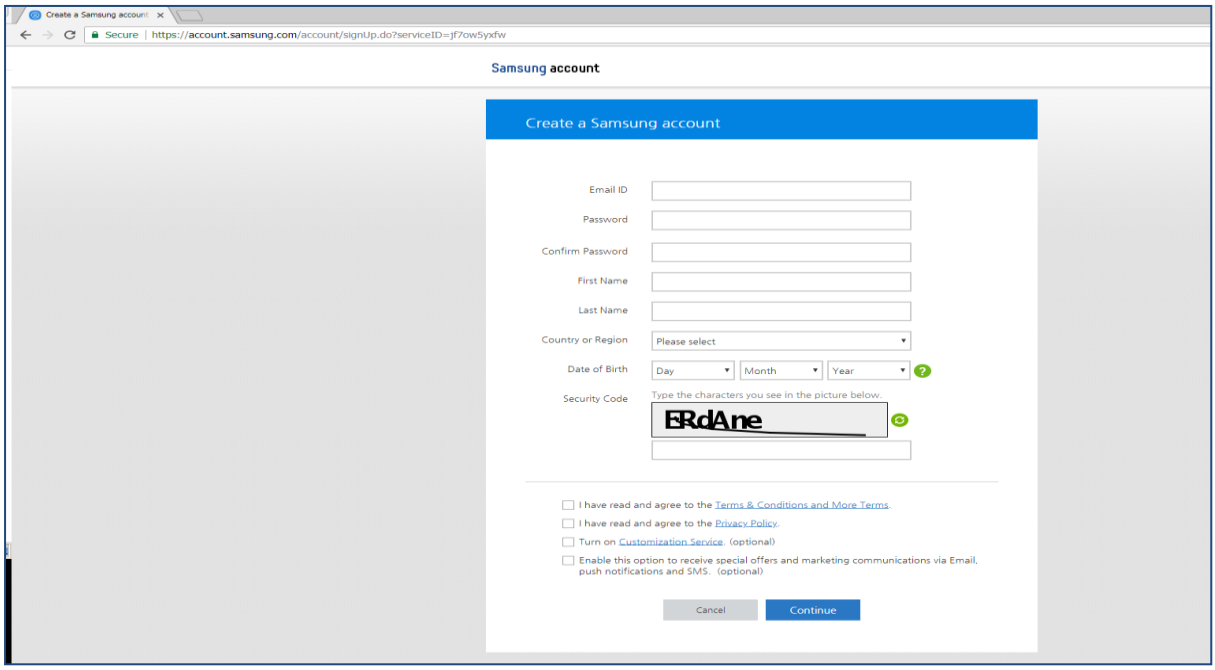
SEAP portal is accessible using this link https://seap.samsung.com/.



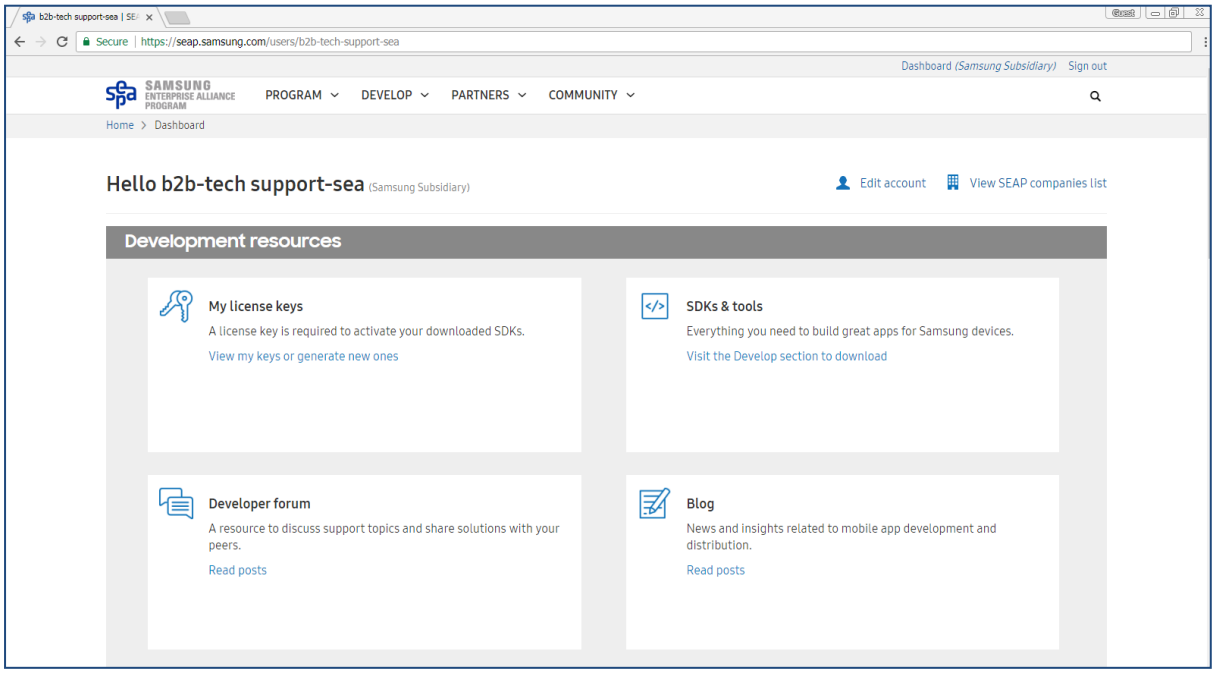
To sign in as registered SEAP partner, users can use **Sign in** menu on the SEAP portal front page. Users can use Samsung Account to log in to the SEAP portal.



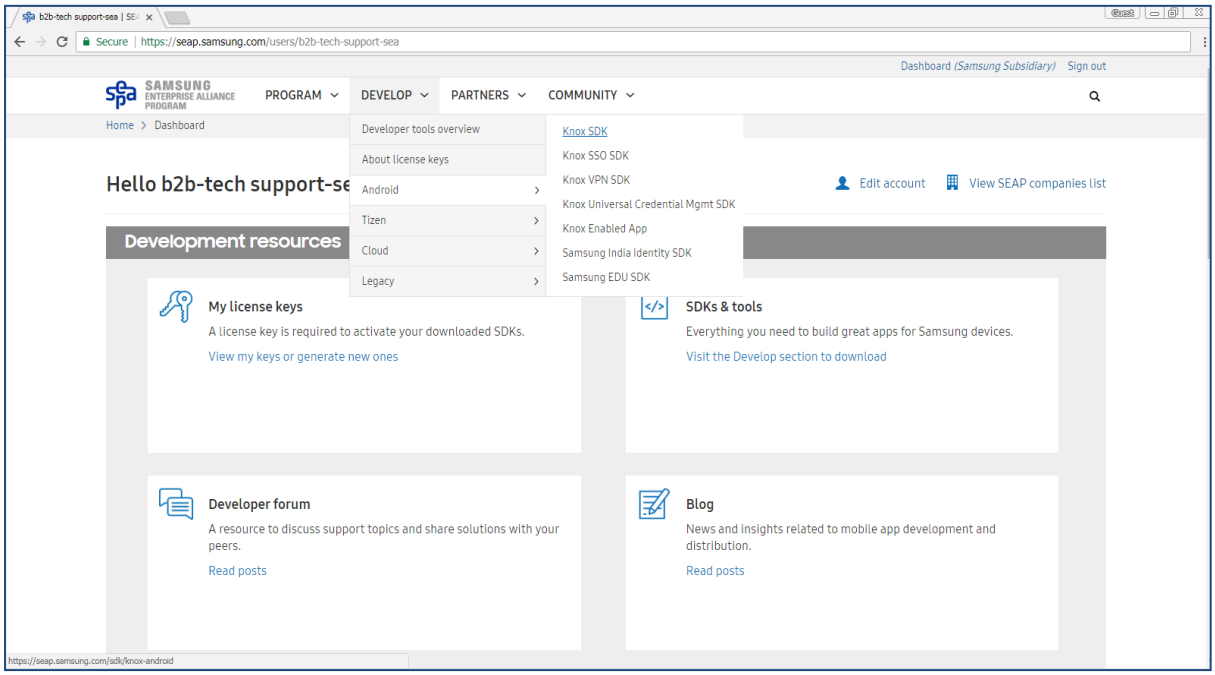
First time users using Samsung Account to log in to the SEAP portal need to enroll their accounts as B2B partner through the **SEAP Enrollment** page which is also accessible using the **Enroll** menu on the SEAP portal front page.



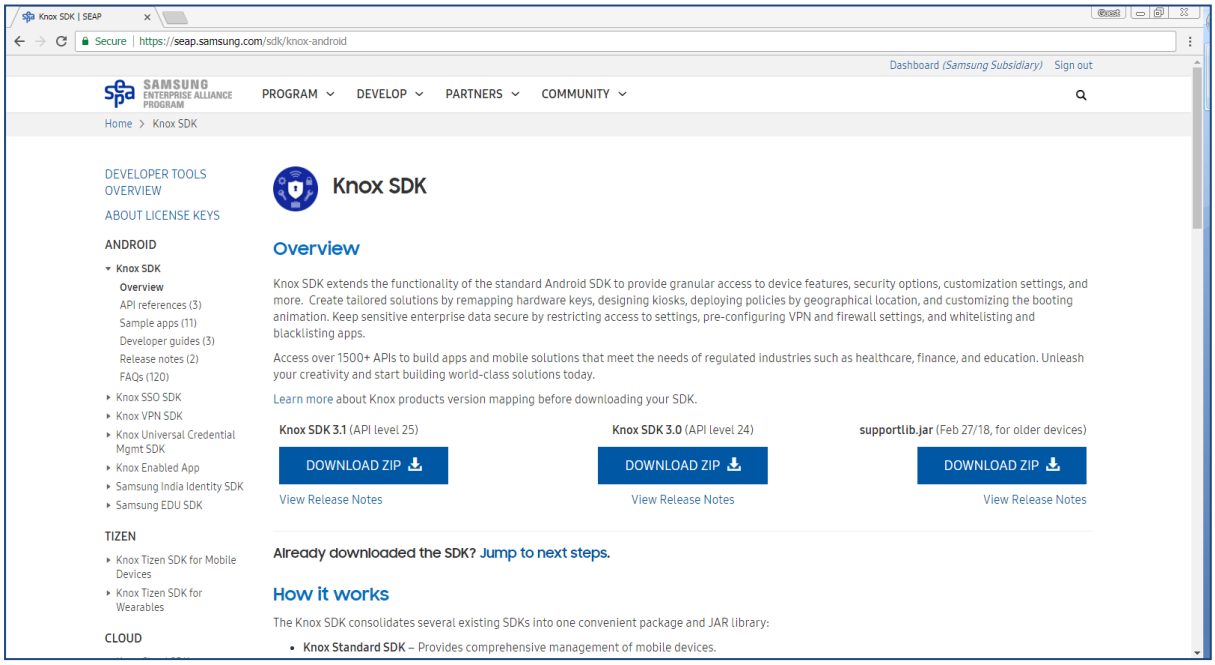
After logged in to the SEAP portal, users will be redirected to the SEAP portal main page where users can download KNOX SDK, request and view generated licenses, open developer forum, among other things.



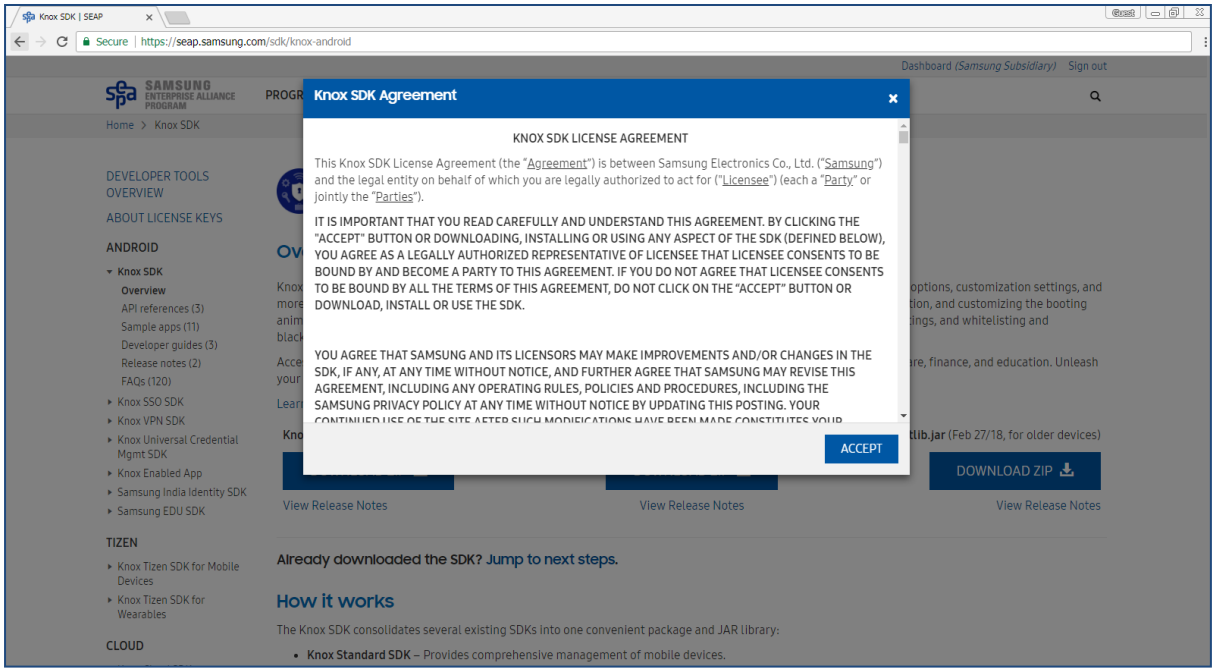
KNOX SDK can be accessed using menu that is available on the SEAP portal main page by selecting **DEVELOP → Android → KNOX SDK**.



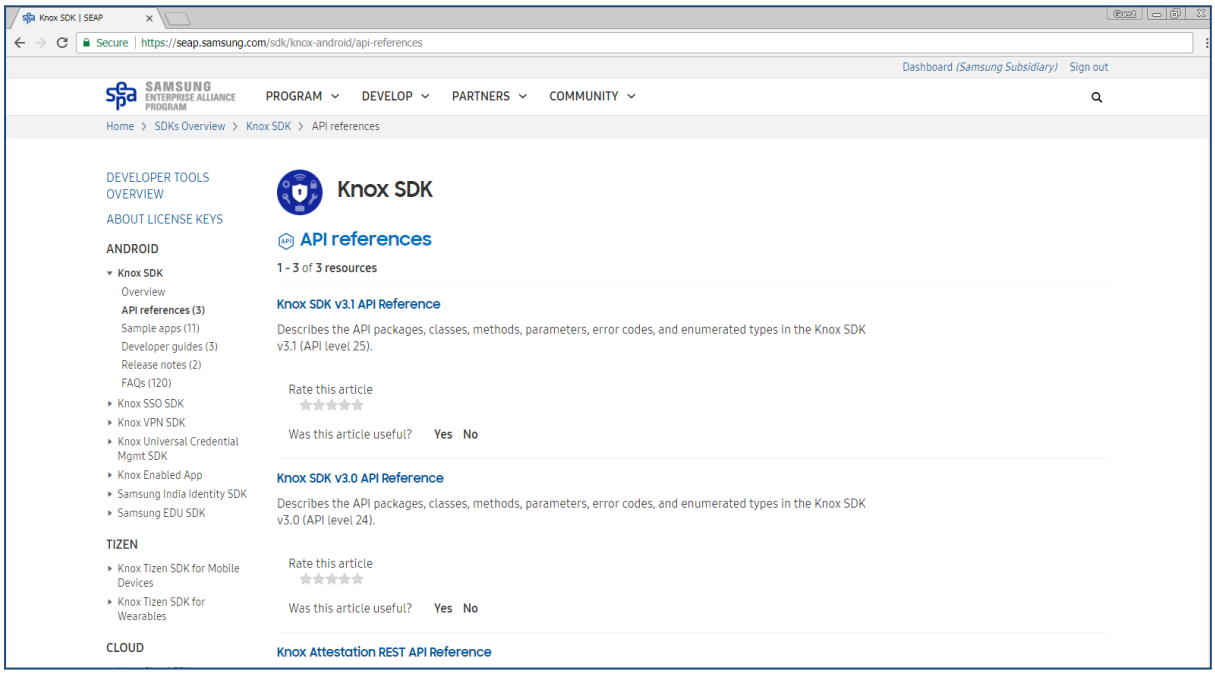
U can download KNOX SDK libraries by clicking the **DOWNLOAD SDK** button.
This step can be repeated for every KNOX SDKs to download all the KNOX SDKs available on the SEAP portal.

