Samsung Gear Application

Hello Accessory Developer's Guide

Version 1.1

Table of Contents

1. Overview

This document explains how to develop Gear Applications using the Samsung Accessory SDK for a Host-side provider application and the Tizen SDK for Wearable for a Wearable-side consumer widget.

The purpose of this document is to quickly introduce Gear Application Programming to application developers.

This document contains development instructions and code samples. It will show you how to:

- Create Gear Application while referencing sample applications.
- Install an application on a Gear from mobile host devices.

The example will show installing a Gear Application without providing a lot of details. This should illustrate the overall parts of a Gear Application.

Then, you will learn how to make the basic Gear Application.

The following figure shows the installation sequence for Gear Applications.

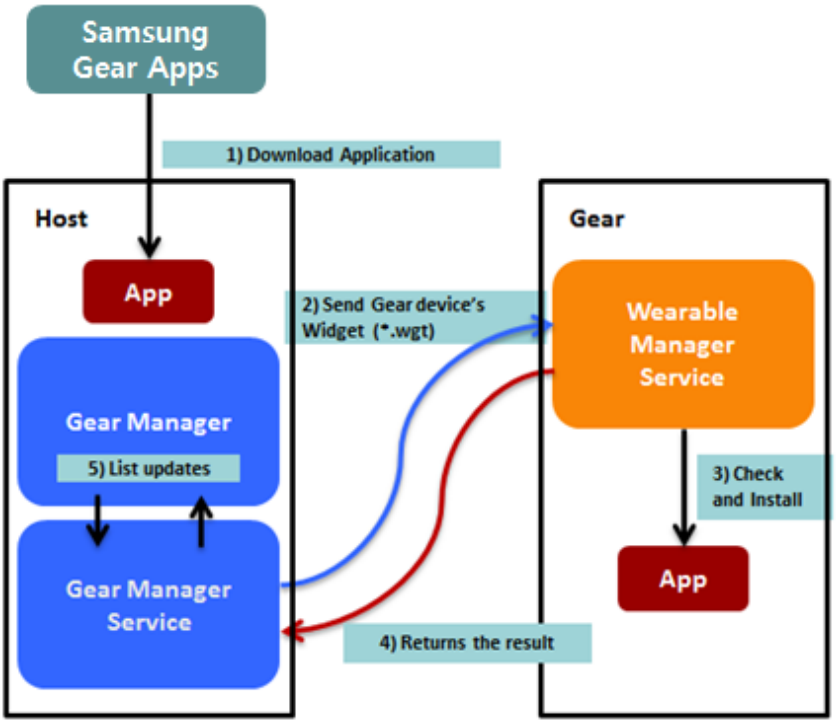

Figure 1: Gear Application installation sequence

1.1. Preconditions

The architecture of a Samsung wearable device is that it works in conjunction with a paired mobile host device (such as a phone or tablet), on which it depends for certain services. The host device runs an application called the Gear Manager, which communicates with the WearableManagerService running on the Gear. Applications intended for Gear has two parts, one of which runs on the host, called the Host-side Application. This installed application will include an application widget intended for the wearable device, called the Wearable-side widget. The Host-side application generally serves a "provider" function, while the Wearable-side widget generally serves a "consumer" function (these functions will be described in more detail later). During the installation of an Host-side application which contains a Wearable-side widget, the Gear Manager sends the Wearable-side widget to the WearableManagerService on the wearable. The WearableManagerService then installs the Wearable-side widget.

In this document, Wearable-side widget is assumed to be a Web application developed using the Tizen SDK for Wearable. The Tizen SDK for Wearable is a Eclipse-based IDE which enables developing Wearable-side widget.

To install a Wearable-side widget, the Gear needs to be connected to the host device through Bluetooth and the Samsung Accessory Framework.

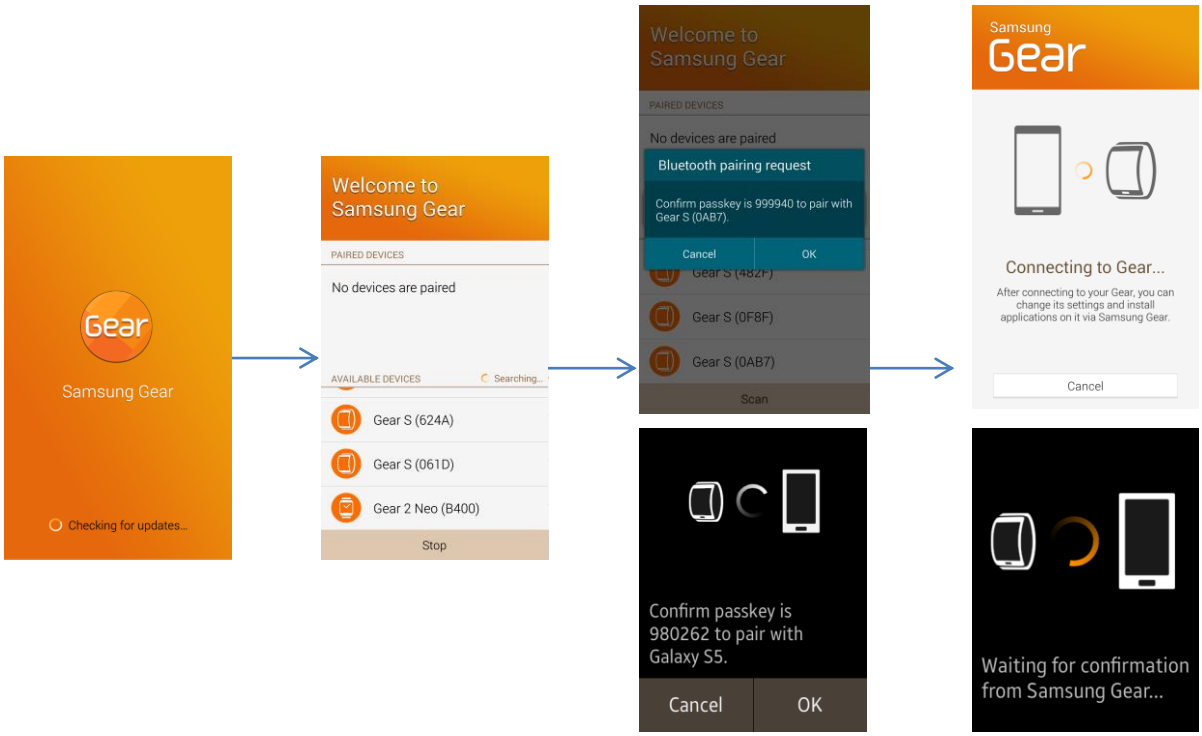

Figure 2: Establishing a Bluetooth connection

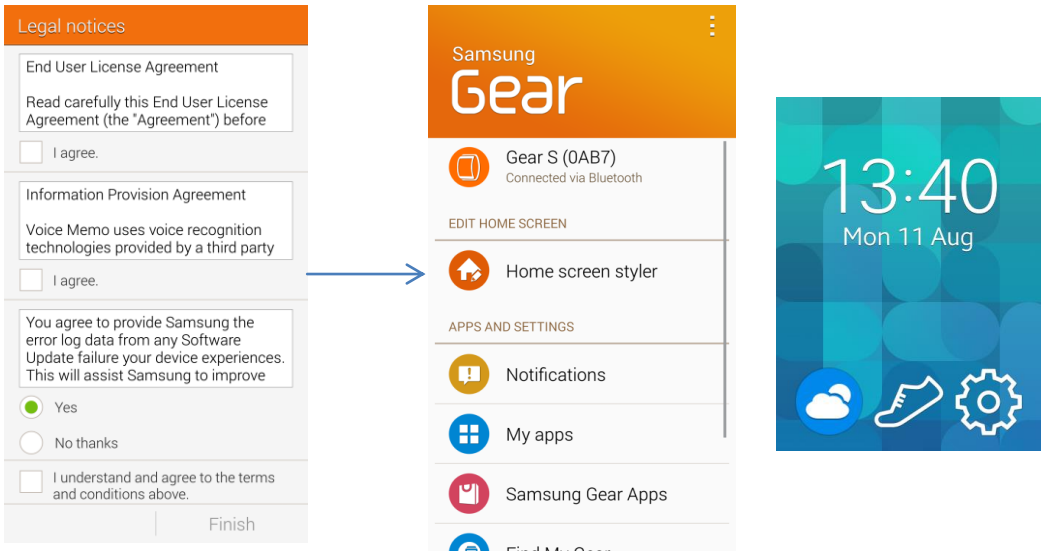

Figure 3: Establishing a connection between host and wearable using Gear Manager

2. Executing HelloAccessory

Hello Gear is a simple application that consists of:

- Host-side application(provider) : HelloAccessoryProvider.apk
- Wearable-side Application(consumer) : HelloAccessoryConsumer.wgt (Web app)

2.1. Importing the HelloAccessoryProvider Project

Import the HelloAccessoryProvider project.

The project contains the following files and components:

- **HelloAccessoryProviderService.java:** The Host-side service that connects with Wearable-side widget.
- **HelloAccessoryConsumer.wgt:** The application installed on the Gear.
- **accessoryservices.xml:** Settings for Host-side service to connect with the Wearable-side service (Samsung Accessory Protocol Service profile).
- **AndroidManifest.xml, layout, values:** Miscellaneous elements.

