TomTom PRO 82xx PRO.connect developer guide

TOMTOM TELEMATICS

Contents

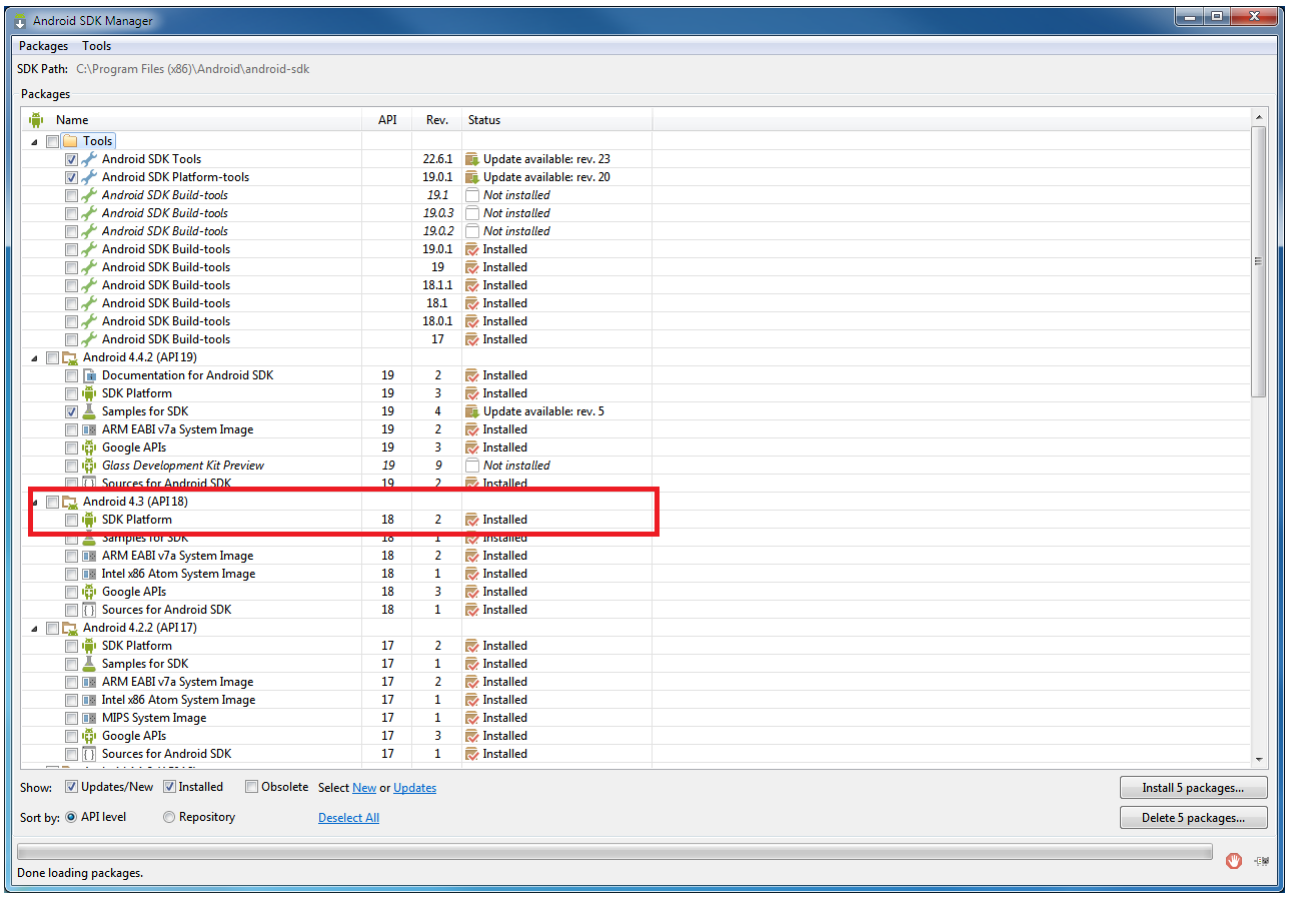
Introduction

This document helps you to set up your PRO.connect SDK working environment. Some sample projects help you to get started.

Preconditions

The following is needed:

- PRO 82xx activated with WEBFLEET, so it can be used with the PRO.connect SDK.
- A WEBFLEET.connect API key. For how to apply for a WEBFLEET.connect API key, refer to WEBFLEET.connect Reference manual.
- Eclipse ADT. You can download Eclipse ADT from http://developer.android.com/sdk/index.html
- The latest Android SDK.
- Android 4.3 must be installed.