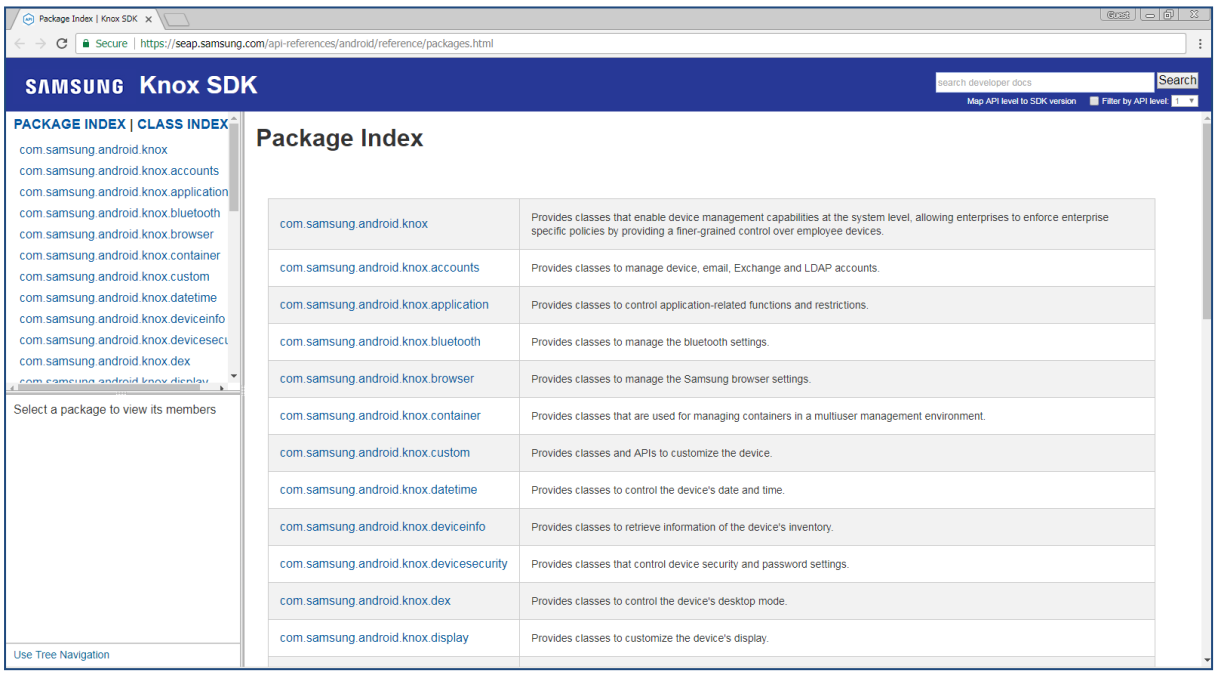


Users need to agree to the SEAP agreement to be able to download and use KNOX SDK.

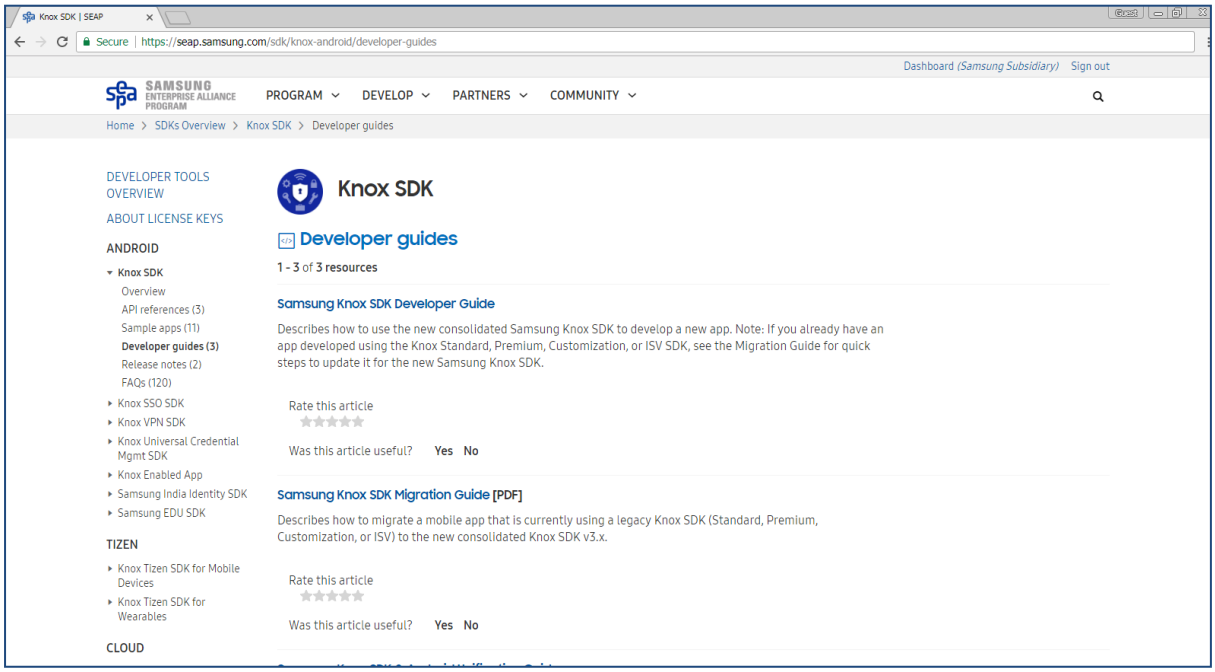


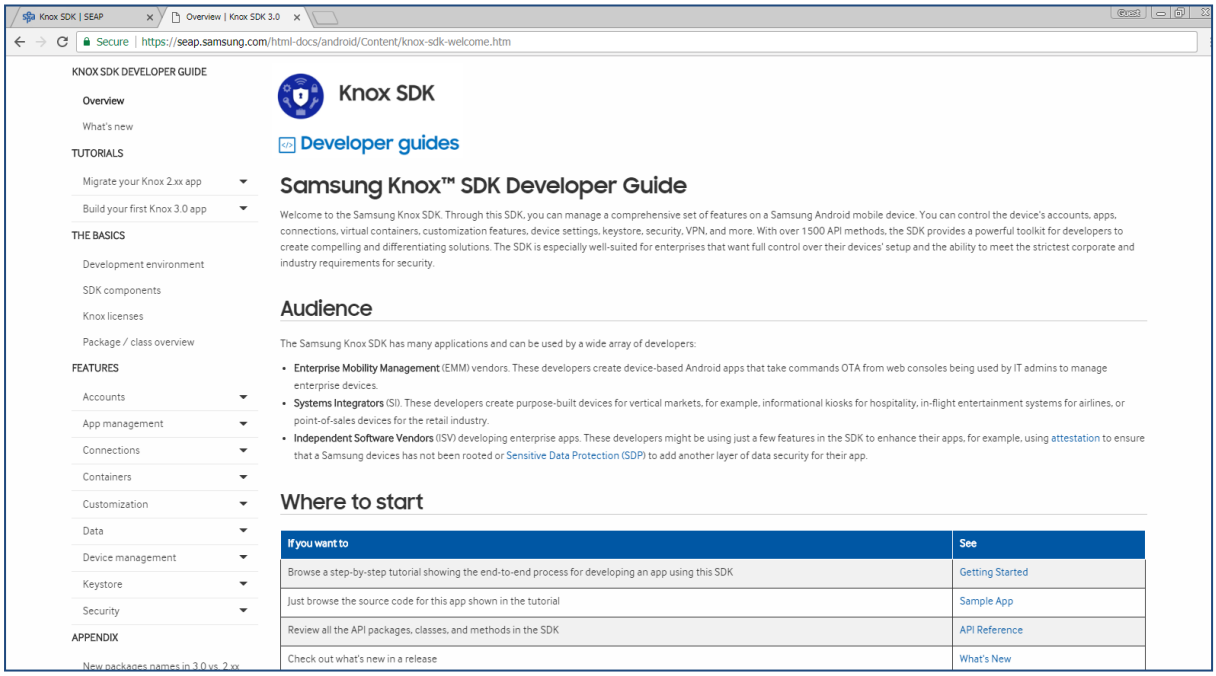
Users can also find several supporting pages to help during development on the side navigation menu.
One of which is the **API References** page where users can list all the API packages, classes, methods, parameters, error codes and enumeration types used by KNOX SDK.



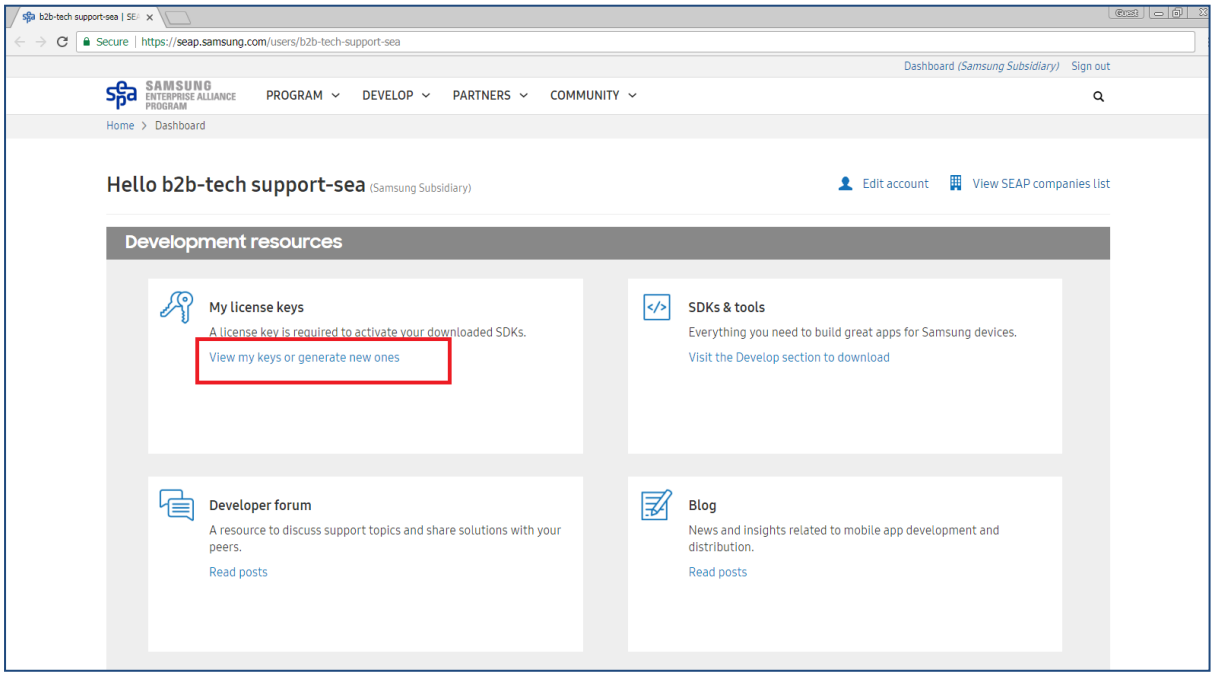


There is also **Developer Guides** page which illustrates on how to getting started with KNOX SDK development, and provides sample codes for KNOX SDK features.