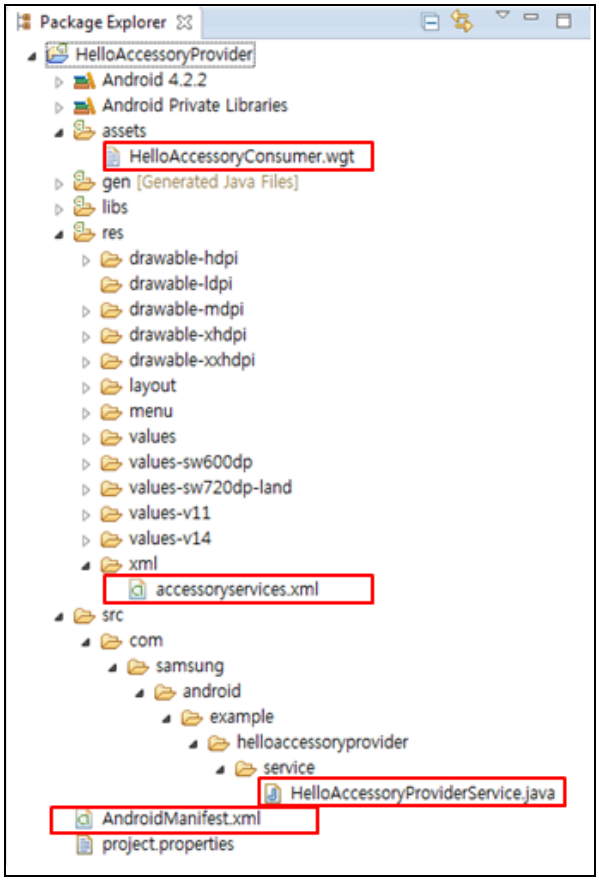

Figure 4: Host-side application Package

2.2. Importing the HelloGearConsumer Project using the Tizen SDK for Wearable

Import the HelloGearConsumer project into the Tizen SDK for Wearable.

The project contains the following files and components:

- **main.js:** The Wearable-side widget logic.
- **accessoryservices.xml**: Settings for the Wearable-side widget to connect with the Host-side service (Samsung Accessory Protocol Service profile).
- **config.xml:** The Wearable-side widget configuration document composed of XML elements**.** Every Wearable-side widget has a config.xml file.
- **HelloAccessoryConsumer.wgt:** The widget application generated by this project.
- **Index.html:** User interface elements for Wearable-side widget.

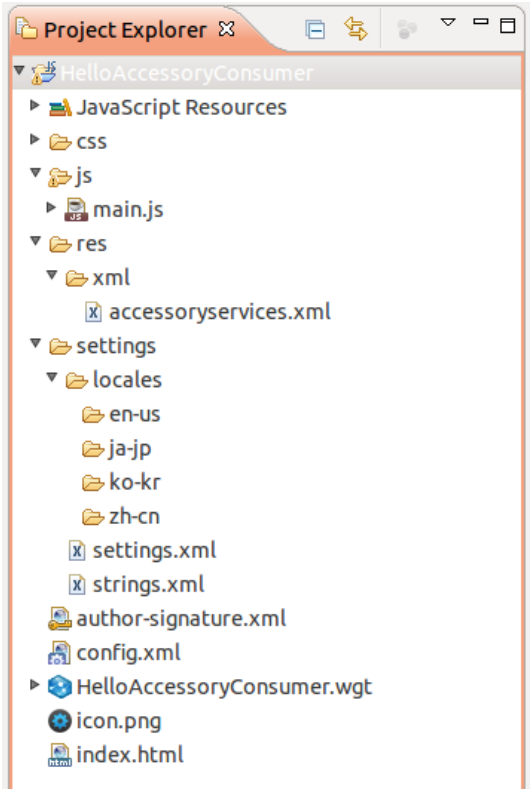

Figure 5: Wearable-side app package

2.3. Generating the HelloAccessoryProvider.apk

Copy the HelloAccessoryConsumer.wgt file to the assets folder of the HelloAccessoryProvider package. Build the HelloAccessoryProvider package to generate the HelloAccessoryProvider.apk file.

2.4. Generating the HelloAccessoryConsumer.wgt

Build the imported HelloAccessoryConsumer package to generate the HelloAccessoryConsumer.wgt file in the project folder in the Tizen SDK for Wearable.

2.5. Installing and Running HelloAccessory

To install and run HelloAccessory:

1. Select the HelloAccessoryProvider.apk file.
2. In the menu, select **Run as Android Application**.
   The HelloAccessoryProvider.apk is installed and run on the Host device.

When the installation is complete, the HostManager sends the HelloAccessoryConsumer.wgt file from the assets folder to the WearableManagerService on the wearable and it installs the HelloAccessoryConsumer.wgt file. (Even if the Host-side application is installed using ADB, the Wearable-side widget is installed automatically after the Host-side application finishes installing properly.)

Note: The Host device must have the Gear Manager installed and must be connected with the Gear through a Bluetooth and Samsung Accessory Protocol(SAP) connection.

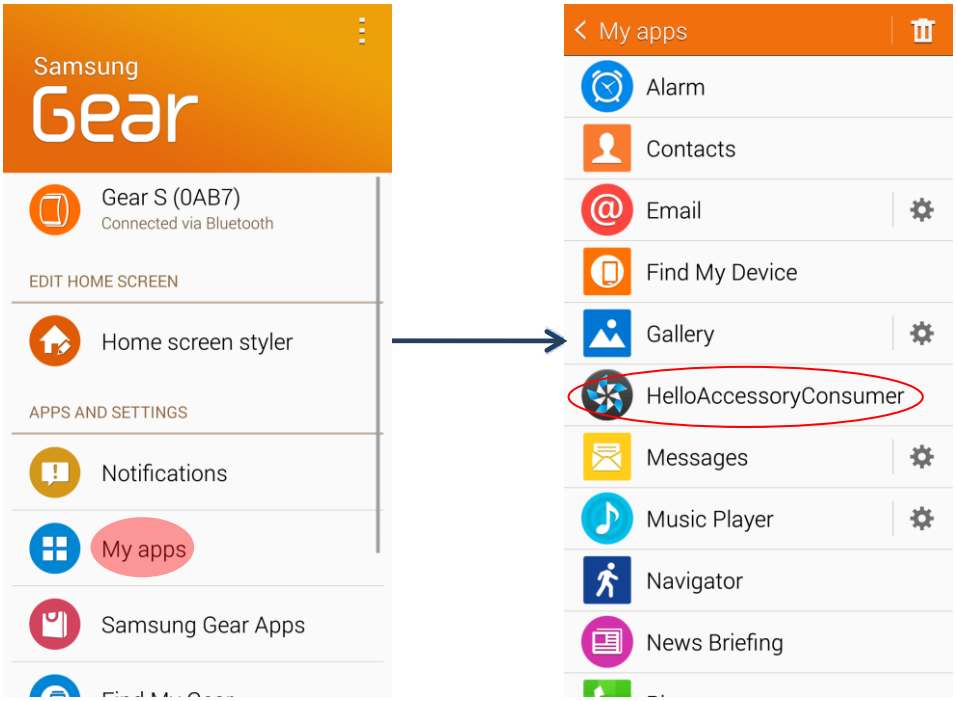

Figure 6: Installing the Gear Application on the device

Figure 7: Using Wearable-side widget

2.6. Uninstalling HelloAccessory

To uninstall the HelloAccessoryConsumer app from the Gear, open the Gear Manager on the host and select **My Apps**.

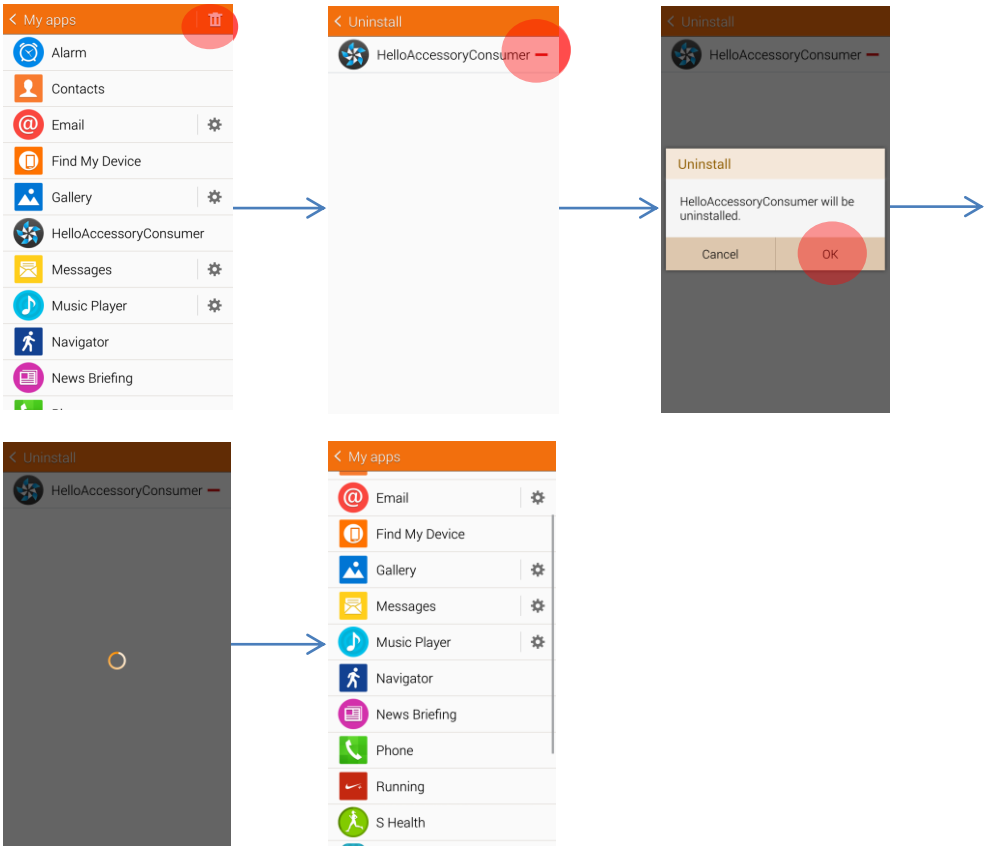

Figure 8: Uninstalling the Wearable-side widget

3. Creating a Gear Application

To create a Gear Application:

1. Create an Android project for the Host-side application and create a Web project using the Tizen SDK for Wearable for the Wearable-side widget.
2. Build the Wearable-side widget to create consumer.
3. Copy the Wearable-side widget (in the Wearable-side project folder) to assets folder (in the Host-side project.)

