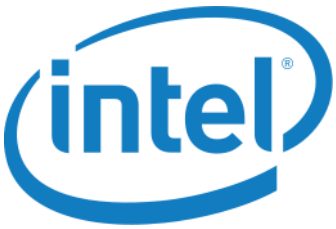
Intel Software Platform for Curie

Android Developer's Guide

Version 1.0

December 2015

BETA DRAFT



Part Number 333550-001US

Contents

Chapter 3: Using the Wearable Platform Core APIs

Chapter 7: Using the Cloud Services SDK

Chapter 8: Third-Party License Information

List of Figures and Code Samples

Figures

Code Samples

Using the Wearable Platform Core APIs

Using the Wearable Platform Body IQ APIs

Using the Wearable Platform Time IQ APIs

Using the Cloud Services SDK

CHAPTER

1 About This Guide

This guide is your starting reference for developing Android mobile applications for Curie-based wearable devices. The guide provides an architecture overview for Intel Software Platform for Curie, and detailed descriptions of supported functionality and APIs.

This chapter contains the following sections:

1.1 Who Should Read This Book

This guide is designed for Android developers.

To use the Intel Curie Software Platform for Curie SDK, Android developers should have proficiency in Java and experience developing mobile applications for the Android mobile platform.

1.2 Terminology Used in This Guide

Table 1: Terms and Definitions

Term	Definition
Android	Android mobile operating system
BLE	Bluetooth Low Energy radio, a wireless personal area network technology
BLOB	Collection of binary data stored as a single entity
CRUD	Create, read, update, and delete operations
DAO	Data Access Object. Provides an abstract interface to a database or other persistence mechanism.
HTTP	Hypertext Transfer Protocol, an application protocol for hypermedia information systems.
HTTPS	Secure Hypertext Transfer Protocol, a protocol for secure communication over a computer network
Javadoc	HTML-based Java API reference
JSON	JavaScript Object Notation, a lightweight data-interchange format
MOT	Mode of transport. In Time IQ this can be walking, driving, or stationary.
PoI	Place of Interest Notable location of significance to the device wearer. This can be an internationally recognized location, or a location of importance to only the device wearer.
REST	Representational State Transfer, a software architectural style of the World Wide Web
SoC	Intel Quark SE System on a Chip
TTL	Time IQ Time-To-Leave
User State	Captures device wearer location and proprioception details. Examples of proprioception details are: device wearer is driving; device wearer is at home, work, or another PoI; device wearer is near a PoI.

Table 1: Terms and Definitions

Term	Definition
UUID	Universally Unique Identifier
Wearable Platform	Intel Software Platform for Curie The name is shortened in this guide for readability.

1.3 Additional Resources

The following documents are included in the Wearable Platform SDK to help you get started:

- *Wearable Platform Javadoc API Reference for Android*
- *Cloud Services Portal Administrator Guide*
- *Intel® Curie™ Platform Customer Reference Board (CRB) Hardware User Guide*
- *Intel® Curie™ Platform Hardware User's Guide*
- *Intel® Curie™ Platform Software User's Guide*

CHAPTER

2 Overview of Intel Software Platform for Curie

This chapter explains Intel Software Platform for Curie basic concepts. The chapter introduces Wearable Platform Core APIs, Intel® IQ Software kits, and Intel Cloud Services APIs. Together, these comprise the Intel Software Platform for Curie SDK.

Tip: In this guide, "Intel Software Platform for Curie" is sometimes called "Wearable Platform" for better readability.

This chapter contains the following sections:

2.1 About Intel Wearable Technology

Intel Wearable Technology is more than just smartwatches and step-counters. Today you can embed payment engines, medical health monitors, identity-based access systems –and so much more– into wearable devices. Wearable devices can be unobtrusive while delivering critical data and services to the device wearer.

Intel Wearable Technology houses the Intel Quark SE System on a Chip (SoC) which integrates the power of a full-sized computer into a single dime-sized chip. The hardware module includes a Bluetooth Low Energy radio (BLE), motion sensors, and battery-charging capabilities. The hardware runs on Intel Software Platform for Curie.

The small form factor Curie module combined with the Wearable Platform SDK is an attractive option for designers and developers who want to quickly turn innovative ideas into products. Intel Wearable Technology is designed to power both consumer wearable devices and industrial wearable devices.

2.1.1 Understanding the Major Components

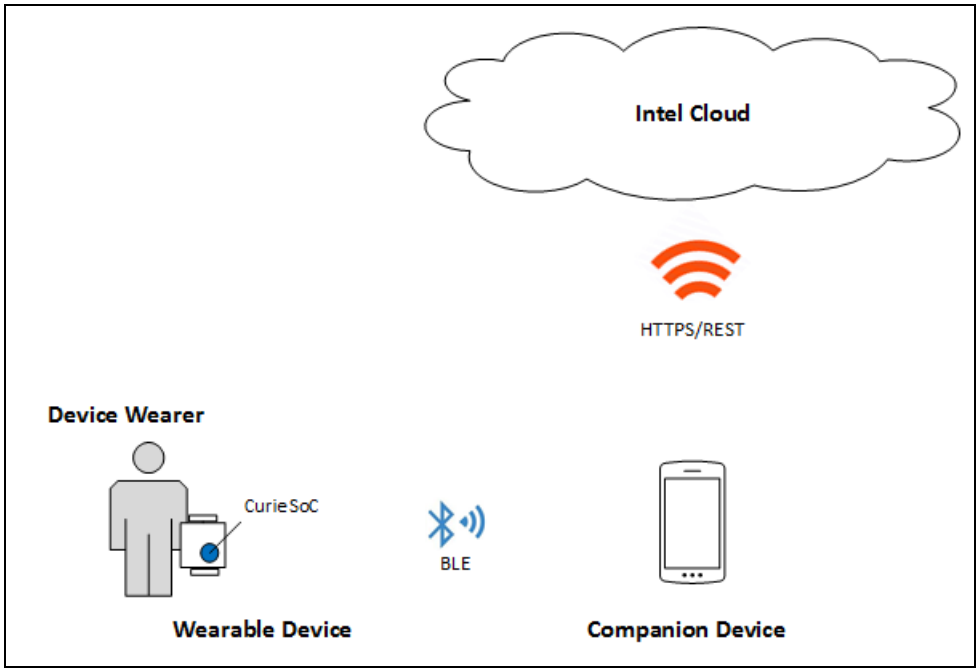
Intel Wearable Technology encompasses the following:

- A wearable device containing Intel Curie SoC
- A companion device running a mobile application created with Wearable Platform SDK
- Intel Cloud Services

The device wearer puts on a wearable device containing Intel Curie SoC. An application implemented with Wearable Platform SDK runs on a companion device, typically a mobile phone. The mobile application communicates with the wearable device over a BLE communication channel. The companion device communicates with Intel Cloud Services using REST and HTTP(S) protocols.

The following image illustrates the communication flow among Intel Wearable Platform components.

Figure: Wearable Platform Components



2.1.2 Intel Wearable Platform Architecture

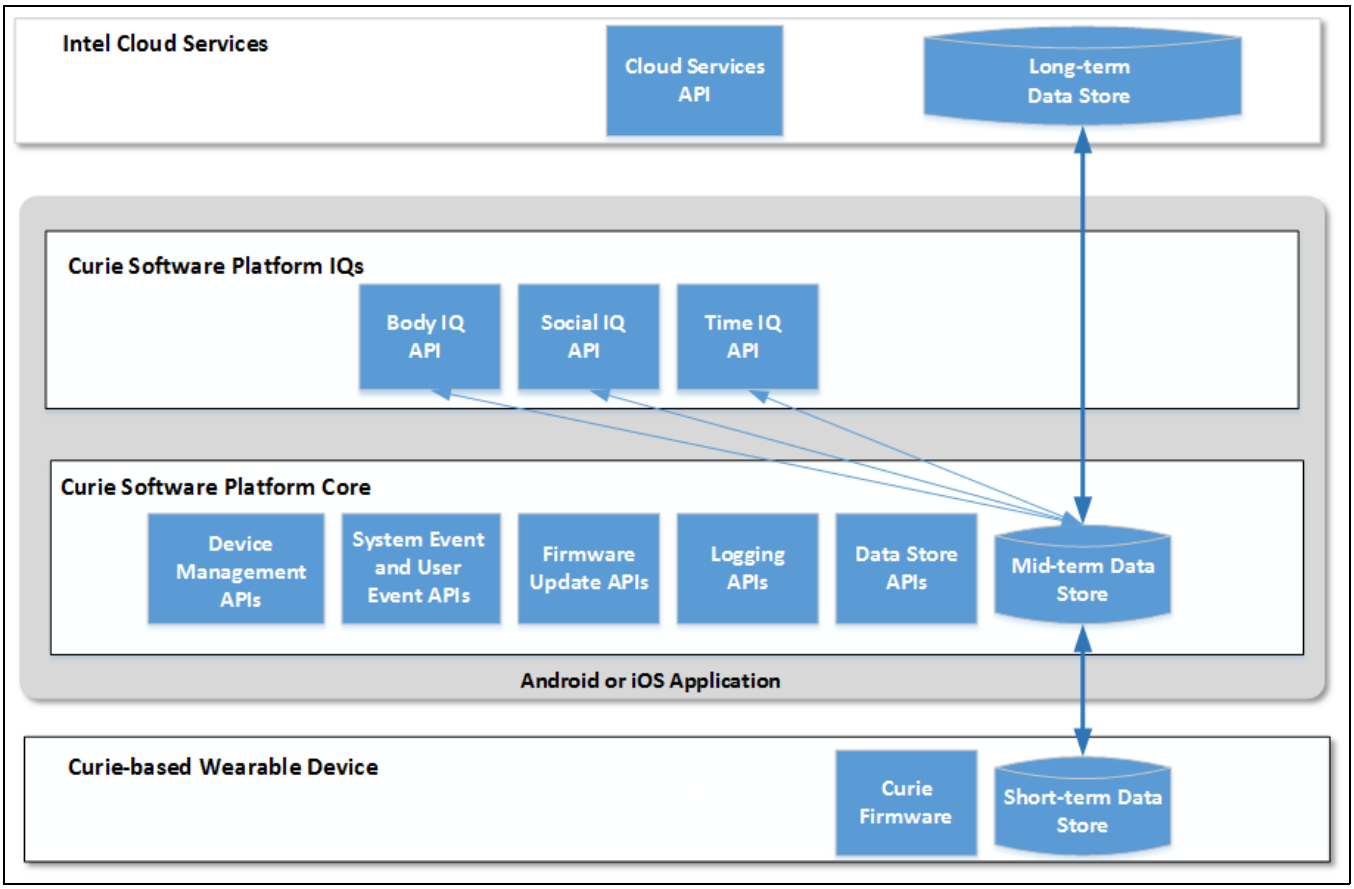
Intel Wearable Platform can be partitioned into three logical, binary components:

- Wearable Platform Core APIs
- IQ Software Kits
- Cloud Services APIs

You can innovate wearable device functionality by implementing one or more IQ Software Kits in your mobile application.

The following image illustrates the Wearable Platform SDK architecture:

Figure: Wearable Platform SDK Architecture



2.2 Wearable Platform Core APIs Overview

The following Wearable Platform Core APIs provide services useful to all mobile applications interacting with Curie-based wearable devices:

- Device Management APIs
- Firmware Update APIs
- System Event APIs
- User Event APIs

- Wearable Notification APIs
- Data Store APIs
- Logging APIs

2.2.1 Device Management APIs

The mobile application uses Device Management APIs to manage and communicate with the wearable device containing Intel Curie SoC. Device Management APIs provide the following operations:

- Scan and discover Intel-based wearable devices
- Pair with the Curie-based wearable device
- Unpair from the Curie-based wearable device
- Connect to the Curie-based wearable device
- Disconnect from the Curie-based wearable device
- Monitor the Curie-based wearable device connection status
- Monitor the Curie-based wearable device battery status
- Perform factory reset of the Curie-based wearable device
- Synchronize the clock between the Curie-based wearable device and the companion device

Note: Clock synchronization between the Curie-based wearable device and the companion device is performed automatically each time the two devices connect, and is refreshed every 24 hours.

2.2.2 Firmware Update APIs

The mobile application uses Firmware Update APIs to programmatically check and update firmware on the wearable

device containing Intel Curie SoC. The Firmware Update APIs provide the following operations:

- Determine what version of firmware is currently installed on the Curie-based wearable device
- Install new firmware revision on the Curie-based wearable device
- Track progress of the firmware update on the Curie-based wearable device

2.2.3 System Event APIs

This set of APIs provides details of system events such as device boot, crash, shutdown, or low battery. The mobile application can monitor for any of these events after the Curie-based wearable device is successfully connected to the companion device.

Each system event is summarized as a four-tuple that includes the following:

- Universally unique identifier (UUID) of the wearable device issuing the event
- UUID of the wearable device user identity
- Type of the system event:
  - Main boot event
  - Boot event after the firmware update
  - Crash event
  - Low battery event
  - Shutdown event triggered by the user
  - Shutdown event triggered due to critically low battery
- Timestamp indicating when the event occurred

2.2.4 User Event APIs

The User Event APIs provide details of events issued by the user, such as a user tapping or activating a button on the wearable device. The mobile application can monitor for these events after the Curie-based wearable device is successfully connected to the companion device.

Each user event is summarized as a four-tuple that includes the following:

- UUID of the wearable device
- UUID of the wearable device user identity issuing the event
- Type of the user event:
  - Double-tap
  - Triple tap
  - Single button press
  - Double button press
  - Triple button press
  - Long button press

Note: Triple button press events are not supported in this release of Intel Software Platform for Curie. The definition is reserved for future use.

2.2.5 Wearable Notification APIs

The Notification APIs support constructing and sending LED and vibration pattern notifications from the companion device to the Curie-based wearable device. The mobile application can specify LED pattern and/or vibration pattern as well as the time interval between the notifications.

Vibration Pattern specifications include:

- Vibration type
  Two types are currently supported: square and special effects
- Amplitude
  Ranges from 0 (no vibration) to 255 (full strength vibration)
- Duration pattern
  Specifies the ON/OFF duration
- Repetition count
  Specifies the number of times the pattern is repeated. Supports 1 to 255 repetitions.

Tip: When configuring LED and Vibration pattern specifications, keep in mind that additional pattern repetitions drain the battery. Keep the repetition count to the minimum to preserve the battery life on the wearable device.

LED Pattern specifications include:

- LED types are LEDBlink and LEDWave
- RGB color list specification
  Color code configuration for each notification
- Intensity specifies LED notification brightness
- Duration pattern specifies the ON/OFF duration
- Repetition count specifies the number of times the pattern is repeated. Supports 1 to 255 repetitions.

2.2.6 Data Store APIs

Wearable Platform Core APIs provide a way to persist data in the local data store. Entities can be created, retrieved, updated, and deleted (CRUD). The APIs support CRUD operations on wearable device details and the user identity details.

Wearable device details data model includes:

- Unique identifier
- Bluetooth address
- Display name

- Manufacturer
- Model
- Serial number
- Firmware revision
- Software revision
- Hardware revision

User identity details data model includes:

- Unique identifier
- User ID
  Authenticated user ID provided by the Cloud Authentication services
- First name
- Last name
- Email address
- Phone number associated with the companion device

2.2.7 Logging APIs

Wearable Platform Core APIs provide logging capabilities supporting multiple log level granularities.

Log levels can be set as follows:

- ERROR
  Indicates that something is wrong and is potentially fatal.
- WARNING
  Indicates a problem that should be looked into.
- INFO
  Indicates something of note. However, there is no problem.
- DEBUG
  Indicates something useful while debugging.

- VERBOSE
  Indicates detailed or frequently occurring information useful while debugging.

The order in terms of verbosity, from least to most is ERROR, WARN, INFO, DEBUG, VERBOSE.

2.2.8 Error Code APIs

This API set provides error codes along with machine-readable and human-readable error descriptions other Wearable Platform Core APIs may return when an operation fails. Error codes include BLE errors, authentication errors, HTTP errors, and many others.

2.3 IQ Software Kits Overview

Intel® IQ Software kits include the embedded software that runs on the Intel® Curie™ module together with companion smartphone applications and associated cloud capabilities.

Each IQ Software Kit provides APIs with specialized functionality capabilities:

- Body IQ APIs
- Social IQ APIs
- Time IQ APIs

2.3.1 Body IQ APIs

Body IQ APIs provide mobile applications with capabilities related to detecting and measuring body activities recorded by the wearable device, including the following:

- Real-time motion activity detection and classification activities can be categorized as walking or running.
- Measurement of step count for walking and running activities
- Measurement of distance covered while walking and running activities
- Measurement of number of calories burned walking and running activities

2.3.2 Social IQ APIs

Social IQ APIs provide mobile applications with capabilities to send notifications to the wearable device, including the following:

- Configuring and sending LED color patterns on Android wearable devices in response to a social network event, an incoming phone call, SMS, and so forth.
- Configuring and sending haptic in the form of vibration patterns in response to a social network event an incoming phone call, SMS, and so forth.

These capabilities are provided by the Wearable Notification APIs on page 11.