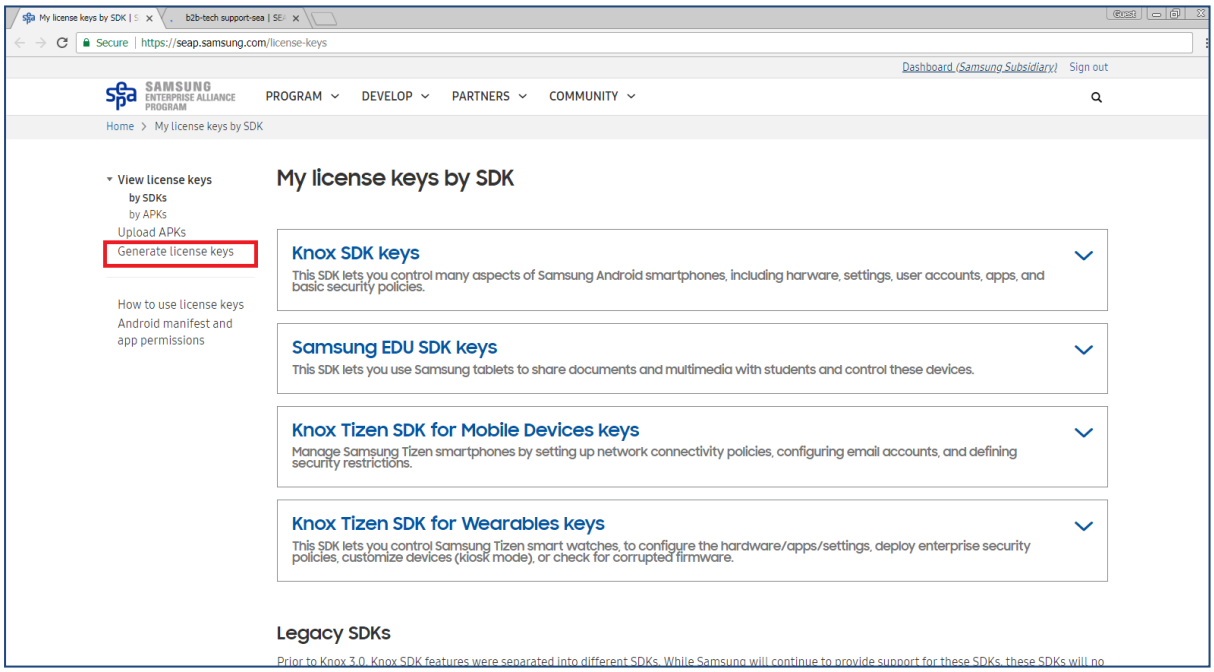


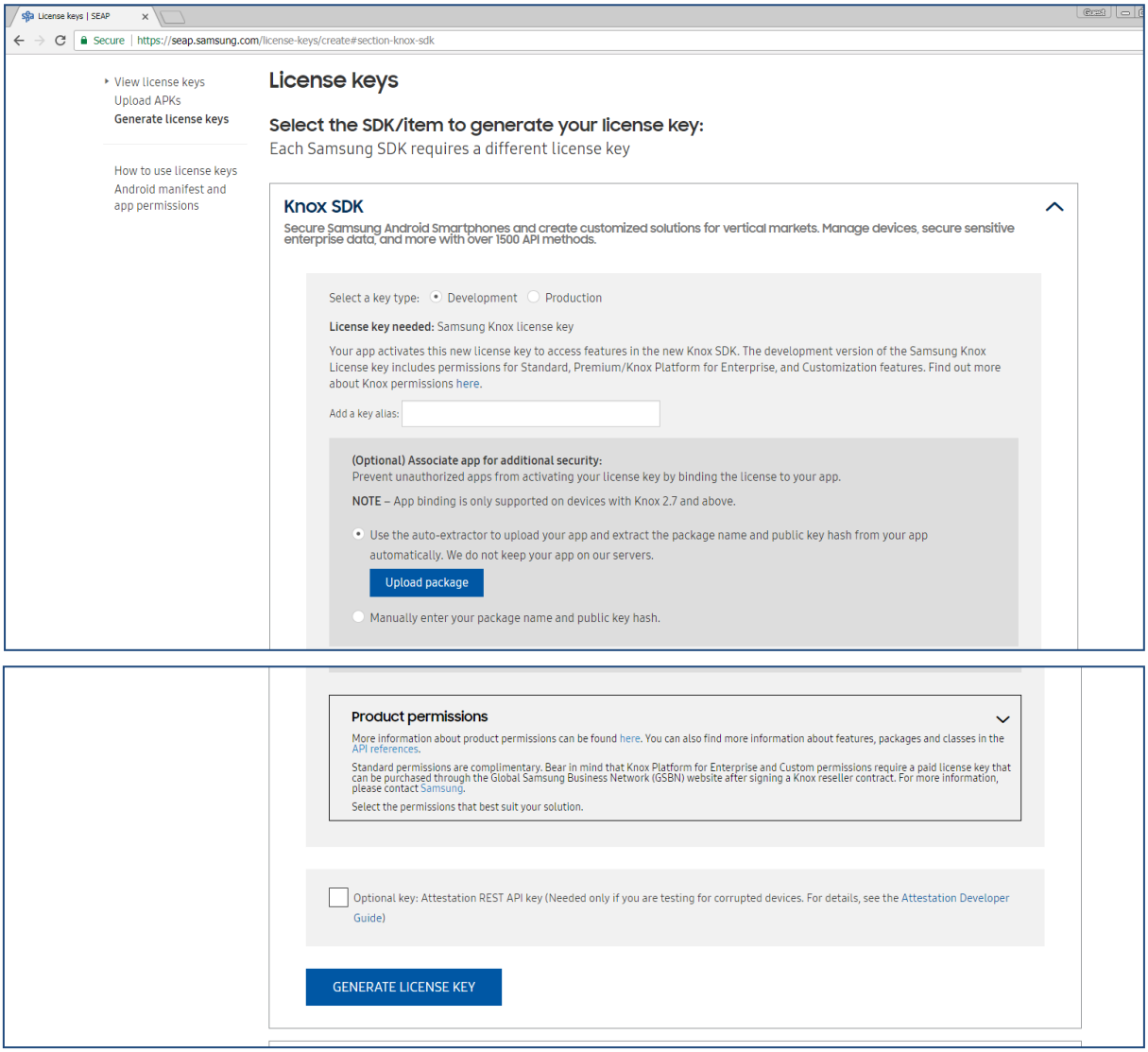


Users can obtain their KNOX licenses on the by open the **Dashboard** page, and then click **View my keys or generated new ones**.



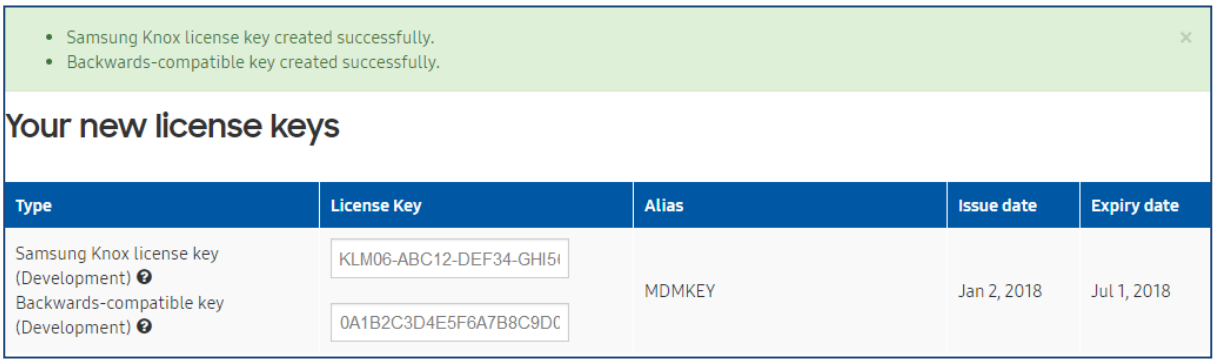
In the license page, click **Generate license key menu**.





- Select a **Development** key type, which you can use during pre-production on a limited number of devices for a limited time period.
  (Later, when you are ready to commercialize your app, select a Production key.)
- Enter a key alias, for example, MDMKEY.
- For added security, you can identify the app that will use this license; only this app will be allowed to activate this license. During development however, you can skip this step considering your app is not yet ready.
- Click **GENERATE LICENSE KEY**. The SEAP Not for Resale Agreement appears.
- After you agree to the terms, the Your New License Keys page displays your keys. Keep them secure

On the **License Keys** page, users can select **VIEW ALL MY LICENSE KEYS** button to view all license keys linked to users account.



Install Prerequisites

To start development using KNOX SDK, some prerequisite applications that need to be installed are:

- Operating System: Microsoft Windows 7/8/10 (32 or 64-bit), Mac OS X 10.8.5 or higher, or Linux 3.2 or higher with GNU C Library (glibc) 2.11
- Java Development Kit (JDK) 8
- Android Studio or Eclipse IDE

Samsung KNOX SDK is compatible with both Android Studio and Eclipse IDE as add-on, however all sample codes in this guide will be presented using Android Studio.

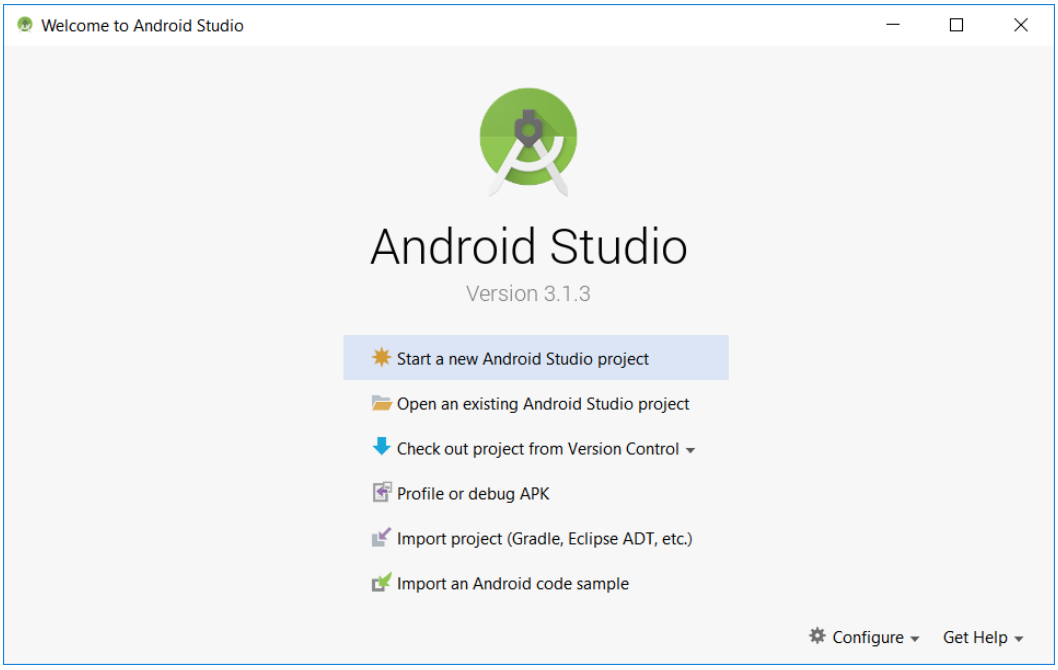
Prior to installing Android Studio, users can open the following link at Android Studio System Requirements to make sure that the system used by users has met the requirement to run Android Studio.

Next, users can start by installing JDK which can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html and continue with installing Android Studio by downloading from http://developer.android.com/sdk/index.html.

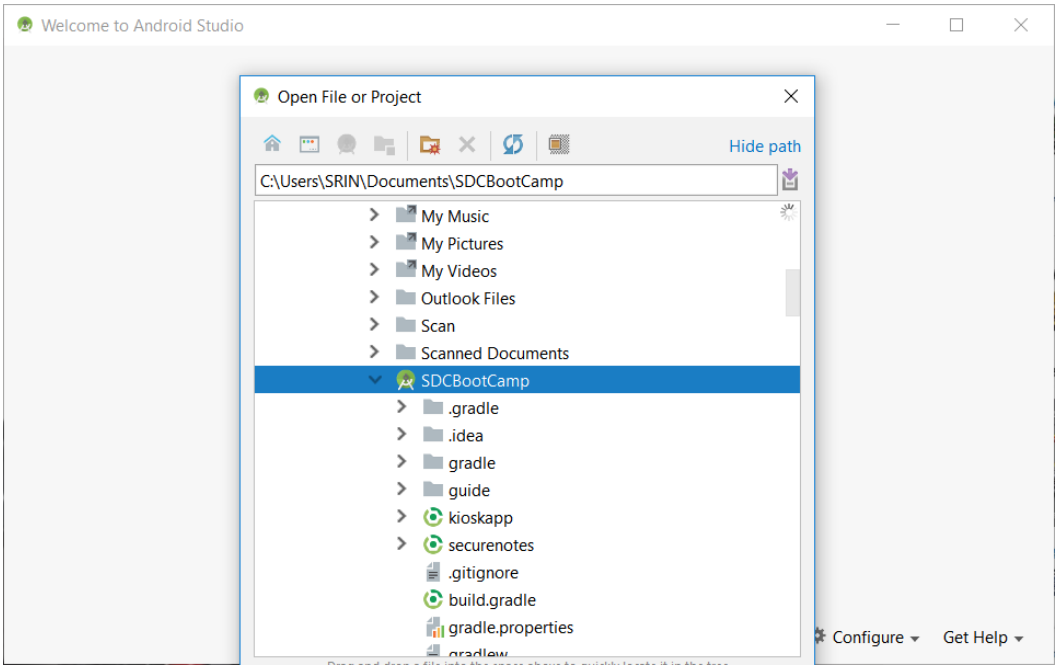
Import Project Template

After Android Studio has been successfully installed on the system, users can now start develop applications using KNOX SDK by creating a new Android Studio project or to use project template provided by this guide that is available at http://bil-id.com/knox-sdk/introduction-template.zip.

Users can import project template to Android Studio by selecting **Open an existing Android Studio project** menu on Android Studio main window.

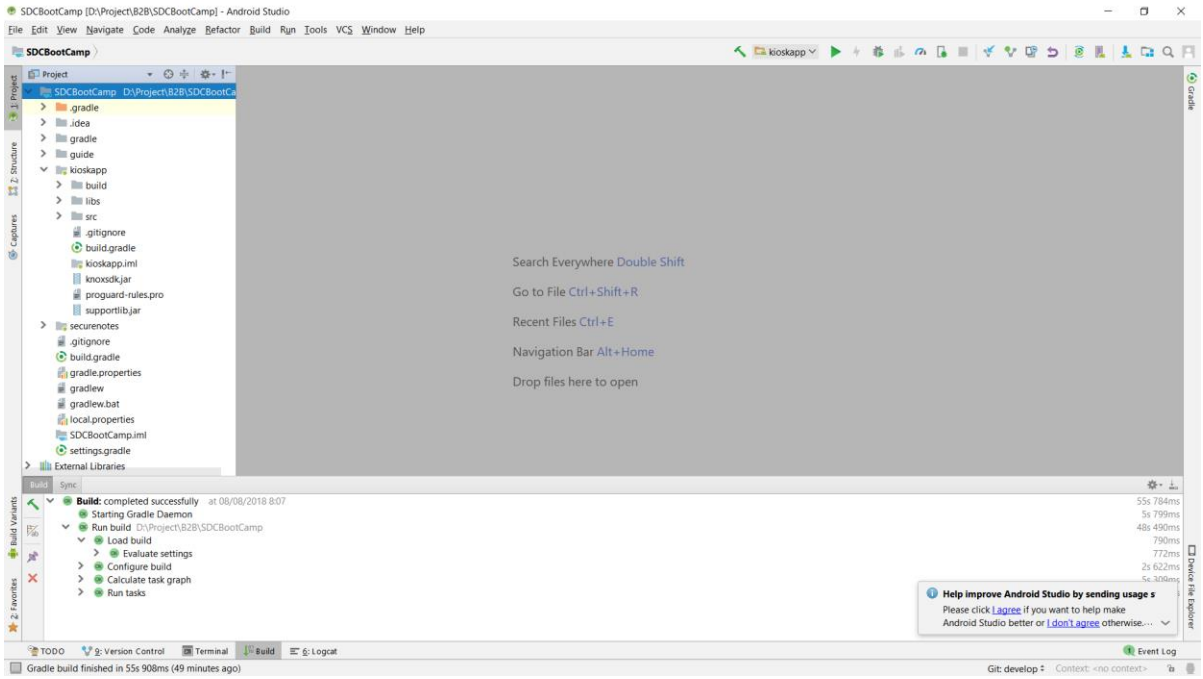

Welcome to Android Studio
Android Studio
Version 3.1.3
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Configure
Get Help

Then select the location of the project template.



Develop with KNOX SDK

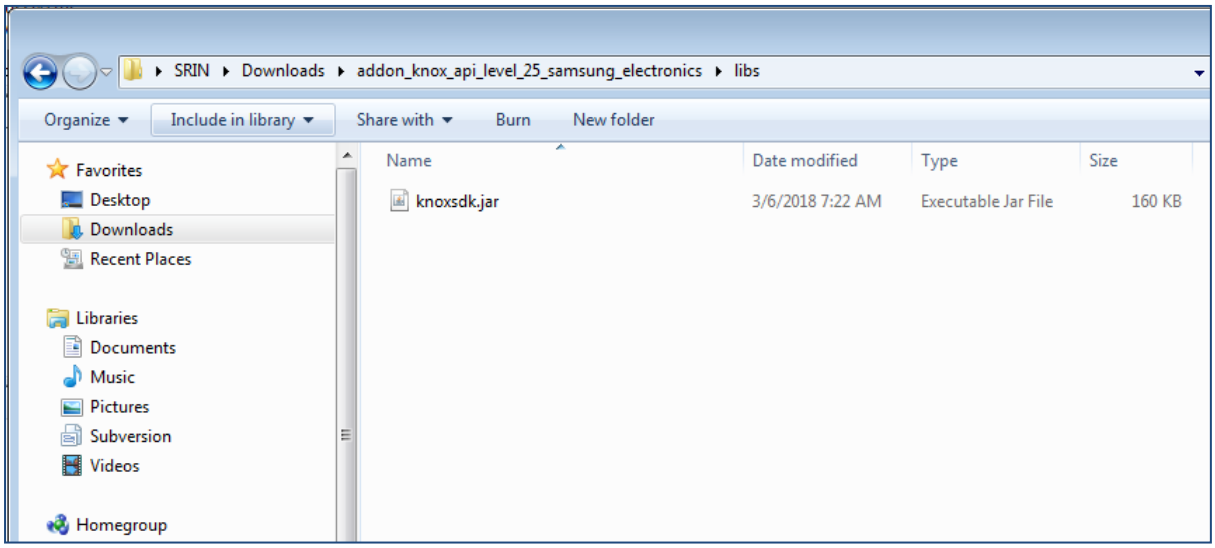
In the project template provided by this guide, there is one Android Studio project available ready to be used by users with only minor changes needed.