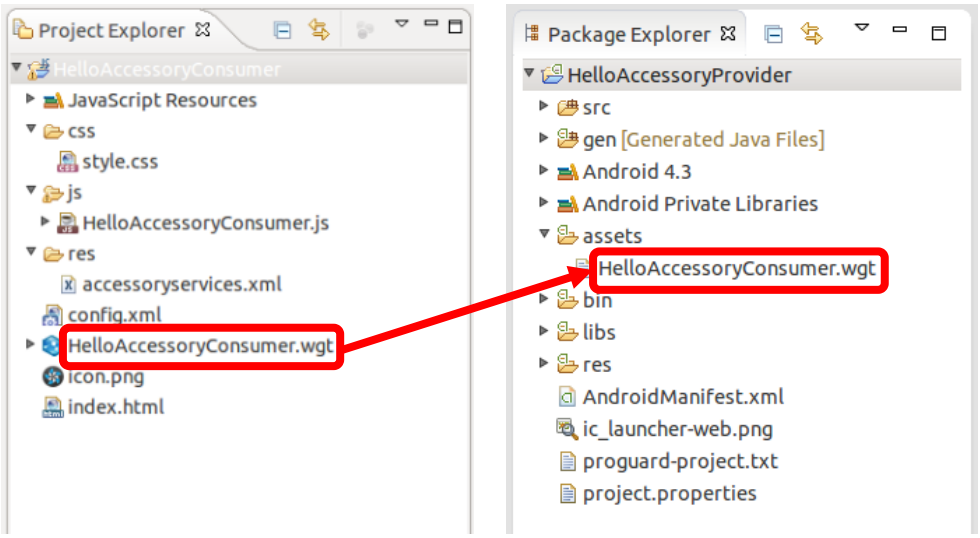

Figure 9: Copying Wearable-side widget to Host-side app

4. Build the Host-side application..
5. Install the Host-side application on the Host device. (Gear Manager should be installed to install Wearable-side widget)
6. After the installation, Gear Manager communicates with WearableManagerService to automatically install the Wearable-side widget on the wearable device.

The Wearable-side widget created in this example is a standalone (non-linked) application. The following chapters describe how to create a more dynamic Wearable-side widget that interacts with its Host-side application through Samsung Accessory Protocol(SAP) connection:

- In Samsung Accessory Framework, you learn how to define the profile of the service in XML to use the provider/consumer services provided by the Samsung Accessory Framework. This procedure is necessary before setting up an Samsung Accessory Protocol(SAP) connection between devices.
- In Creating the HelloAccessoryProvider App and Creating the HelloAccessoryConsumer App, you learn how to add a provider service to a Host-side application and a consumer service to a Wearable-side widget.

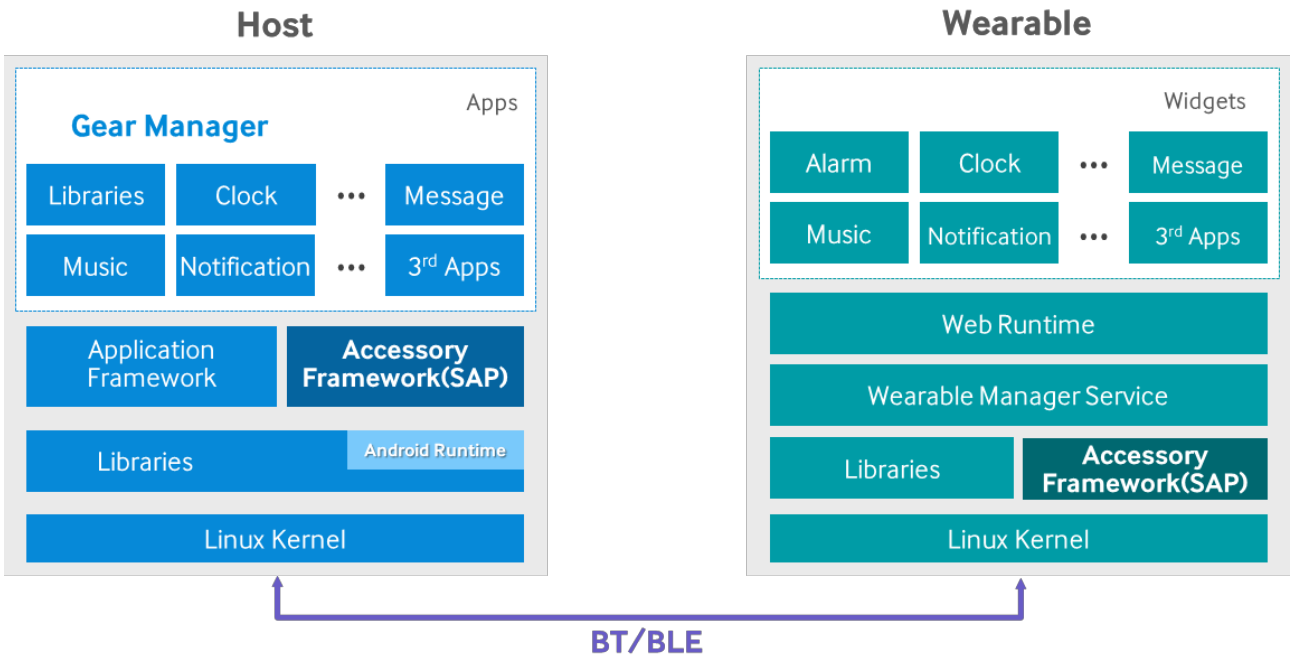

Figure 10: Architecture of the connection between a Host-side application and a Wearable-side widget

4. Introduce Samsung Accessory Framework

The communication between a Host-side application and a Wearable-side widget requires a provider/consumer service. The Samsung Accessory Framework establishes a logical connection between the applications across the network layers.

If the predefined profile of a consumer service is implemented on a Wearable-side widget and a provider service of the same profile is implemented on a counterpart Host-side application, the Samsung Accessory Framework can create a connection between the Wearable-side widget and the Host-side application to communicate through it. Note that the two services must share the same profile.

4.1. SAP, Samsung Accessory SDK (For Android Platform)

4.1.1. Configuring the Android Manifest File for Host-side application

To use the Samsung Accessory Framework in Wearable-side widget:

1. Add permission for your Service Provider application in the Android manifest file.

```
<uses-permission android:name="com.samsung.accessory.permission.ACCESSORY_FRAMEWORK" />
```

2. Declare the Service Provider class that you derived from the SAAgent as a service in the Android manifest file. The SAAgent class extends the Android service and handles asynchronous accessory-related intents. The SAAgent implementation executes all of its activities in a worker thread, which means it does not overload your application's main thread.

```
<service android:name="com.samsung.android.example.hellogearprovider.service.
                       HelloGearProviderService" >
```

3. Add your Service Provider broadcast receivers for the intents handled by the Gear in the Android manifest file inside the <application> element.

```
<receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver" >
    <intent-filter>
        <action android:name="android.accessory.device.action.REGISTER_AFTER_INSTALL" />
    </intent-filter>
</receiver>
<receiver android:name="com.samsung.android.sdk.accessory.ServiceConnectionIndicationBroadcastReceiver" >
    <intent-filter>
        <action android:name="android.accessory.service.action.ACCESSORY_SERVICE_CONNECTION_IND" />
    </intent-filter>
</receiver>
```

4. Add the Service Profile information in an XML file. Declare the path of your Service Profile XML file in the Android manifest file. For example, /res/xml/<yourname>.xml.

```
<meta-data
            android:name="AccessoryServicesLocation"
            android:value="/res/xml/accessoryservices.xml" />
```

4.1.2. Registering the Service Provider

Register your Service Provider by specifying the Accessory Service Profile description with the Accessory Service Framework. The Accessory Service Framework enters the description in the local Capability Database. The Accessory Capability Exchange module advertises the services registered to the connected Samsung Accessory Devices.

The registration process expects the Accessory Service Profile description in the Accessory Service XML file to be located in the /res/xml folder of your Android project.

Note: The following code is just an example of how to make an Accessory XML file. There is no actual implementation of such an application.

Provider - accessoryservices.xml

```
<resources>
    <application name="HelloGearProvider" >
        <serviceProfile
            id="/system/hellogear"
            name="smartview"
            role="provider"
            serviceImpl="com.example.sec.android.app.hellogearprovider.backend.HelloGearProviderImpl"
            version="1.0" >
            <supportedTransports>
                <transport type="TRANSPORT_BT" />
            </supportedTransports>

            <serviceChannel
                id="104"
                dataRate="low"
                priority="low"
                Reliability="DISABLE" >
            </serviceChannel>
        </serviceProfile>
    </application>
</resources>
```

Set the <application> element's "name" attribute to your application name to allow the Samsung Accessory Framework to advertise it in the Accessory eco-system. The name is usually the same as your application's Android AppName. If your application implements multiple Service Providers, declare multiple <serviceProfile> elements inside the <application> element.