Establishing a connection

You need to do the following before you can start developing applications on your TomTom PRO 82xx device.

Preparations on Windows

If you are developing using Windows, you need to install a device driver to connect your computer to your TomTom PRO 82xx device.

1. Go to http://developer.tomtom.com/products/Bridge.
2. Go to the **Downloads** tab.
3. Download **Device Driver Windows NT/XP/7/8**.
4. Unpack the Zip file to your computer.

   Remember the location where you have unpacked the Zip file.
5. Check if there is the **adb_usb.ini** file in **User****[YOUR_USER_NAME]**\.**android**\ and add 0x1390.

   If you cannot find the **adb_usb.ini** file in that location, do the following.

   1. Create a new **ini** file by opening a text editor and typing 0x1390.
   2. Save the file with the name **adb_usb.ini** under **User****[YOUR_USER_NAME]**\.**android**\.

Preparations on Linux

If you are developing using Linux, no device driver is needed.

Check if there is the **adb_usb.ini** file in /**home**/**[YOUR_USER_NAME]**/.**android**/

If there is no such file, do the following.

1. Create a new **ini** file by opening a text editor and typing 0x1390.
2. Save the file with the name **adb_usb.ini** under /**home**/**[YOUR_USER_NAME]**/.**android**/

Connecting your TomTom PRO 82xx device to your computer

Connect your TomTom PRO 82xx to your computer using a micro USB cable. We recommend to connecting the micro USB cable directly to your PRO 82xx device.

Note: If you want to connect your PRO 82xx device while it is in the cradle, you must connect the USB cable to the cradle and additionally you must connect the external power supply from TomTom to the cradle.

When you connect your PRO 82xx device to a Windows computer for the first time you are asked for drivers. Select the driver that you have downloaded before.

If your PRO 82xx device cannot be recognised by your computer automatically, go to the Device Manager and install the driver for the Android device manually.

When you connect your PRO 82xx to your computer, your device asks you if you want to grant access to the computer. Tap **OK**.

Checking the connection

To check if your PRO 82xx is connected to your computer and the ADB (Android Debug Bridge) is properly working, open a command line interface and execute **adb devices**. Your device should be listed with its serial number.

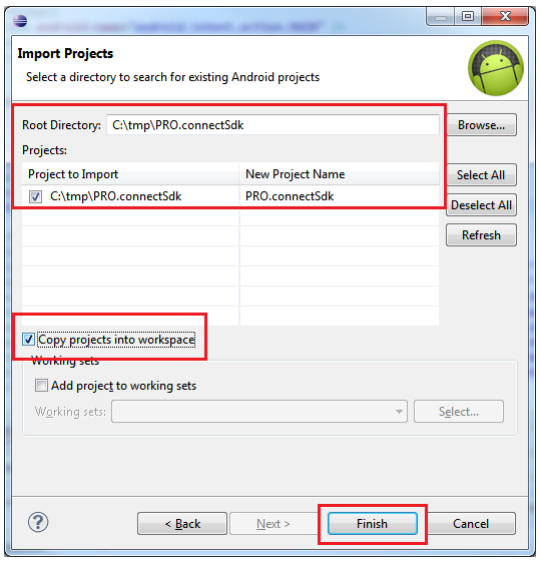
Note: You can find the serial number on the bottom side of your PRO 82xx, device left to the Kensington Lock. An Example of a serial number is: OC123G45678

Installing the PRO.connect SDK

You can download the PRO.connect SDK from the PRO.connect developer forum or from the TomTom Telematics website.

1. Download the Zip file and unpack it into a directory called **PRO.connectSdk** on your computer.
2. Start Eclipse and import the PRO.connect SDK.
   1. Go to **File**.
   2. Select **Import**.
   3. Select the **Android** folder.
   4. Select **Existing Android Code Into Workspace** and click **Next**.

   A pop-up window opens.