Since KNOX SDK is only allowed to be used by users who have been registered as SEAP partner, therefore KNOX SDK libraries will not be included in the project template. Thus users are required to download necessary KNOX SDKs themselves and include them in the project.

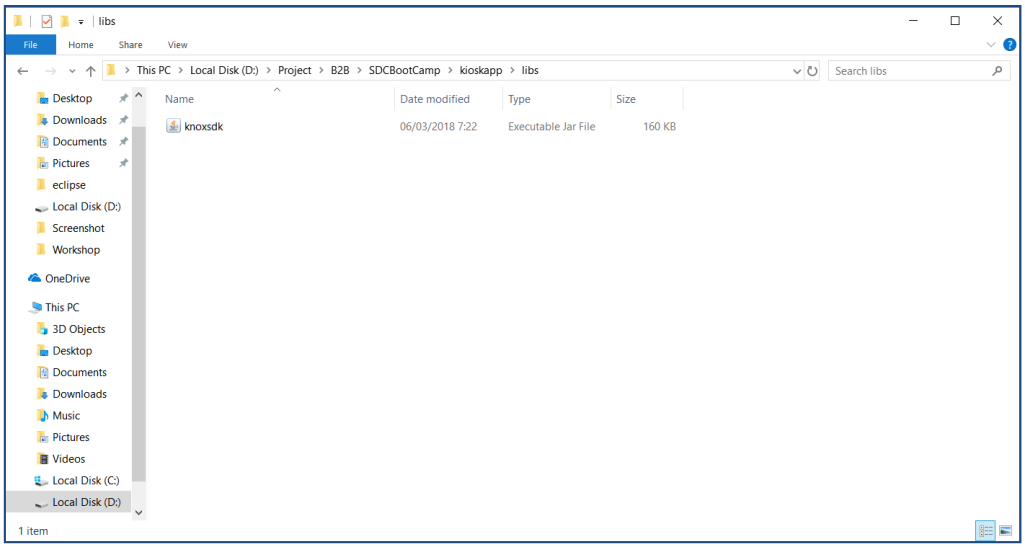
In the downloaded KNOX SDK file there are several jar libraries available for users to be able to use KNOX SDK features in their applications.

For KNOX SDK version 3.0(API level 25), jar library can be found in the **addon_knox_api_level_25_samsung_electronics\libs** directory where it contains a single knoxsdk.jar file:



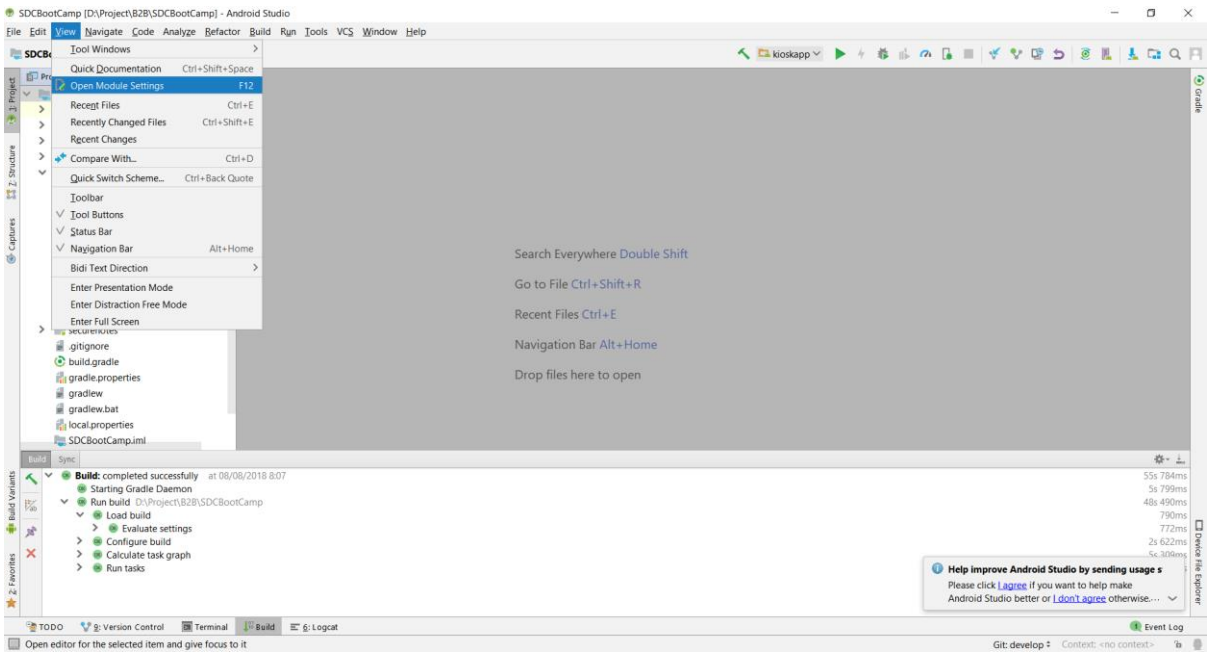
This Introduction to Samsung KNOX SDK guide covers KNOX Standard SDK.

To begin with the courses, users can start by copying the jar library from KNOX SDK file to **app\libs** directory of project template.

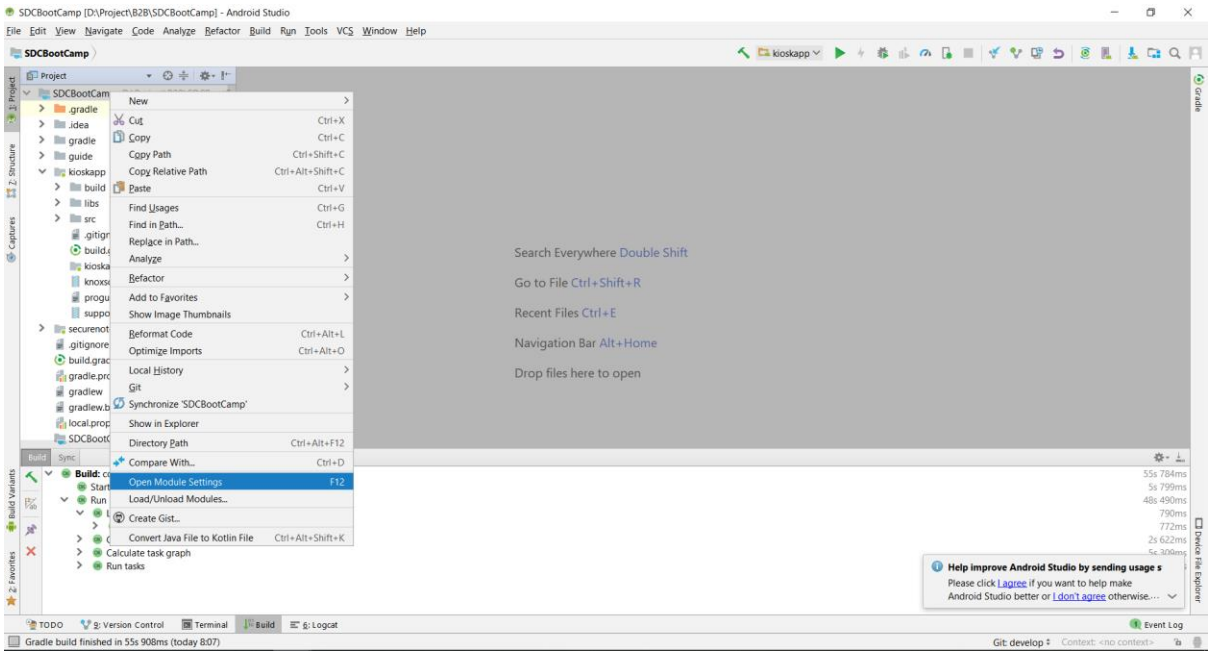


Next, users can include the jar library into the project using these steps.

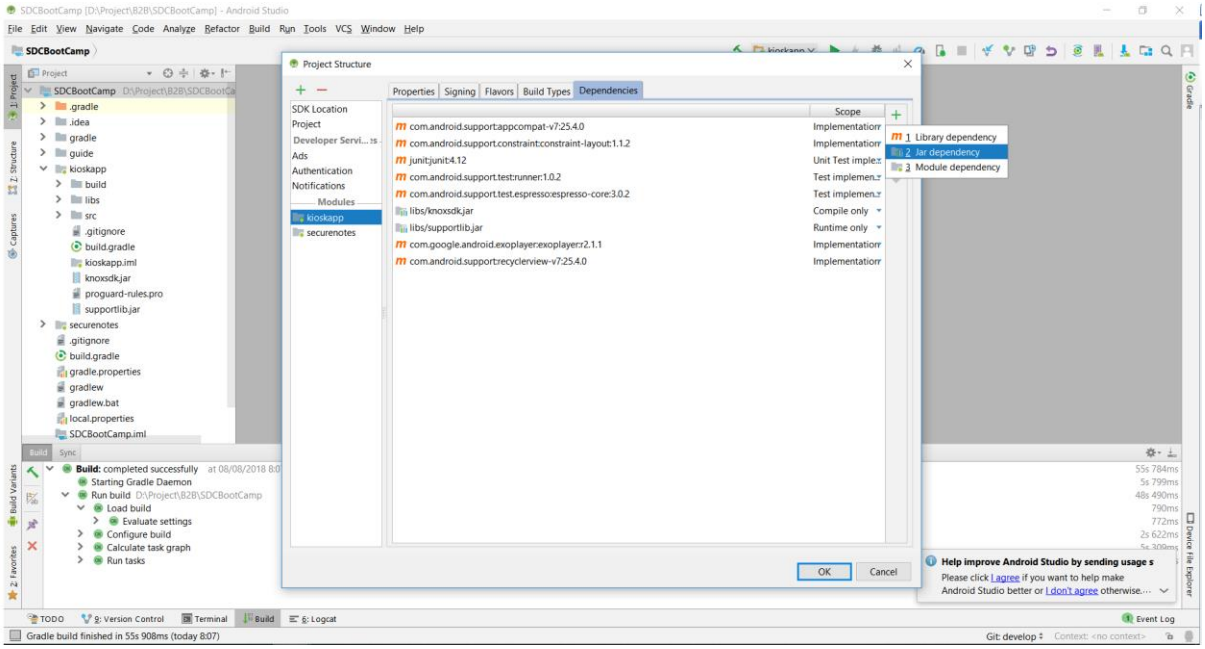
Open module settings window by selecting **View** menu and then choose **Open Module Settings** menu item.



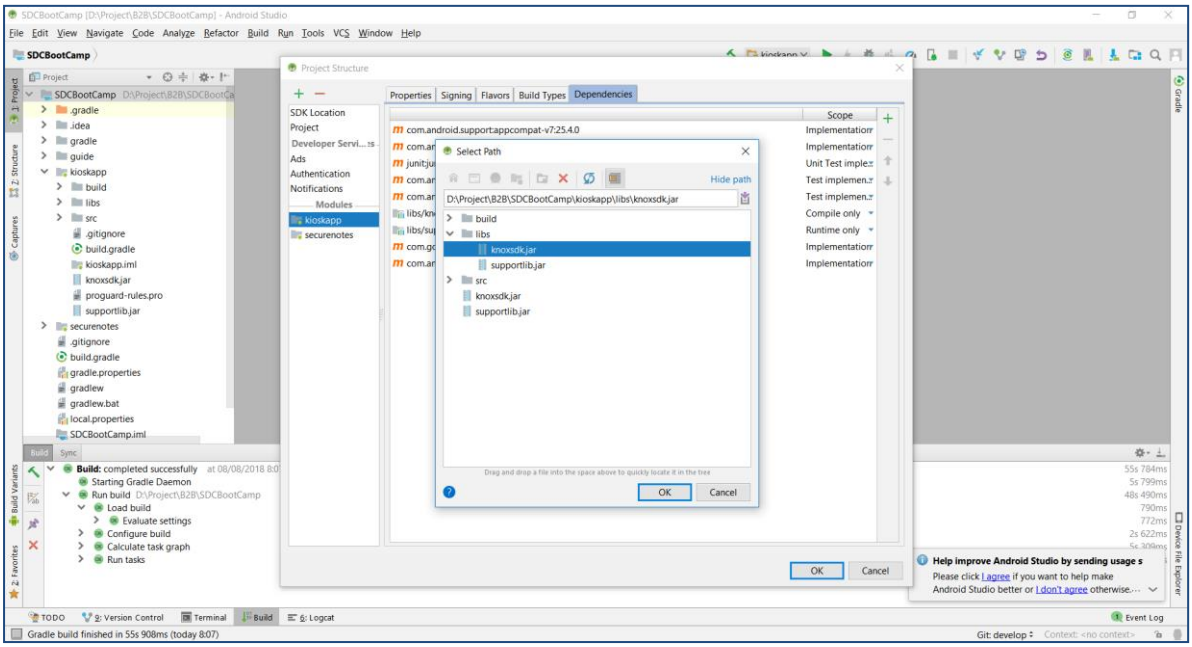
Or also available by right clicking in the **Project** window and choose **Open Module Settings**.



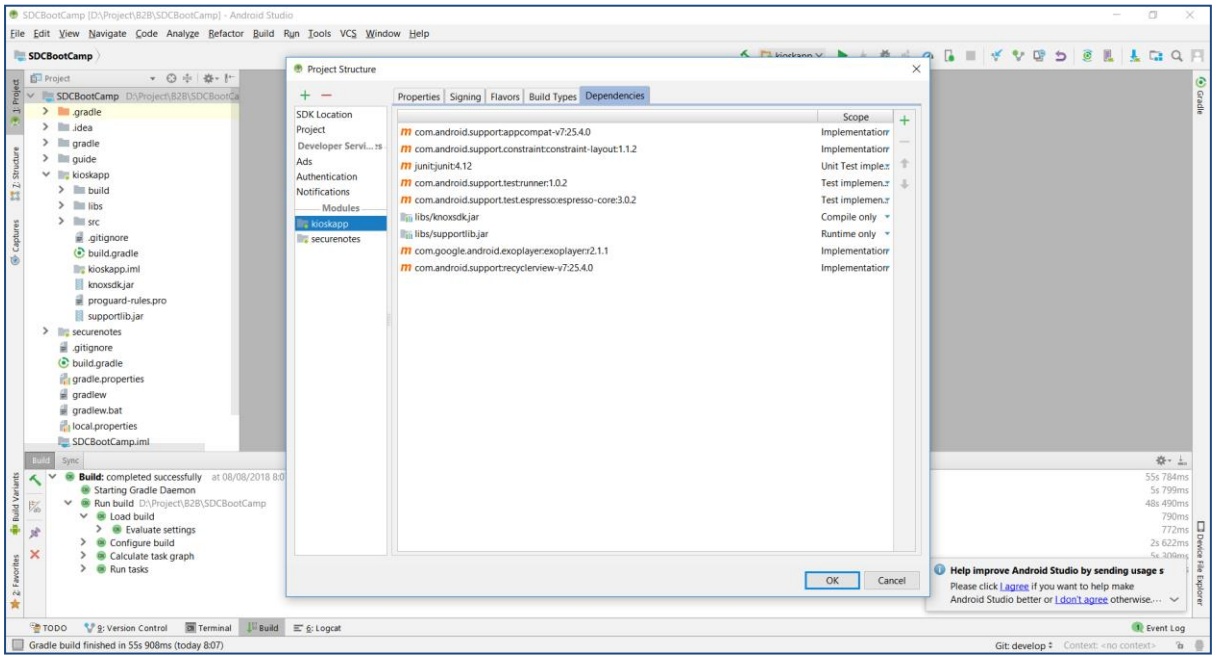
Next, open **Dependencies** settings by selecting **app** module and choose **Dependencies** tab.
Users can then add jar libraries by selecting **+** symbol and choose **Jar dependency**.