In each <serviceProfile> element:

- The "serviceImpl" attribute is your subclass that extends the SAAgent.
- The "role" attribute should be "provider".
- The "name" attribute is a user-friendly name of your Service Provider.
- The "id" attribute is the Service Profile identifier of the Service Provider.
- The "version" attribute specifies the Service Profile specification version that your Service Provider application supports.

In the <supportedTransports> element, declare the transports the Service Provider operates on. Currently, the Samsung Accessory Framework supports TRANSPORT_WIFI, TRANSPORT_BT, TRANSPORT_BLE, and

TRANSPORT_USB. If your Service Provider supports multiple transports, declare multiple <transport> elements. In the above example, the Service Provider supports onlyBluetooth.

In each <serviceChannel> element:

- dataRate is either "low" or "high".
- priority is "low", "medium" or "high".
- reliability "ENABLE" or "DISABLE". In case of packet drop, reliable transfer will re-transmit the packet, but incurs additional overhead as a result.

The Samsung Accessory Framework automatically registers Accessory Peer Agents during application installation and automatically deregisters them during uninstallation. The framework creates an error log if the Accessory Service Profile implementation registration process fails. Document Type Definition (DTD) schema validation significantly lowers the chances of registration failure.

```
<!DOCTYPE resources [
<!ELEMENT resources (application)>
<!ELEMENT application (serviceProfile)+>
<!ELEMENT supportedTransports (transport)+>
<!ELEMENT serviceProfile (supportedTransports, serviceChannel+) >
<!ELEMENT transport EMPTY>
<!ELEMENT serviceChannel EMPTY>
<!ATTLIST application name CDATA #REQUIRED>
<!ATTLIST application xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceProfile xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceProfile serviceImpl CDATA #REQUIRED>
<!ATTLIST serviceProfile role (PROVIDER | CONSUMER |provider | consumer) #REQUIRED>
<!ATTLIST serviceProfile name CDATA #REQUIRED>
<!ATTLIST serviceProfile id CDATA #REQUIRED>
<!ATTLIST serviceProfile version CDATA #REQUIRED>
<!ATTLIST serviceProfile serviceLimit (ANY | ONE_ACCESSORY | ONE_PEERAGENT | any | one_accessory |
one_peeragent) #IMPLIED>
<!ATTLIST serviceProfile serviceTimeout CDATA #IMPLIED>
<!ATTLIST supportedTransports xmlns:android CDATA #IMPLIED>
<!ATTLIST transport xmlns:android CDATA #IMPLIED>
<!ATTLIST transport type (TRANSPORT_WIFI|TRANSPORT_BT|TRANSPORT_BLE|TRANSPORT_USB
                          |transport_wifi|transport_bt|transport_ble|transport_usb) #REQUIRED>
<!ATTLIST serviceChannel xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceChannel id CDATA #REQUIRED>
<!ATTLIST serviceChannel dataRate (LOW | HIGH | low | high) #REQUIRED>
<!ATTLIST serviceChannel priority (LOW | MEDIUM | HIGH | low | medium | high) #REQUIRED>
<!ATTLIST serviceChannel reliability (ENABLE | DISABLE | enable | disable) #REQUIRED>
]>
```

Include the DTD validation rules at the very top of your Accessory XML file:

1. In the Samsung Accessory SDK, go to **Window > Preferences > XML > XML Files > Validation**, and select **Enable markup validation**.
2. In the **No grammar specified** box and the **Missing root element** box, select **Ignore**.

When the validation rules are included, the Samsung Accessory SDK validates the Accessory XML file when you build your application to check whether the Accessory XML follows the rules required by the Samsung Accessory Framework. You can also validate the XML at any time by right-clicking on the XML file and selecting **Validate**.

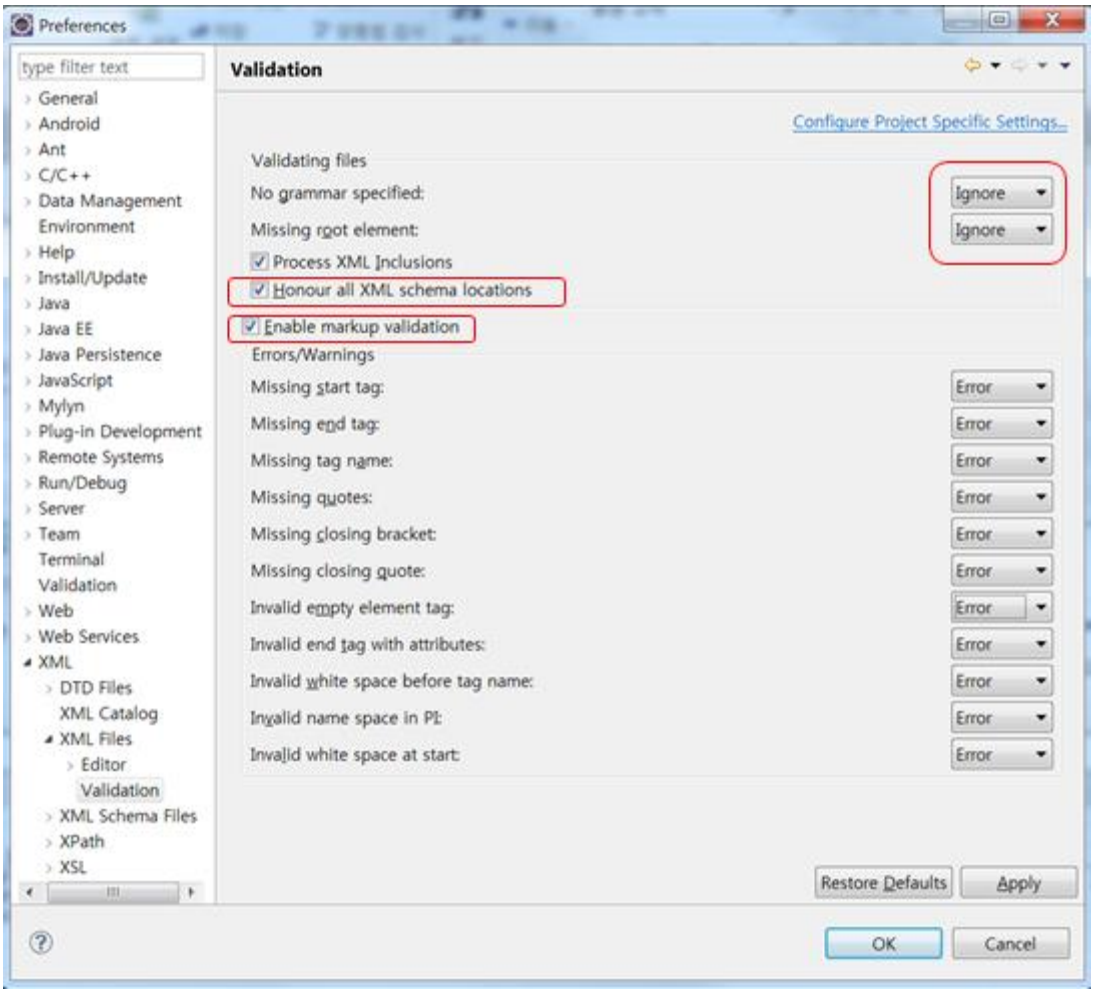

Figure 11: Samsung Accessory SDK XML validation settings

4.2. SAP in Tizen SDK for Wearable

- You can get the SAP API information in the help contents of Tizen SDK for Wearable:

  API Spec: Web App Programming → API References → Device API Reference → Communication →SAP

  Tutorial: Web App Programming → Tutorials → Device API Tutorials → Communication Tutorials →SAP Tutorial

4.2.1. Configuring the config.xml File for Wearable-side widget

To use the Samsung Accessory Framework in Wearable-side widget:

Add the Service Profile information in an XML file. Declare the path of your Service Profile XML file in the config.xml file. For example, res/<yourname>.xml

```
<tizen:metadata key="AccessoryServicesLocation" value="res/accessoryservices.xml" />
```

4.2.2. Registering the Service Consumer

Register your Service Consumer by specifying the Accessory Service Profile description with the Samsung Accessory Framework. The Samsung Accessory Framework enters the description in the local Capability Database. The Accessory Capability Exchange module advertises the services registered to the connected Samsung Accessory Devices.

The registration process expects the Accessory Service Profile description in the Accessory Service XML file declared in the config.xml file associated with AccessoryServicesLocation metadata key.

Note: The following code is just an example of how to make an Accessory XML file. There is no actual implementation of such an application.

Consumer - accessoryservices.xml

```
<resources>
    <application name="HelloAccessoryConsumer" >
        <serviceProfile
            id="/system/hellogear"
            name="smartview"
            role="consumer"
            version="1.0" >
            <supportedTransports>
                <transport type="TRANSPORT_BT" />
            </supportedTransports>
            <serviceChannel
                id="104"
                dataRate="low"
                priority="low"
                Reliability="DISABLE" >
            </serviceChannel>
        </serviceProfile>
    </application>
</resources>
```

Set the <application> element's "name" attribute to your application name to allow the Samsung Accessory Framework to advertise it in the Accessory eco-system. The name is usually the same as your application's AppName. If your application implements multiple Service Consumers, declare multiple <serviceProfile> elements inside the <application> element.

In each <serviceProfile> element:

- The "role" attribute should be "consumer".
- The "name" attribute is a user-friendly name of your Service Consumer.
- The "id" attribute is the Service Profile identifier of the Service Consumer.
- The "version" attribute specifies the Service Profile specification version that your Service Consumer application supports.