2.3.3 Time IQ APIs

Time IQ APIs provide a diverse set of context-aware capabilities to help consumers with time management, multi-task-

ing, arriving on time to targeted destinations, and other daily routine optimization. Time IQ APIs include management of the following entities:

- Calendar events
  Meetings and appointments recorded in the device wearer calendar.
- Reminders
  Calendar reminders, Time-To-Leave (TTL) cues in order to arrive to the next destination on time, task reminders, and so forth.
- Places of Interest (PoI)
  Notable location of significance to the device wearer This can be an internationally recognized location, or a location of importance to only the device wearer.
- User state
  Captures device wearer location and proprioception details. Examples of proprioception details are: device wearer is driving; device wearer is at home, work, or another PoI; device wearer is near a PoI.

2.4 Intel Cloud Services APIs Overview

The Cloud Services APIs provide a communication layer between the companion device and the Intel Cloud Service. Functionality includes:

- User authentication, cloud request scheduling, cloud response processing and error handling functionality used by all of the other Cloud Services SDK modules.
- Storage and retrieval of custom JavaScript Object Notation (JSON) documents that contain details about the authenticated user.
- Upload, storage and retrieval capabilities for general JSON documents.

- Upload and download capabilities for Binary Large OBject (BLOB) data and software assets.

For more information about Intel Cloud Services SDK, see Using the Cloud Services SDK on page 97.

2.5 Benefits of Using Intel Software Platform for Curie

Intel Software Platform for Curie is designed to facilitate application development so you can quickly deploy an extendable and forward-compatible wearable product.

- The platform is comprehensive and supports all currently defined Curie IQ APIs.
- You can extend support to new Curie IQ APIs as they become available.
- The Wearable Platform Core APIs provide services common to all Curie IQ APIs, eliminating redundant implementations.

CHAPTER

3 Using the Wearable Platform Core APIs

The Wearable Platform Core APIs implement services essential to managing and communicating with all Curie-based wearable devices regardless of Curie IQ.

This chapter contains the following sections:

3.1 Initializing Core in Your Application

Before accessing any of the services Wearable Platform Core APIs provide, you should initialize Core class in your application. Call the `Core.init()` method, supplying the application context as the parameter. If you plan to use additional services provided with Intel Software Platform for Curie, such as Intel Cloud services, they should be initialized immediately after. For detailed information, see Using the Cloud Services SDK on page 99.

Example: **Initializing the Core Class**

```
import com.intel.wearable.platform.core.Core;

public class Application extends android.app.Application {

    private static Application instance;

    @Override
    public void onCreate() {
        super.onCreate();
        instance = this;
        //initialize Intel Cloud services here, such as CloudBlobPublicStore, etc.
        Core.init(getApplicationContext());
    }
    public static final Context getContext(){
         return instance.getApplicationContext();
    }
}
```

3.2 Using the Device Management APIs

Device Management APIs are provided in the `com.intel.wearable.platform.core.device` package of the Wearable Platform Core SDK for Android mobile platform.

Device Management APIs support operations for managing and communicating with all Curie-based wearable devices. For a summary of Device Management API capabilities, see Device Management APIs on page 9.

Tip: BLE is also known as Bluetooth Smart or Version 4.0+ of the Bluetooth specification. platform.

The companion device (the mobile phone) communicates with the Curie-based wearable device using Bluetooth Low Energy (BLE) protocol. BLE provides significantly lower power consumption for both the wearable device and the companion mobile phone.

To establish a connection with the Curie-based wearable device, the mobile application must follow standardized development steps consisting of:

1. Scanning for nearby BLE-enabled devices
2. Selecting the device of interest
3. Pairing with and connecting to the wearable to the companion device

Device Management APIs further simplify this process.

This section contains the following examples which demonstrate how Device Management APIs can be used in your applications.

3.2.1 To Start a Wearable Device Scan

Perform the following steps to discover the Curie-based wearable device located in the close proximity:

Step 1. Create an instance variable of type `IWearableScanner`.

Step 2. Instantiate this variable using the `WearableScannerFactory`.

Step 3. Start the scan for the BLE devices in range, while specifying `IWearableScannerListener` parameter.

This listener is the callback function continuously monitoring the results of the scan until the scan is stopped. The scan itself is performed in the background. Events are reported to `IWearableScannerListener` in the main UI thread.

Tip: In Java, the callback pattern is frequently implemented using an anonymous inner class. You can override methods of this class to customize callback behavior.

Step 4. To customize how your application handles wearable device detection, the application should override two `of IWearableScannerListener` methods:

- `onWearableFound`
  This method is called each time a new BLE device is discovered. Overwrite this method to specify a set of actions to take when the BLE device is detected.
- `onScannerError`
  This method is called if an error has occurred during the scan. Overwrite this method to specify a set of actions to handle any errors that may occur during the course of device scanning.

You can include the following code sample in your application to start scanning for Curie-based devices. See the example on the next page.

Example: **Starting a Scan**

```
import com.intel.wearable.platform.core.error.Error;
import com.intel.wearable.platform.core.device.IWearableScanner;
import com.intel.wearable.platform.core.device.WearableScannerFactory;
import com.intel.wearable.platform.core.device.WearableToken;
import com.intel.wearable.platform.core.device.listeners.IWearableScannerListener;

        private IWearableScanner scanner;
...
        scanner = WearableScannerFactory.getDefaultScanner();
...
        scanner.startScan(new IWearableScannerListener() {
            @Override
            public void onWearableFound(IWearableScanner wearableScanner,
                                        WearableToken wearableToken) {
                    // A new BLE device is discovered
            }

            @Override
            public void onScannerError(IWearableScanner wearableScanner,
                                       com.intel.wearable.platform.core.error.Error error) {
                    // Error: Bluetooth support is disabled on the companion device.
                    if(error.gerErrorCode() == Error.BLE_ERROR_BT_DISABLED){
                        Intent enableBluetoothIntent =
                                        new Intent(BluetoothAdapter.ACTTION_REQUEST_ENABLE);
                        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
                    } else {
                        // Some other error occurred during the scan
                    }
            }
        });
```

3.2.2 To Stop a Wearable Device Scan

To abort wearable device discovery, call the `scanner.stopScan()`method, referencing the same scanner as in the example above.

3.2.3 About the Wearable Token

Intel Wearable Platform Core APIs define the `WearableToken` class to uniquely identify any Intel wearable device and to track it from inception. When the wearable device is initially discovered during the scan, the scanner creates a new corre-

sponding `WearableToken` to be associated with that wearable device. It contains a unique ID, address, and name of the wearable device. From the initial discovery onwards to pairing, connecting, or any other interaction with the wearable device, the `WearableToken` is used to reference it.

3.2.4 Obtaining the Wearable Controller

Once the wearable device is successfully discovered and is associated with the instance of the `WearableToken` class, your application can start communicating with that wearable device using an instance of the `IWearableController` interface. Intel Wearable Platform has defined the `IWearableController` interface to assist applications with communicating and managing the wearable device, and providing support for a wide array of operations, including:

- `pair()`
  Pairs with the wearable device
- `unpair()`
  Unpairs from the wearable device
- `connect()`
  Establishes a connection with the wearable device
- `disconnect()`
  Disconnects from the wearable device
- `factoryReset()`
  Initiates a factory reset of the wearable device
- `getBatteryStatus()`
  Reads the current battery status, which is reported in the related method of supplied `IWearableControllerListener`
- `getFirmwareController()`
  Returns instance of `IFirmwareController` used to update firmware on the wearable device. See Obtaining the Firmware Controller on page 29.
- `getNotificationController()`
  Returns an instance of a `INotificationController`

used to receive notifications of asynchronous events, such as user tapping on the device or system shut-down. See Obtaining the Notification Controller on page 37.

Use the `WearableControllerFactory.getWearableController()` method to obtain an instance of the Wearable Controller corresponding to the wearable device detected during the scan.

Supply as the first parameter the `WearableToken` obtained during the scanning process.

Example: **Obtaining the Wearable Controller**

```
import com.intel.wearable.platform.core.notification.WearableNotification;
import com.intel.wearable.platform.core.sample.util.EventBusUtil;
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.device.WearableControllerFactory;
import com.intel.wearable.platform.core.device.WearableIdentity;
import com.intel.wearable.platform.core.device.WearableToken;

        private IWearableController wearableController;
...
        wearableController = WearableControllerFactory.getWearableController(
                                    token, new WearableEventManager());
...
```

Supply as the second parameter the callback implementing the `IWearableControllerListener` interface in order to monitor status of the connection between the wearable device and the companion device.

Example: Monitoring the Wearable Device Connection

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.device.WearableBatteryStatus;
import com.intel.wearable.platform.core.device.listeners.IWearableControllerListener;

public class WearableEventManager implements IWearableControllerListener {

    @Override
    public void onConnecting(IWearableController wearableController) {
        //received wearable device is connecting event
    }

    @Override
    public void onConnected(IWearableController wearableController) {
        //received wearable device has connected event
    }

    @Override
    public void onDisconnecting(IWearableController wearableController) {
        //received wearable device is disconnecting event
    }

   @Override
    public void onDisconnected(IWearableController wearableController) {
        //received wearable device has disconnected event
    }

    @Override
    public void onPairedStatusChanged(IWearableController wearableController,
                                      boolean isPaired) {
        //received wearable device is paired event
    }

    @Override
    public void onBatteryStatusUpdate(IWearableController wearableController,
                                      WearableBatteryStatus batteryStatus) {
        //battery status update event
    }

    @Override
    public void onFailure(IWearableController wearableController,
                          com.intel.wearable.platform.core.error.Error error) {
        //error event occured
    }
 }
```

Attach the Wearable Controller to the wearable device before performing any operations with the wearable device. From that point on, use the Wearable Controller to control the wearable device, including connectivity to a companion application running on the mobile device, clock synchronization, battery monitoring, message exchange, and so forth.

3.2.5 To Pair with the Wearable Device

To pair the wearable device with the companion device, call `wearableController.pair()` method.

3.2.6 To Unpair the Wearable Device

To unpair wearable device from the companion device, call the `wearableController.unpair()` method.

3.2.7 To Connect to the Wearable Device

To connect wearable device to the companion device, call the `wearableController.connect()` method.

3.2.8 To Disconnect the Wearable Device

To disconnect wearable device from the companion device, call the `wearableController.disconnect()`method.

3.2.9 To Read the Wearable Device Battery Status

To obtain the current measurement of the battery level on the wearable device, call the `wearableController.getBatteryStatus()`method.

Alternatively, the application can subscribe to receive battery status update events. After your application calls the `wearableController.subscribeToBatteryStatusUpdateEvents()`method, updates on the battery level of the wearable device are reported in the `onBatteryStatusUpdate()`method of the `IWearableControllerListener` implementation of the callback supplied to the Wearable Controller:

Example: **Reading the Wearable Device Battery Status**

```
@Override
public void onBatteryStatusUpdate(IWearableController wearableController,
                                  WearableBatteryStatus batteryStatus) {
    //process the new battery status containing the latest battery level
}
```

3.2.10 To Set the Warble Device Name

To set the name for the wearable device, call the `wearableController.setWearableName(java.lang.String name)` method.

3.2.11 To Perform a Factory Reset of a Wearable Device

To perform a factory reset on the wearable device, call the `wearableController.factoryReset()` method.

3.3 Using the Firmware Update APIs

Firmware Update APIs are provided in the `com.intel.wearable.platform.core.firmware` package of the Wearable Platform Core SDK for Android mobile platform.

Firmware Update APIs are provided with the `IFirmwareController` interface and support operations for reading the latest available firmware version available of the wearable device, as well as updating it to a new revision.

3.3.1 Obtaining the Firmware Controller

Use the `wearableController.getFirmwareController()` method to obtain an instance of the `IFirmwareController associated` with the wearable device (after it has been connected to the companion device).

Example: **Obtaining the Firmware Controller**

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.firmware.IFirmwareController;

       private IWearableController wearableController;
       private IFirmwareController firmwareController;

...
           if (wearableController != null) {
               if (wearableController.getFirmwareController() != null) {
                   firmwareController = wearableController.getFirmwareController();
               } else {
                   throw new NullPointerException(
                   "This device does not support firmware upgrades.");
               }
           ...
           }
```

Below are few examples demonstrating how this API can be used in your applications.

3.3.2 To Read the Wearable Device Firmware Version

To get a string representation of the latest version of firmware installed on the wearable device, call the `firmwareController.getWearableFirmwareVersion()` method.

3.3.3 To Update the Wearable Device Firmware

To install firmware on the wearable device, call the `firmwareController.installFirmware(File firmwareFile, IFirmwareInstallListener listener)` method.

Supply as the first parameter a reference to the file containing the new firmware revision.

Example: **Updating the Wearable Device Firmware**

```
private File firmwareFile = new File("absolute path to firmware file");
...
firmwareController.installFirmware(firmwareFile, new
                                   FirmwareInstallEventManager());
```

Supply as the second parameter callback implementing the `IWearableListener` interface to monitor status of the firmware file transfer onto the wearable device as it progresses.

Example: **Monitoring Status of the Firmware File Transfer**

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.firmware.IFirmwareInstallListener;


public class FirmwareInstallEventManager implements IFirmwareInstallListener {
    @Override
    public void onFirmwareInstallStarted(IWearableController controller) {
        //firmware file transfer onto wearable device has started
    }

    @Override
    public void onFirmwareInstallProgress(IWearableController controller,
                                          int progress, int total) {
           // [progress] bytes of firmware file [total] bytes
           // have been transfered onto wearable device
    }

    @Override
    public void onFirmwareInstallComplete(IWearableController controller) {
           // firmware file transfer onto wearable device has completed successfully
    }

    @Override
    public void onFirmwareInstallError(IWearableController controller,
                         com.intel.wearable.platform.core.error.Error error) {
           // error occurred during the first transfer
    }
}
```

The firmware file containing a new revision of wearable device firmware can be obtained using Intel Cloud Services. See Using the Cloud Services SDK on page 99.

When initializing your application, after calling the `Core.init(context)` method, your application can create a new instance of the `CloudPublicBlobStore` class. This class

can be used to store wearable firmware files for firmware updates as described in this section.

3.4 Using the System Event APIs

System Event APIs are provided in the `com.intel.wearable.platform.core.event.system` package of the Wearable Platform Core SDK for Android mobile platform.

The System Event APIs provide details of system events such as device boot, crash, shutdown or low battery.

After the Curie-based wearable device has been successfully connected to the companion device, the mobile application can monitor system events using `SystemEventController` class.

3.4.1 To Subscribe to System Events

To subscribe to receiving system events as they occur on the connected wearable device, the application calls the `SystemEventController.subscribe(IWearableSystemEventListener listener)` method. The listener is the callback function, invoked as the system events are received from the wearable device.

3.4.2 To Unsubscribe from System Events

To unsubscribe from receiving system events from the connected wearable device, call the `SystemEventController.unsubscribe()` method.

3.4.3 About System Event Types

Each system event, as described using the `WearableSystemEvent` class, contains details of each system event:

- `getTimestamp()`
  Returns atimestamp when the event occurs.
- `getWearableIdentityUuid()`
  Returns UUID of the wearable device.
- `getUserIdentityUuid()`
  Returns UUID of the wearable device user identity.
- `getSystemEvent()`
  Returns the specific type of the system event, represented as one of the following possible subclasses of the `WearableSystemEvent.SystemEvent` class:
  - `WearableSystemEvent.BootEvent`
    Two boot type enumerations are defined: MAIN or OTA (boot event after firmware update)
  - `WearableSystemEvent.CrashEvent`
    `WearableSystemEvent.LowBattEvent`
    Low battery
  - `WearableSystemEvent.ShutdownEvent`
    Two shutdown type enumerations are defined: USER_SHUTDOWN (shutdown triggered by the user) or CRITICAL_BATTERY_SHUTDOWN (shut down due to critically low battery)

The following examples demonstrate how System Event APIs can be used in your applications.

Example: **Unsubscribing from Receiving System Events**

```
import com.intel.wearable.platform.core.event.system.SystemEventController;
import com.intel.wearable.platform.core.event.system.WearableSystemEvent;

   //after wearable device is discovered and connected to
   SystemEventController.subscribe(new WearableSystemEventManager());
   ...
   SystemEventController.unsubscribe();
```

Example: **Using System Events APIs**

```
import com.intel.wearable.platform.core.event.system.IWearableSystemEventListener;
import com.intel.wearable.platform.core.event.system.WearableSystemEvent;

public class WearableSystemEventManager implements IWearableSystemEventListener {