Jar libraries can be added to project one by one using above method.



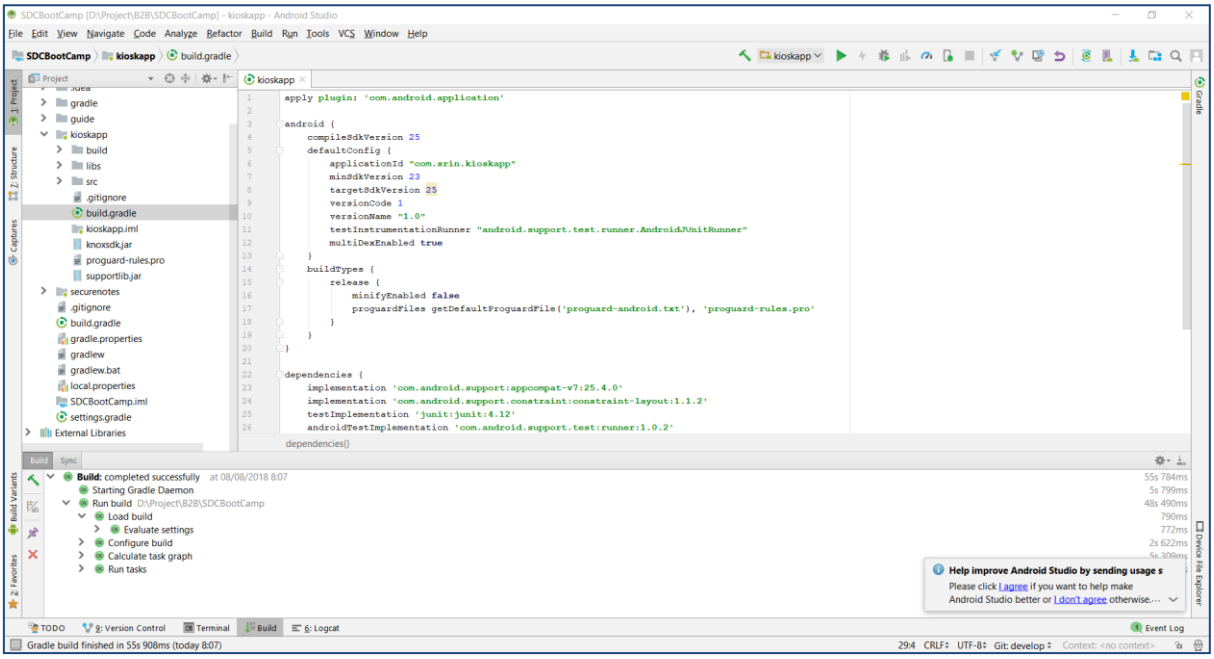
After users have included all the jar libraries into the project, users can now run the applications on the target device.

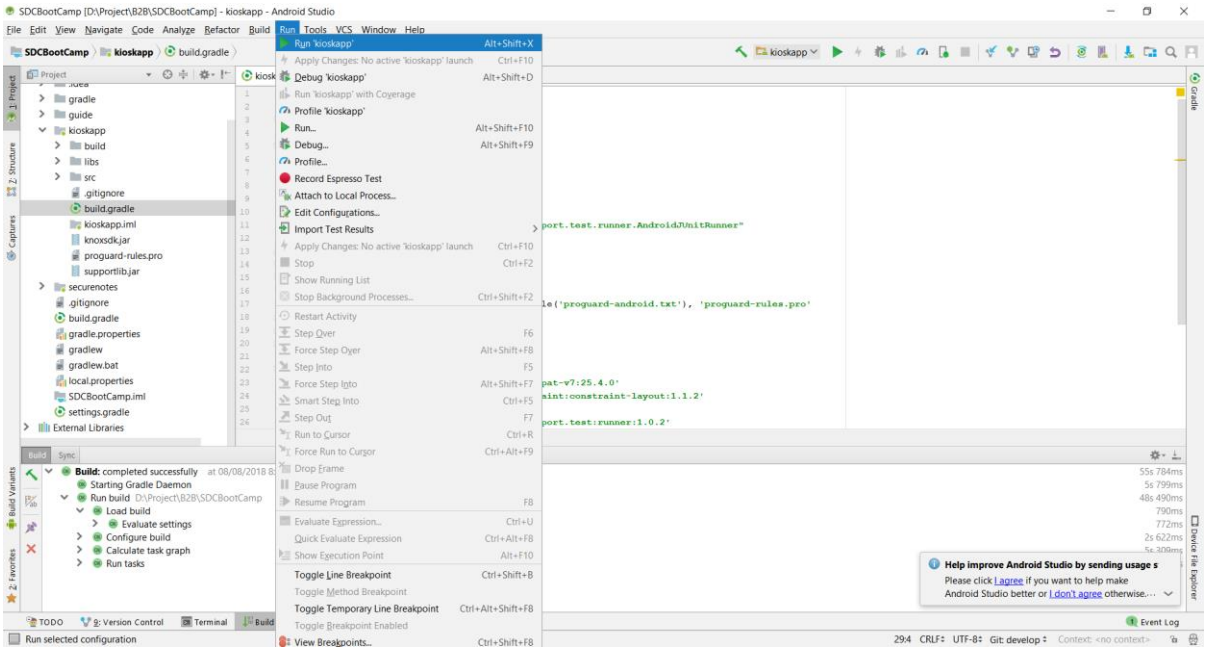


Jar libraries can also be included into the project by inputting these lines of code to the **app\build.gradle** file.

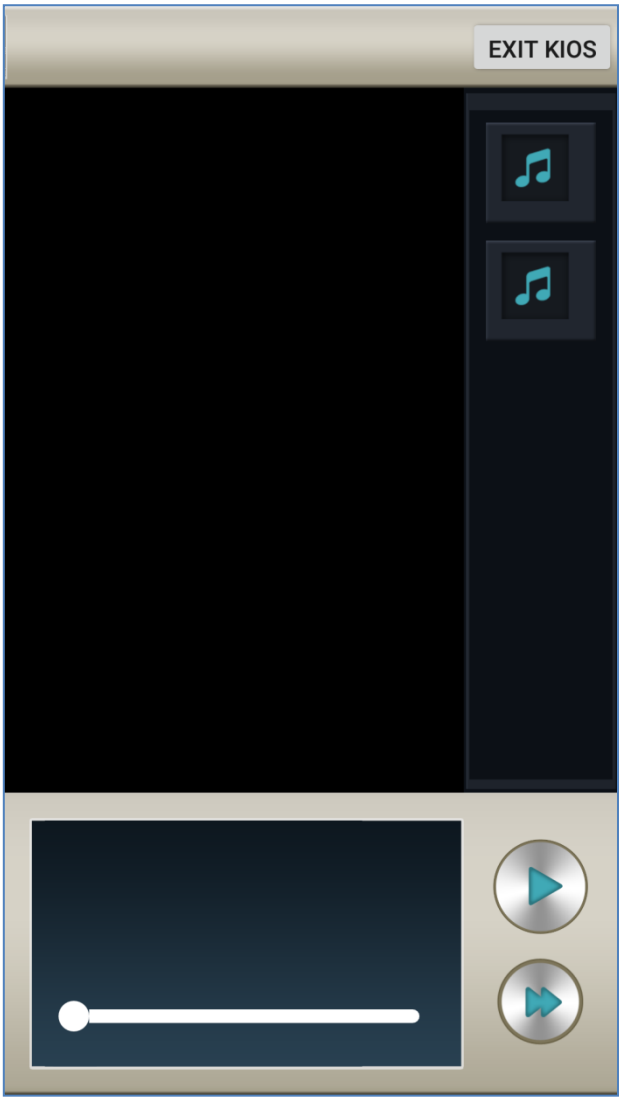
```
//     TODO : add knox library dependencies
    compileOnly files('libs/knoxsdk.jar')
```

Lastly, users can run the applications by selecting **Run** menu and choose **Run 'app'**.





0Application built from the project template will now run on the device, where basic user interface has been provided to facilitate the Introduction to Samsung KNOX SDK courses to be more focus on implementing KNOX SDK features.



Activate Device Administrator

Every application using KNOX SDK features are required to have device administration features enabled. More details about Android Device Administration API can be found at http://developer.android.com/guide/topics/admin/device-admin.html.

Any function calls to KNOX SDK APIs without having device administration features enabled will throw SecurityException error.

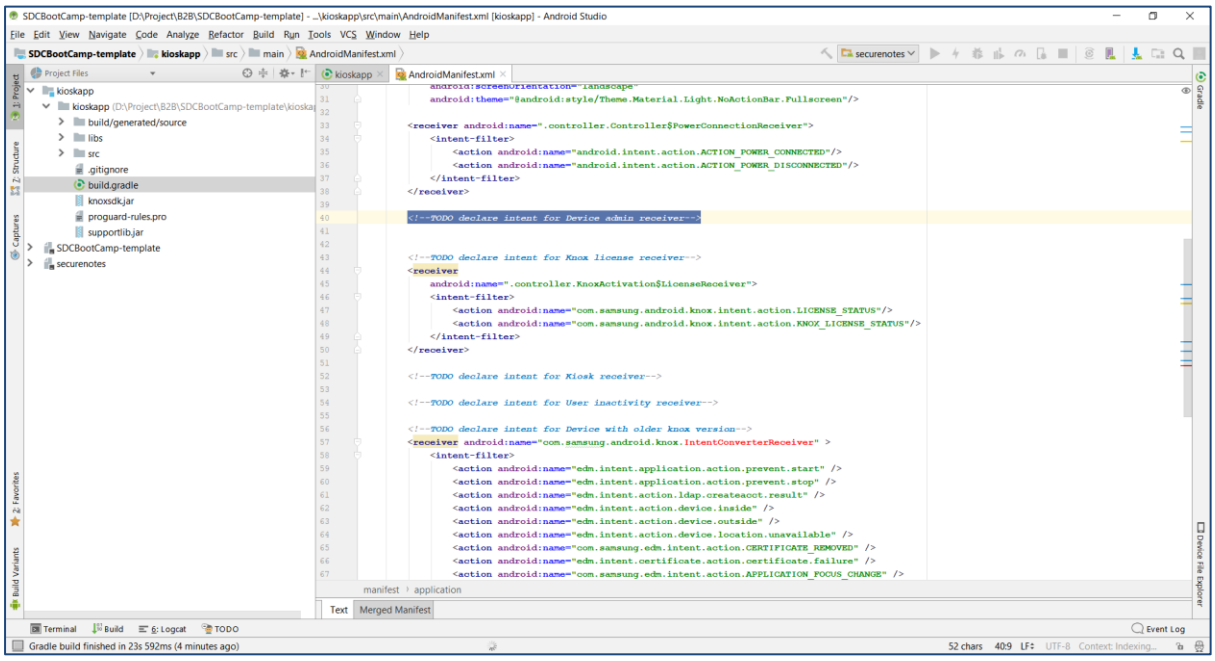
Steps involved in creating device administration applications can be found at http://developer.android.com/guide/topics/admin/device-admin.html#developing or as below

1. Create subclass of DeviceAdminReceiver to application manifest file
2. Include the security policies in metadata file
3. Subclass DeviceAdminReceiver
4. Enable the application as Device Administrator

Create subclass of DeviceAdminReceiver to application manifest file

Applications using Device Administration API have to create a subclass of DeviceAdminReceiver and include it to the application manifest file.

Users who are using the project template can open **app\src\main\AndroidManifest.xml** file and search for **Activate Device Administrator, Step 1** keyword and make necessary changes on the commented block of code.



Code changes in the manifest file can follow code excerpt below.

```
<!--TODO declare intent for Device admin receiver-->
<receiver
    android:name=".controller.KnoxActivation$KnoxAdmin"
    android:description="@string/enterprise_device_admin_description"
    android:label="@string/enterprise_device_admin"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin_receiver"/>
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Receiver class `.controller.KnoxActivation$KnoxAdmin` which is a subclass of DeviceAdminReceiver should be given `BIND_DEVICE_ADMIN` permission to perform actions in response to `DEVICE_ADMIN_ENABLED` intent broadcasted by system events.

Include the security policies in metadata file

Security policies used by applications should be declared in metadata file which is defined in the manifest file.

In the project template, the metadata file has been provided in **kioskapp\src\main\res\xml\ device_admin_receiver.xml** as defined in the manifest file `android:resource="@xml/enterprise_device_admin"`, and users can fill in the file content as follows.

```
<!--TODO define security policies metadata file -->
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
   <uses-policies>

   </uses-policies>
</device-admin>
```

Users do not need to include all the security policies in metadata file, only the ones that are relevant for the applications.

Failure to include necessary security policies required by KNOX SDK APIs will result to any function calls to those APIs throw SecurityException error.

More details on Android Device Administration API security policy can be found at http://developer.android.com/guide/topics/admin/device-admin.html#policies.

Subclass DeviceAdminReceiver

Next, users have to implement the subclass of DeviceAdminReceiver as included in the manifest file to the code.
This subclass consists of a series of callbacks that are triggered when device administration events occur.

In the project template, this class has been provided in **kioskapp\src\main\java\com\srin\kioskapp\controller\KnoxActivation.java** file as an inner class named **KnoxAdmin**.
Users can make necessary changes as the following.