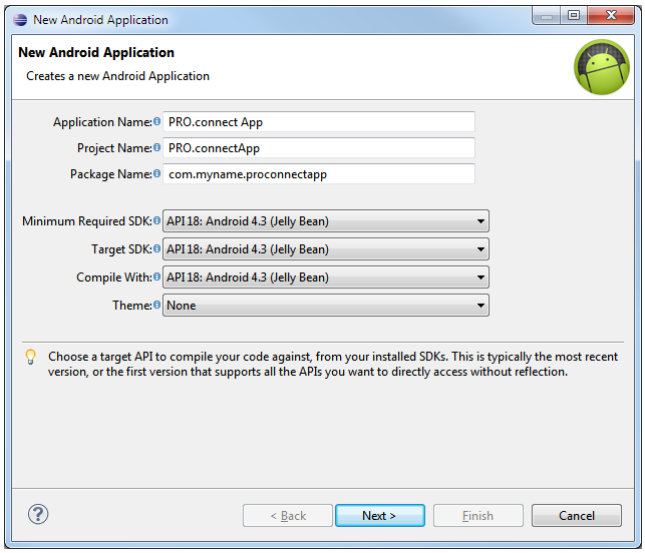


3. Select **Copy projects into workspace**.
4. Click **Finish**.

Creating a sample app

Do the following to create a sample app using the installed PRO.connect SDK.

1. Create an new Android Application project in Eclipse.

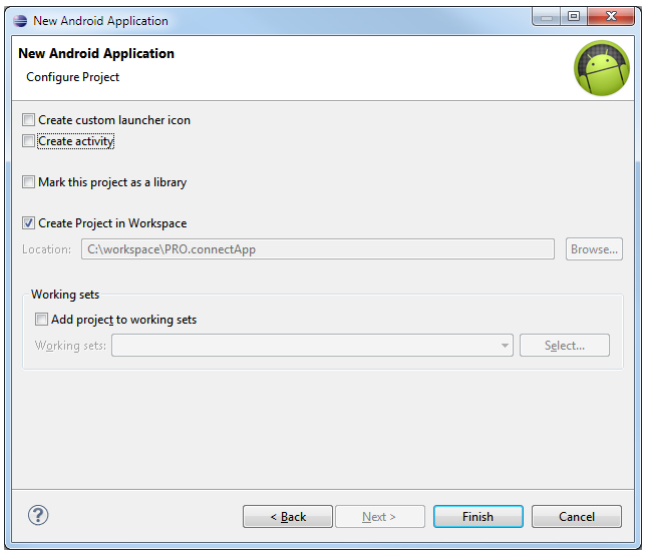


   Define a name for the application, the project and package.

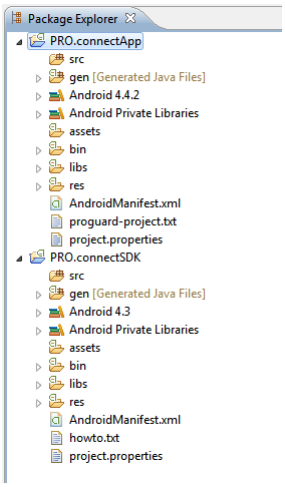
   IMPORTANT: Make sure that Android 4.3 (Jelly Bean) is selected in all three lists as shown below.

2. Click **Next**.
3. Deselect **Create custom launcher icon** and **Create activity**.

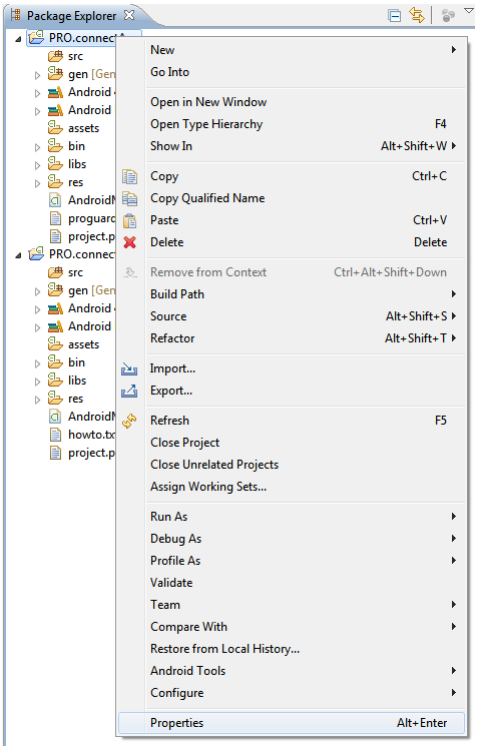
   Do this exclusively for the sample app, to keep the sample as simple as possible.



Your package Explorer in Eclipse should be looking as shown below.