   @Override
   public void onWearableSystemEvent(WearableSystemEvent systemEvent) {
      //process notification of the system event issued on the wearable device
      if (systemEvent.getSystemEvent() instanceof
                                    WearableSystemEvent.CrashEvent) {
          int crashType = ((WearableSystemEvent.CrashEvent)
                    systemEvent.getSystemEvent()).getCrashType();
          holder.eventDescription.setText("CrashType: " + crashType);
      } else if (systemEvent.getSystemEvent() instanceof
                                    WearableSystemEvent.LowBattEvent) {
           holder.eventType.setText("LowBattEvent");
           int batterLevel = ((WearableSystemEvent.LowBattEvent)
           systemEvent.getSystemEvent()).getBatteryLevel();
           holder.eventDescription.setText("BatteryLevel: " + batterLevel + "%");
      } else if (systemEvent.getSystemEvent() instanceof
                                    WearableSystemEvent.ShutdownEvent) {
           holder.eventType.setText("ShutdownEvent");
           WearableSystemEvent.ShutdownEvent.ShutdownType shutDownType =
                                    ((WearableSystemEvent.ShutdownEvent)
           systemEvent.getSystemEvent()).getShutdownType();
           holder.eventDescription.setText("ShutdownType: " + shutDownType.name());
      } else if (systemEvent.getSystemEvent() instanceof
                                    WearableSystemEvent.BootEvent) {
           holder.eventType.setText("BootEvent");
           WearableSystemEvent.BootEvent.BootType bootType =
                                              ((WearableSystemEvent.BootEvent)
           systemEvent.getSystemEvent()).getBootType();

           holder.eventDescription.setText("BootType: " + bootType.name());
      }
   }
}
```

3.5 Using the User Event APIs

User Event APIs are provided in the `com.intel.wearable.platform.core.event.user` package of the Wearable Platform Core SDK for Android mobile platform.

The User Event APIs provides details of user events such as the user double-tapping the wearable device, the user tri-

ple-tapping the wearable device, or the user pressing the wearable device button.

After the Curie-based wearable device has been successfully connected to the companion device, the mobile application can monitor user events using the `UserEventController` class.

3.5.1 To Subscribe to User Events

To subscribe to receiving user events on the connected wearable device as the occur, the application calls the `UserEventController.subscribe(IWearableUserEventListener listener)` method. The listener is the callback function, invoked as the user events are received from the wearable device.

3.5.2 To Unsubscribe from User Events

To unsubscribe from receiving user events from the connected wearable device, call the `UserEventController.unsubscribe()` method.

3.5.3 User Event Specification

Each user event, as described using `WearableUserEvent class`, contains details of each user event:

- `getWearableIdentityUuid()`
  Returns UUID of the wearable device.
- `getUserIdentityUuid()`
  Returns UUID of the wearable device user identity.
- `getUserEvent()`
  Returns the specific type of the user event, repre-

sented as one of the following possible subclasses of `WearableUserEvent.UserEvent class`:

– `WearableUserEvent.TappingEvent` = user taps the device. `UserEventType` = TAPPING.
The following types of tapping events are defined:

```
WearableUserEvent.TappingEventType.DOUBLE_TAP
WearableUserEvent.TappingEventType.TRIPLE_TAP
WearableUserEvent.TappingEventType.UNKNOWN
```

– WearableUserEvent.ButtonEvent = user taps the device. UserEventType = BUTTON.
The following types of button activation events are defined:

```
WearableUserEvent.ButtonEventType.SINGLE_PRESS
WearableUserEvent.ButtonEventType.DOUBLE_PRESS
WearableUserEvent.ButtonEventType.TRIPLE_PRESS
WearableUserEvent.ButtonEventType.LONG_PRESS
WearableUserEvent.ButtonEventType.UNKNOWN
```

Note: Triple-Press events are not supported in this release of Intel Software Platform for Curie. The definition is reserved for future use.

The following example demonstrates how User Event APIs can be used in your applications.

Example: **Unsubscribing from User Events**

```
import com.intel.wearable.platform.core.event.system.UserEventController;
import com.intel.wearable.platform.core.event.system.WearableUserEvent;

   //after wearable device is discovered and connected to
   UserEventController.subscribe(new WearableUserEventManager());
   ...
   UserEventController.unsubscribe();
```

3.5.4 Putting It All Together

The following example demonstrates how to use the User Events APIs.

Example: **Using the User Events APIs**

```
import com.intel.wearable.platform.core.event.system.IWearableUserEventListener;

import com.intel.wearable.platform.core.event.system.WearableUserEvent;


public class WearableUserEventManager implements IWearableUserEventListener {


   @Override
   public void onWearableUserEvent(WearableUserEvent userEvent) {

      //process notification of the user event issued on the wearable device
        WearableUserEvent wearableUserEvent = userEvent.getUserEvent();
        switch (wearableUserEvent.getUserEventType()) {
                case TAPPING:
                      // The exact type of tapping event can be shown as follows:
                      System.out.println((
                      (WearableUserEvent.TappingEvent)
                          wearableUserEvent).getTappingEventType().name());
                      break;
                case BUTTON:
                      System.out.println ((
                      (WearableUserEvent.ButtonEvent)
                          wearableUserEvent).getButtonEventType().name());
                      break;
                default:
                      //unknown user event, ignore
                      break;
        }
...

}
```

3.6 Using the Wearable Notification APIs

Wearable Notification APIs are provided in the `com.intel.wearable.platform.core.notification` package of the Wearable Platform Core SDK for Android mobile platform.

The Wearable Notification APIs support constructing and sending LED display pattern notifications and vibration pattern notifications from the companion device to the Curie-based wearable device.

3.6.1 Obtaining the Notification Controller

Use the `wearableController.getNotificationController()` method to obtain an instance of the `INotificationController` associated with the wearable device (after it has been connected to the companion device).

Example: **Obtaining the Notification Controller**

```
import com.intel.wearable.platform.core.device.IWearableController;
import com.intel.wearable.platform.core.notification.INotificatiionController;

       private IWearableController     wearableController;
       private INotificationController notificationController;

...

       if (wearableController != null) {
           if (wearableController.getNotificationController() != null) {
               notificationController =
                           wearableController.getNotificationController();
           } else {
               throw new NullPointerException(
               "This device does not support LED/haptic notifications.");
           }
       ...
       }
```

From this point on, your application can transmit notifications of choice by calling the method

```
notificationController.sendNotification(WearableNotification notification)
```

as described in more details in the sections below.

3.6.2 To Specify an LED Pattern

To transmit LED notifications to the wearable device, construct an object of class `WearableNotification.LedPattern` and supply it as a parameter to the `WearableNotification` class constructor.

Example: **Transmitting LED Notifications to the Wearable Device**

```
WearableNotification.LedPattern(
           WearableNotification.LedPattern.Type type,
           int id,
           List<WearableNotification.RGBColor> rgbColorList,
           int repetitionCount,
           int intensity)
```

The LED pattern type is specified as the first parameter to the constructor of `WearableNotification.LedPattern` class. The Wearable Notification APIs supply a set of predefined LED pattern types from which you can choose including the following:

- LED_BLINK_C2
- LED_BLINK_X2
- LED_FIXED
- LED_WAVE_C2
- LED_WAVE_X2

You can also specify other attributes such as pattern color scheme, intensity, repetition count and duration.

The RGB color scheme for each notification is specified as the `RGBColor(int red, int green, int blue)` class object. It can be specified as an argument to the

`WearableNotification.LedPattern` class constructor, or separately added with the `addRGBColor()` method.

Tip: When configuring LED and Vibration pattern specifications, keep in mind that additional pattern repetitions drain the battery. Keep the repetition count to the minimum to preserve the battery life on the wearable device.

- Intensity
  Specifies LED notification brightness. Ranges from 1 to 255
- Repetition count
  Specifies the number of times the pattern is repeated. Supports 1 to 255 repetitions.

Duration pattern, ON/OFF duration, is specified using the `WearableNotification DurationPattern(int durationOn, int durationOff)` class, where:

  - `durationOn` is the number of milliseconds the LED is active
  - `durationOff` is the number of milliseconds the LED is inactive.

***Example:* Specifying an LED Pattern**

```
WearableNotification.LedPattern ledPattern =
        new WearableNotification.LedPattern(
                WearableNotification.LedPattern.Type.LED_BLINK_C2,
                0, //id
                null, //rgb color list
                repetitionCount, //repetition count
                amplitude);//intensity
        ledPattern.addDuration(
                new WearableNotification.DurationPattern(500,500));
        int r = (durationsAdapter.getItem(i).rgb_color >> 16) & 0xFF;
        int g = (durationsAdapter.getItem(i).rgb_color >> 8) & 0xFF;
        int b = (durationsAdapter.getItem(i).rgb_color >> 0) & 0xFF;
        ledPattern.addRGBColor(new WearableNotification.RGBColor(r, g, b));
```

Once the `WearableNotification.LedPattern` object is defined, the LED-only pattern notification can be created using it in the constructor, as follows:

```
WearableNotification ledNotification =
                        new WearableNotification(ledPattern);
```

3.6.3 To Specify Vibration Pattern

To transmit vibration notifications to the wearable device, construct an object of class `WearableNotification.VibrationPattern` and supply it as a parameter to the `WearableNotification` class constructor.

Two types of vibration patterns are currently supported: square and special effects. Each type can be created using a corresponding `WearableNotification.VibrationPattern` class constructor.

To create a new instance of a Square Vibration Pattern:

Example: **Creating a Square Vibration Pattern**

```
VibrationPattern(
    WearableNotification.VibrationPattern.Type type,
    byte amplitude,
    int repetitionCount,
    WearableNotification.DurationPattern... durationPatterns)
```

To create a new instance of a Special Effects Vibration Pattern:

Example: **Creating a Special Effects Vibration Pattern**

```
VibrationPattern(
    WearableNotification.VibrationPattern.Type type,
    byte amplitude,
    java.util.List<java.lang.Integer> effectsList,
    WearableNotification.DurationPattern... durationPatterns)
```

The vibration type is specified as the first parameter to the constructor of `WearableNotification.VibrationPattern` class. The Wearable Notification API supplies a set of predefined vibration pattern types from which you can choose, including the following types:

- VIBRA_SQUARE
- VIBRA_SPECIAL_EFFECTS

You can also specify other attributes such as pattern amplitude, repetition count and duration:

- Amplitude
  Ranges from 0 (no vibration) to 255 (full strength vibration)
- Repetition count
  Specifies the number of times the pattern is repeated. Supports 1 to 255 repetitions.
- Duration pattern, ON/OFF duration, is specified using the `WearableNotification DurationPattern(int durationOn, int durationOff)` class, where:
  - `durationOn` is the number of milliseconds the vibration is active
  - `durationOff` is the number of milliseconds the vibration is inactive.

Tip: When configuring LED and Vibration pattern specifications, keep in mind that additional pattern repetitions drain the battery. Keep the repetition count to the minimum to preserve the battery life on the wearable device.

***Example:* Specifying a Vibration Type**

```
WearableNotification.VibrationPattern vibrationPattern =
        new WearableNotification.VibrationPattern(
            WearableNotification.VibrationPattern.Type.VIBRA_SQUARE,
            (byte) amplitude,
            repetitionCount);
  vibrationPattern.addDuration(
            new WearableNotification.DurationPattern(500,500));
```

Once the `WearableNotification.VibrationPattern` object is defined, the vibration-only pattern notification can be created using it in the constructor, as follows:

***Example:* Creating a Vibration-Only Pattern**

```
WearableNotification  vibrationNotification = new WearableNotification(vibrationPattern));
```

3.6.4 Using LED and Vibration Patterns in One Notification

To construct a wearable notification consisting of both LED and vibration patterns, construct the wearable notification as follows:

Example: **Using LED and a Vibration Pattern in One Notification**

```
WearableNotification  comboNotification =
       new WearableNotification(vibrationPattern, LedPattern, delayStartup);
```

The delay startup parameter specifies the duration of time that elapses between the conclusion of the Vibration Pattern and the commencement of the LED Pattern.

3.6.5 Putting It All Together

Complete the following steps to construct and send notifications to the wearable device:

Step 1. Obtain the Notification Controller specific to the wearable device your application is connected to.

See Obtaining the Notification Controller on page 37 for more information.

Step 2. Create an instance of the class or classes describing the notification pattern. Do one of the following:

- Construct a class describing the Vibration pattern.
- Construct a class describing the LED pattern class.
- Construct both of the above classes, along with the startup delay specification, if the notification consists of both LED and haptic patterns.

See To Specify an LED Pattern on page 38 for more information.

Step 3. Use the class or classes from Step 2 to create an instance of the Wearable Notification class.

Step 4. Call the `notificationController.sendNotification(weara`

bleNotification) method to transfer the notification onto the wearable device, where:

- notificationController references the notification controller obtained in Step 1
- wearableNotification references the wearable notification created in Step 3.

The Notification Controller provides a channel or transferring vibration and/or LED notifications to the wearable device in order to make the device vibrate and/or blink. Use its sendNotification() method to send notification to the wearable device.

Example: **Constructing and Sending a Notification**

```
import com.intel.wearable.platform.core.notification.INotificatiionController;

       private IWearableController wearableController;
       private INotificationController notificationController;
       private WearableNotification.VibrationPattern vibrationPattern;
       private WearableNotification.LedPattern ledPattern;
...
       if (wearableController != null) {
           if (wearableController.getNotificationController() != null) {
               notificationController = wearableController.getNotificationController();
               ...
               vibrationPattern = new WearableNotification.VibrationPattern(...);
               ...
               ledPattern = new WearableNotification.LedPattern(...);
               ...
               notificationController.send(vibrationPattern, ledPattern, delayStartup);
           } else {
               throw new NullPointerException(
               "This device does not support LED/haptic notifications.");
           }
       ...
       }
```

3.7 Using the Data Store APIs

The Wearable Platform Core SDK provides a data store for storing User Identity and Wearable Identity details. This functionality is provided in the com.intel.wearable.platform.core.persistence.Local

DataStore package of the Wearable Platform Core SDK for the Android mobile platform. The objects that can be stored, UserIdentity and WearableIdentity, are defined in the com.intel.wearable.platform.core.model.datastore package.

3.7.1 About the User Identity Object

User identity is usually created after the user has successfully logged-in and authenticated with the cloud services. The class

com.intel.wearable.platform.core.model.datastore.UserIdentity

represents the device wearer, and is intended to be used in conjunction with LocalDataStore.setCurrentUserIdentity(UserIdentity).

3.7.2 Creating the User Identity

A new instance of User Identity can be constructed as in the follows:

```
UserIdentity(UUID uuid,
        String userId,
        String firstName,
        String lastName,
        String email,
        String phoneNumber)
```

Constructor's parameters are:

The UUID parameter value is the unique user ID and is *nullable*.

- If the supplied uuid parameter is NULL, the other parameters are used to create a new UserIdentity object and. A new uuid is generated automatically when this UserIdentity is set as part of

LocalDataStore.setCurrentUserIdentity(UserIdentity).

- If the supplied `uuid` parameter identifies a UserIdentity that already exists in the LocalDataStore, that object will be updated with the values of the other parameters supplied to the class constructor.

The `userId` parameter value is the external user ID, usually obtained upon the user successfully logging-in with the Cloud API. The remaining parameters are self-explanatory.

The following example demonstrates constructing a sample user identity.

Example: **Constructing a Sample User Identity**

```
final UserIdentity identity = UserIdentity(null,"userID",
                                           "John", "Doe",
                                           "john.doe@domain.com",
                                           "(555)123-4567");
```

3.7.3 Setting the Current User

After the user is authenticated and the corresponding `UserIdentity` is created (or retrieved), the application can start associating this user identity with the events and activities on the wearable device. This can be done as follows:

```
final UUID result = LocalDataStore.setCurrentUserIdentity(identity);
```

This method returns the unique UUID of the current user. As you recall, the user identity's UUID may be null when the `UserIdentity` is first created, and is auto-generated by the system when the user is set.

3.7.4 Obtaining the Current User

To get the `UserIdentity` or UUID of the current device wearer, the following methods in the `LocalDataStrore` class are available:

- `static UserIdentity getCurrentUserIdentity`
  Returns the current `UserIdentity`.
- `static java.util.UUID getCurrentUserIdentity UUID()`
  Returns the UUID of the current user identity.

The following example illustrates retrieving the UUID of the current user, followed by retrieving it from the data store.

Example: **Retrieving UUID of the Current User**

```
final UUID userIdentityUUID =
               LocalDataStore.getCurrentUserIdentityUUID()

if (userIdentityUUID == null) {
       throw new IllegalStateException(
       "No current UserIdentity exists.");
}
final UserIdentity identity =
               LocalDataStore.getUserIdentity(userIdentityUUID);
if (identity == null) {
       throw new IllegalStateException(
       String.format("No UserIdentity found for %s.",
               userIdentityUUID));
}
```

3.7.5 Retrieving the User Identity

To get the `UserIdentity` and UUID of the current device wearer, the following two methods of the `LocalDataStrore` class are available:

- `static UserIdentity getUserIdentity(String userId)`
  Returns the `UserIdentity` object associated with the provided `userId`, or null no such user exists in the local data store.
- `static UUID getUserIdentity(UUUI uuid)`
  Returns the `UserIdentity` object associated with the provided `uuid`, or null no such user exists in the local data store.

3.7.6 Deleting the User Identity

To remove the `UserIdentity` from the `LocalDataStore`, call `LocalDataStore.removeUserIdentity(identity.getUuid())` method. If deleting was successful the return value is TRUE.

3.7.7 About the Wearable Identity Object

Wearable Identity represents the wearable device based on data obtained directly from the wearable device. Class `com.intel.wearable.platform.core.model.data store.WearableIdentity` represents the wearable device and is intended to be used in conjunction with the `LocalDataStore.registerWearable()` method.

3.7.8 Creating the Wearable Identity

A new instance of User Identity can be constructed as follows:

```
WearableIdentity(UUID uuid,
                 String address,
                 String displayName,
                 String manufacturer,
                 String model,
                 String serialNumber,
                 String firmwareRevision,
                 String softwareRevision,
                 String hardwareRevision)
```

Another way to construct the `WearableIdentity` object is to register details of the wearable device in the local data store, as described in the following section.

3.7.9 Registering Wearable Identity

The `WearableIdentity` is returned after the wearable device details are registered in the local data store. The `WearableIdentity` object is returned after calling `LocalDataStore.registerWearable()`, as follows:

```
static LocalDataStore.registerWearable(
       String address,
       String displayName,
       String manufacturer,
       String model,
       String serialNumber,
       String firmwareRevision,
       String softwareRevision,
       String hardwareRevision)
```

3.7.10 Obtaining the Wearable Identity

To get the `WearableIdentity` object the following methods in the `LocalDataStrore` class are available:

- `static WearableIdentitygetWearableIdentity( String deviceAddress)`
  Returns the WearableIdentity object associated with the provided deviceAddress, or null if not found.
- `static WearableIdentitygetWearableIdentity( UUID uuid)`
  Returns the `WearableIdentity` object associated with the `uuid`, or `null` if not found.

Example: **Retrieving the WearableIdentity by Device Address**

```
final String deviceAddress = "CC:4B:DB:77:82:28";
final WearableIdentity wearable =
   LocalDataStore.getWearableIdentity(deviceAddress);
if (wearable == null) {
       throw new IllegalStateException(
       String.format("No WearableIdentity found for %s.",
              deviceAddress));
}
```

3.7.11 Deleting the Wearable Identity

To remove the `WearableIdentity` from the `LocalDataStore`, call the method

```
LocalDataStore.remove earableIdentity(wearable.getUuid())
```

If deleting was successful the return value is TRUE.

3.7.12 The Local Data Store

The Wearable Platform Core SDK provides methods to initialize, reset and delete the local data store.

- Call the method `LocalBataStore,init(byte[] encryptionKey)`to initialize the local data store. The encryption key, which you should diligently preserve and protect, is used to encrypt the data store to secure its data.
- Call `LocalDataStore.reset()` to clear all data managed by the local data store and reset the data store to its original state.
- Call `LocalDataStore.deleteDataStore()` to remove the data store file and all persisted information.

3.8 Using the Logging APIs

The Wearable Platform Core SDK for the Android mobile platform provides the class `com.intel.wearable.platform.core.util.Logger` to support logging functionality. Logging output goes to a log file in the default or custom location. Optionally, the application can specify the maximum file size of the log file. When the log file reaches the maximum allowed size, older log entries are overwritten with the newer entries.

3.8.1 About Logging Levels

Wearable Platform logging supports different logging granularities. They are specified in the `Logger.Level`, enumeration and include the following levels:

ERROR
Indicates that something is wrong and is potentially fatal.

WARNING
Indicates a problem that should be looked into.

INFO
Indicates something of note. However, there is no problem.

DEBUG
Indicates something useful while debugging.

VERBOSE
Indicates detailed or frequently occurring information useful while debugging.

If logging goes to a file, the logger can be told how often to change log files, how large the files can get, and how many files are kept before deleting older log files.

3.8.2 Initializing the Logger

The logger can be initialized with or without the log levels listed in the previous sections.