In the <supportedTransports> element, declare the transports the Service Consumer operates on. Currently, the Samsung Accessory Framework supports TRANSPORT_WIFI, TRANSPORT_BT, TRANSPORT_BLE, and

TRANSPORT_USB. If your Service Consumer supports multiple transports, declare multiple <transport> elements. In the above example, the Service Consumer supports only Bluetooth.

In each <serviceChannel> element:

- dataRate is either "low" or "high".
- priority is "low", "medium" or "high".
- reliability "ENABLE" or "DISABLE". In case of packet drop, reliable transfer will re-transmit the packet, but incurs additional overhead as a result.

The Samsung Accessory Framework automatically registers Accessory Peer Agents during application installation and automatically deregisters them during uninstallation. The framework creates an error log if the Accessory Service Profile implementation registration process fails. Document Type Definition (DTD) schema validation significantly lowers the chances of registration failure.

```
<!DOCTYPE resources [
   <!ELEMENT resources (application)>
   <!ELEMENT application (serviceProfile)+>
   <!ELEMENT serviceProfile (supportedTransports, serviceChannel+) >
   <!ELEMENT supportedTransports (transport)+>
   <!ELEMENT transport EMPTY>
   <!ELEMENT serviceChannel EMPTY>
   <!ATTLIST application name CDATA #REQUIRED>
   <!ATTLIST serviceProfile role (PROVIDER | CONSUMER | provider | consumer) #REQUIRED>
   <!ATTLIST serviceProfile name CDATA #REQUIRED>
   <!ATTLIST serviceProfile id CDATA #REQUIRED>
   <!ATTLIST serviceProfile version CDATA #REQUIRED>
   <!ATTLIST serviceProfile autoLaunchAppId CDATA #IMPLIED>
   <!ATTLIST serviceProfile serviceLimit (ANY | ONE_ACCESSORY | ONE_PEERAGENT |
                                          any | one_accessory | one_peeragent) #IMPLIED>
   <!ATTLIST serviceProfile serviceTimeout CDATA #IMPLIED>
   <!ATTLIST transport type (TRANSPORT_WIFI | TRANSPORT_BT | TRANSPORT_BLE |
                             TRANSPORT_USB | transport_wifi | transport_bt | transport_ble |
                             transport_usb) #REQUIRED>
   <!ATTLIST serviceChannel id CDATA #REQUIRED>
   <!ATTLIST serviceChannel dataRate (LOW | HIGH | low | high) #REQUIRED>
   <!ATTLIST serviceChannel priority (LOW | MEDIUM | HIGH | low | medium | high) #REQUIRED>
   <!ATTLIST serviceChannel reliability (ENABLE | DISABLE | enable | disable) #REQUIRED>
]>
```

When the validation rules are included, the Tizen SDK for Wearablevalidates the Accessory XML file when you build your application to check whether the Accessory XML follows the rules required by the Samsung Accessory Framework.

5. Creating the HelloAccessoryProvider App

This chapter describes how to implement a service provider and complete a HelloGearProvider application. For more information on the Samsung Accessory Framework and creating a basic Host-side application, see Samsung Accessory Framework and HelloAccessory.

5.1. The Android Manifest File

Declare the permission, service (Provider service), receiver and intent-filter, and meta-data (Service Profile) in the Android manifest file before creating an application.

Permissions

- BLUETOOTH, BLUETOOTH_ADMIN: Permission for the Bluetooth connections
- ACCESSORY_FRAMEWORK: Permission for the Samsung Accessory Protocol
- wmanager.APP: Permission for the Wearable application
- wmanager.ENABLE_NOTIFICATION: Permission for the HostManager notifications

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="com.samsung.accessory.permission.ACCESSORY_FRAMEWORK" />
<uses-permission android:name="com.samsung.wmanager.APP"/>
<uses-permission android:name="com.samsung.wmanager.ENABLE_NOTIFICATION"/>
```

Service (Provider service)

Add the service to the Android manifest file to implement your provider (providers must be implemented as a service).

```
<service android:name="com.samsung.android.example.hellogearprovider.service.HelloGearProviderService" >
</service>
```

Broadcast receiver and intent-filter

Add the receiver to the Android manifest file to make sure the provider works properly.

```
<receiver android:name="com.samsung.android.sdk.accessory.ServiceConnectionIndicationBroadcastReceiver" >
    <intent-filter>
        <action android:name="android.accessory.service.action.ACCESSORY_SERVICE_CONNECTION_IND" />
    </intent-filter>
</receiver>
<receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver" >
    <intent-filter>
        <action android:name="android.accessory.device.action.REGISTER_AFTER_INSTALL" />
    </intent-filter>
</receiver>
```

Meta-data

Declare the provider/consumer service profile as shown below to use the Samsung Accessory Protocol(SAP) connection between the Host-side application and the Wearable-side widget.

For more information on the XML file related to the profile, see Registering the Service Provider.

```
<meta-data
            android:name="AccessoryServicesLocation"
            android:value="/res/xml/accessoryservices.xml" />
```

For linked type (Master-Follower) application, add the meta-data information for the master application to the Android manifest file of your Host-side application within the <application> tag. This information is used to notify users when the master application does not exist or is not installed by WearableManagerService:

- For “master_app_name”, enter the name of the master application.
- For “master_app_packagename”, enter the package name of the master application.
- For “master_app_samsungapps_deeplink” and “master_app_playstore_deeplink”, enter the market deep link to the master application. (This is optional. You can insert one or more links.)

  ※ In order to upload your Host applications on Samsung GALAXY Apps, you have to develop the apps using one or more of Samsung SDKs.

```
<meta-data
    android:name="master_app_name"
    android:value="master App" />
<meta-data
    android:name="master_app_packagename"
    android:value="com.example.masterapp" />
<meta-data
    android:name="master_app_samsungapps_deeplink"
    android:value="Samsungapps deeplink URL" />
<meta-data
    android:name="master_app_playstore_deeplink"
    android:value="playstore deeplink URL" />
```

5.2. Application Types

The following figure shows the application package structure required by Gear Manager.

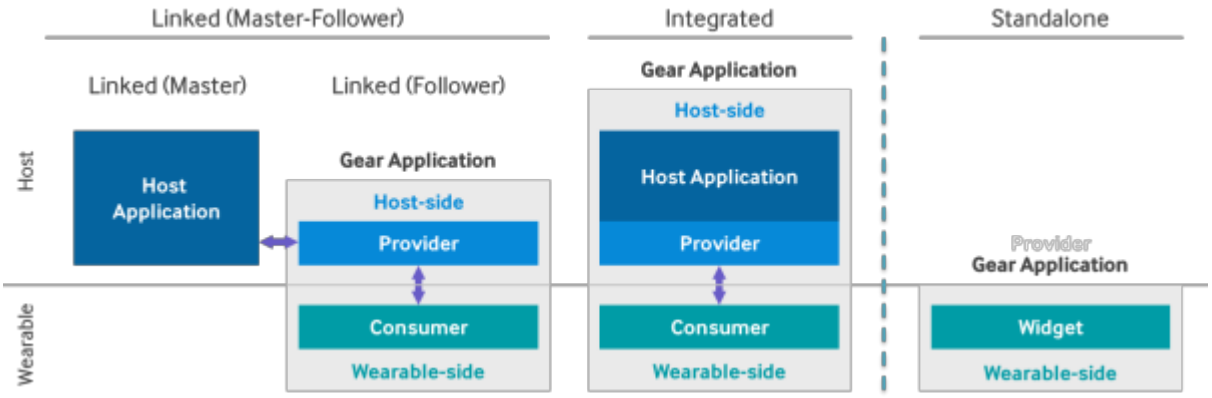

Figure 12: Application package structure

5.2.1. Linked (Master-Follower)

The Linked Type consists of the Host-side application and the Wearable-side application.

The Wearable-side application is not included with the general host application. You need to install the general host application and the Wearable-side application.

To enable host devices to recognize Linked applications and store recommendations, add the following permission to Android manifest file in the Host-side application (AndroidManifest.xml).

```
<uses-permission android:name=”com.samsung.WATCH_APP_TYPE.Linked_Follower”/>
```

To upload Gear applications to the Samsung Gear Apps Store, add the following metadata to the AndroidManifest.xml for the application that works with your wearable-side widget.