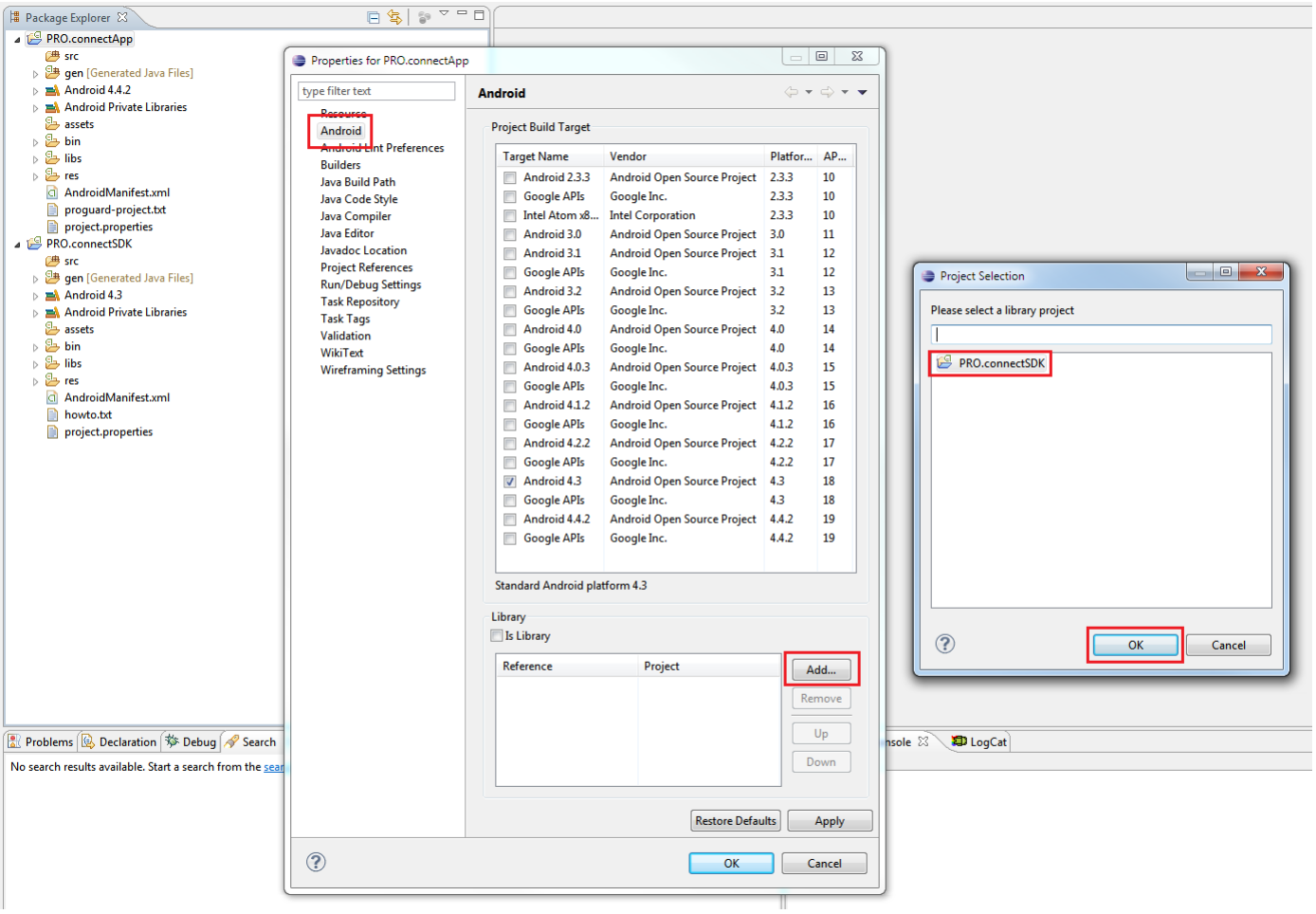
4. In Eclipse set a reference in your newly created PRO.connect App to the PRO.connectSdk project. Right-click **PRO.connectApp** and select **Properties**.



The Properties window for PRO.connectApp opens.

5. Select **Android** from the list on the left and click the **Add** button in the lower right corner. The Project Selection window opens.

6. Select **PRO.connectSDK** and click **OK**.



Getting started

In this chapter you will learn basic steps on how you can implement your own application for your TomTom PRO 82xx device using the PRO.connect SDK.