```
Logger.init()
```

- Calling this method writes all log entries to the internal log file named `logs.txt`.
- No maximum file size is specified.
- The file is stored with the rest of the application data. When the application is uninstalled, this file is automatically deleted.

```
Logger.init(File loggerFile, Logger.Level level)
```

- Calling this method writes all log entries to the specified file.
- No maximum file size is specified.
- The file location can be internal or external to the application. If the location of the file is external to the application, when the application is uninstalled, the log file must be deleted manually.
- The Logger Level specifies the lowest granularity level to be recorded in the log file. If the application initializes the Logger with the WARNING log level, only WARNING and ERROR log messages will be recorded. Specifying INFO log level will record INFO, WARNING and ERROR messages. Alternatively, if the application specifies VERBOSE log level, all five types of log messages will be recorded.

```
Logger.init(File loggerFile, Logger.Level level, long maxSize)
```

- Same as the above but with the maximum log file size specified.
- Maximum log file size is in bytes.
- When the log file reaches the specified maximum allowed size, older log entries are overwritten with the newer entries.

3.8.3 Message logging

Messages are supplied to the Logger in a form of a String, optionally followed by zero or more objects relevant to the String log message. Messages can be logged with or without a corresponding exception. The messages are logged using the appropriate granularity and urgency level methods in the Logger class.

Error-level logging

```
Logger.e(String message, Object objects)
Logger.e(Throwable throwable, String message, Object objects)
```

Warning-level logging

```
Logger.w(String message, Object objects)
Logger.w(Throwable throwable, String message, Object objects)
```

Info-level logging

```
Logger.i(String message, Object objects)
Logger.i(Throwable throwable, String message Object objects)
```

Debug-level logging

```
Logger.d(String message, Object objects)
Logger.d(Throwable throwable, String message, Object objects)
```

Verbose-level Logging

```
Logger.v(String message, Object…objects)
Logger.v(Throwable throwable, String message, Object objects)
```

Example: Logging Debug and Error Messages

To log a Debug level message displaying device firmware version, the application can call:

```
Logger.d("Device Firmware Version = " + version);
```

To Log an Error-level message and to throw the `NoRecordException` if the Wearable Identity with the specified UUID does not exist, the application can call:

```
Logger.e(NoRecordException, "Couldn't find a
wearableIdentity with address %s in the local data
store", mAddress);
```

Note: The `NoRecordException` is provided with the `com.intel.wearable.platform.core.persistence` package.

3.8.4 Retrieving and Removing the Log File

Use the `Logger.getLogFile()` method to obtain the reference to the log file.

Use the`Logger.cleanLogFile()` method to erase all of the content in the log file.

3.9 Using the Error Code APIs

The `Error` class is provided in the `com.intel.wearable.platform.core.error` package of the Wearable Platform Core SDK for Android mobile platform.

The `Error` class defines a summary of failure conditions that may be encountered by the system. Each error is represented using an error code and a human-readable error message containing error details.

The error code can be obtained using the `getErrorCode()` method. The error message can be obtained using and the `getErrorMessage()` method.

At this time, Wearable Platform SDK defines the following error codes:

- AUTHENTICATION_ERROR
- BLE_ERROR
- BLE_ERROR_BATTERY_STATUS_FAILURE
- BLE_ERROR_BT_DISABLED
- BLE_ERROR_CHARACTERISTIC_READ_ERROR
- BLE_ERROR_DEVICE_ALREADY_CONNECTED
- BLE_ERROR_INVALID_WEARABLE_TOKEN
- BLE_ERROR_NOT_CONNECTED
- BLE_ERROR_SCANNER_ALREADY_STARTED

- BLE_ERROR_SINGLE_DEVICE_ALREADY_CONNECTED
- BT_ERROR
- CLOUD_CREDENTIALS_DECODE_ERROR
- CLOUD_ERROR
- CLOUD_HTTP_ERROR
- CLOUD_INTERRUPTED_ERROR
- CLOUD_JSON_ERROR
- CLOUD_TIMEOUT_ERROR
- CLOUD_UNEXPECTED_ERROR
- CLOUD_USER_NOT_LOGGED_IN_ERROR
- CREATOR
- DEFAULT_ERROR
- DUPLICATE_RECORD
- IN_PROGRESS
- INTERNAL_ERROR
- INVALID_RANGE
- LOW_BATTERY
- METHOD_NOT_ALLOWED
- NO_SPACE
- NOT_FOUND
- NOT_READY
- PRECONDITION_FAILED
- REQUEST_TIMEOUT
- RESPONSE_TIMEOUT
- TOO_LONG
- UNAUTHORIZED
- UNSUPPORTED
- WEARABLE_ERROR

CHAPTER

4 Using the Wearable Platform Body IQ APIs

This chapter provides information to get you started using the Body IQ APIs.

This chapter contains the following sections:

- About the Body IQ APIs
- Using the Body IQ APIs

4.1 About the Body IQ APIs

The Body IQ APIs collect information about the physical activity of the device wearer. The Body IQ APIs provide information about type of activity the device wearer has performed or is currently performing. This information is provided in the form of a time series, and includes the following details for each data point:

- Type of an activity - walking or running
- Activity Start timestamp

- Activity End timestamp
- Step count for interim Walking or Running intervals

Based on time series data, Body IQ APIs can compute additional values for each data point such as activity duration, distance covered or number of calories burned. Time series data is persisted in the local data store on the companion device.

4.1.1 Body IQ Time Series Overview

To compute details of the device wearer's activity, such as distance covered or calories burned, the device wearer must enter the following minimal profile details:

- Height
- Weight
- Gender

The Body IQ APIs use this information, in conjunction with data observed by the wearable device sensor technology, to enhance each data point in the time series with details specific to the device wearer. The sensors on the wearable device continuously collect activity data about the device wearer. After Body IQ initialization, the application can subscribe to receiving user activity updates and record time series capturing user activity details.

4.1.2 How Body IQ Data is Stored

Time series observed with Body IQ follows the following Data Store storage policy:

Short-term storage: up to 3 days worth of activity data can be stored on the wearable device. If the wearable device is not connected to the companion device within three days of conducting activity, the oldest data points are purged to make room for new activity information.

Mid-term storage: Time series observed and collected by the Body IQ APIs on the companion application, are stored in the local data store on the companion device. They are stored for 30 days.

Long-term storage: The application can read time series data stored on the companion device in the mid-term data store, and transfer it to the cloud data store for long-term storage using Intel Cloud services.

4.2 Using the Body IQ APIs

All Body IQ APIs are provided in the `com.intel.wearable.platform.body` package of the Wearable Platform SDK for the Android mobile platform.

This section contains the following topics:

- To Add Body IQ to Your Application
- To Initialize Body IQ
- To Create a User Profile
- To Retrieve the User Profile
- To Retrieve the User Profile
- To Subscribe to Activity Data Updates
- To Query Activity Data
- To Convert Activity Data to a JSON Object
- To Delete Activity Data from the Data Store

4.2.1 To Add Body IQ to Your Application

Step 1. Add Body IQ SDK as a dependency with Gradle.

Example: **Adding BodyIQ SDK as a Gradle Dependency**

```
compile(group: 'com.intel.wearable.platform', name: 'bodyiq', version: '0.2.2', ext: 'aar') {
    transitive = true
}
```

Step 2. Add the `packagingOptions` to the android section of your project `build.gradle`.

Example: **Adding packagingOptions to Your Project**

```
packagingOptions {
    exclude 'META-INF/LICENSE'
    exclude 'META-INF/LICENSE.txt'
    exclude 'META-INF/NOTICE'
    exclude 'META-INF/NOTICE.txt'
}
```

The artifact is only available from Intel's artifactory maven repository.

Example: **Using Intel Artifact**

```
maven {
    url 'https://ubit-artifactory-or.intel.com/artifactory/ndg-repos'
    credentials {
        username = "${artifactory_user}"
        password = "${artifactory_password}"
    }
}
```

4.2.2 To Initialize Body IQ

After including Body IQ in your project, as described above, call `Body.init(android.content.Context)` in the `onCreate()` method of your `android.app.Application`. This initializes Body IQ data store and performs other necessary set up steps.

4.2.3 To Create a User Profile

Step 1. Create a `com.intel.wearable.platform.core.model.datastore.UserIdentity` instance, and save it with `com.intel.wearable.platform.body.persistence.BodyDataStore`.

Step 2. Set biological sex, height, and weight for the current `com.intel.wearable.platform.core.model.datastore.UserIdentity`.

Example: **Creating a Body IQ User Profile**

```
import com.intel.wearable.platform.body.persistence.BodyDataStore;
import com.intel.wearable.platform.body.model.BiologicalSex;
import com.intel.wearable.platform.core.model.datastore.UserIdentity;

       // Create the CoreIQ User
       final UserIdentity identity = new UserIdentity(null, // UUID is set on save
                                                      "External Service Id",
                                                      "First Name",
                                                      "Last Name",
                                                      "Email Address",
                                                      "Phone Number");
       BodyDataStore.setCurrentUserIdentity(identity);
       BodyDataStore.setBiologicalSex(BiologicalSex.FEMALE);
       BodyDataStore.setHeight(height);
       BodyDataStore.setWeight(weight);
```

4.2.4 To Retrieve the User Profile

Use the `BodyDataStore.getCurrentProfile()` method to get the `com.intel.wearable.platform.body.model.Profile` object.

Example: **Retrieving a Body IQ User Profile**

```
import com.intel.wearable.platform.body.persistence.BodyDataStore;
import com.intel.wearable.platform.body.model.Profile;

       final Profile profile = BodyDataStore.getCurrentProfile();
```

4.2.5 To Subscribe to Activity Data Updates

To receive updates on user activity, implement a `com.intel.wearable.platform.body.listen.ActivityIntervalListener` instance and specify it as a parameter to the `Body.addActivityListener` method.

Any activity events will be supplied in the `onActivityInterval` callback method as the `ActivityInterval` object.

`BodyIQ` must be initialized before adding an `ActivityIntervalListener`.

Example: **Subscribing to Activity Data Updates**

```
import android.util.Log;
import com.intel.wearable.platform.body.Body;
import clisten.ActivityIntervalListener;
import com.intel.wearable.platform.body.model.ActivityInterval;

class Listener implements ActivityIntervalListener {
    public void onActivityInterval(final ActivityInterval interval) {
        Log.i("Listener", String.format("ActivityInterval(%s) received.", interval.uuid));
    }
}

Body.addActivityListener(new Listener());
```

4.2.6 To Query Activity Data

After some events have been processed by the.`body.ingest.BodyEventSubscriber`, you can retrieve a `.body.model.TimeSeries` from `.body.persistence.ActivityIntervalDao`. See the following example.

Example: **Querying Activity Data**

```
import java.util.Date;
import com.intel.wearable.platform.body.Body;
import com.intel.wearable.platform.body.model.ActivityInterval;
import com.intel.wearable.platform.body.model.ActivityStepInterval;
import com.intel.wearable.platform.body.model.TimeSeries;
import com.intel.wearable.platform.body.model.TimeSeriesStep;
import com.intel.wearable.platform.body.persistence.ActivityIntervalDao;

       final ActivityIntervalDao intervalDao = new ActivityIntervalDao(Body.context());
       final long now = (new Date()).getTime();
       final long seriesStart = now - (1000L * 60L * 15L);
       final long seriesEnd = now;
       // Query for activity from the last 15 minutes.
       final TimeSeries<ActivityInterval> series = intervalDao.fetchSeries(seriesStart,
                                                   seriesEnd);
```

4.2.7 To Convert Activity Data to a JSON Object

After acquiring a TimeSeries, make a single method call to convert the TimeSeries into an `org.json.JSONObject`.

Example: **Converting Activity Data to a JSON Object**

```
import java.util.ArrayList;
import com.intel.wearable.platform.body.model.ActivityInterval;
import com.intel.wearable.platform.body.model.TimeSeries;
import com.intel.wearable.platform.body.model.TimeSeriesHelper;
import org.json.JSONObject;

       final TimeSeries<ActivityInterval> series =
       new TimeSeries<>(new ArrayList<ActivityInterval>());
       final JSONObject json = TimeSeriesHelper.toJSONObject(series);
```

The `.body.model.TimeSeriesHelper#toJSONObject` produces a JSON structure that is compatible with the TimeSeries API and conforms to the JSON schema for TimeSeries data.

4.2.8 To Delete Activity Data from the Data Store

If you know the threshold for old data, then removing old `ActivityInterval` data from the database is straightforward.

Example: Deleting Activity Data from the Data Store

```
import org.joda.time.DateTime;
import com.intel.wearable.platform.body.Body;
import com.intel.wearable.platform.body.model.ActivityInterval;
import com.intel.wearable.platform.body.persistence.ActivityIntervalDao;

       final DateTime threshold = DateTime.now().minusDays(30);
       final ActivityIntervalDao intervalDao = new ActivityIntervalDao(Body.context());
       final Iterable<ActivityInterval> deletedIntervals =
                                              intervalDao.deleteOldActivity(threshold);
```

Alternatively, a default threshold of thirty (30) days ago can be much more conveniently used through `com.intel.wearable.platform.body.persistence.BodyDataStore`.

Example: Using the Default Threshold for Deleting Old Activity

```
import com.intel.wearable.platform.body.persistence.BodyDataStore;

       BodyDataStore.clearOldActivityIntervals();
```

CHAPTER

5 Using the Wearable Platform Social IQ APIs

Social IQ provides support to transfer LED and haptic notifications to the wearable device in response to an event. For example, after receiving a calendar reminder or a Time-to-Leave notification, the application can send an LED and/or vibration pattern to the wearable device to notify the device wearer of the event.

See Using the Wearable Notification APIs on page 37 for details on configuring and sending LED and vibration notifications to the wearable device.

CHAPTER

6 Using the Wearable Platform Time IQ APIs

The Curie Software Platform TimeIQ APIs implement scheduling services on Curie-based wearable devices.

This chapter contains the following sections:

6.1 Time IQ APIs Overview

This chapter provides information to help you get started using TimeIQ. Before exploring the TimeIQ APIs in depth, review following steps necessary to start using the SDK. Each of these steps will be more fully explained this chapter.

Step 1. Create an instance of the `TimeIQApi` class.

You will use this `TimeIQApi` instance to interact with the Time IQ API.

Step 2. Initialize the Time IQ API.

Step 3. Activate all modules of the `TSOPlaces Engine`.

Step 4. Initialize the `TimeIQMessageListener`.

Step 5. Implement the `TimeIQMessagelistener` methods.

Step 6. Configure the listener methods to subscribe to the requisite notifications.

This enables the TimeIQ API to update the status of events and reminders.

Step 7. Register the listener you created in the previous step.

6.1.1 About Time IQ Results

A Time IQ `Result` object provides a straightforward way to discern the outcome of any event in a Time IQ API. Two types of `Result` objects exist in the Time IQ SDK:

- Standalone Result
- ResultData

6.1.2 Using a Standalone Result

After making an API call using TimeIQ, you can check a standalone `Result` in two ways:

- To find the outcome, evaluate the `ResultCode` property of the Result.
- To determine whether the API call was successful, call the `isSuccess()` method on the `Result` object.

Example: **Using a Standalone Result**