```
<meta-data android:name="GearAppType" android:value="wgt"/>
```

the version codes of Gear application must have different numbers.
If you try to register Gear applications with the same version code, the Samsung Gear Apps Store cannot register your new application.

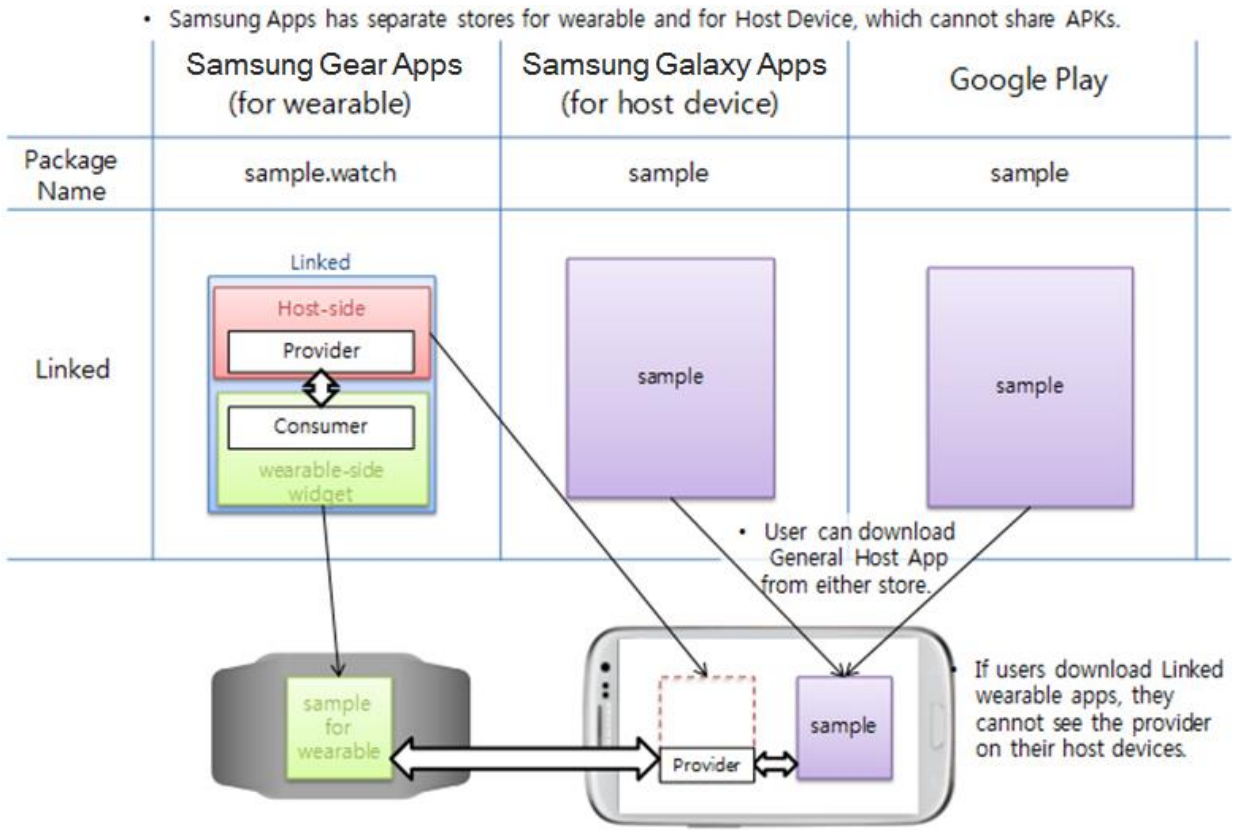

Figure 13: Linked Gear Application Installation

5.2.2. Integrated

Integrated applications consist of a host-side APK and a wearable-side widget.

The host-side APK contains the general host app. When users download an Integrated application, the general host app in the host-side APK is installed on Host devices

Integrated applications operate in the same manner as Linked applications, but they only have one (1) APK instead of the two (2) APKs of Linked applications. Integrated Gear package names must be different than those used for general host apps. If you use the same names, there can be problems when users update Integrated applications.

To enable host devices to recognize Integrated applications and store recommendations, add the following permission to your general host Android manifest file (AndroidManifest.xml).

```
<uses-permission android:name="com.samsung.WATCH_APP_TYPE.Integrated"/>
```

To upload Gear applications to Samsung Gear Apps Store, add the following metadata to the AndroidManifest.xml for the application that works with your wearable-side widget

```
<meta-data android:name="GearAppType" android:value="wgt"/>
```

the version codes of Gear application must have different numbers.
If you try to register Gear applications with the same version code, the Samsung Gear Apps Store cannot register your new application.

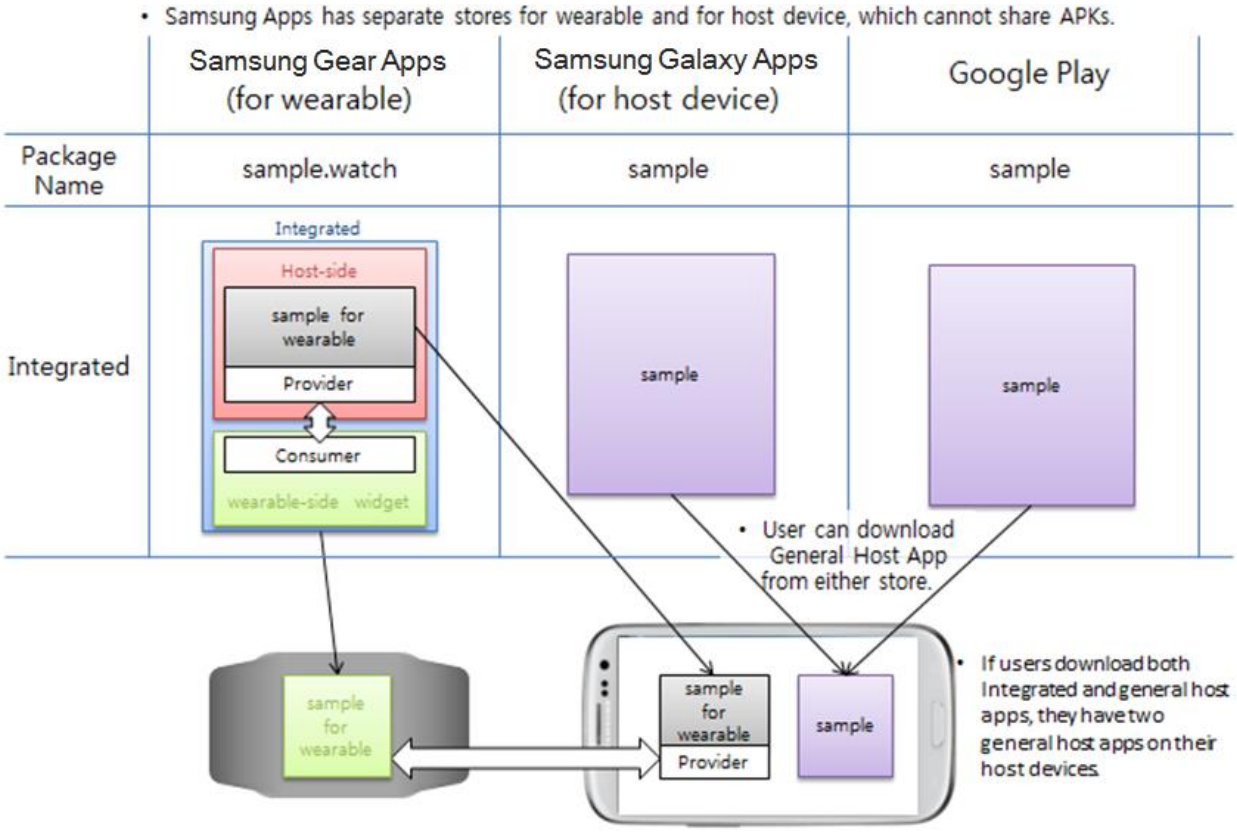

Figure 14: Integrated Gear application Installation

5.2.3. Standalone

You can use the standalone type, such as a Clock app, when your Gear application does not need a general host application, and can operate independently.

5.3. The Provider Service Profile

In the Android manifest file, declare the service profile for the Samsung Accessory Protocol(SAP) connection.

In the service profile the ID need not match the name; but the ID and the name of a service must match the corresponding values of its peer service to establish a connection with it. The ID and name values are case-sensitive.

To send data through multiple channels within a connection, add more service channels to the service profile.

The main settings for a service profile are:

- **role**: The service role. Set it to "provider" (case-sensitive).
- **id**: The service ID. Enter the same ID entered for the consumer (case-sensitive).
- **name**: The service name. Enter the same name entered for the consumer (case-sensitive).
- **serviceImpl**: The implementation class for the service agent.
- <transport> element **type**: The data transfer type. Only TRANSPORT_BT is supported.
- <serviceChannel> element **id**: The service channel ID. An ID must be unique within a service profile. The ID is used for data transfer after a connection is built.

res/xml/accessoryservices.xml

```
<resources>

    <application name="HelloGearProvider" >
        <serviceProfile
            id="/system/hellogear"
            name="smartview"
            role="provider"
            serviceImpl="com.example.sec.android.app.hellogearprovider.backend.HelloGearProviderImpl"
            version="1.0" >
            <supportedTransports>
                <transport type="TRANSPORT_BT" />
            </supportedTransports>

            <serviceChannel
                id="104"
                dataRate="low"
                priority="low"
 reliability="enable">
            </serviceChannel>
        </serviceProfile>
    </application>

</resources>
```

5.4. Creating the Provider Activity

The HelloGearProvider application of Hello Gear does not contain an activity. If Host-side application require UI(User Interface), you can add an activity to a provider application.

5.5. Starting the Provider Service

The Samsung Accessory Framework creates a peer agent list based on the manifest and the provider service profile of a provider application. When a Samsung Accessory Protocol(SAP) connection is set up, it can start a provider service through broadcasting. The provider service can receive an ACCESSORY_SERVICE_CONNECTION_IND broadcast from the Samsung Accessory Framework with an

ACCESSORY_SERVICE_CONNECTION_IND permission. This means that the Samsung Accessory Protocol(SAP) connection starts the provider service to the consumer service.

5.6. Initiating the Service Connection

When the Samsung Accessory Framework receives a connection request from the consumer service that matches the provider service, the Samsung Accessory Framework calls `onServiceConnectionRequested()` to send an SAAgent (matching consumer service) object to the provider service.

The provider service uses the following 2 methods in the method, `onServiceConnectionRequested()` to accept or reject a connection request. If a request is neither accepted nor rejected explicitly in `onServiceConnectionRequested()`, it is accepted by default.

- `acceptServiceConnectionRequest()`: Accepts a connection request.
- `rejectServiceConnectionRequest()`: Rejects a connection request.

If a requesting is accepted, the Samsung Accessory Framework calls `onServiceConnectionResponse()` to send the SASocket object to the provider service.

Through the received socket, the provider service exchanges data with the consumer service. The following sample code illustrates how to manage SASocket objects with the SASocket management map.

If a previous connection exists, you can use the existing SASocket object to connect.

HelloAccessoryProviderService.java