Preparing your project for the SDK

In the **Android Manifest** in the **application** element insert the **meta-data** element as follows and replace YOUR_API_KEY with your individual WEBFLEET.connect API key.

```
<meta-data android:name="com.tomtom.telematics.proconnectsdk.apikey" an-
droid:value="YOUR_API_KEY"/>
```

Displaying the list of messages from WEBFLEET

The following Activity snippet shows how to create the SDK service, how to retrieve and display the WEBFLEET messages.

TextMessageListActivity

```
/**
 * This Activity demonstrates the basic concepts of the PRO.connect SDK.
 * <p>
 * Before running it, be sure to have your device activated for the use with WEBFLEET.
 *</p>
 * <ol>
 * <li>In {@link #onCreate(Bundle)} the ProConnectSdk is created.</li>
 * <li>After that, a task is started which queries and logs all text messages that
have been sent to the device.</li>
 * <li>In order to properly free all resources of the ProConnectSdk, in {@link
#onDestroy()} the ProConnectSdk is released again.</li>
 * </ol>
 * <p>
 * It is recommended to keep a reference to the ProConnectSdk for the lifetime of the
hosting Activity or Service to avoid recreating and
 * disposing resources in the background repeatedly.
 * </p>
 */
public class TextMessageListActivity extends Activity
{
    //reference to the ProConnectSdk, initialized in onCreate() and disposed in
onDestroy()
    private ProConnectSdk proConnectSdk;

    /**
     * <p>
     * This implementation of a task demonstrates how to deal with background tasks
when
```

```
     * using the ProConnectSdk. All methods of the ProConnectSdk are potentially
blocking calls and therefore cannot
     * be executed on the main thread of an Android application. In order to load off
tasks to another
     * thread you must implement tasks that inherit from {@link ProConnectTask} or
{@link AsyncTask}
     * or at least execute them in a separate thread and then synchronize with the
UI thread.
     * </p>
     * <p>
     * This sample task inherits from {@link ProConnectTask}. It retrieves all text
messages that
     * are currently stored on the device in {@link #doInProConnectSdk(ProConnectSdk)}
and displays
     * them in {@link #onResult(List)}. Note, {@link #onResult(List)} is already
synchronized with
     * the UI thread.
     * </p>
     */
    private static class ListTextMessageTask extends ProConnect-
Task<List<TextMessage>>
    {
        private static final String LOG_TAG = "PRO.connectApp";

        @Override
        protected List<TextMessage> doInProConnectSdk(ProConnectSdk proConnectSdk)
        {
            //this is executed in a separate worker thread
            //and just returns the currently stored text messages
            return proConnectSdk.getTextMessageClient().getTextMessageList();
        }

        @Override
        protected void onResult(List<TextMessage> textMessageList)
        {
            //this is executed on the UI thread
            //it displays the text messages queried in doInProConnectSdk()
            Log.i(LOG_TAG, "Found " + (textMessageList.size() == 0 ? "no" :
textMessageList.size()) + " text messages.");

            for(TextMessage textMessage : textMessageList)
            {
                Log.i(LOG_TAG, "Message: " + textMessage.textMessage);
                Log.i(LOG_TAG, "Author: " + textMessage.author);
                Log.i(LOG_TAG, "Received: " + new
Date(textMessage.receivedTimestamp));
            }
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
```

```
        //create a new instance of the ProConnectSdk
        //best practice is to keep the reference for the lifetime of the Activity or
Service
        this.proConnectSdk = new ProConnectSdk(this);

        //this task is executed right after the activity has been started, without
any user interaction
        //watch the output of LogCat to see the text messages
        this.proConnectSdk.executeTask(new ListTextMessageTask());

        this.startService(new Intent(this.getApplicationContext(), NewTextMessag-
eService.class));
    }

    @Override
    protected void onDestroy()
    {
        super.onDestroy();

        this.stopService(new Intent(this.getApplicationContext(), NewTextMessag-
eService.class));

        //be sure to free the ProConnectSdk in onDestroy() to release all resources
the ProConnectSdk has acquired in the background
        this.proConnectSdk.release();
    }
}
```

Receiving a message from WEBFLEET using an Android service

The following BroadcastReceiver snippet shows how to receive an order in the background and display a toast without the need of a long running background service.

Note: This service is started and stopped by the TextMessageListActivity activity.