```
Result  result = mTimeIQApi.reminderManager.getReminder(ReminderId);
ResultCode resultCode = result.getResultCode();
bool reminderRetrieved = result.isSuccess();
```

6.1.3 Using ResultData

The `ResultData` object extends the TimeIQ `ResultClass`. The `ResultData` object is identical to `Result` with the exception of one property:

```
private final T data;
```

Certain types of API calls in the TimeIQ package promote the information from the call into `ResultData`. When this occurs, complete the following steps:

Step 1. Compose API calls using `ResultData`.

Step 2. Call the `getData()` method to retrieve the information you're attempting to retrieve through the API. In this example, the information is reminders.

Example: **Using ResultData**

```
ResultData<Collection<IReminder>> allActiveRemindersResultData =
       YourTimeIQRemindersUtilityClass.getAllActiveReminders();

Collection<IReminder> remindersCollection = allActiveRemindersResultData.getData();
```

Tip: In this example, a good practice is to first call `isSuccess` on the `ResultsData` object, and then call the `getData()` method. This ensures no error occurs upon execution of the API call.

The following table summarizes common result codes you can use.

Table 1: Useful Result Codes

Type	Codes
General	`ResultCode.SUCCESS` `ResultCode.GENERAL_ERROR`
Search	`ResultCode.SEARCH_NO_RESULTS` `ResultCode.SEARCH_TERM_NOT_SUPPORTED`
Location	`ResultCode.LOCATION_IS_NULL`
Time to Leave (TTL)	`ResultCode.TTL_IS_OVERDUE`

6.2 Initializing TimeIQ In Your Application

Before accessing any of the scheduling services Curie Software Platform TimeIQ APIs provide, initialize `TimeIQApi` class in your application. Call the `TimeIQApi.init()` method on an instance of `TimeIQApi`, supplying three parameters:

- An instance of your authentication provider
- The application context
- The URL for the Curie Platform Cloud Service

See the following example.

Example: **Initializing TimeIQ**

```
import com.intel.wearable.platform.timeiq.api.TimeIQApi;
. . .
public class YourService extends Service implements IMessageListener {
    public static final TimeIQApi mTimeIQApi = new TimeIQApi();
    private IAuthCredentialsProvider myAuthProvider;

    @Override
    public void onCreate() {
        super.onCreate();
        Context context = this.getApplicationContext();
        //Use the AuthCredentialsProvider of your choice to create auth credentials
        myAuthProvider = AuthUtil.getAuthProvider(context);
        //Cloud server url parameter left null below intentionally for code example
        mTimeIQApi.init(context, myAuthProvider, null);
    }
}
```

Note: It is important to call the `onDestroy()` method on your instance of the `TimeIQAPI` in the `onDestroy()` lifecycle method of the service or activity you create. See the `TimeIQReadme.txt` file for more implementation details about unregistering the Message Handler listener.

6.3 Handling Messages in the TimeIQ APIs

Message Handling APIs are provided in the `com.intel.wearable.platform.timeiq.api.common.messageHandler` package of the Wearable Platform Core SDK for Android. These APIs support operations for communicating information about events, reminders, Place-related notifications, and other types of alerts across the TimeIQ APIs. Message Handling APIs are provided with the `IMessageHandler` interface.

This section includes the following topics:

- About the Message Handler
- Registering a Listener with the Message Handler
- To register messages

- Initializing the Message Handler
- Listening to Messages from Message Handler

6.3.1 About the Message Handler

`IMessageHandler` manages a range of tasks including:

- `register()`
  Registers the listener to the messages sent by the Message Handler
- `init()`
  Initialization of Message Handler, which should be called after registering a listener
- `unregister()`
  Unregisters the specified listener from the Message Handler
- `dispose()`
  Clears the Message Handler and disposes any pending messages

Use `getMessageHandler()` method to obtain an instance of the Message Handler to manage messages across the TimeIQ APIs.

Example: **TimeIQ Message Handling**

```
import com.intel.wearable.platform.timeiq.api.common.messageHandler.IMessageHandler;

. . .
public class YourService extends Service implements IMessageListener {
    public static final TimeIQApi mTimeIQApi = new TimeIQApi();

    @Override
    public void onCreate() {
        super.onCreate();
        . . .
        IMessageHandler messageHandler = mTimeIQApi.getMessageHandler();
    }
}
```

6.3.2 Registering a Listener with the Message Handler

You can listen for `TimeIQ` messages in your application. Each of the TimeIQ APIs sends messages to communicate when an event occurs or when a reminder or other type of notification is triggered. A good practice is to register to listen to messages using a sticky service. This ensures that you will receive messages even if your application temporarily stops running.

6.3.3 To register messages

Step 1. First implement the `IMessageListener` interface in your class.

Step 2. Register to listen to messages using the `register()` method.

The following example demonstrates how to register to listen to messages using a service.

Example: **Registering a Listener with Time IQ Message Handler**

```
import com.intel.wearable.platform.timeiq.api.common.messageHandler.IMessageHandler;

. . .
public class YourService extends Service implements IMessageListener {
    public static final TimeIQApi mTimeIQApi = new TimeIQApi();

    @Override
    public void onCreate() {
        super.onCreate();
        . . .
        IMessageHandler messageHandler = mTimeIQApi.getMessageHandler();
               //register the current class to listen to messages
               messageHandler.register(this);

    }
}
```

Important: Unregister the listener in the `onDestroy()` `lifecycle` method of the service or activity you create. See

the `TimeIQReadme.txt` file for more implementation details about unregistering the Message Handler listener.

6.3.4 Initializing the Message Handler

Initialization of the Message Handler should be called after registering at least one listener. All messages sent before `init()` will be accumulated. When `init()` is called, all of the accumulated messages will be received by the listener if a listener has been registered. If `init()` is called and no listener has been registered yet, the accumulated messages will be lost.

Example: **Initializing the Time IQ Message Handler**

```
import com.intel.wearable.platform.timeiq.api.common.messageHandler.IMessageHandler;

. . .
public class YourService extends Service implements IMessageListener {
    public static final TimeIQApi mTimeIQApi = new TimeIQApi();

    @Override
    public void onCreate() {
        super.onCreate();
        . . .
        IMessageHandler messageHandler = mTimeIQApi.getMessageHandler();
                                         messageHandler.register(this);
       //start receiving messages
       messageHandler.init()

    }
}
```

6.3.5 Listening to Messages from Message Handler

`YourService`, or any other class that you registered to the messageHandler, should implement `IMessageListener`.

The `IMessageListener` interface contains one method that you should implement `void onReceive(IMessage message)`.

The message you receive has two methods: `getType` and `getData`.

Below is a list of the message types that may be returned when you call the `getType` method:

- ON_REMINDERS_TRIGGERED
  A reminder was triggered. For this `MessageType` the Message Data is `RemindersResult`.
- ON_EVENT_TRIGGERED
  TTL triggered for an event. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.
- ON_EVENT_START
  A defined event has started. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.
- ON_EVENT_END
  A defined event has ended. For this `MessageType` the Message Data is `TSOEventTriggeredMsg`.

6.4 Using the Reminders APIs

Reminders are a central feature within the TimeIQ SDK and can be used to complement functionality in the Events, Places, and User State Time IQ APIs. You can create some types of Reminders without reference to any event, place or user state. For example, you can include a functionality that creates a notification prompting the device wearer to complete a task at a particular time. This is a called a Do reminder.

This section contains the following topics:

6.4.1 About the Reminders Manager

The `IRemindersManager` interface provides a mechanism for creating, adding, storing and retrieving reminders. You can also use `IRemindersManager` to edit or remove existing reminders.

A reminder is always created with a trigger, which enables the developer to set the timing for the reminder to activate.

When it is time for the reminder to activate, the developer will get a message through the Message Handler, with a message with type ON_REMINDERS_TRIGGERED

6.4.2 Reminder and Trigger Types

The following are supported reminder types:

- DoReminder
  Reminds device wearer to perform an action specified in a user-defined string.
- CallReminder
  Reminds device wearer to call a specified contact.
- NotificationReminder
  Reminds device wearer that a notification will be sent to specified contact.

The following are supported trigger types:

- ChargeTrigger
  Based on pre-defined charging conditions
- MotTrigger
  Based on pre-defined means of transport (MOT) conditions
- PlaceTrigger
  Based on pre-defined actions for a specific location
- TimeTrigger
  Specific pre-defined time

6.4.3 To Create a Reminder

Step 1. Create the trigger for the reminder.

Step 2. Create a reminder with the trigger.

The following example adds a reminder to turn on the car light. This reminder is triggered when the user starts driving.

Example: **Creating a Reminder**

```
import com.intel.wearable.platform.timeiq.api.common.protocol.enums.MotType;
import com.intel.wearable.platform.timeiq.api.reminders.IReminder;
import com.intel.wearable.platform.timeiq.api.reminders.ReminderBuildException;
import com.intel.wearable.platform.timeiq.api.reminders.doReminder.DoReminder;
import com.intel.wearable.platform.timeiq.api.triggers.ITrigger;
import com.intel.wearable.platform.timeiq.api.triggers.TriggerBuildException;
import com.intel.wearable.platform.timeiq.api.triggers.mot.MotTransition;
import com.intel.wearable.platform.timeiq.api.triggers.mot.MotTrigger;

...
public IReminder createReminder(){
    //create a trigger:
       ITrigger trigger = null;
    try {
       trigger = new MotTrigger.MotTriggerBuilder(MotType.CAR, MotTransition.START).build();
       } catch (TriggerBuildException e) {  }
        // trigger for mot to change to car started (will not be triggered now,
        // if you are currently driving)
        // create a reminder
       IReminder reminder = null;
    if (trigger != null) {
        try {
            reminder = new DoReminder.DoReminderBuilder(trigger,
                                        "Turn on the carlights").build();
        }catch (ReminderBuildException e) {  }
    }
}
```

6.4.4 To Add a Reminder

Step 1. Get a reference to the `RemindersManager` object from an instance of the `TimeIQApi` object.

Step 2. Call the `addReminder` method to include the reminder you want to add as a parameter in the method call.

The output from the `addReminder` method is assigned to a `Result` object.

Step 3. Verify that the reminder was successfully added.

Check the `isSuccess()` method on the `result` object.

Example: **Adding a Reminder**

```
import com.intel.wearable.platform.timeiq.api.common.result.Result;
import com.intel.wearable.platform.timeiq.api.reminders.IReminder;
import com.intel.wearable.platform.timeiq.refapp.YourService;

...
private String addReminder(){
    IReminder reminder = createReminder();
    String message = null;
    if(reminder != null) {
        // add the reminder
     Result result = YourService.mTimeIQApi.getRemindersManager().addReminder(reminder);
       // YourService.mTimeIQApi - see below
     message = result.isSuccess() ?
                "reminder_was_added" : // reminder added OK
    "reminder_was_not_added" + result.getMessage(); // reminder was not added with the error
  }
    return message;
}
```

6.4.5 To Edit a Reminder

Editing a reminder requires that you create a new reminder with your changes and replace the old version with the updated version.

Step 1. Create a new reminder.

Save the details of the existing reminder.

Step 2. Remove the old version of the reminder.

Step 3. Add the new reminder including the changes you made.

6.4.6 Adding Snooze Functionality for Reminders

Snooze functionality is commonly recognized as an alarm clock feature. You usually set an alarm clock to wake you from sleep, and then enable the snooze feature to allow you to “snooze” a few minutes past the alarm. Subsequent alarms sound at regular intervals until you are done snoozing, and disable all alarms associated with your wake time.

The Time IQ Reminders snooze functionality works in a similar way. The device wearer sets a time to be reminded of an upcoming task or event, and then enables the snooze feature. When the first reminder notification is activated, the device wearer can ignore the alarm knowing that subsequent alarms will pester him or her at regular intervals. When the device wearer is ready to act on the reminder, he or she disables all alarms associated with the task or event.

6.4.7 To Snooze Reminders

To snooze a reminder, you will first need to get the snoozing options.

Step 1. Call the `getSnoozeOptions` method at the Reminders Manager, and pass the reminder ID.

You will get a `ResultData` containing a list of snooze options.

Step 2. Ask the user to pick a `SnoozeOption`.

Step 3. Call the `snoozeReminder` method at the Reminders Manager.

Pass the reminder ID and the `SnoozeOption` that the user picked to snooze the reminder.

6.4.7.1 Supported Snooze Options

The snooze options are relevant to the specific reminder, and ordered by importance. they take into account the state of the user, the type of reminder and the type of the trigger. The following are supported snooze options:

- `WHEN_CHARGING`
  Next time the user will be charging his device
- `FROM_CAR`
  Next time the user will start driving
- `NEXT_DRIVE`
  Next time the user will start driving
- `DEFINE_HOME`
  Next time the user will arrive home, but home is not yet defined. in this case, we recommend that the developer will prompt the user to define his home

Tip: For FROM_PLACE, the place can be obtained by casting the snoozeOption to PlaceSnoozeOption, and using the getPlaceId() method on the PlaceSnoozeOption instance. Then use the PlaceRepo to obtain the place itself

- `DEFINE_WORK`
  Next time the user will arrive to work, but work is not yet defined. In this case, you can prompt the user to define his work.
- `FROM_PLACE`
  Next time the user arrives to a specific place.
- `NEXT_TIME_AT_CURRENT_PLACE`
  Next time the user gets to the current place
- `LEAVE_CURRENT_PLACE`
  When the user leaves the current place

Tip: For IN_X_MIN, the delay offered can be obtained by casting the snoozeOption to TimeDelaySnoozeOption, and using the getDelayMinutes() method on the TimeDelaySnoozeOption instance.

- `IN_X_MIN`
  In a specific time delay
- `TIME_RANGE`
  In a specific time range.

For `TIME_RANGE`, the delay offered can be obtained by casting the `snoozeOption` to `TimeRangeSnoozeOption`. Use the `getTimeRange()` method on the `TimeDelaySnoozeOption` instance to get the `SnoozeTimeRange`.

6.4.7.2 Supported Snooze TimeRange Types

- THIS_MORNING
- TODAY
- THIS_EVENING
- THIS_NIGHT
- TOMORROW_MORNING

6.4.8 To End a Reminder

Once the user sees the reminder and acts upon it, you should mark the reminder as ended.

To end the reminder, call the `endReminder` method at the Reminders Manager, and pass the reminder ID along with the `ReminderEndReason` type.

The following are supported `ReminderEndReason` types:

- Dismiss
  Device wearer dismissed the reminder.
- Done
  Device wearer completed the reminder.

See the following example.

Example: **Ending a Reminder**

```
public static Result endReminder(IReminder reminder, ReminderEndReason
    reminderEndReason) {
    return endReminder(reminder.getId(), reminderEndReason);
}

public static Result endReminder(String reminderId, ReminderEndReason
    reminderEndReason) {
    return YourService.mTimeIQApi.getRemindersManager().endReminder(reminderId,
    reminderEndReason);
```

6.5 Using the Events APIs

The Events APIs are tailored to condition reminders upon a scheduled occurrence at a particular place or location. For example, a user may want to add an event to help plan a workout at the gym. The user can specify the gym's location, the time and the date of the workout.

This section contains the following topics:

- Using the Events Engine
- Using the EventBuilder
- To Add an Event
- To Edit a Reminder
- Adding Snooze Functionality for Reminders
- Adding Snooze Functionality for Reminders

6.5.1 About Time to Leave (TTL) Notification

One useful feature of the Events API is the Time-To-Leave (TTL) notification. This notification sends the user an alert before the start time of the event to inform the device wearer that it is time to depart for the event. A TTL notification is automatically generated upon the creation of an event. Although this can be used as standalone functionality, it's important to keep in mind that this notification is automatically sent for each event that is added on `TimeIQ`.

6.5.2 About Calendar Integration

The Events APIs integrate with basic calendar functionality. The device wearer is able to define a set of read calendars from which the TimeIQ SDK will upload events and generate TTL notifications accordingly. (See Calendar API for more details).

Upon creation of certain types of events, such as a `BeEvent`, you can add the event to the calendar. For the `BeEvent`, use the `addToCalendar` option when creating the event using the `BeEvent` builder. The event will then also be added to the calendar.

6.5.3 Using the Events Engine

Use the Time IQ Events Engine interface to add and manage configurable reminders. Defined events are ready for customization, and are associated with notifications to the device wearer at key time points relevant to a scheduled event.

In addition, the Events Engine interface enables user interaction upon receiving the different notifications sent for each event. For example, upon receiving TTL notification for a doctor's appointment, the user can choose the time snooze option, and receive a subsequent notification at the specified time.

6.5.4 Using the EventBuilder

Use the `EventBuilder` function to create a new event. `EventBuilder` is designed to give you flexibility to choose the properties or attributes required for any type of event you implement in an application.

The `BaseEvent` class requires two attributes:

- Location
- Start time

Additional optional parameters are documented in the API Reference. Some of the optional parameters have default values.

6.5.5 To Add an Event

Step 1. Get a reference to the application Event Engine by calling a method on the `TimeIQApi` object instance.

Step 2. Create the event using the appropriate Event Builder and calling the `addEvent` method. Place the event as an argument inside that method.

Step 3. Verify the successful completion of the operation by examining the outcome returned through the Result object.

Example: **Adding an Event**

```
IEventsEngine eventsEngine = YourYourService.mTimeIQApi.getEventsEngine();