```
public static class KnoxAdmin extends DeviceAdminReceiver {
 /*TODO define Device admin receiver class*/
    @Override
    public void onEnabled(Context context, Intent intent) {
        Toast.makeText(context, "Device Admin enabled",
Toast.LENGTH_SHORT).show();
    }

    @Override
    public CharSequence onDisableRequested(Context context, Intent intent)
{
        return "Disable from Device administrator";
    }

    @Override
    public void onDisabled(Context context, Intent intent) {
        Toast.makeText(context, "Device admin disabled",
Toast.LENGTH_SHORT).show();
    }
}
```

In above code excerpt, only some of callbacks are implemented to KnoxAdmin subclass. Users can implement other callbacks as can be found at http://developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html.

Enable the application as Device Administrator

The last step is to ask mobile device users to explicitly enable the application as Device Administrator for the security policies declared in metadata file to be enforced.
Mobile device users have to activate device administrator before applications can use any KNOX SDK APIs.

To activate device administrator, users can add the following code excerpt into the project.

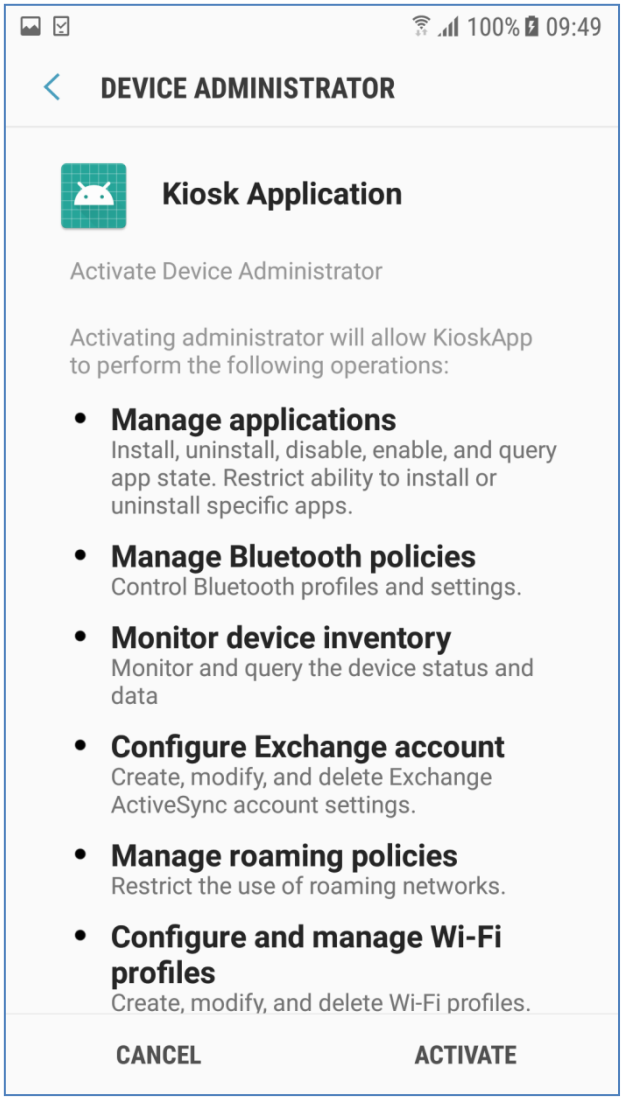
In the project template, there is an empty method named **activateAdmin** to put the code to run this command.

```
public void activateAdmin(Activity activity) {
/*TODO Implement activate device admin */
   if (null == mDPM) {
       Log.e(TAG, "Failed to get DevicePolicyManager");
       return;
   }

   boolean active = mDPM.isAdminActive(mDeviceAdmin);
   if (!active) {
       try {
           Intent intent = new
Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
           intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN,
mDeviceAdmin);
           intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
"Activate device administrator.");
           activity.startActivityForResult(intent, RESULT_ENABLE);
       } catch (Exception e) {
           Log.w(TAG, "Exception: " + e);
       }
   } else {
       Log.w(TAG, "Admin already activated");
   }
}
```

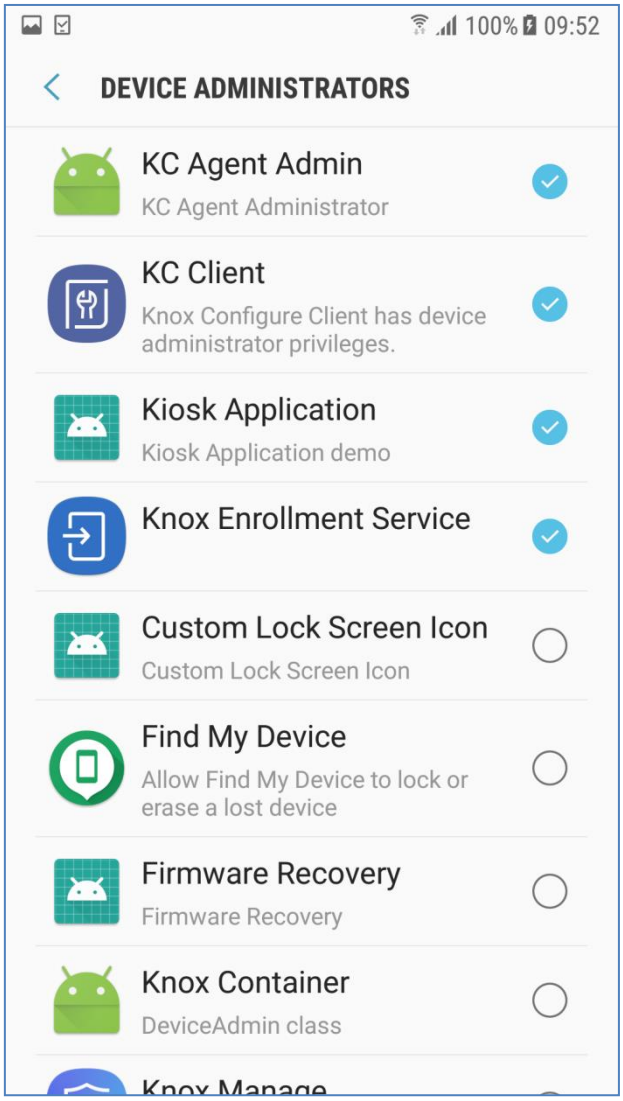
The **mDPM** variable used in above code excerpt is already provided in the project template and initialized as below.

```
mDPM = (DevicePolicyManager)
mContext.getSystemService(Context.DEVICE_POLICY_SERVICE);
```

This command will display a window to prompt users to activate device administrator and enforce security policies as declared in metadata file.



Users can manage device administrator applications by means of **Settings → Lock screen and security → Other security settings → Device administrators**, where mobile device users are allowed to disable device administrator applications added to the system. However developers can also opt to prevent mobile device users to disable device administrator applications using KNOX SDK.



Warning. This guide book does not advise to prevent disable device administrator applications during development, due to any errors in applying KNOX policies by developers might not be reversible and applications may not be able to be uninstalled.

Activate License

Aside from requirement to have device administration enabled, every application using KNOX SDK features are also required to activate KNOX licenses according to KNOX SDKs used by the applications.

Any function calls to KNOX SDK APIs without having KNOX licenses activated for respective SDK will throw SecurityException error.

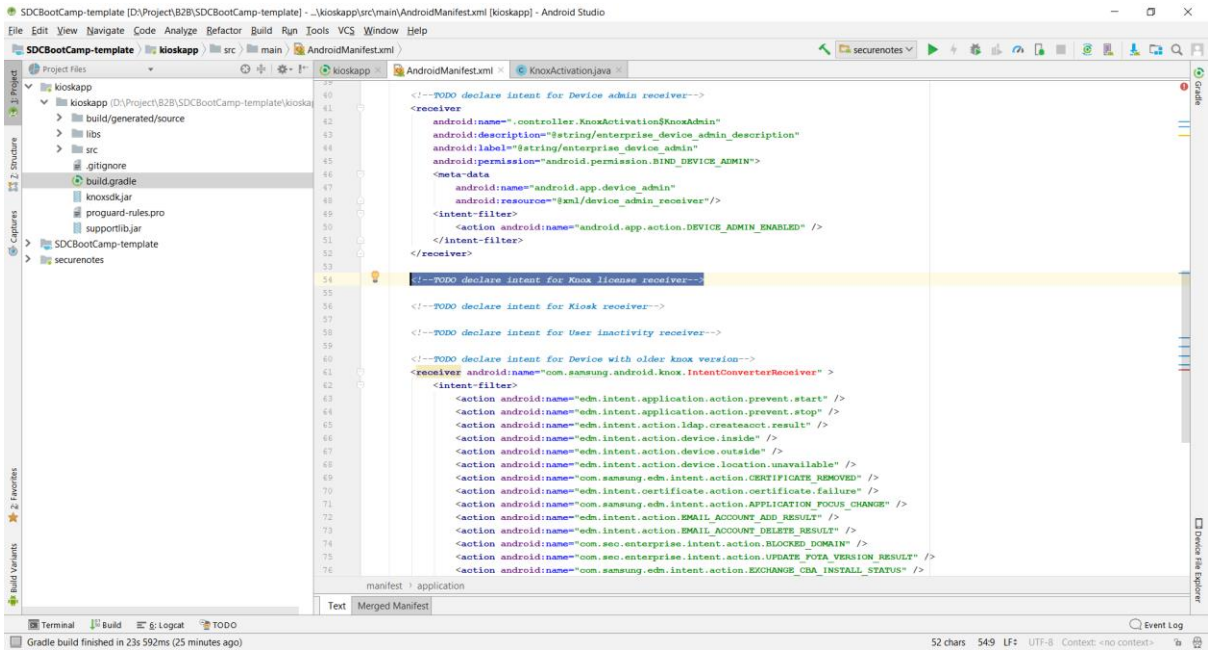
To activate KNOX licenses, users can follow steps as outlined on SEAP portal at these links Samsung KNOX activation APIs and to activate KNOX licenses, or as below.

1. Create subclass of BroadcastReceiver to application manifest file
2. Subclass BroadcastReceiver to respond to KNOX licenses activation events
3. Activate KNOX licenses

Create subclass of BroadcastReceiver to application manifest file

Applications that perform activate KNOX licenses have to create a subclass of BroadcastReceiver and include it to the application manifest file.
This BroadcastReceiver class should be able to respond to
`com.samsung.android.knox.intent.action.LICENSE_STATUS`
intent broadcasted on ELM license activation for Standard SDK.

In the project template, users can open **app\src\main\AndroidManifest.xml** file and search for **Activate License, Step 1** keyword and make necessary changes on the commented block of code.



Code changes in the manifest file can follow code excerpt below.

```
<!--TODO declare intent for Knox license receiver-->
<receiver
   android:name=".controller.KnoxActivation$LicenseReceiver"
   android:enabled="true">
   <intent-filter>
       <action
android:name="com.samsung.android.knox.intent.action.LICENSE_STATUS" />
       <action
android:name="com.samsung.android.knox.intent.action.KNOX_LICENSE_STATUS"
/>
   </intent-filter>
</receiver>
```

Receiver class `.controller.KnoxActivation$LicenseReceiver` which is a subclass of BroadcastReceiver is to respond to KNOX licenses activation events.

Applications using KNOX SDK requires SKL license, required to include `com.samsung.android.knox.intent.action.KNOX_LICENSE_STATUS i`n the receiver class. In addition for device with knox version before 2.7.1 requires to include `com.samsung.android.knox.intent.action.LICENSE_STATUS` with backward compatibility key.

Subclass BroadcastReceiver to respond to KNOX licenses activation events

Next, users have to implement the subclass of BroadcastReceiver as included in the manifest file to the code.
This subclass will have the logic to store KNOX licenses activation results to *SharedPreferences* because KNOX SDK has no method to confirm whether applications have activated KNOX licenses.

In the project template, this class has been provided in **kioskapp\src\main\java\com\srin\kioskapp\controller\KnoxActivation.java** file as an inner class named **LicenseReceiver**.
Users can search for **LicenseReceiver** keyword and make necessary changes as the following.

```
public static class LicenseReceiver extends BroadcastReceiver {
    private static final String SUCCESS = "success";
    private static final String FAILURE = "fail";

    /*TODO handle knox license receiver*/
    @Override
    public void onReceive(Context context, Intent intent) {

if(EnterpriseLicenseManager.ACTION_LICENSE_STATUS.equals(intent.getAction())) {
            final String status =
intent.getExtras().getString(EnterpriseLicenseManager.EXTRA_LICENSE_STATUS);
            final int errorCode =
intent.getExtras().getInt(EnterpriseLicenseManager.EXTRA_LICENSE_ERROR_CODE,
Integer.MIN_VALUE);
            if(status==null) return;
```

```
            if(status.equals(SUCCESS)) {
                SharedPreferences sharedPreferences =
context.getSharedPreferences(LICENSEKNOX, Context.MODE_PRIVATE);
                SharedPreferences.Editor editor = sharedPreferences.edit();
                editor.putBoolean(ELMKEY, true);
                editor.commit();

                applyPolicy(context);
            } else {
                Log.e(TAG, "ELM Activate license error" +errorCode);
                if(mContext == null) return;
                Toast.makeText(mContext,"ELM Activate license error :
"+errorCode, Toast.LENGTH_SHORT).show();
            }
        }


if(KnoxEnterpriseLicenseManager.ACTION_LICENSE_STATUS.equals(intent.getAction()))
{
            final String status =
intent.getExtras().getString(KnoxEnterpriseLicenseManager.EXTRA_LICENSE_STATUS);
            final int errorCode =
intent.getExtras().getInt(KnoxEnterpriseLicenseManager.EXTRA_LICENSE_ERROR_CODE,
Integer.MIN_VALUE);
            if(status==null) return;
            if(status.equals(SUCCESS)) {
                SharedPreferences sharedPreferences =
context.getSharedPreferences(LICENSEKNOX, Context.MODE_PRIVATE);
                SharedPreferences.Editor editor = sharedPreferences.edit();
                editor.putBoolean(SKLKEY, true);
                editor.commit();
                KnoxActivation.getInstance(context).activateELMLicense();
            } else {
                Log.e(TAG, "SKL Activate license error : "+errorCode);
                if(mContext == null) return;
                Toast.makeText(mContext,"SKL Activate license error :
"+errorCode, Toast.LENGTH_SHORT).show();
            }
        }

    }
}
```

To find error code definition sent to receiver class, users can open this link at https://seap.samsung.com/api-references/android-standard/reference/android/app/enterprise/license/EnterpriseLicenseManager.html#ACTION_LICENSE_STATUS for list of ELM license activation error codes.

Activate KNOX licenses

Next, mobile device users have to explicitly accept to terms of privacy policy for applications to be able to use KNOX SDK APIs.

To activate KNOX licenses, users can add the following code excerpt into the project.

In the project template, there are empty method named **activateSKLLicense.** And if want to support devices older than 2.71 user need to **activateELMLicense.** Below is code to run this command.

```
public void activateSKLLicense() {
    /*TODO Implement activate SKL key*/
    if(mDPM.isAdminActive(mDeviceAdmin) && !isSKLLicenseActive(mContext))
{
        mSKL.activateLicense(sklLicense, pkgName);
    }
}
```

```
public void activateELMLicense() {
    /*TODO Implement activate ELM key*/
    if(mDPM.isAdminActive(mDeviceAdmin) && !isELMLicenseActive(mContext))
{
        mELM.activateLicense(elmLicense, pkgName);
    }
}
```

These methods use **mDPM, mSKL** and **mELM** which are already provided in the project template and initialized as below.

```
mDPM = (DevicePolicyManager)
mContext.getSystemService(Context.DEVICE_POLICY_SERVICE);
mELM = EnterpriseLicenseManager.getInstance(mContext);
mSKL = KnoxEnterpriseLicenseManager.getInstance(mContext);
```

Aside from those, these methods also use **pkgName** variable which is the application package name, and **sklLicense** which is SKL license keys that can be obtained from SEAP portal.

In this example, the app will call **activateSKLLicense** if device admin is already activated or just has been successfully activated.