NewTextMessageService

```
/**
 * This Service demonstrates push notifications using the PRO.connect SDK.
 * <p>
 * Before running it, be sure to have your device activated for the use with WEBFLEET.
 *</p>
 * <ol>
 * <li>In {@link #onCreate()} the ProConnectSdk is created.</li>
 * <li>After that, a task is started which installs a push notification listener (a
{@link NewTextMessageListener}) for new WEBFLEET text messages in the Service.</li>
 * <li>{@link NewTextMessageListener#onResult(TextMessage)} is called, when you send
a text message to a device this service is installed and started on.</li>
 * <li>Note that when receiving a new text message in {@link NewTextMessageLis-
tener#onResult(TextMessage)}, the creation of the Toast is synchronized with main
thread of the application using a handler.</li>
 * <li>In order to properly free all resources of the ProConnectSdk, in {@link
#onDestroy()} the ProConnectSdk is released again.</li>
 * </ol>
```

```
 *
 * <p>
 * It is recommended to keep a reference to the ProConnectSdk for the lifetime of the
hosting Activity or Service to avoid recreating and
 * disposing resources in the background repeatedly.
 * </p>
 */
public class NewTextMessageService extends Service
{
    //reference to the ProConnectSdk, initialized in onCreate() and disposed in
onDestroy()
    private ProConnectSdk proConnectSdk;

    /**
     * <p>
     * This implementation of a task demonstrates how to deal with background tasks
when
     * using the ProConnectSdk. Almost all methods of ProConnectSdk are blocking calls
and therefore cannot
     * be executed on the main thread of an Android application. In order to load off
tasks to another
     * thread you must implement tasks that inherit from {@link ProConnectTask} or
{@link AsyncTask}
     * or at least execute them in a separate thread and then synchronize with the
UI thread.
     * </p>
     * <p>
     * This sample task inherits from {@link ProConnectTask}. It installs a {@link
NewTextMessageListener} that
     * is called back when a new text message arrives on the device. Since {@link
NewTextMessageListener#onResult(TextMessage)}
     * is not synchronized with the UI thread, a handler is used to synchronize with
the UI and display a Toast.
     * </p>
     */
    private static class CreateNewTextMessageListenerTask extends ProConnect-
Task<Void>
    {
        private final NewTextMessageListener newTextMessageListener;

        public CreateNewTextMessageListenerTask(NewTextMessageListener new-
TextMessageListener)
        {
            this.newTextMessageListener = newTextMessageListener;
        }

        @Override
        protected Void doInProConnectSdk(ProConnectSdk proConnectSdk)
        {
            proCon-
nectSdk.getTextMessageClient().setOnNewTextMessageCallback(this.newTextMessageLis
tener);
            return null;
        }
    }
```

```
    private class NewTextMessageListener implements Callback<TextMessage>
    {
        //a handler is used in order to synchronize with the UI thread
        private Handler handler;

        public NewTextMessageListener()
        {
            this.handler = new Handler();
        }

        /**
         * Called by the PRO.connect SDK when an error occurred while receiving a new
text message.
         */
        @Override
        public void onFailure(final ErrorInfo errorInfo)
        {
            //this code is executed on the main thread of the application and thus
synchronized with the UI thread
            this.handler.post(new Runnable()
            {
                @Override
                public void run()
                {
                    Toast.makeText(getApplicationContext(), "Error: " + errorInfo,
Toast.LENGTH_LONG).show();
                }
            });
        }

        /**
         * Called by the PRO.connect SDK when a new text messages was received.
         */
        @Override
        public void onResult(final TextMessage textMessage)
        {
            //this code is executed on the main thread of the application and thus
synchronized with the UI thread
            this.handler.post(new Runnable()
            {
                @Override
                public void run()
                {
                    Toast.makeText(getApplicationContext(), "Text message: " +
textMessage, Toast.LENGTH_LONG).show();
                }
            });
        }
    }

    @Override
```

```
    public void onCreate()
    {
        super.onCreate();
        //create a new instance of the ProConnectSdk
        //best practice is to keep the reference for the lifetime of the Activity or
Service
        this.proConnectSdk = new ProConnectSdk(this);

        //this task is executed right after the activity has been started, without
any user interaction
        //watch out for Toasts after a text message has been sent to the device in
WEBFLEET
        this.proConnectSdk.executeTask(new CreateNewTextMessageListenerTask(new
NewTextMessageListener()));
    }



    @Override
    public IBinder onBind(Intent intent)
    {
        return null;
    }

    @Override
    public void onDestroy()
    {
        super.onDestroy();
        //be sure to free the ProConnectSdk in onDestroy() to release all resources
the ProConnectSdk has acquired in the background
        this.proConnectSdk.release();
    }
}
```

Receiving orders from WEBFLEET using a BroadcastReceiver

The following BroadcastReceiver snippet shows how to receive an order in the background without the need of a long running background service and display as Toast.