TSOPlace eventLocation = new TSOPlace(40.764367, -73.981076,"New York", "New York");

BeEvent doctorAppEvent = new BeEvent.BeEventBuilder(eventLocation, System.currentTimeMillis()
    + TEN_MINUTES).duration(DURATION_TWENTY_MINUTES).addToCalendar(true).build();

Result res =  eventsEngine.addEvent(doctorAppEvent);

assertEquals(res.getResultCode(), ResultCode.SUCCESS);
```

6.6 Using the Places APIs

Locations can be used as triggers for reminders or for other TimeIQ API events. Use the `TimeIQ` Places APIs to enable the device wearer to input custom locations.

The device wearer manages his or her places repository in the mobile application. The device wearer can manually add places to, or remove places from, the application. The device wearer can then access the collection of all the known places. Known places may have been added by the device wearer, or automatically added by the SDK when resolving the location of detected calendar events.

The `TimeIQ` Places APIs also provide the device wearer access to the two potentially auto-detected places: home and workplace.

This section contains the following topics:

- Using the Places Repository
- About the TSOPlaces Engine
- Using TSOPlace Builder
- Using IPlaceRepo
- To Add a New Place
- To Delete a Place
- To Retrieve All Places
- To Retrieve a Place by ID
- Managing Special Places: Home and Work

6.6.1 Using the Places Repository

Implement `IPlaceRepo` when you want to store a list of places within your application. After a location has been resolved and added to the device wearer Places, you can use the `IPlaceRepo` interface to retrieve the location. This is useful when you want to enable the device wearer to select a location parameter when creating an event, or setting an event to trigger a location-based reminder.

6.6.2 About the TSOPlaces Engine

Use `TSOPlaces` to create preferred places defined by the device wearer, and store them in the Places Repository with their corresponding semantic keys.

Important: You must start the TSOPlacesEngine upon the first use of the TimeIQ SDK in the application.

6.6.3 About the TSOPlace Interface

The interface `com.intel.wearable.platform.timeiq.api.places.datatypes.TSOPlace` represents a place in the `TimeIQ` SDK.

The `TSOPlace` object contains the following data items:

- `TSOCoordinate`
  Structure that holds the geographic Latitude/Longitude coordinates of the place.
  Example: 37.3865906,-121.9812071
- `Address`
  String that represents the street address of the place.
  Example: 3100 Lakeside Drive, Santa Clara, CA 95054, USA
- `Name`
  String that represents the name of the place.
  Example: The Plaza Suites Hotel
- `ManualPlaceSource`
  Holds an indication to the entity that generates the place.
- `ManualPlaceSource.USER`
  For places added explicitly by the application developer.
- `ManualPlaceSource.CALENDAR`
  For places added implicitly by the calendar resolver module within the SDK.

6.6.4 Using TSOPlace Builder

The device wearer defines a preferred place by entering a physical address, and providing a place name. The `TSOPlaces` engine determines the geolcation coordinates for the place, and associates the coordinates with the name defined by the device wearer.

6.6.5 Using IPlaceRepo

The `IPlaceRepo` interface, `com.intel.wearable.platform.timeiq.api.places.IPlaceRepo`, is the main entry point for the Places Repository APIs. Obtain the `iPlaceRepo` interface from the `com.intel.wearable.platform.timeiq.api.TimeIQApi` object by calling its `getPlacesRepo()` method.

Example: **Obtaining the iPlaceRepo Interface**

```
IPlaceRepo placesRepo = mTimeIQApi.getPlacesRepo();
```

Use this interface to add new places, delete existing ones and retrieve a collection of the existing places in the SDK.

These places can be used to create place-based triggers and reminders. For example, you can create a reminder to send a text to a friend when you leave your workplace and are headed to your meeting place.

6.6.6 To Add a New Place

Step 1. Create a new object called TSOPlace.

Step 2. Generate a Geographic coordinate object, address and name.

Example: **Creating a New Place**

```
TSOCoordinate placeCoordinate = new TSOCoordinate(37.3865906,-121.9812071);
String placeAddress = "3100 Lakeside Drive, Santa Clara, CA 95054, USA";
String placeName = "The Plaza Suites Hotel";
TSOPlace newPlace = new TSOPlace(placeName, placeAddress, placeCoordinate);
```

Step 3. After creating a place, add it to the repository.

Step 4. Gets a `ResultData` object with a `PlaceID` object, which is a place identifier generated by the SDK. See example on the next page.

Example: **Obtaining a ResultData Object with a PlaceID**

```
ResultData<PlaceID> placeIDResultData = placesRepo.addPlace(newPlace);
if (placeIDResultData.isSuccess())
{
   Log.v(TAG, "New place was added successfully.");

}
```

6.6.7 To Delete a Place

Use the same `placeID` that was generated while adding the place to the repository.

Example: **Deleting a Place**

```
Result result = placesRepo.removePlace(placeId);
if (result.isSuccess())
{
   Log.v(TAG, The place was removed successfully.");
}
```

6.6.8 To Retrieve All Places

To get the list of all places in the Places repository, both user defined and those resolved from calendar events, call the `getAllPlaces` method.

Note: Potential auto-detected home and work, will not be returned by this API method.

Example: **Retrieving All Places**

```
Collection<TSOPlace> myPlacesList = null;

ResultData<Collection<TSOPlace>> placesListResult = placesRepo.getAllPlaces();

if(placesListResult.isSuccess()) {

    myPlacesList  = placesListResult.getData();

}
```

6.6.9 To Retrieve a Place by ID

If you hold a `PlaceID` object, you can retrieve the relevant `TSOPlace` from the repository using the `getPlace()` method.

Example: **Retrieving a Place by ID**

```
ResultData<TSOPlace> placeResultData = placesRepo.getPlace(placeId);
if(placeResultData != null && placeResultData.isSuccess())
{
     TSOPlace place = placeResultData.getData();
}
```

6.6.10 Managing Special Places: Home and Work

You can add to the SDK two special places: home and work.

To add one of these places, use a specific API which enables adding a place with a argument called `SemanticKey`. You have two options:

- `SemanticKey.HOME` for home
- `SemanticKey.WORK` for work

Example: **Adding Home or Work Places**

```
TSOCoordinate placeCoordinate = new TSOCoordinate(37.3866427,-121.9886155);

String placeAddress = "1287 Oakmead Pkwy, Sunnyvale, CA 94085, USA";

String placeName = "Home sweet Home";

TSOPlace place = new TSOPlace(placeName, placeAddress, placeCoordinate);

if (placesRepo.addPlace(place, SemanticKey.HOME)) {

       Log.v(TAG, "My Home was set successfully.");

}
```

You can get the special places by using this method. See example on the next page.

Example: **Getting Special Places**

```
ResultData<PlaceID> getPlaceId(SemanticKey semanticKey);
```

Once you have the `PlaceID`, you can get the `TSOPlace` object.

In addition to these special places, the system might generate auto-detected home and auto-detected work, with following semantic keys:

- SemanticKey.AUTODETECTED_HOME
- SemanticKey.AUTODETECTED_WORK

You can get access to these auto-detected places (if they exist), using the same method. See the following example.

Example: **Generating Places Using Semantic Keys**

```
ResultData<PlaceID> getPlaceId(SemanticKey semanticKey);
```

6.7 Using the User State APIs

The User State APIs provide a means to get the current state of the user, as well as setting a notification when the user state changes.

The user state includes the following data:

- Means of transport (MOT)
  CAR, WALK, or STATIONARY
- Visited places
  A list of places that the user is currently visiting

This section contains the following topics:

- About the UserState Object
- Getting User State MOT Data
- Getting Visited Places Data
- About the User State Manager
- Getting the Current User State

- Registering a Listener to User State Changes
- Unregistering a Listener for User State Changes

6.7.1 About the UserState Object

The object `com.intel.wearable.platform.timeiq.api.user state.UserState` object holds information about the user state as created by the SDK at a specific time. The SDK creates a new `UserState` object each time user state data changes. You usually will not create `UserState` objects by yourself, but rather obtain them from the `UserStateManager`.

You can get the creation time of the `UserState` by calling its `getTimeStamp()` method.

Example: **Getting the UserState Creation Time**

```
UserState state = obtainState()
long creationTime = state.getTimeStamp();
```

6.7.2 About UserStateData

Each data item that comprises the `UserState` object can be obtained in the form of

`com.intel.wearable.platform.timeiq.api.userstate.User StateData<T>`

where T is a specific data type.

To get the actual data from a `UserStateData` object call its `getData()` method.

In addition to the actual data, the `UserStateData` object also contains the timestamp in which the specific data was changed. **Note:** This is different from the timestamp in which the `UserState` object was changed.

Note also that the `UserState` object may return `null` when asked for a specific data if it has no information about it.

Example: **Getting Data from a UserStateData Object**

```
UserStateData<MotType> motData = state.getMot();
if (motData != null) {
  long time = motData.getTimeStamp();
  MotType mot = motData.getData();
}
```

6.7.3 Getting User State MOT Data

The means of transport (MOT) data contains the device wearer means of transport such as walking, driving, or stationary. The MOT is represented by the `com.intel.wearable.platform.timeiq.api.common.protocol.enums.MotType` enumeration.

The possible values of `MotType` are:

- `MotType.STATIONARY`
- `MotType.WALK`
- `MotType.CAR`

Note: The value `MotType.PUBLIC_TRANSPORT` is not currently implemented. This MOT type may be implemented in a future release.

Example: **Getting Means of Transport (MOT) Data**

```
UserStateData<MotType> motData = state.getMot();
if (motData != null) {
  MotType mot = motData.getData();
  if (mot.equals(MotType.WALK)) {
              Log.v(TAG, "Look mom, I'm walking!");
   }
}
```

6.7.4 Getting Visited Places Data

The visited places data is a list of `PlaceIDs`that the user is actively visiting. In the TimeIQ SDK, visiting means device wearer remains within his or her geographic bounds for a

significant duration. This data is represented by the class `com.intel.wearable.platform.timeiq.api.user state.VistedPlaces`.

For information about `PlaceIDs` and adding new places, see For more information see "Using the Places Repository" on page 83.

Example: **Getting Visited Places Data**

```
UserStateData<VisitedPlaces> visitsData = state.getVisits();
if (visitsData != null) {
   VisitedPlaces visits = visitsData.getData();
   for (PlaceID id : visits) {
               if (id.getSemanticKey.isHome()) {
                      Log.v(TAG, "Honey, I'm home!");
                      break;
               }
  }
}
```

6.7.5 About the User State Manager

The interface `com.intel.wearable.platform.timeiq.api.userstate.IUserStateManager` is the main entry point for the User State APIs. Obtain this interface from the `com.intel.wearable.platform.timeiq.api.TimeIQApi` object by calling its `getUserStateManager()` method.

Example: **Obtaining the User State Manager**

```
IUserStateManager stateManager = mTimeIQApi.getUserStateManager();
```

Using this interface, you can get the device wearer current state, register a listener to be notified when a state change occurs, and also unregister a listener to stop being notified.

6.7.6 Getting the Current User State

Obtain the current user state by calling the `IUserStateManager` method `getCurrentState()`. This method returns a `UserState` object which contains the most recently updated user state data.

Example: **Getting Recently Updated User State Data**

```
ResultData<UserState> stateResult = stateManager.getCurrentState();
if (stateResult.isSuccess()) {
   UserState state = stateResult.getData();
}
```

6.7.7 Registering a Listener to User State Changes

A new UserState object is created whenever some user state data is changed. To be notified when a new `UserState` object is created, call the `IUserStateManager` method `registerForStateChanges()` and pass an instance of: `com.intel.wearable.platform.timeiq.api.userstate.IUserStateChangeListener`. The `registerForStateChanges()` method returns the current `UserState` object. This is the same object that would be returned from a call to `getCurrentState`. Returning the current `UserState` object enables you to get the current state and to register for changes in one call.

***Example:* Registering a Listener to User State Changes**

```
class MyUserStateChangeListener implements IUserStateChangeListener {
   public void onStateChanged(UserState oldState, UserState newState, UserStateChanges
changes) {
     if (changes.isChanged(UserStateType.MOT)) {
        Log.v(TAG, "Rangers citadel to team echo, detected MOT change.");
     }
   }
}
IUserStateChangeListener listener = new MyStateChangeListener();
ResultData<UserState> stateResult = stateManager.registerForStateChanges(listener);
if (stateResult.isSuccess()) {
   Log.v(TAG, "Team echo to rangers citadel, state monitor installed successfully.");
}
```

The `onStateChanged` callback method parameters are:

- `oldState`
  The `UserState` object just before the change.
- `newState`
  The `UserState` object after the change.
- `changes`
  An instance of `com.intel.wearable.platform.timeiq.api.user state.UserStateChanges` that describes the data changes between the old and the new states.

6.7.8 Unregistering a Listener for User State Changes

To unregister any previously registered listener, use `IUserStateManger` method `unregisterForStateChanges()`. Unregister a listener when listener notifications about user state changes are no longer necessary.

***Example:* Unregistering a Listener for User State Changes**

```
stateManager.unregisterForStateChanges(listener);
```

6.8 Using Route APIs

The Route APIs provide a structured way to get route information that can be used on its own, or in tandem with other TimeIQ APIs.

The `IRouteProvider` interface specifies methods that can be used to initiate API calls to obtain routing information.

Tip: Though the RouteProvider uses a cache, it is a good practice to call the methods of IRouteProvider from an AsyncTask, since the call might take some time to complete execution.

You can call the following methods from RouteProvider:

- `getTTL()`
  Gets the TTL (Time to Leave) from a trip origin, in order to arrive at a certain time at a destination.
- `getETA()`
  Gets the ETA (Estimated Time of Arrival) from a trip origin to the destination for a given departure time.

The following example demonstrates how to get a TTL route from a trip origin to a destination for an event that will start in 2 hours. This example specifies the preferred MOT as driving.

***Example:* Using the Route APIs**

```
import com.intel.wearable.platform.timeiq.api.common.protocol.datatypes.location.
    TSOCoordinate;
import com.intel.wearable.platform.timeiq.api.common.protocol.enums.TransportType;
import com.intel.wearable.platform.timeiq.api.common.result.ResultData;
import com.intel.wearable.platform.timeiq.api.route.IRouteProvider;
import com.intel.wearable.platform.timeiq.api.route.TtlRouteData;
import com.intel.wearable.platform.timeiq.refapp.YourService;

import java.util.concurrent.TimeUnit;


    public TtlRouteData getTTL(TSOCoordinate origin, TSOCoordinate destination){
       // get IRouteProvider
       IRouteProvider routeProvider = YourService.mTimeIQApi.getRouteProvider();
       // arrival time is 2 hours from now
       long arrivalTime = System.currentTimeMillis() + TimeUnit.HOURS.toMillis(2);

       // gets the TTL route from origin to destination (preferably by car)
       ResultData<TtlRouteData> ttlRouteDataResultData =
                 routeProvider.getTTL(origin, destination, arrivalTime, TransportType.CAR);
       // returns the routeData or null if unsuccessful
       return ttlRouteDataResultData.isSuccess() ? ttlRouteDataResultData.getData() : null;
    }
```

You can call the method with `preferredTransportType` as `null`. In this case the API will return the route with the most appropriate transport type for this route.

The method returns a `ResultData` which upon success, contains a `TtlRouteData` or `EtaRouteData`, depending upon the specific method you called.

The `RouteData` contains, along other useful methods, the `getRouteSegments()` method, which returns all the route segments that constitute the route to the destination. The route to the destination is defined when the device wearer specifies the alert time.

The list of `RouteSegment` is ordered by time, from the time to leave until the arrival time.

The segments contains useful methods such as `getSegmentDuration()`, which returns the duration of the segment. The `getRouteInfo() method` which, if it is a travel segment, contains the route information such as the `getTrafficIndication()` method.

Each segment has a type. Supported `SegmentTypes` are:

- TimeToTTL
  Time to Time to Leave (TTL)
- OriginInDoor
  `inDoor` navigation at trip origin
- TimeToCar
  Time to get to the car
- Travel
  Travel with desired means of transport (MOT)
- Park
  Park near destination + get from car to building
- DestinationInDoor
  `inDoor` navigation at destination

6.9 Using the Calendar Details APIs

The Calendar Details APIs enable all the calendar interaction required by the Time IQ SDK.

Through the Calendar Details API, you can retrieve the list of available calendars on the device, define the set of calendars to read events from, and the write calendar for the Time IQ SDK to update.

To use the Calendar Details APIs, first get the reference by calling the `getCalendarsDetailsProvider()` method on the `TimeIQApi` object instance. Once received, it is ready for use and provides interaction with the available calendars.

Example: **Using the Calendar APIs**