```
public static class KnoxAdmin extends DeviceAdminReceiver {

   @Override
   public void onEnabled(Context context, Intent intent) {
       KnoxActivation.getInstance(context).activateSKLLicense();
   }
...
public void activateAdmin(Activity activity) {
   ...
   if (!active) {
       ...
```

```
    } else {
        Log.w(TAG, "Admin already activated");
        activateSKLLicense();
    }
}
```

This command will display a privacy policy window that needs to be accepted by mobile device users for applications to be able to use KNOX SDKs.

Support older devices

The Knox SDK landing page provides a downloadable library called supportlib.jar, which provides backwards compatibility with older Samsung devices running Knox v2.7 or earlier. These older devices don't recognize the new namespace used by the Knox SDK v3.x.

As described in Import the new libraries, you can include the supportlib.jar at run-time, to translate the new namespace used by the Knox SDK v3.x to the old namespace recognized by the older devices.

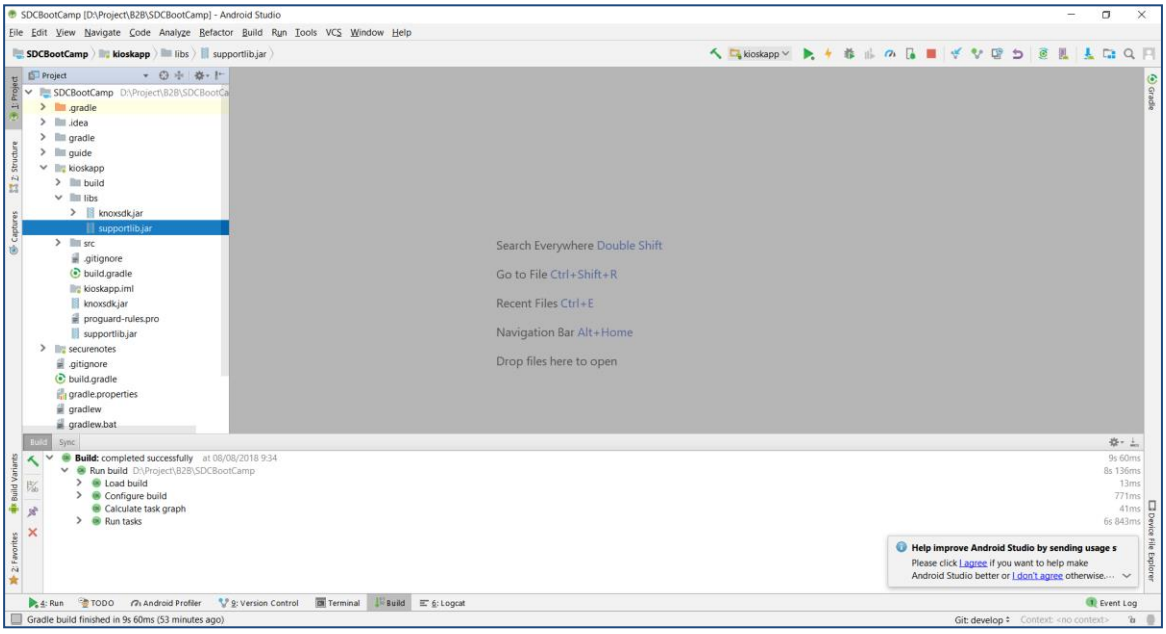
There are a couple of other things you need to do, which are described in this section:

- Add support library to project
- Define the old intents in the Manifest

Add support library to project

Put the supportlib.jar under the libs folder:

Now add the supportlib.jar to dependencies in by put this following line inside dependencies



bracket (app/build.gradle file):

```
dependencies {
    .
    .
    compileOnly files('libs/knoxsdk.jar')
    runtineOnly files('libs/supportlib.jar')

}
```

Define the old intents in the Manifest

Since supportlib.jar is added only for run-time scope and not used in compilation, you need to declare all the old intents in your app's Android manifest file.

Simply copy and paste the following code into your manifest.

```
<receiver android:name="com.samsung.android.knox.IntentConverterReceiver" >
    <intent-filter>
        <action android:name="edm.intent.application.action.prevent.start" />
        <action android:name="edm.intent.application.action.prevent.stop" />
        <action android:name="edm.intent.action.ldap.createacct.result" />
        <action android:name="edm.intent.action.device.inside" />
        <action android:name="edm.intent.action.device.outside" />
        <action android:name="edm.intent.action.device.location.unavailable" />
        <action android:name="com.samsung.edm.intent.action.CERTIFICATE_REMOVED" />
        <action android:name="edm.intent.certificate.action.certificate.failure" />
        <action android:name="com.samsung.edm.intent.action.APPLICATION_FOCUS_CHANGE" />
        <action android:name="edm.intent.action.EMAIL_ACCOUNT_ADD_RESULT" />
        <action android:name="edm.intent.action.EMAIL_ACCOUNT_DELETE_RESULT" />
        <action android:name="com.sec.enterprise.intent.action.BLOCKED_DOMAIN" />
        <action android:name="com.sec.enterprise.intent.action.UPDATE_FOTA_VERSION_RESULT" />
        <action android:name="com.samsung.edm.intent.action.EXCHANGE_CBA_INSTALL_STATUS" />
        <action android:name="android.intent.action.sec.CBA_INSTALL_STATUS" />
        <action android:name="edm.intent.action.EXCHANGE_ACCOUNT_ADD_RESULT" />
        <action android:name="edm.intent.action.EXCHANGE_ACCOUNT_DELETE_RESULT" />
        <action android:name="com.samsung.edm.intent.action.ENFORCE_SMIME_ALIAS_RESULT" />
        <action android:name="edm.intent.action.knox_license.status" />
        <action android:name="edm.intent.action.license.status" />
        <action android:name="com.samsung.edm.intent.event.NTP_SERVER_UNREACHABLE" />
        <action android:name="edm.intent.action.enable.kiosk.mode.result" />
        <action android:name="edm.intent.action.disable.kiosk.mode.result" />
        <action android:name="edm.intent.action.unexpected.kiosk.behavior" />
        <action android:name="com.samsung.edm.intent.action.SIM_CARD_CHANGED" />
        <action android:name="android.intent.action.sec.SIM_CARD_CHANGED" />
        <action android:name="com.samsung.action.knox.certenroll.CEP_CERT_ENROLL_STATUS" />
        <action android:name="com.samsung.action.knox.certenroll.CEP_SERVICE_DISCONNECTED" />
        <action android:name="com.sec.enterprise.knox.intent.action.KNOX_ATTESTATION_RESULT" />
        <action android:name="com.sec.action.NO_USER_ACTIVITY" />
        <action android:name="com.sec.action.USER_ACTIVITY" />
        <action android:name="com.samsung.android.mdm.VPN_BIND_RESULT" />
    </intent-filter>
</receiver>
```

Kiosk Mode

Kiosk Mode API allows enterprises to restrict user access to selected applications and features on the mobile devices. These applications and features can be made available through the kiosk home screen.

Some of Kiosk Mode APIs that will be covered in this guide book are:

1. Makes the app as a launcher
2. Disable hardware keys
3. Hide status bar

There are several steps we have to do before we start using Kiosk Mode,
First we need to declare two Instances: EnterpriseDeviceManager and KioskMode object.

Inside the Controller.java class constructor:

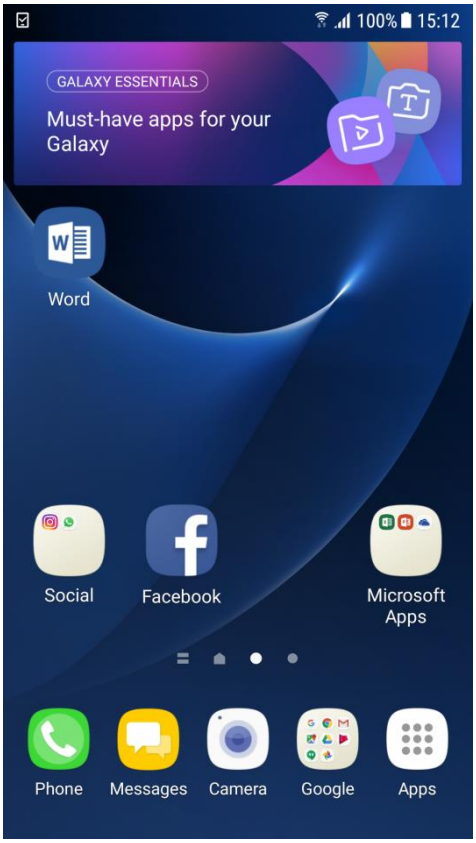
```
private EnterpriseDeviceManager mEDM;
private KioskMode mKioskMode;

public Controller(Context context) {
    ...
    /* initiate Enterprise Device Manager instance*/
    mEDM =  EnterpriseDeviceManager.getInstance(mContext);


    /* initiate Kiosk Mode instance*/
    mKioskMode = mEDM.getKioskMode();
}
```

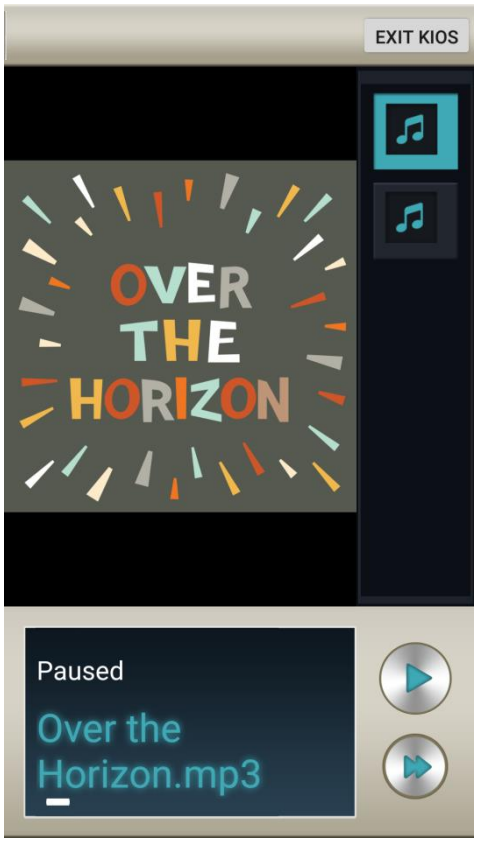
Next, we add the permission **“com.samsung.android.knox.permission.KNOX_KIOSK_MODE”** inside the AndroidManifest.xml file:

```
<!--TODO add permission for kiosk mode-->
  <uses-permission android:name="com.samsung.android.knox.permission.KNOX_KIOSK_MODE"/>
```

Make the Apps as Kiosk Launcher

Now we make our apps as a default launcher. When the device has finished reboot, the apps will automatically launched (instead of the default home screen).





Enable Kiosk Mode

To enable Kiosk Mode, we just call method enableKioskMode() from KioskMode class. As we define our kiosk mode instance as **mKioskMode,** we execute **mKioskMode.enableKioskMode()**.

```
public void enableKioskMode(String pkgName){
    /*TODO Implement enable kiosk mode*/
    try {
        if(!isKioskModeEnabled()) {
            Log.w("TRACE", "enableKioskMode: ");
            mKioskMode.enableKioskMode(pkgName);
        }
    } catch (SecurityException e) {
        Log.w(TAG, "SecurityException: " + e);
    }
}
```

The isKioskModeEnabled() method is already define in the sample code, this method will check if we have already enable kiosk mode before. And here's the implementation looks like:

```
public boolean isKioskModeEnabled(){
    return mKioskMode.isKioskModeEnabled();
}
```

Disable Kiosk Mode

The KioskMode class also provide the API to disable the KioskMode that we create before. To do that, we just execute the **mKioskMode.disableKioskMode()**.

```
public void disableKioskMode(){
    /*TODO Implement disable kiosk mode*/
    try {
        Log.w("TRACE", "disableKioskMode: ");
        mKioskMode.disableKioskMode();
    } catch (SecurityException e) {
        Log.w(TAG, "SecurityException: " + e);
    }
}
```

Define the Kiosk Mode Receiver

Now we need to know whether we have success or not while we enable/disable the Kiosk Mode. Knox API provides you the Broadcast Receiver to handle the result of enabling/disabling Kiosk Mode process:

Frist, we define the intent filter of the broadcast in our AndroidManifest.xml file

```
<!--TODO declare intent for Kiosk receiver-->

<receiver android:name=".controller.Controller$KioskModeReceiver">
    <intent-filter>
        <action
android:name="com.samsung.android.knox.intent.action.ENABLE_KIOSK_MODE_RESULT"
/>
        <action
android:name="com.samsung.android.knox.intent.action.DISABLE_KIOSK_MODE_RESULT"
/>
    </intent-filter>
</receiver>
```

Next, we handle the result inside the **onReceive** method under the KioskModeReceiver class:

```
public static class KioskModeReceiver extends BroadcastReceiver {
   /*TODO handle kiosk mode receiver*/
   @Override
   public void onReceive(Context context, Intent intent) {
      if (intent.getAction().equals(KioskMode.ACTION_ENABLE_KIOSK_MODE_RESULT))
{
         final int kioskResult =
intent.getExtras().getInt(KioskMode.EXTRA_KIOSK_RESULT);
         if (kioskResult == KioskMode.ERROR_NONE) {
            Log.w("TRACE", "enableKioskMode: success");
            //Do something
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_HOME);
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_BACK);
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_VOLUME_UP);
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_VOLUME_DOWN);
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_MENU);
            Controller.getInstance(context).disableHardwareKey(true,
KeyEvent.KEYCODE_APP_SWITCH);
            Controller.getInstance(context).hideStatusBar(true);
            Controller.getInstance(context).setUserTimeOut(10);
         } else {
            Toast.makeText(context, "Kiosk result error " + kioskResult,
Toast.LENGTH_SHORT).show();
         }
} else if
(intent.getAction().equals(KioskMode.ACTION_DISABLE_KIOSK_MODE_RESULT)) {
         final int kioskResult =
intent.getExtras().getInt(KioskMode.EXTRA_KIOSK_RESULT);

         if (kioskResult == KioskMode.ERROR_NONE) {
            Log.w("TRACE", "disableKioskMode: success");
            //Do something
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_HOME);
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_BACK);
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_VOLUME_UP);
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_VOLUME_DOWN);
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_MENU);
            Controller.getInstance(context).disableHardwareKey(false,
KeyEvent.KEYCODE_APP_SWITCH);
            Controller.getInstance(context).hideStatusBar(false);
         }
      }
   }
}
```

Now you can put any logic while the kiosk mode has been successfully enabled or disabled. For this sample app, we are going to disable/enable hardware button and hide the status bar after the kiosk mode enabled, that function we will implemented on the next chapter.

Disable hardware keys

To disable hardware keys so users are not allowed to use specific device keys, application developers can use KioskMode.allowHardwareKeys(List<Integer> hwKeyId, boolean allow) API provided by KNOX SDK.

Let's implement the code inside the disableHardwareKey() method under the Controller class