```
@Override
protected void onServiceConnectionResponse(SASocket thisConnection, int result) {
    if (result == CONNECTION_SUCCESS) {
        if (thisConnection != null) {
            HelloGearProviderConnection myConnection = (HelloGearProviderConnection) thisConnection;
            if (mConnectionsMap == null) {
                mConnectionsMap = new HashMap<Integer, HelloGearProviderConnection>();
            }
            myConnection.mConnectionId = (int) (System.currentTimeMillis() & 255);
            Log.d(TAG, "onServiceConnection connectionID = " + myConnection.mConnectionId);
            mConnectionsMap.put(myConnection.mConnectionId, myConnection);

            Toast.makeText(getBaseContext(), R.string.ConnectionEstablishedMsg, Toast.LENGTH_LONG).show();
        } else
            Log.e(TAG, "SASocket object is null");
    } else if (result == CONNECTION_ALREADY_EXIST) {
        Log.e(TAG, "onServiceConnectionResponse, CONNECTION_ALREADY_EXIST");
    } else {
        Log.e(TAG, "onServiceConnectionResponse result error =" + result);
    }
}
```

5.7. Using the Service Connection

If a service connection is established, you can exchange data with the SASocket object received from the Samsung Accessory Framework.

The SASocket class provides the following main methods:

- `onError()`: Called when a connection error occurs.
- `onReceive()`: Called when data is received through the channel.
- `send() / secureSend()`: Sends data through the channel sent as the parameter. In the code sample below, `onReceive()` sends the message from the consumer back to the consumer using the `send()` or secureSend() method.
- `onServiceConnectionLost()`: Called when a connection is lost.

HelloGearProviderService.java

```
public class HelloGearProviderConnection extends SASocket {
    private int mConnectionId;

    public HelloGearProviderConnection() {
        super(HelloGearProviderConnection.class.getName());
    }

    @Override
    public void onError(int channelId, String errorString, int error) {
        Log.e(TAG, "Connection is not alive ERROR: " + errorString + "  " + error);
    }

    @Override
    public void onReceive(int channelId, byte[] data) {
        String strToUpdateUI = new String(data);
        Log.d(TAG, "onReceive");
        HelloGearProviderConnection uHandler =
                                    mConnectionsMap.get(Integer.parseInt(String.valueOf(mConnectionId)));
        try {
            uHandler.send(HELLOB_CHANNEL_ID, strToUpdateUI.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onServiceConnectionLost(int errorCode) {
        Log.e(TAG, "onServiceConectionLost  for peer = " + mConnectionId + "error code =" + errorCode);
        if (mConnectionsMap != null) {
            mConnectionsMap.remove(mConnectionId);
        }
    }
}
```

6. Creating the HelloAccessoryConsumer App

This chapter describes how to implement a HelloAccessoryConsumer application using Tizen SDK for Wearable.

With the Tizen SDK for Wearable, you can develop rich consumer Web applications and build great application experiences with well-known Web programming languages: HTML, CSS, and JavaScript. The Gear Web APIs support the latest HTML5 capabilities, which can make your application run across various devices with minimal customization. With the Web Device API, you can also enable advanced device access from Wearable-side widgets. The API references are available in Tizen SDK for Wearable(**Help → Help Contents** → Tizen Wearable**Web App Programming → API References**) as well as Programming Guide and Tutorials for beginners.

Note that this document does not cover general explanation of how to develop Gear Web Application

6.1. The config.xml File

Declare the meta-data (Service Profile) in the Wearable-side widget configuration document (config.xml). The general form of a configuration document for a web application may be found in the W3C Recommendation: Packaged Web Apps (widgets) – Packaging and XML Configuration (http://www.w3.org/TR/widgets).

Meta-data (service profile)

Declare a consumer service profile by creating an XML file in your project, as described in the next section. Pick a location in the project, then add a declaration showing where to find the file to your config.xml. In this example, the file is accessoryservices.xml and it has been placed in the res directory, both of which must exist.

```
<tizen:metadata key="AccessoryServicesLocation" value="res/accessoryservices.xml"/>
```

Privilege

Declare a required privilege

```
<tizen:privilege name="http://developer.samsung.com/privilege/accessoryprotocol"/>
```

Watch Clock Vs. Watch Widget

There are two types of applications you can create for the Gear device: Watch Clock and Watch Widget

Watch Clock is a clock widget always running on the clock screen. Meanwhile, Watch Widget is a normal web application, which you need to launch from the application menu.

If you create a Watch Clock app, add the following line in config.xml file.

```
<tizen:category name="com.samsung.wmanager.WATCH_CLOCK" />
```

If you create a Watch Widget app, you don’t need to declare a category (default type is watch widget).

6.2. The Consumer Service Profile

Declare the service profile for the Samsung Accessory Protocol(SAP) connection in XML.

In the service profile, the ID need not match the name; but the ID and the name of a service must match the corresponding values of its peer service to establish a connection with it. The ID and name are case-sensitive.

To send data through multiple channels within a connection, add more service channels in the service profile.

The main settings for a service profile are:

- **role**: The service role. Set it to "consumer" (case-sensitive).
- **id**: The service ID. Enter the same ID entered for the provider (case-sensitive).
- **name**: The service name of the service. Enter the same name entered for the provider (case-sensitive).
- **autoLaunchAppId**: The application ID that should be invoked by incoming service connection request. If a Peer Agent requests a service connection to this service profile while it is not running, the system launchs this application. Unless this value is specified, the system never launches this application even if the Peer Agent requests a service connection.
- <transport> element **type**: The data transfer type. Only TRANSPORT_BT is supported.
- <serviceChannel> element **id**: The service channel ID. An ID must be unique within a service profile. The ID is used for data transfer after a connection is built.

res/accessoryservices.xml

```
<resources>
    <application name="HelloAccessoryConsumer" >
        <serviceProfile
            id="/system/hellogear"
            name="smartview"
            role="consumer"
            version="2.0"
             autoLanchAppId="VcGvmQlo1.HelloAccessory”>
            <supportedTransports>
                <transport type="TRANSPORT_BT" />
            </supportedTransports>
            <serviceChannel
                id="104"
                dataRate="low"
                priority="low"
                reliability="DISABLE" >
            </serviceChannel>
        </serviceProfile>
```

```
    </application>
</resources>
```

6.3. Defining the Application Layout

In this example, the start file (index.html) contains application UI elements. With the Tizen SDK for Wearable , you can develop rich Web applications using standard web technologies (HTML5/CSS/Javascript) as well as Web Device APIs. Please refer to the API specifications in the Help Contents menu in the Tizen SDK for Wearable .

6.4. Communicating between Host-side application and Wearable-side widget using Samsung Accessory Protocol (SAP)

To send data to peer agents, the consumer must be connected with the provider service over Samsung Accessory Protocol(SAP) connection (**API References → Device API Reference → Communication → SAP**). The main features of the SAP API include:

- **Setting up a connection :**

  You can establish a connection between Host-side application and Wearable-side widget.

- **Simple data (string) exchange :**

  You can exchange string messages between Host-side application and Wearable-side widget.

- **File exchange:**

  You can exchange files between Host-side application and Wearable-side widget

- **Closing the connection:**

  You can close the connection between Host-side application and Wearable-side widget.

6.4.1. Setting up a Connection

Learning how to request SAAgent and connect peer agents is a basic Samsung Accessory Framework management skill:

1. To get the SAAgent specified in an accessory service profile, use the requestSAAgent() method:

```
var SAAgent;
function onsuccess(agents) {
        SAAgent = agents[0];
        for(var i = 0; i < agents.length; i++) {
                console.log(i + ". " + agents[i].name );
                /* Process the SA Agents */
        }
}
function onerror(e) {
        console.log("Error name: " + e.name + ", Error message : " + e.message);
}
try {
    webapis.sa.requestSAAgent(onsuccess, onerror);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

2. Define the handler for peer device status notifications using the SADeviceStatusCallback listener interface, and set the listener to the SAAgent. The status handler informs you when the Gear device disconnects from the host (DETACHED) or connects to it (ATTACHED). In the ATTACHED case, you need to call findPeerAgents() to re-establish an Samsung Accessory Protocol(SAP) connection. See step 4 for more details:

```
function ondevicestatus(type, status) {
    if (status == "ATTACHED") {
            console.log("Attached remote peer device. : " + type);
            SAAgent.findPeerAgents();    } else if (status == "DETACHED") {
            console.log("Detached remote peer device. : " + type);
        }
}

try {
    webapis.sa.setDeviceStatusListener(ondevicestatus);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

3. Define the handler for connection event notifications using the ServiceConnectionCallback listener interface, and add a listener with the defined handler. In case of the autoLaunchAppId attribute of service profile xml, if a peer agent requests a service connection to the service profile while it is not running, the system launches your application and calls the onrequest() event handler of the ServiceConnectionCallback listener interface :

```
var SASocket;
var connectioncallback = {
        /* when a remote peer agent requests a service connection */
        onrequest : function(peerAgent){
            SAAgent.acceptServiceConnectionRequest(peerAgent);    },
        /* when the connection between provider and consumer is established */
        onconnect : function(socket) {
            SASocket = socket;
        },
        /* when an error occurs during connect and request operations */
        onerror : function(errorCode) {
            console.log("Service connection error. : " + errorCode);
        }
};
try {
    SAAgent.setServiceConnectionListener(connectioncallback);
} catch (e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