```
ICalendarDetailsProvider calendarDetailsProvider =
YourYourService.mTimeIQApi.getCalendarsDetailsProvider();
ResultData<List<CalendarDetails>> resultData =
calendarDetailsProvider.getAvailableCalendars();

if (resultData.isSuccess()) {
List<CalendarDetails> availableCalendarsDetails = resultData.getData();
// add usage of the available calendars
}
```

CHAPTER

7 Using the Cloud Services SDK

The Cloud Services SDK provides a communication layer between the Intel Software Platform for Curie and Intel Cloud Services. The following sections provide details and code samples for the Cloud Services SDK in an Android environment.

7.1 Understanding the Cloud Services SDK

Features of the Cloud Services SDK include authentication, user and device profile storage, data storage, Binary Large Objects (BLOB) storage, native login, notification, time series data storage, error handling and logging. In addition to hiding the handling of complex protocols such as OAuth, Google Cloud Messaging and the Apple Push Notification service, the Cloud Services SDK handles header formatting, auto token refresh, background requests and queuing, device power management, and security review and scan. The Cloud Services SDK consists of the following modules.

- The Cloud Core module provides user authentication, cloud request scheduling, cloud response processing and error handling functionality used by all of the other Cloud Services SDK modules. It is packaged in `com.intel.wearable.cloudsdk.core`. Authenticating with the Cloud Services SDK on page 102 and Handling Cloud Services SDK Errors on page 110 contain additional information.
- The User Profile Store module provides storage and retrieval of a custom JSON documents that contain details about the authenticated user. It is packaged in `com.intel.wearable.cloudsdk.profilestore`. Accessing User Profiles on page 111 contains additional information.
- The Document Store module provides the functionality to upload, store and retrieve general JavaScript Object Notation (JSON) documents. It is packaged in `com.intel.wearable.cloudsdk.docstore`. Accessing Document Stores on page 113 contains additional information.
- The Device Profile Store module provides storage and retrieval functionality for custom JSON documents that contain details about the companion device used to connect to the cloud. It is packaged in `com.intel.wearable.cloudsdk.deviceprofilestore`. Accessing Device Profiles on page 114 contains additional information.
- The BLOB Store module provides upload and download capabilities for BLOB data and software assets. It

is packaged in `com.intel.wearable.cloudsdk.blobstore`. Accessing BLOB Data and Software Assets on page 116 contains additional information.

- The Time Series module provides storage, retrieval and analysis of time series data points. It is packaged in `com.intel.wearable.cloudsdk.timeseries`. Accessing Times Series Data on page 123 contains additional information.

This chapter documents procedures with sample code that can be followed to perform certain tasks that pertain to Cloud Services interactions. For complete details on the Cloud Services SDK for Android, see the *Cloud Services SDK Java API Reference*.

7.2 Using the Cloud Services SDK

To use the Cloud Services SDK, you must incorporate the Cloud Core module into your application either directly, or via an implicit dependency when using one of the other Cloud Services SDK modules listed in Understanding the Cloud Services SDK on page 97. The first step when using the SDK to connect to the cloud is to authenticate and log in the connecting user by calling the appropriate authentication provider. Once the user is logged in, the desired action can be taken. Details on calling the appropriate authentication provider are documented in Authenticating with the Cloud Services SDK on page 102.

The following sample code illustrates one option to upload a file to the BLOB store in the cloud. Note that both the

Cloud Core module and the BLOB store module are imported. The code does the following:

1. The native user name / password authentication provider `CloudUAAAuthProvider` is instantiated and the user is logged in successfully.
2. The BLOB store is initialized with the `CloudBlobStore` class.
3. A file is created
4. The file is uploaded to the Cloud.
5. Final responses to the upload are defined.

See the following example.

Example: Uploading a File to the BLOB Data Store

```
   ...
        // create the desired auth provider
        final CloudUAAAuthProvider authProvider =
       new CloudUAAAuthProvider("https://authserver1.wearables.host.com",
            "ndg1", "ndg1SecRet", this);
        // login the user (note: sample account below exists already)
        authProvider.login("sophieteste@email.com", "inteltest123A~",
               new ICloudAuthLoginCb() {
            @Override
            public void onFailed(CloudError cloudError) {
                Log.e(TAG, cloudError.getMessage());
            }
            @Override
            public void onSuccess() {
                // login succeeded, upload a file
                // first, initialize the blob store
                CloudBlobStore.init
                             ("https://appserver1.wearables.host.com", "1",
                                    authProvider.getCredentialIdentifier(), this);
                // then create a dummy file to upload (note: exception handling omitted)
                final File f = new File(this.getFilesDir(), "examplefile.txt");
                FileOutputStream outputStream = new FileOutputStream(f);
                outputStream.write("anything".getBytes());
                outputStream.close();
                // finally, upload the file
                CloudBlobStore.getInstance().upload(f, "exampletag",
                      new ICloudResponseCb() {
                    @Override
                    public void onSuccess(CloudResponse cloudResponse) {
                        Log.d(TAG, "Done. Response is " + cloudResponse.toString());
                    }
                    @Override
                    public void onFailed(CloudResponse cloudResponse) {
                        Log.e(TAG, "Error. Error is " + cloudResponse.getError()
                             .getMessage());
                    }
                });
            }
        });
        ...
}
```

7.3 Authenticating with the Cloud Services SDK

To authenticate a user with the Cloud Services SDK, the application must install and instantiate one of the provided authentication providers and then log in the user. Once login is deemed to be successful, the application will need to obtain a credential identifier by calling `authProvider.getCredentialIdentifier()`. This credential identifier can be passed to the various modules of the Cloud Services SDK as proof of authentication. For details on how to use the Cloud Services SDK authentication providers, see the appropriate procedures in the following sections.

Note: Each authentication provider inherits from the default authentication provider. Unless overridden, calling login() without parameters for any provider will execute the default authentication provider.

- Using the Default Cloud Authentication Provider
- Using the Intel UAA Authentication Provider
- Using the Facebook Authentication Provider
- Using the Google Authentication Provider
- Using the Application Authentication Provider

7.3.1 Using the Default Cloud Authentication Provider

The default cloud authentication provider is in the `com.intel.wearable.cloudsdk.core` package. When called, the `CloudAuthProvider` presents a login/create-account page within a full-screen, borderless web browser window. The page will accept user credentials for any of the supported login mechanisms pre-configured on the cloud by the application administrator. To use the default cloud authentication provider, follow the steps below. This sample

code creates an `ActionBarActivity` for which authentication is required.

Step 1. Implement the `CloudAuthDevApp` activity.

Example: **Implementing CloudAuthDevApp Activity**

```
import com.intel.wearable.cloudsdk.core.*;
...
public class CloudAuthDevApp extends ActionBarActivity {
```

Step 2. Instantiate `CloudAuthProvider` inside `onCreate` for the `CloudAuthDevApp` activity.

Example: **Instantiating CloudAuthProvider**

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
        ...
        CloudAuthProvider authProvider = new CloudAuthProvider
       ("https://authserver1.wearables.host.com", "ndgFightClub",
       "ndgFightClubSecRet", this);
        ...
```

Step 3. Call the `login()` function.

Example: **Calling the Login() Function**

```
authProvider.login(new ICloudAuthLoginCb() {
        @Override
        public void onFailed(CloudError err) {
          // login failed, do something
        }
        @Override
        public void onSuccess() {
          // login successful, do something else
        }
    });
```

Step 4. After a successful authentication, retrieve the credential identifier which is needed by other Cloud Services SDK modules to confirm user authentication.

Example: **Confirming User Authentication**

```
String credentialIdentifier = authProvider.getCredentialIdentifier();
```

Step 5. To customize the look of the login/create-account page, inherit from `CloudAuthWebViewClient`, make the necessary customizations, and pass an instance

of the inheriting class using `authMgr.setWebViewClient()`.

7.3.2 Using the Intel UAA Authentication Provider

Intel User Account and Authentication (UAA) is the native Intel Cloud Services authentication provider. It is located in the `com.intel.wearable.cloudsdk.core` package. Users can create accounts and login to access Intel Cloud services. To use the Intel UAA authentication provider, follow the steps below.

Step 1. Instantiate the `CloudUAAAuthProvider` authentication provider.

Example: **Instantiating Intel UAA Authentication Provider**

```
import com.intel.wearable.cloudsdk.core.*;

...

CloudUAAAuthProvider authProvider = new CloudUAAAuthProvider("https://
authserver1.wearables.host.com", "ndg1", "ndg1SecRet", androidContext);
```

Step 2. Create an account if applicable.

Example: **Creating a UAA User Account**

```
authProvider.createAccount("janedoe", "password_for_janedoe", "Jane", "Doe",
        new ICloudAuthLoginCb() {

@Override
public void onFailed(CloudError err) {
// handle the error
}

@Override
public void onSuccess() {
// maybe login the user (see next step)
}
});
```

Step 3. Authenticate and log in the user.

Example: **Authenticating and Logging in a UAA User**

```
authProvider.login("janedoe", "password_for_janedoe", new ICloudAuthLoginCb() {

@Override
public void onFailed(CloudError err) {
// handle login failure
}

@Override
public void onSuccess() {
// start using the rest of the API (e.g. blob storage)
}
});
```

Step 4. After a successful authentication, retrieve the credential identifier which is needed by other Cloud Services SDK modules to confirm user authentication.

Example: **Confirming UAA User Authentication**

```
String credentialIdentifier = authProvider.getCredentialIdentifier();
```

7.3.3 Using the Facebook Authentication Provider

The Facebook authentication provider enables a user to login using the credentials associated with a Facebook account.

The following procedure illustrates how to instantiate the Facebook authentication provider using the constructor in the `com.intel.wearable.cloudsdk.facebookauth` package. This sample code creates an `ActionBarActivity` called `CloudAuthDevApp`.

Step 1. Install the provider.

Example: **Installing the Facebook Authentication Provider**

```
compile group: 'com.intel.wearable.cloudsdk', name: 'facebookauth', version: '2.2.0'
```

Step 2. Implement the `CloudAuthDevApp` activity. See the following example.

Example: **Implementing CloudAuthDevApp**

```
import com.intel.wearable.cloudsdk.core.*;
...
public class CloudAuthDevApp extends ActionBarActivity {
        CloudFacebookAuthProvider mAuthProvider;
        ...
}
```

Step 3. Instantiate `CloudFacebookAuthProvider` inside `onCreate` for the `CloudAuthDevApp` activity and assign the instance to `mAuthProvider`.

Example: **Instantiating CloudFacebookAuthProvider**

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
        ...
        mAuthProvider = new CloudFacebookAuthProvider("https://
authserver1.wearables.host.com", "ndg1", "ndg1SecRet", this);
        ...
  }
```

Step 4. Override `onActivityResult`.

Example: **Overriding onActivityResult**

```
@Override
    public void onActivityResult(int requestCode, int responseCode, Intent intent) {
        super.onActivityResult(requestCode, responseCode, intent);
        mAuthProvider.onActivityResult(requestCode, responseCode, intent);
    }
```

Step 5. Call the `login()` function.

Example: **Calling the login() Function**

```
mAuthProvider.login(new ICloudAuthLoginCb() {
        @Override
        public void onFailed(CloudError err) {
          // login failed, do something
        }
        @Override
        public void onSuccess() {
          // login successful, do something else
        }
    });
```

Step 6. After a successful authentication, retrieve the credential identifier which is needed by other Cloud Services SDK modules to confirm user authentication.

Example: **Confirming the Facebook User Authentication**

```
String credentialIdentifier = mAuthProvider.getCredentialIdentifier();
```

7.3.4 Using the Google Authentication Provider

The Google authentication provider enables a user to login using the credentials associated with a Google account. The following procedure illustrates how to instantiate the Google authentication provider using the constructor in the `com.intel.wearable.cloudsdk.googleauth` package. This sample code creates an `ActionBarActivity` called `CloudAuthDevApp`.

Step 1. Install the provider.

Example: **Installing the Google Authentication Provider**

```
compile group: 'com.intel.wearable.cloudsdk', name: 'googleauth', version: '2.2.0'
```

Step 2. Implement the `CloudAuthDevApp` activity.

Example: **Implementing CloutAuthDevApp**

```
import com.intel.wearable.cloudsdk.core.*;
...
public class CloudAuthDevApp extends ActionBarActivity {
        CloudGoogleAuthProvider mAuthProvider;
        ...
}
```

Step 3. Instantiate `CloudGoogleAuthProvider` inside `onCreate` for the `CloudAuthDevApp` activity and assign the instance to `mAuthProvider`.

Example: **Instantiating CloudGoogleAuthProvider**

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
        ...
        mAuthProvider = new CloudGoogleAuthProvider("https://e2e-uaa.fc.wearables.infra-
host.com", "ndgFightClub", "ndgFightClubSecRet", this);
        ...
  }
```

Step 4. Override `onActivityResult`.

Example: **Overriding onActivityResult**

```
@Override
    public void onActivityResult(int requestCode, int responseCode, Intent intent) {
        super.onActivityResult(requestCode, responseCode, intent);
        mAuthProvider.onActivityResult(requestCode, responseCode, intent);
    }
  ```
```

**Step 5.** Call the `login()` function.

***Example:*** **Calling the login() Function**

```
mAuthProvider.login(new ICloudAuthLoginCb() {
    @Override
    public void onFailed(CloudError err) {
      // login failed, do something
    }
    @Override
    public void onSuccess() {
      // login successful, do something else
    }
});
```

Step 6. After a successful authentication, retrieve the credential identifier which is needed by other Cloud Services SDK modules to confirm user authentication.

Example: **Confirming Google User Authentication**

```
String credentialIdentifier = mAuthProvider.getCredentialIdentifier();
```
```

7.3.5 Using the Application Authentication Provider

The Application authentication provider is in the `com.intel.wearable.cloudsdk.core` package. The Application authentication provider authenticates the application, not the user, to the cloud. It is used for services that do not need user-level access (for example, public BLOB storage or to download new firmware). To use the Application authentication provider, follow the steps below.

Step 1. Instantiate the `CloudAppAuthProvider`.

Example: **Instantiating CloudAppAuthProvider**

```
CloudAppAuthProvider authProvider = new CloudAppAuthProvider(androidContext);
```

Step 2. Login.

Example: **Logging In an Application**

```
CloudResponse res = authProvider.login();
if (res.hasErrorOccurred())
    // login failed, do something else
    // login succeeded, do something else
```

Step 3. After a successful authentication, retrieve the credential identifier which is needed by other Cloud Services SDK modules to confirm application authentication.

Example: **Confirming Application Authentication**

```
String credentialIdentifier = authProvider.getCredentialIdentifier();
```

Note: This provider is used internally by the SDK and is not intended for use by the application.

7.4 Handling Cloud Services SDK Errors

Errors can be returned by the Cloud Services SDK in the following ways:

- Inside a `CloudResponse` object
- As return values
- As callback parameters.

In all of these cases, the error is captured by a `CloudError` object and the message inside can be accessed as illustrated in the `login()` example below.

Example: **Returning Cloud Services SDK Errors**

```
webViewAuthProvider.login(new ICloudAuthLoginCb() {
        @Override
        public void onFailed(CloudError error) {
            Log.d("CloudErrorExample", error.getMessage());
        }
        @Override
        public void onSuccess() {
            // do something
        }
});
```

Use the following call to get the specific error.

```
error.throwSpecificError();
```

Some editors might prompt you to surround the specific error call with a try/catch block. If you take the editor's suggestion, it will generate code for you that might look like this.

Example: **Generated Error Code**

```
try {
      error.throwSpecificError();
  } catch (CloudHttpError cloudHttpError) {
      cloudHttpError.printStackTrace();
  } catch (CloudTimeoutError cloudTimeoutError) {
      cloudTimeoutError.printStackTrace();
  } catch (CloudInterruptedError cloudInterruptedError) {
      cloudInterruptedError.printStackTrace();
  } ...
```

Some `CloudError` objects may have a useful error code. Use the following call to retrieve the error code.

```
int errorCode = error.getCode();
```

When the error is encapsulated in a `CloudResponse` object, access it as shown in the `login()` example below which uses `CloudAppAuthProvider`.

***Example:* Accessing Encapsulated Error Information**

```
CloudResponse res = cloudApplicationAuthProvider.login();
if (res.hasErrorOccurred()) {
        CloudError error = res.getError();
        Log.d("EncapsulatedErrorExample", error.getMessage());
}
```

7.5 Accessing User Profiles

The Cloud Services SDK provides a User Profile Store module to store and retrieve custom user profile information as JSON objects. It is packaged in `com.intel.wearable.cloudsdk.profilestore`. To access a user profile, the user must have a credential identifier as documented in Authenticating with the Cloud Services SDK. To use the API, follow the steps below.

Step 1. Create a `CloudUserProfileStore` instance.

***Example:* Creating a CloudUserProfileStore Instance**

```
import com.intel.wearable.cloudsdk.profilestore.*;
...
// see prerequisites above to learn how to obtain a `credentialIdentifier`
CloudUserProfileStore.init("https://appserver1.wearables.host.com", credentialIdentifier,
androidContext);
CloudUserProfileStore profileStore = CloudUserProfileStore.getInstance();
```

Step 2. Create a JSON user profile.
See the following example.

Example: **Creating a JSON User Profile**

```
final JSONObject sampleProfile = new JSONObject();
  try {
        sampleProfile.put("Name", "Superman");
        sampleProfile.put("Age", "200");
        sampleProfile.put("Height", "6");
        sampleProfile.put("Weight", "230");
  } catch (JSONException e) {
        e.printStackTrace();
  }
```

Step 3. Upload the JSON user profile to the cloud.

Example: **Uploading the JSON User Profile to the Cloud**

```
profileStore.put(sampleProfile, new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("put", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("put", "failure");
            Log.v("put", "ErrorMessage" + res.getError().getMessage());
        }
   });
```

Step 4. Retrieve the user profile as necessary.

Example: **Retrieving the User Profile**

```
profileStore.get(new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("get", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("get", "failure");
            Log.v("get", "ErrorMessage" + res.getError().getMessage());
        }
});
```

7.6 Accessing Document Stores

The Cloud Services SDK provides the functionality to upload, store and retrieve general JSON documents. It is packaged in `com.intel.wearable.cloudsdk.docstore`. Applications can save, retrieve, and delete documents based on a document ID. To use the API, follow the steps below.

