HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

FHO Fachhochschule Ostschweiz

API Tutorial

Overview

There are two main usage scenarios for the SmartNaviWatch library. This section tries to outline those scenarios and give a brief overview of the architecture you will be working with.

Using the provided Wear App: This is most useful when you try to quickly provide your own application with the features that SmartNaviWatch has. The communication stack in this instance looks like the following:

```
SmartNaviWatch Wear Application <-> Bluetooth <-> SmartNaviWatch Mobile App <->  Your Application
```

The only part you need to worry about is your own application. You install the Mobile App (comes with the Wear App) and implement the connection to it. Then you send messages with navigation instructions and the library will take care of displaying them to the user.

Using only the messaging layer: Use this approach if you want to create your own shiny user interface for both mobile and watch applications. The communication stack will look like this:

```
Your Wear Application <-> EndPoint <-> Bluetooth <-> EndPoint <-> Your Mobile Application
```

In this scenario you take full control over both ends of the user experience. The API is only used to send and receive messages.

Warning: Please note that events occur asynchronously, in another thread. Changing UI elements as a result of an event requires you to make sure that () is called in order to run your code in the proper thread.

Setting up the development environment

- » Download the source code for SmartNaviWatch here
- » Include the source code or a reference to a compiled version into your project
- » If you want to use the premade Watch App, install the SmartNaviWatch APK on your phone

Basic Messaging

Before you can start to send or receive messages, you need to understand the basic concept of all important messaging components.

NavigationMessage This is the main class for containing messages. Messages consist of a message type and a payload object. The payload object is required to be marked with the Serializable interface.

When working with the default APK (sending messages via NavigationServiceConnector) you should use a HashTable<String, Object> as payload. The keys correspond to the MessageDataKeys enumeration.

Custom keys will be ignored by the default APK, but can come in handy in other scenarios where you build both ends of the connection. See "Sending Data" for some examples of creating messages with useful content.

Warning: This protocol does not ensure reliable messaging. If you need to make sure messages are sent and received you need to take care of this yourself by adding another layer on top.

MessageEndPoint EndPoints are used to receive or send messages. If you are working with the SmartNaviWatch APK you don't need to create your own EndPoint since you can send messages via the service connector.

The creation of a custom watch and mobile application based on our library requires you to use EndPoints on both sides of the connection. Also note that both your mobile application as well as your wear application need to be from the same APK (and therefore have the same application id and certificate) to get packages passed through.

NavigationServiceConnector Working with the provided APK, you send and receive messages via this connector from within your mobile application. This is to ensure that the Bluetooth connection is accepted by the underlying messaging layer based on the application id.

Sending messages from a mobile application

» Create and store an instance of NavigationServiceConnector:

```
private NavigationServiceConnector messageService;
...
private NavigationServiceConnector getServiceConnector() {
    if (messageService == null)
    {
        messageService = new NavigationServiceConnector(appContext);
    }
    return messageService;
}
```

» Create a message:

```
NavigationMessage msg = NavigationMessage.create(messageType, payload);
```

» Send your message:

```
getServiceConnector().sendMessage(msg);
```

Receivingmessages in a mobile application

» Create and store an instance of NavigationServiceConnector (see "Sending messages from a mobile application"

» Subscribe to the provided events by implementing IMessageListener

```
public class Example implements IMessageListener {
...
    private void initListener() {
        getServiceConnector().addMessageListener(this);
    }
 ...
    @Override
    public void messageReceived(NavigationMessage message) {
        // Your message handling here
    }
}
```

Sending / receiving messages without the default APK

» Create and store an instance of EndPoint (you can use the same EndPoint for sending and receiving)

```
private MessageEndPoint endPoint;
...
endPoint = new MessageEndPoint(getApplicationContext());
...
```

» Subscribe to the provided events

```
public class Example implements IMessageListener {
 ...
    private void initListener() {
        endPoint.addMessageListener(this);
    }
 ...
    @Override
    public void messageReceived(final NavigationMessage message) {
        // Your message handling here
    }
}
```

» Create a message

```
NavigationMessage msg = NavigationMessage.create(messageType, payload);
```

» Send your message

```
endPoint.sendMessage(msg);
```

Sending Data

When sending data between your own applications, you are free to use any payload objects, as long as they implement the Serializable interface and can be serialized by the default Java serializer. The same goes for the message type, choose whatever fits your needs. Just make sure the type passed is a path like this "sample/path/for/reference".

However, sending data to the provided Wear App requires you to use a more strict data format.

The type of the message should be one of the constants defined in MessageTypes. The content of the messages is made of a HashTable<String, Object>. The following table shows which keys should be set and what their expected values are:

Key	Expected Value
MessageDataKeys.TurnType	String:
	`"C"    -> straight on`
	`"TL"   -> turn left`
	`"TSLL" -> turn left slightly`
	`"TSHL" -> turn left hard`
	`"TR"   -> turn right`
	`"TSLR" -> turn right lightly`
	`"TSHR" -> turn right hard`
	`"KL"   -> turn left slightly`
	`"KR"   -> turn right slightly`
	`"TU"   -> U-turn`
	`"TRU"  -> U-turn`
	`"OFF"  -> off route warning`
	`"RNDB" -> Roundabout`
MessageDataKeys.RoutingDescription	String: Description of what the user should do next.
MessageDataKeys.RouteLeftTime	Integer: Estimated time left for the user to reach the destination.
MessageDataKeys.MapPolygonData	MapPolygonCollection: Polygons to be rendered on the map background.
MessageDataKeys.LocationName	String: Name of the location the user is currently at.
MessageDataKeys.LocationAccuracy	Float: Accuracy of the location in meters.