4. To retrieve peer agents, use the findPeerAgents() method of the SAAgent interface. To establish a service connection with a remote peer agent, use the requestServiceConnection() method. The callback will be invoked once for each found peer. Note that onpeeragentupdated is mostly used on Host-side

application to detect Wearable-side widget availability. AVAILABLE means the corresponding Wearable-side widget is installed, while UNAVAILABLE means opposite:

```
function onpeeragentfound(peerAgent) {
    if(peerAgent.appName == "expected app name") {
        SAAgent.requestServiceConnection(peerAgent);
    }
}

function onpeeragentupdated(peerAgent, status) {
    if(status == "AVAILABLE") {
        try {
            SAAgent.requestServiceConnection(peerAgent);
        } catch(e) {
            console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
        }
    } else if(status == "UNAVAILABLE") {
        console.log("Uninstalled application package of peerAgent on remote device.");
    }
}

function onerror(errorCode) {
    console.log("Error code : " + errorCode);
    if(errorCode == "PEER_NOT_FOUND") {
        console.log("If remote application is not already installed on the remote device, Please wait
onpeeragentupdated callback.");
    }
}

var peeragentfindcallback = {
    onpeeragentfound : onpeeragentfound,
    onpeeragentupdated : onpeeragentupdated,
    onerror : onerror
};

try {
    SAAgent.setServiceConnectionListener(connectioncallback);
    SAAgent.setPeerAgentFindListener(peeragentfindcallback);
    SAAgent.findPeerAgents();
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

5. As a result of the request, onconnect() of ServiceConnnectionCallback, defined in step 3, will be executed.

6.4.2. Simple Data(String) Exchange

1. To send a string message to a remote accessory peer agent, use the sendData() method of the SASocket interface. Provide the channel ID to connect:

```
// Assume SAAgent object has been obtained by using requestSAAgent() method.
var SASocket;
var connectioncallback = {
    /* when the connection between provider and consumer is established */
    onconnect : function(socket) {
        SASocket = socket;
        for (var i = 0; i < SAAgent.channelId.length; i++) {
                SASocket.sendData(SAAgent.channelIds[i], "send message to " + i + "th channel.");
        }
    }
};

    try {
        SAAgent.setServiceConnectionListener(connectioncallback);
    } catch(e) {
        console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
```

```
}
```

2. To send an encrypted string message to a remote accessory peer agent, use the sendSecureData() method of the SASocket interface. Provide the channel ID to connect:

```
// Assume SAAgent object has been obtained by using requestSAAgent() method.
var SASocket;
var connectioncallback = {
    /* when the connection between provider and consumer is established */
    onconnect : function(socket) {
        SASocket = socket;
        for (var i = 0; i < SAAgent.channelId.length; i++) {
                SASocket.sendSecureData(SAAgent.channelIds[i], "send message to " + i + "th channel.");
        }
    }
};

try {
    SAAgent.setServiceConnectionListener(connectioncallback);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

3. To receive a string message from a remote peer agent, use the setDataReceiveListener() method. The setDataReceiveListener() method registers the SADataReceiveCallback listener interface, which is invoked when a string message from a peer agent is received :

```
/* Assume SASocket object has been obtained by using setServiceConnectionListener() method. */
function onreceive(channelId, data) {
        console.log("Message received - " + channelId + " : " + data);
};

try{
    SASocket.setDataReceiveListener(onreceive);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

6.4.3. File Exchange

1. To get the SAFileTransfer object specified in the SAAgent, use the `getSAFileTransfer()` method :

```
// Assume SAAgent object has been obtained by using requestSAAgent() method.
try{
    var filetransfer = SAAgent.getSAFileTransfer();
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

2. To send a file to a remote peer agent:
   Define the event handlers for file transfer notifications using the SAFileSendCallback listener interface:

```
/* Assume filetransfer object has been obtained by using getSAFileTransfer() method. */
var sendfilecallback = {
    onprogress : function(transferId, progress){
        console.log("onprogress transferId : " + transferId + ", progress : " + progress);
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileSendError transferId : " + transferId + " code : " + errorCode);
    }
};
try{
```

```
    filetransfer.setFileSendListener(sendfilecallback);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

3. Next, use the sendFile() method :

```
/* Assume agent object has been obtained by using requestSAAgent() method. */
var filePath = "file:///opt/usr/media/Downloads/Image.jpg";
var transferId = null;
var filetransfer = null;

var peeragentfindcallback = {
    onpeeragentfound : function(peerAgent) {
       if(peerAgent.appName == "expected app name") {
            try {
                transferId = filetransfer.sendFile(peerAgent, filePath);
            } catch(e) {
                console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
            }
        }
    },
    onerror : function(errorCode){
        //DO SOMETHING
    }
};

var sendfilecallback = {
    onprogress : function(transferId, progress){
        console.log("onprogress transferId : " + transferId + ", progress : " + progress);
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileSendError transferId : " + transferId + " code : " + errorCode);
    }
};

try{
    filetransfer = agent.getSAFileTransfer();
    filetransfer.setFileSendListener(sendfilecallback);
    agent.setPeerAgentFindListener(peeragentfindcallback);
    agent.findPeerAgents();
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

4. To receive a file from a remote agent, define the event handlers for notifications(onreceive, onprogress, oncomplete) using the SAFileReceiveCallback listener interface :

```
/* Assume filetransfer object has been obtained by using getSAFileTransfer() method. */
var receivefilecallback = {
    onreceive : function(transferId, fileName){
        console.log("Incoming file transfer request form the remote peer agent. transferId : " + transferId
+ " file name : " + fileName);
    },
    onprogress : function(transferId, progress){
        console.log("onprogress transferId : " + transferId + ", progress : " + progress);
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileReceiveError  transferId : " + transferId + " code : " + errorCode);
    }
};

try{
    filetransfer.setFileReceiveListener(receivefilecallback);
} catch(e) {
```

```
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

5. To accept the download of a file from a remote peer agent, use the receiveFile() method in onreceive:

```
/* Assume filetransfer object has been obtained by using getSAFileTransfer() method. */
var newFilePath = "file:///opt/usr/media/Downloads/ReceivedImage.jpg";

var receivefilecallback = {
    onreceive : function(transferId, fileName) {
        try {
            console.log("Incoming file transfer request form the remote peer agent. transferId : " +
transferId + " file name : " + fileName);
            filetransfer.receiveFile(transferId, newFilePath);
        } catch(e) {
            console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
        }
    },
    onprogress : function(transferId, progress){
        console.log("onprogress transferId : " + transferId + ", progress : " + progress);
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileReceiveError  transferId : " + transferId + " code : " + errorCode);
    }
};

try{
    filetransfer.setFileReceiveListener(receivefilecallback);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

6. To reject the incoming transfer request from a remote peer agent, use the rejectFile() method with the transfer ID:

```
/* Assume filetransfer object has been obtained by using getSAFileTransfer() method. */
var receivefilecallback = {
    onreceive : function(transferId, fileName) {
        try {
            console.log("Incoming file transfer request form the remote peer agent. transferId : " +
transferId + " file name : " + fileName);
            if(1) { // User wanted condition
                filetransfer.rejectFile(transferId);
            }
        } catch(e) {
            console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
        }
    },
    onprogress : function(transferId, progress){
        console.log("onprogress transferId : " + transferId + ", progress : " + progress);
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileReceiveError  transferId : " + transferId + " code : " + errorCode);
    }
};

try {
    filetransfer.setFileReceiveListener(receivefilecallback);
} try (e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

7. To cancel the download, use the cancelFile() method with the transfer ID in onprogress:

```
/* Assume filetransfer object has been obtained by using getSAFileTransfer() method. */
var receivefilecallback = {
    onreceive : function(transferId, fileName){
        try {
            console.log("Incoming file transfer request form the remote peer agent. transferId : " +
transferId + " file name : " + fileName);
            filetransfer.receiveFile(transferId, newFilePath);
        } catch(e) {
            console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
        }
    },
    onprogress : function(transferId, progress){
        try {
            if(1) { // User wanted condition
                filetransfer.cancelFile(transferId);
            }
        } catch(e) {
            console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
        }
    },
    oncomplete : function(transferId, localPath){
        console.log("File transfer complete.  transferId : " + transferId);
    },
    onerror : function(errorCode, transferId){
        console.log("FileReceiveError  transferId : " + transferId + " code : " + errorCode);
    }
};

try {
    filetransfer.setFileReceiveListener(receivefilecallback);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
}
```

6.4.4. Closing the Service Connection

To close the service connection with the remote peer agent, use the close() method:

```
// Assume SAAgent object has been obtained by using requestSAAgent() method.
var callback = {
    onrequest : function(peerAgent){
    },
    onconnect : function(socket){
        if(socket.peerAgent.appName != "expected_appName") {
            try {
                socket.close();
            } catch(e) {
                console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);
            }
        }
    },
    onerror : function(errorCode){
    },
};

try {
    SAAgent.setServiceConnectionListener(callback);
} catch(e) {
    console.log("Error Exception, error name : " + e.name + ", error message : " + e.message);}
```

6.5. Localizing Widgets

To localize your widgets for I18N, you need to refer to “Localizing Widgets” section in “Help Contents” in the Tizen SDK for Wearable:

Tizen Wearable Web App Programming → Application Development Process → Packaging Applications → Localizing Widgets

Copyright

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co., Ltd. cannot accept responsibility for any errors or omissions or for any loss occurred to any person, whether legal or natural, from acting, or refraining from action, as a result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co. Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co. Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

For more information, please visit http://developer.samsung.com/