To set up the BroadcastReceiver you have to insert the following in the **Android Manifest**.

Insert the following snippet before the **application** element.

```
        <!-- Ensures that this app does receive the relevant Intents only from the
PRO.connect SDK. -->
        <uses-permission an-
droid:name="com.tomtom.telematics.proconnnectsdk.permission.SEND_MESSAGE" />
```

In the **application** element insert the following.

```
        <!-- Declaring the permission that this app does receive the relevant Intents
only from the PRO.connect SDK. -->
        <receiver
            android:name="com.myname.proconnectapp.NewOrderReceiver"
            an-
droid:permission="com.tomtom.telematics.proconnnectsdk.permission.SEND_MESSAGE" >
```

```
            <!--
                Declaring the version of the PRO.connect SDK this app is developed
against
                is mandatory.
                The PRO.connect SDK needs to know on which compatibility level it needs
to communicate with this app.
            -->
            <meta-data
                android:name="com.tomtom.telematics.proconnectsdk.api.version"
                an-
droid:value="@string/com.tomtom.telematics.proconnectsdk.api.version" />

            <!-- This receiver reacts on new WEBFLEET orders. -->
            <intent-filter>
                <action an-
droid:name="com.tomtom.telematics.proconnectsdk.action.NEW_ORDER" />
            </intent-filter>
        </receiver>
```

NewOrderReceiver

```
/**
 * This BroadcastReceiver demonstrates receiving a push notification from WEBFLEET
without the use of a long-running service.
 * <p>If you send an order to the device this app is installed on, this BroadcastReceiver
will show a Toast displaying the
 * order.</p>
 * <p>
 * Before running it, be sure to have your device activated for the use with WEBFLEET.
 *</p>
 * <ol>
 * <li>In the AndroidManifest declare that the app is using the permission
<code>com.tomtom.telematics.proconnnectsdk.permission.SEND_MESSAGE</code>.</li>
 * <li>Also, in the AndroidManifest declare that this BroadcastReceiver shall react
on Intents with the action {@link ProConnectSdkIntents#ACTION_NEW_ORDER}.</li>
 * <li>Also, in the AndroidManifest configure that this BroadcastReceiver requires
the permission
<code>com.tomtom.telematics.proconnnectsdk.permission.SEND_MESSAGE</code>.</li>
 * <li>Also, in the AndroidManifest configure the version of the PRO.connect SDK this
BroadcastReceiver is implemented against using a <code>meta-data</code> element.</li>
 * <li>Now, if you have installed the app and send an order in {@link #onRe-
ceive(Context, Intent)} the Intent {@link ProConnectSdkIntents#ACTION_NEW_ORDER} is
received.</li>
 * <li>The order is extracted from the extra's of the Intent and Toast is
displayed.</li>
 * </ol>
 *
 * <p>Note, the app must not be running in order to display the Toast.</p>
 */
public class NewOrderReceiver extends BroadcastReceiver
{
    @Override
```

```
    public void onReceive(Context context, Intent intent)
    {
        Order order = (Order) in-
tent.getExtras().get(ProConnectSdkIntents.EXTRA_ORDER);
        Toast.makeText(context, "Order: " + order, Toast.LENGTH_LONG).show();
    }
}
```

PRO.connect Order Reference Implementation (RI)

The PRO.connect Order RI can be downloaded from the PRO.connect developer forum or from the TomTom Telematics website.

1. Download the Zip file and unpack it.
2. Start Eclipse and import the PRO.connect Order RI.
    1. Go to **File**.
    2. Select **Import**.
    3. Select the **Android** folder.
    4. Select **Existing Android Code Into Workspace** and click **Next**.

    A pop-up window opens.
3. In the PRO.connect Order RI that you have imported to Eclipse, set a reference to the PRO.connectSdk similar to what you did when you have created the sample app.

Documentation

For the documentation of the PRO.connectSDK go to libs\docs\index.html

For the documentation of the order reference implementation go to doc\index.html in the folder where you have unpacked the Order Reference Implementation.

Copyright notices