Step 1. Create a `CloudDocStore` instance.

***Example:* Creating a CloudDocStore Instance**

```
import com.intel.wearable.cloudsdk.docstore.*;
...
// see prerequisites above to learn how to obtain a `credentialIdentifier`
CloudDocStore.init("https://e2e-app.fc.wearables.infra-host.com",
credentialIdentifier,androidContext);
CloudDocStore docStore = CloudDocStore.getInstance();
```

Step 2. Create a JSON document.

***Example:* Creating a JSON Document**

```
JSONObject sampleDocument = new JSONObject();
try {
        sampleDocument.put("test1", "value1");
        sampleDocument.put("test2", "value2");
        sampleDocument.put("test3", "value3");
        sampleDocument.put("test4", "value4");
        sampleDocument.put("test5", "value5");
} catch (JSONException e) {
        e.printStackTrace();
}
```

Step 3. Upload the document.

***Example:* Uploading the JSON Document**

```
docStore.put("2222", sampleDocument, new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("put", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("put", "failure");
            Log.v("put", "ErrorMessage" + res.getError().getMessage());
        }
  });
```

Example:

```
docStore.get("2222", new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("get", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("get", "failure");
            Log.v("get", "ErrorMessage" + res.getError().getMessage());
        }
  });
```

7.7 Accessing Device Profiles

The Cloud Services SDK provides a device profile store module to store and retrieve custom JSON documents that contain details about the companion device used to connect to the cloud. It is packaged in `com.intel.wearable.cloudsdk.deviceprofilestore`. To access a device profile, the user must have a credential identifier as documented in Authenticating with the Cloud Services SDK on page 102. Documents can be saved, retrieved and deleted by specifying a document identifier. Query capabilities are not supported yet. To use the API, follow the steps below.

Step 1. Create a `CloudDeviceProfileStore` instance.

Example: **Creating a CloudDeviceProfileStore Instance**

```
import com.intel.wearable.cloudsdk.deviceprofilestore.*;
...
// see prerequisites above to learn how to obtain a `credentialIdentifier`

CloudDeviceProfileStore deviceProfileStore  = new
CloudDeviceProfileStore(credentialIdentifier, androidContext);
```

Step 2. Create a JSON device Profile.

Example: **Creating a JSON Device Profile**

```
JSONObject sampleDeviceProfile = new JSONObject();
try {
        sampleDeviceProfile.put("test1", "value1");
        sampleDeviceProfile.put("test2", "value2");
        sampleDeviceProfile.put("test3", "value3");
        sampleDeviceProfile.put("test4", "value4");
        sampleDeviceProfile.put("test5", "value5");
} catch (JSONException e) {
        e.printStackTrace();
}
```

Step 3. Upload the device profile.

Example: **Uploading the Device Profile**

```
deviceProfileStore.put("some_device_id", CloudDeviceProfileStore.DeviceType.MOBILE,
sampleDeviceProfile,
    new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("put", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("put", "failure");
            Log.v("put", "ErrorMessage" + res.getError().getMessage());
        }
    });
```

Step 4. Get the device profile back as necessary.

Example: **Getting the Device Profile**

```
deviceProfileStore.get("some_device_id", new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.v("get", "success");
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.v("get", "failure");
            Log.v("get", "ErrorMessage" + res.getError().getMessage());
        }
  });
```

Calls similar to those documented in this section can be used to get the current list of device profiles and to delete a device profile. See the *Cloud Services SDK Java API Reference* for details.

7.8 Accessing BLOB Data and Software Assets

The Cloud Services SDK is packaged in `com.intel.wearable.cloudsdk.blobstore` and provides upload and download capabilities for the storage of BLOB data and software assets in the cloud. Depending on the access restrictions imposed on the binary, the following BLOB stores can be accessed.

- The `CloudAnonymousBlobStore` class allows for uploading of anonymous binary files. They are accessible by an administrator only. See Using CloudAnonymousBlobStore on page 116 for details.
- The `CloudPublicBlobStore` class allows for downloading software assets like device firmware. See Using CloudPublicBlobStore on page 118 for details.
- The `CloudBlobStore` class allows for uploading and downloading files belonging to a given user. Authentication is required for access. See Using CloudBlobStore on page 120 for details.

7.8.1 Using CloudAnonymousBlobStore

The `CloudAnonymousBlobStore` class allows uploading anonymous binary files to the cloud. Authentication is required although the files are stored anonymously. An application administrator can use this class to allow users to upload anonymous feedback about the application. To use the anonymous BLOB store, follow the steps below.

Step 1. Create a `CloudAnonymousBlobStore` instance. The product ID argument is a string of your choice. When downloading, the administrator will be able

to use calls like `getList()` to list BLOBs by product ID and tag (discussed later).

Example: **Creating a CloudAnonymousBlobStore Instance**

```
import com.intel.wearable.cloudsdk.blobstore.*;
...
CloudAnonymousBlobStore.init("https://app1.wearables.host.com", "some_product_id",
androidContext);
CloudAnonymousBlobStore anonymousBlobStore = CloudAnonymousBlobStore.getInstance();
```

Step 2. Upload a file named `f`. The file tag argument can be any string.

Example: **Uploading the f File**

```
anonymousBlobStore.upload(f, "some_tag", new ICloudResponseCb() {
    @Override
    public void onSuccess(CloudResponse res) {
    }
    @Override
    public void onFailed(CloudResponse res) {
    }
});
```

Step 3. To upload a file using a different file name add the `withFileName` argument as in the following API call.

Example: **Uploading a File Using a Different File Name**

```
anonymousBlobStore.getInstance().upload(f, "some_tag", "some_custom_name",
        new ICloudResponseCb() {
    @Override
    public void onSuccess(CloudResponse res) {
    }
    @Override
    public void onFailed(CloudResponse res) {
    }
});
```

7.8.2 Using CloudPublicBlobStore

The `CloudPublicBlobStore` class allows downloading of binary files like firmware. These files are available to all users of the application thus no authentication is required. To download a file from the public BLOB store, follow the steps below.

Step 1. Create a `PublicBlobStore` instance. Make sure the value of `productId` is valid in that it has been defined for an asset which has been uploaded using the administrative interface.

Example: **Creating a PublicBlobStore Instance**

```
import com.intel.wearable.cloudsdk.blobstore.*;
...
CloudPublicBlobStore.init("https://app1.wearables.host.com", productId,
       androidContext);
```

Step 2. Download a file named `my_asset`. This call can be invoked multiple times: first time results in the `onStarted()` callback and successive calls result in the `onProgress()` or `onFinished()` callback as necessary. The download call could therefore be placed, for example, in an Android application's lifecycle callback `onResume()`. See the following example.

Example: **Downloading the my_asset File**

```
protected void onResume() {
        super.onResume();
        CloudPublicBlobStore publicBlobStore = CloudPublicBlobStore.getInstance();
        // download to Context.getExternalFilesDir(null).getAbsolutePath()/my_asset
        publicBlobStore.download("my_asset", new ICloudDownloadCb() {
            @Override
            public void onStarted(CloudDownloadResponse cloudDownloadResponse) {
                Log.i(TAG, "download started.");
            }
            @Override
            public void onFailed(CloudDownloadResponse cloudDownloadResponse) {
                Log.e(TAG, "download failed.");
                Log.e(TAG, cloudDownloadResponse.getError().toString());
            }
            @Override
            public void onFinished(CloudDownloadResponse cloudDownloadResponse) {
                File vp = new File(androidContext.getExternalFilesDir(null), "my_asset");
                if (vp.exists())
                    Log.i(TAG, "finished downloading my_asset: " + vp.getAbsolutePath());
                else
                    Log.e(TAG, "oops! Should have called onFailed instead.");
            }
            @Override
            public void onProgress(int bytesDownloaded, int bytesTotal) {
                Log.i(TAG, "downloaded " + bytesDownloaded + " bytes out of " + bytesTotal);
            }
        });
        ...
}
```

Step 3. Use the following call to download a file and save it with a different name.

Example: **Saving a Downloaded File with a Different Name**

```
assetMgr.download("my_asset", "save_as_this_name", new ICloudDownloadCb() {
    ...
});
```

The following list documents important semantics of the download calls.

- When an absolute path is not provided, the name of the file to be downloaded (for example, my_latest_voice_pack) is appended to the path obtained by the `Context.getExternalFilesDir(null)` call.

- Duplicates are not downloaded. Thus, if the destination path already exists, the SDK will immediately callback `onFinished()`.
- If the application is no longer in memory after the download call is made, it must make the same download call again once it is started. Without this call, the SDK will not have callbacks registered and thus won't be able to inform the application when the download is finished. In this situation, the SDK will log an error message saying that it could not call the application back.
- `onProgress()` is called periodically as the file is downloading. The call is made each time a chunk of the file is downloaded. The chunk size is unknown.
- Use the `com.intel.wearable.cloudsdk.core.CloudDownloadPolicy` class to change how SDK downloads are handled. For example, you can request WiFi only downloads, change where in-progress downloads are placed, or stop displaying downloads in the Android download manager interface. You set the download policy by passing it in the constructor as illustrated in the sample below.

Example: **Changing How SDK Downloads are Handled**

```
CloudPublicBlobStore.init(..., myDownloadPolicy, androidContext);
```

7.8.3 Using CloudBlobStore

The `CloudBlobStore` class allows uploading and downloading of binary files belonging to an authenticated user. After a successful authentication, retrieve the credential identifier using `authProvider.getCredentialIdentifier()` as confirmation. The following steps contain sample code that illus-

trate how to upload and download a file with the `CloudBlobStore` class.

Step 1. Create a `CloudBlobStore` instance. The product ID argument is a string of your choice. All uploads will have this product ID set in the metadata.

Example: **Creating a CloudBlobStore Instance**

```
import com.intel.wearable.cloudsdk.blobstore.*;
...
// see prerequisites above to learn how to obtain a `credentialIdentifier`
CloudBlobStore.init("https://app1.wearables.host.com", "some_product_id",
        credentialIdentifier, androidContext);
CloudBlobStore userBlobStore = CloudBlobStore.getInstance();
```

Step 2. Upload the file named `f`. The file tag argument can be any string.

Example: **Uploading the f File**

```
userBlobStore.upload(f, "some_tag", new ICloudResponseCb() {
    @Override
    public void onSuccess(CloudResponse res) {
    }
    @Override
    public void onFailed(CloudResponse res) {
    }
});
```

Step 3. Download the file named `f`. This call can be invoked multiple times: first time results in the `onStarted()` callback and successive calls result in the `onProgress()` or `onFinished()` callback as necessary.

Note: The semantics documented in Using CloudPublicBlobStore on page 118 are also relevant to this `CloudBlobStore` class.

Example: Downloading the f File

```
// download to Context.getExternalFilesDir(null).getAbsolutePath()/f.getName()
publicBlobStore.download(f.getName(), new ICloudDownloadCb() {
    @Override
    public void onStarted(CloudDownloadResponse cloudDownloadResponse) {
        Log.i(TAG, "download started.");
    }
    @Override
    public void onFailed(CloudDownloadResponse cloudDownloadResponse) {
        Log.e(TAG, "download failed.");
        Log.e(TAG, cloudDownloadResponse.getError().toString());
    }
    @Override
    public void onFinished(CloudDownloadResponse cloudDownloadResponse) {
        File vp = new File(androidContext.getExternalFilesDir(null), f.getName());
        if (vp.exists())
            Log.i(TAG, "finished downloading: " + vp.getAbsolutePath());
        else
            Log.e(TAG, "oops! Should have called onFailed instead.");
    }
    @Override
    public void onProgress(int bytesDownloaded, int bytesTotal) {
        Log.i(TAG, "downloaded " + bytesDownloaded + " bytes out of " + bytesTotal);
    }
});
```

Step 4. To upload or download a file using a different file name than the one it has, use the following calls.

Example: Uploading or Downloading a File Using a Different File Name

```
userBlobStore.upload(f, "some_tag", "upload_as_this_name", new ICloudResponseCb() {
    ...
});
...
publicBlobStore.download("upload_as_this_name", "save_as_this_name",
        new ICloudDownloadCb() {
    ...
});
```

For more sample code using the `CloudBlobStore` class see Using the Cloud Services SDK on page 99.

7.9 Accessing Times Series Data

The Cloud Services SDK provides a data storage module to store, retrieve and analyze time series data points. (This data typically consists of successive measurements made over a time interval; for example, GPS measurements.) Time series data is published and retrieved as a list of Observation objects with one Observation being a set of data points, each called a Measurement (for example, speed, distance or orientation). The time series API are packaged in `com.intel.wearable.cloudsdk.timeseries`. To access time-series data for a user profile, the user must have a credential identifier as documented in Authenticating with the Cloud Services SDK on page 102. To use the API, follow the steps below.

Step 1. Create a `CloudTimeSeries` instance.

Example: **Creating a CloudTimeSeries Instance**

```
import com.intel.wearable.cloudsdk.timeseries.*;
...
// see prerequisites above to learn how to obtain a `credentialIdentifier`

CloudTimeSeries timeSeriesDataStore = new CloudTimeSeries(credentialIdentifier,
       androidContext);
```

Step 2. Create an `Observation`.

Example: **Creating an Observation**

```
// This Observation is for a WALKING event.

Observation observation = new Observation(new Date(System.currentTimeMillis()),"WALKING");
observation.addMeasurement(new Measurement("duration", "44"));
observation.addMeasurement(new Measurement("distance", "22"));
observation.addMeasurement(new Measurement("calories", "565"));
observation.addMeasurement(new Measurement("stepCount", "343"));
```

Step 3. Optionally set a session ID or other information about the Observation.

Example: **Setting a Session ID**

```
observation.setSessionId("some_session_id");
observation.setDataSource("intel.clark.sensor1");
observation.setLocation(45.55066, -122.9134);
```

Step 4. Publish the Observation.

Example: **Publishing an Observation**

```
Observation[] observations = { observation };
timeSeriesDataStore.post("wearable_id", "companion_device_id", observations,
  new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            Log.d(TAG, "Post Success" + res.getPayload().toString());
        }
        @Override
        public void onFailed(CloudResponse res) {
            Log.e(TAG, "ERROR: " + res.getPayload() + " " + res.getError().getMessage());
        }
  });
```

Step 5. To retrieve all observations that occurred within a specific time span, use `getObservations` as below.

Example: **Retrieving all Observations**

```
timeSeriesDataStore.getObservations(new Date(System.currentTimeMillis() - 100000),
  new Date(System.currentTimeMillis()), null, new ICloudResponseCb() {
        @Override
        public void onSuccess(CloudResponse res) {
            // do something
        }
        @Override
        public void onFailed(CloudResponse res) {
           // do something else
        }
  });
```

Step 6. To pass a filter to retrieve specific observations that occurred within a specific time span, use `filterBuilder` as below.

Step 7. Get some observations that occurred within a specific time window by passing a filter.

Example: Retrieving Observations Within a Specific Time Window

```
// create observation filter
CloudTimeSeries.FilterBuilder fb = new CloudTimeSeries.FilterBuilder();
fb.createFilter().toSelectObservationsWith("distance").lessThan("25")
.and("duration").greaterThan("30").forEvent("WALKING");


// filter observations that occurred within the given time window
timeSeriesDataStore.getObservations(new Date(System.currentTimeMillis() - 100000),
new Date(System.currentTimeMillis()), fb.build(), new ICloudResponseCb() {
...

  });
```

CHAPTER

8 Third-Party License Information

This chapter contains the following third-party software licenses:

- Apache License
- Deusty License
- Deusty License
- MIT License

8.1 Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent

license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
* You must give any other recipients of the Work or Derivative Works a copy of this License; and
* You must cause any modified files to carry prominent notices stating that You changed the files; and
* You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
* If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for

determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Note: Other license terms may apply to certain, identified software files contained within or distributed with the accompanying software if such terms are included in the directory containing the accompanying software. Such other license terms will then apply in lieu of the terms of the software license above.

END OF TERMS AND CONDITIONS

8.2 Deusty License

Copyright (c) 2010-2015, Deusty, LLC
All rights reserved.

Redistribution and use of this software in source and binary forms,
with or without modification, are permitted provided that the following
conditions are met:

* Redistributions of source code must retain the above
  copyright notice, this list of conditions and the
  following disclaimer.

* Neither the name of Deusty nor the names of its
  contributors may be used to endorse or promote products
  derived from this software without specific prior
  written permission of Deusty, LLC.

8.3 Eclipse Public License

Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS
"Contribution" means:
a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
b) in the case of each subsequent Contributor:
i) changes to the Program, and
ii) additions to the Program;
where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.
"Contributor" means any person or entity that distributes the Program.
"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.
"Program" means the Contributions distributed in accordance with this Agreement.
"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.
2. GRANT OF RIGHTS
a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.
3. REQUIREMENTS
A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:
a) it complies with the terms and conditions of this Agreement; and
b) its license agreement:
i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.
When the Program is made available in source code form:
a) it must be made available under this Agreement; and
b) a copy of this Agreement must be included with each copy of the Program.
Contributors may not remove or alter any copyright notices contained within the Program.
Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.
4. COMMERCIAL DISTRIBUTION
Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.
For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.
5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement , including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year

after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

8.4 MIT License

Copyright (c) 2010 Xtreme Labs and Pivotal Labs
Copyright (c) 2007 Mockito contributors
Copyright 2012-2014 Zafar Khaja

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.