```
    public void disableHardwareKey(boolean disable, int hwKey) {
    /*TODO Implement toggle hardware keys*/
        try {
            if (mKioskMode.allowHardwareKeys(new
ArrayList<Integer>(Arrays.asList(new Integer[]{hwKey})),
!disable).contains(hwKey)) {
                mToast.setText(String.format("%s Hardware Keys %s success", disable
? "Disable" : "Enable", KeyEvent.keyCodeToString(hwKey).replace("KEYCODE_",
"")));
            } else {
                mToast.setText(String.format("%s Hardware Keys %s failed", disable ?
"Disable" : "Enable", KeyEvent.keyCodeToString(hwKey).replace("KEYCODE_", "")));
            }
        } catch (SecurityException e) {
            Log.w(TAG, "SecurityException: " + e);
            mToast.setText("Error: SecurityException occurred - " + e);
            mToast.show();
        }
    }
```

In above code excerpt, method arguments that passed to the API are **`Arrays.asList(new Integer[]{hwKey}))`** for **hwKeyId** parameter and **`!disable`** for **allow** parameter.

hwKeyId parameter expects list of Key code constants as can be found at http://developer.android.com/reference/android/view/KeyEvent.html#constants. In the project template, some of Key code constants used are KeyEvent.KEYCODE_HOME, KeyEvent.KEYCODE_POWER, KeyEvent.KEYCODE_MENU, KeyEvent.KEYCODE_APP_SWITCH, KeyEvent.KEYCODE_BACK, KeyEvent.KEYCODE_VOLUME_UP and KeyEvent.KEYCODE_VOLUME_DOWN.

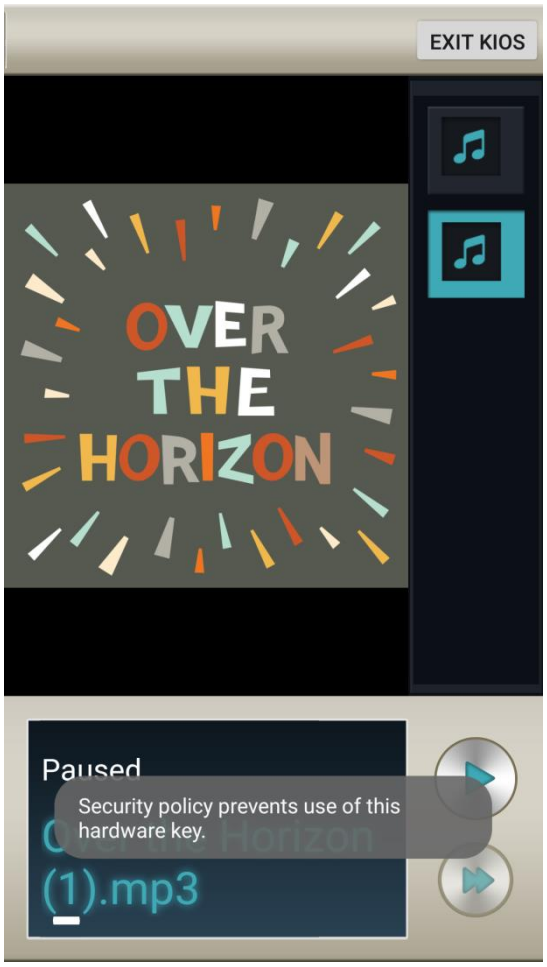
Whilst **allow** parameter expects *boolean* value to determine whether the API being called to allow or deny access to device keys as specified in the **hwKeyId** parameter.

In above code excerpt, **allow** parameter uses negative value of **`!disable`** because KioskMode.allowHardwareKeys(List<Integer> hwKeyId, boolean allow) API expects **`true`** value to allow access to device keys and **`false`** value to deny access to device keys.

Whereas **disableHardwareKey** method used in the project passes `true` value when users disable device keys and `false` value when users enable device keys.

This API returns list of Key code constants of allowed or denied device keys depending on **allow** parameter passed to the API. These returns values can also be used by developers to determine whether the API call was success or fail.

If applications have successfully called KioskMode.allowHardwareKeys(List<Integer> hwKeyId, boolean allow) API, when users try to use the keys, system will pop an alert with error message **Security policy prevents use of this hardware key**.



Warning. Please be advised that using this API without caution may lock the device and render the device inaccessible.

Hide status bar

To hide the device status bar so users are not allowed to interact with the status bar, application developers can use KioskMode.hideSystemBar(boolean hide) API provided by KNOX SDK.

Let's implement the code inside the hideStatusBar() method under the Controller class

```
public void hideStatusBar(boolean state){
   /*TODO Implement toggle status bar*/
   try {
      mKioskMode.hideStatusBar(state);
   } catch (SecurityException e) {
      Log.w(TAG, "SecurityException: " + e);
   }
}
```

state parameter expects *boolean* value to determine whether the API being called to hide or show device status bar.

Users can notice that when KioskMode.hideSystemBar(boolean hide) API has been called successfully, device status bar will no longer be visible to the users.

Custom Device Manager

Custom Device Manager Api provide functionally to control device settings on mobile devices.

Some of Custom Device Manager APIs that will be covered in this guide book are:

1. User Inactivity Timeout
2. Boot Animation
3. Disable application uninstallation
4. Power Off
5. Auto Power On

User inactivity Timeout

In the application demo , we have function to show screen saver. To make all devices had same screen time before we show screen saver, application developer can use SystemManager.setUserInactivityTimeout(int timeout) API provided by KNOX Customization SDK.

This API requires **com.samsung.android.knox.permission.KNOX_CUSTOM_SYSTEM** permission to be added to application manifest file.

```
<!--TODO add permission for knox custom-->
<uses-permission android:name="
com.samsung.android.knox.permission.KNOX_CUSTOM_SYSTEM " />
```

To use this API, users can add the following code excerpt into the project.

In the project template, there is an empty method named **setUserTimeOut**

```
public void setUserTimeOut(int userTimeOut) {
   /*TODO Implement set user inactivity timeout*/
   try {
      CustomDeviceManager cdm = CustomDeviceManager.getInstance();
      SystemManager systemManager = cdm.getSystemManager();
      int result = systemManager.setUserInactivityTimeout(userTimeOut);
      Log.d(TAG,  result== 0 ?
            "set timeout success " : "set timeout failed");
      int screenResult = systemManager.setScreenTimeout(10);
      Log.d(TAG, "set screen timeout result : "+screenResult);
   } catch(SecurityException e) {
      Log.w(TAG, "SecurityException:" + e);
   }catch (Exception ee){
      ee.printStackTrace();
   }
}
```

Details for each line in above code excerpt are as the following.

As with the SystemManager API, first is to create CustomDeviceManager instance which can be done by calling CustomDeviceManager.getInstance() API, where this instance will be used in every succeeding calls to KnoxCustomization API. Then we create SystemManager which can be done by calling CustomDeviceManager.getSystemManager().

Application developer need to pass integer timeout, to set how long before broadcast receiver triggered

Then we create Broadcast Receiver to receive intent, and class to handle the intent

```
<!--TODO declare intent for User inactivity receiver-->
<receiver android:name=".receiver.UserActivityReceiver">
    <intent-filter>
        <action
android:name="com.samsung.android.knox.intent.action.NO_USER_ACTIVITY"/>
        <action android:name="com.sec.action.NO_USER_ACTIVITY"/>
    </intent-filter>
</receiver>
```

After code has been implemented in the project, users can run applications. After timeout (set by user) application will show screen saver

Boot Animation

In the application we want to change boot animation, application developers can use SystemManager.setBootingAnimation (String animationFile, String loopFile, String soundFile, int delay) API provided by KNOX Customization SDK.

This API requires **com.samsung.android.knox.permission.KNOX_CUSTOM_SYSTEM** permission to be added to application manifest file.

```
<uses-permission android:name="
com.samsung.android.knox.permission.KNOX_CUSTOM_SYSTEM " />
```

To use this API, users can add the following code excerpt into the project.

In the project template, there is an empty method named **changeBootAnimation**

```
public void changeBootAnimation() {
   /*TODO : Implement change boot animation*/
   CustomDeviceManager cdm = CustomDeviceManager.getInstance();
   SystemManager systemManager = cdm.getSystemManager();
   try {
      String animationFile = mContext.getFilesDir() + "/bootsamsung.qmg";
      String loopFile = mContext.getFilesDir() + "/bootsamsungloop.qmg";
      String soundFile = mContext.getFilesDir() + "/sound.ogg";

      File fileAnimation = new File(animationFile);
      fileAnimation.setReadable(true, false);
      ParcelFileDescriptor animationFD = ParcelFileDescriptor.open(fileAnimation,
ParcelFileDescriptor.MODE_READ_ONLY);
      File fileLoop = new File(loopFile);
      fileLoop.setReadable(true, false);
      ParcelFileDescriptor loopFD = ParcelFileDescriptor.open(fileLoop,
ParcelFileDescriptor.MODE_READ_ONLY);
      File fileSound = new File(soundFile);
      fileSound.setReadable(true, false);
      ParcelFileDescriptor soundFD = ParcelFileDescriptor.open(fileSound,
ParcelFileDescriptor.MODE_READ_ONLY);

      int error = systemManager.setBootingAnimation(animationFD, loopFD, soundFD,
2000);

      mToast.setText("errorcode "+ error);
      mToast.show();
   } catch(FileNotFoundException e) {
      Log.w(TAG, "FileNotFoundException:" + e);
   } catch(NoSuchMethodError e) {
      Log.w(TAG, "NoSuchMethodError:" + e);
   } catch (SecurityException e) {
      Log.w(TAG, "SecurityException:" + e);
   }
}
```

Details for each line in above code excerpt are as the following.

As with the SystemManager API, first is to create CustomDeviceManager instance which can be done by calling CustomDeviceManager.getInstance() API, where this instance will be used in every succeeding calls to KnoxCustomization API. Then we create SystemManager which can be done by calling CustomDeviceManager.getSystemManager().

Application developers need 3 file for boot animation, “animation.qmg”, “loop.qmg” and “sound.ogg”.
Application developers can use Create custom boot and shutdown animations guide and tools to help creating necessary files.

After creating animation, loop and sound files for boot animation, application developers need to put the files into Context.getFilesDir() path, and then call File.setReadable(true,false) for SystemManager.setBootingAnimation (String animationFile, String loopFile, String soundFile, int delay) API to successfully implemented.

Next, Create ParcelFileDescriptor object and then call SystemManager.setBootingAnimation (String animationFile, String loopFile, String soundFile, int delay) API to change boot animation.

animationFile parameter expects *File* value, it use *.qmg* format containing graphics to be displayed. **loopFile** parameter expects *File* value , it use *.qmg* format containing animation loop graphics to be displayed. **soundFile** parameter expects File value, it use .ogg format containing sound to be played. **delay** parameter expects *Integer* value for delay of boot sound in milliseconds

This API returns Integer value that can be used by developers to determine whether the API call was success or fail.

After code has been implemented in the project, users can run applications, and press **Download Boot Animation, Download Boot Loop Animation, Download Boot Sound** to download animation, loop and sound files

Users can notice that when restart device, boot logo is changed.

Shutdown Animation

To change shutdown animation, application developers can use SystemManager.setShuttingDownAnimation (String animationFile,String soundFile) API provided by KNOX Customization SDK.

This API requires **com.sec.enterprise.knox.permission.CUSTOM_SYSTEM** permission to be added to application manifest file

```
<uses-permission
android:name="com.sec.enterprise.knox.permission.CUSTOM_SYSTEM"/>
```

To use this APIs, users can add the following code excerpt into the project.

In the project template, there is an empty method named **changeShutdownAnimation**

```
public void changeShutdownAnimation() {
   /*TODO : Implement change shutdown animation*/
   CustomDeviceManager cdm = CustomDeviceManager.getInstance();
   SystemManager systemManager = cdm.getSystemManager();
   try {
      String animationFile = mContext.getFilesDir() + "/shutdown.qmg";
      String soundFile = mContext.getFilesDir() + "/sound.ogg";

      File fileAnimation = new File(animationFile);
      fileAnimation.setReadable(true, false);
      ParcelFileDescriptor animationFD = ParcelFileDescriptor.open(fileAnimation,
ParcelFileDescriptor.MODE_READ_ONLY);
      File fileSound = new File(soundFile);
      fileSound.setReadable(true, false);
      ParcelFileDescriptor soundFD = ParcelFileDescriptor.open(fileSound,
ParcelFileDescriptor.MODE_READ_ONLY);


      int error = systemManager.setShuttingDownAnimation(animationFD, soundFD);
      mToast.setText("errorcode " + error);
      mToast.show();
   } catch(FileNotFoundException e) {
      Log.w(TAG, "FileNotFoundException:" + e);
   } catch(NoSuchMethodError e) {
      Log.w(TAG, "NoSuchMethodError:" + e);
   } catch(SecurityException e) {
      Log.w(TAG, "SecurityException:" + e);
   }

}
```

Details for each line in above code excerpt are as the following.

As with the SystemManager API, first is to create CustomDeviceManager instance which can be done by calling CustomDeviceManager.getInstance() API, where this instance will be used in every succeeding calls to KnoxCustomization API. Then we create SystemManager which can be done by calling CustomDeviceManager.getSystemManager().

Application developers need 3 file for boot animation, “animation.qmg” and “sound.ogg”. Application developers can use Create custom boot and shutdown animations guide and tools to help creating necessary files.

After creating animation and sound files for shutdown animation, application developers need to put the files into Context.getFilesDir() path, and then call File.setReadable(true,false) for SystemManager.setShutingDownAnimation (String animationFile,String soundFile) API to successfully implemented.

Next, call SystemManager.setShutdownAnimation (String animationFile,String soundFile) API to change shutdown animation.

animationFile parameter expects *File* value, it use *.qmg* format containing graphics to be displayed. **soundFile** parameter expects File value, it use .ogg format containing sound to be played.

This API returns Integer value that can be used by developers to determine whether the API call was success or fail.

Users can notice that when restart device, shutdown logo is changed.

Power Off

We want to powering off device when there was no power connected to device, application developers can use SystemManager.powerOff () API provided by KNOX Customization SDK.

This API requires **com.sec.enterprise.knox.permission.CUSTOM_SYSTEM** permission to be added to application manifest file

```
<uses-permission
android:name="com.sec.enterprise.knox.permission.CUSTOM_SYSTEM"/>
```

To use this APIs, users can add the following code excerpt into the project.

In the project template, there is an empty method named **powerOff**

```
public void powerOff() {
    /*TODO Implement power off device */
    CustomDeviceManager cdm = CustomDeviceManager.getInstance();
    SystemManager systemManager = cdm.getSystemManager();
    try {
        systemManager.powerOff();
    }catch (NoSuchMethodError e) {
        Log.e(TAG,"NoSuchMethodError:" + e);
    }catch (SecurityException e) {
        Log.e(TAG,"SecurityException:" + e);
    }
}
```

Details for each line in above code excerpt are as the following.

As with the SystemManager API, first is to create CustomDeviceManager instance which can be done by calling CustomDeviceManager.getInstance() API, where this instance will be used in every succeeding calls to KnoxCustomization API. Then we create SystemManager which can be done by calling CustomDeviceManager.getSystemManager() **,**then call SystemManager.powerOff () to powering off device**.**

Auto Power On

We want to powering on device when power connected to device, application developers can use SystemManager.setForceAutoStartUpState () API provided by KNOX Customization SDK.

This API requires **com.sec.enterprise.knox.permission.CUSTOM_SYSTEM** permission to be added to application manifest file

```
<uses-permission
android:name="com.sec.enterprise.knox.permission.CUSTOM_SYSTEM"/>
```

To use this APIs, users can add the following code excerpt into the project.

```
public void setForceAutoStartUpState(int state) {
    /*TODO Implement auto on device*/
    CustomDeviceManager cdm = CustomDeviceManager.getInstance();
    SystemManager systemManager = cdm.getSystemManager();
    try {
        int code = systemManager.setForceAutoStartUpState(state);
        if(code == 0) {
            Toast.makeText(mContext,(systemManager.getForceAutoStartUpState() == 1 ?
"Enable":"Disable") + " Set Auto Power On",Toast.LENGTH_LONG).show();
        }
    }catch (NoSuchMethodError e) {
        Log.e(TAG,"NoSuchMethodError:" + e);
    }catch (SecurityException e) {
        Log.e(TAG,"SecurityException:" + e);
    }
}
```

Details for each line in above code excerpt are as the following.

As with the SystemManager API, first is to create CustomDeviceManager instance which can be done by calling CustomDeviceManager.getInstance() API, where this instance will be used in every succeeding calls to KnoxCustomization API. Then we create SystemManager which can be done by calling CustomDeviceManager.getSystemManager()

Then we call SystemManager.setForceAutoStartUpState(int state) , Application developers need to pass parameter integer state to enable or disable auto power on.

Version History

- Version 1.0 – Initial Release, 16 August 2016
- Version 2.0 – Migrate to SDK 3.0, March 2018
- Version 2.1 – Minor changes, March 2018