Developer documentation

Examiner: Jan-Philipp Steghöfer
Date: 2015-10-30
Authors: Emil Axelsson och Erik Karlkvist
Group name: Spoti5
Group members: Emil Axelsson, Matilda Horppu, Johannes Hildén,
Erik Karlkvist, Hampus Rönström

Build process

GitHub

To clone the object into your computer, enter:

```
$ git clone https://github.com/emiilax/spoti5.git
```

(This project is in a private repository, therefore you can only do this if you are invited)

Terminal

First thing you have to do is make sure that you have Gradle installed and a device connected (either a virtual device or other android phone). Make your way into the the project root(enter /your/path/spoti5/EcoBussing).

To build project enter:

```
$ gradle build
```

To install the application into your plugged-in device, enter:

```
$ gradle installDebug
```

To uninstall the application from your plugged-in device, enter:

```
$ gradle uninstallDebug
```

Android studio

Open *Android Studio,* if “Welcome to Android Studio” window is active, press open existing project and go to wherever you have put it.

If you already have started Android Studio, press *File* > *Open* > *(find the project and open)* Make sure to open the EcoBussing folder when opening in Android studio.

Tests

We have run the JUnit tests from Android Studio. When trying to find a way to run the tests from the command line we failed to do so. So the only way we know how to run the tests are from Android Studio. To do so you have to change Test Artifact in Build Variants to Unit Tests. Then you run the class AllJUnitTests in the package com.example.spoti5.ecobussing(test).

Acceptance tests are performed manually and these tests can be found in the GitHub repository.

Major parts

Database

We needed a database in the project in order to store things as distance traveled, carbon dioxide saved, names, passwords etc. No one in the group were particularly experienced in creating or coding a database. A lot of time went into finding something that we understood and could work with within the timeframe that we had. We ended up choosing Firebase because it seemed easy to implement since it was developed for smartphone applications.

The way we put data on the firebase is really simple but it took some time and testing to understand. We ended up creating a class called DatabaseUser, and we simply set the value of a node to the variables in the class. However firebase does not use the name of the variables when naming nodes, but the name of the get-methods. It was very confusing seeing just some variables appear in the database and not others, especially as they often did not have the same names. There was also a problem with things like hashmaps and lists. These had to be parsed into junit, and then stored as a string in the database. Because of this we had a lot of loading errors from the database where the data was incorrect or old. However retrieving data was really easy because you can just parse it back to a DatabaseUser if everything worked out correctly.

We wanted the database to be exchangeable in case firebase turned out to be really hard to work with or just did not do what we wanted. Whenever the application needs something from the database we wanted it to go through a class that was easy to exchange. The database url is, because of the way firebase works, also easy to change. This means that if the data gets corrupted and can not be parsed we can create a new database on firebase and link to the new url instead.

Api

In this project three different api’s were used, Electricity, Västtrafik and Google Maps Directions. These api’s are the pillars in the application. The Electricity api is used to get the

position of the bus that the user is currently on. The response results in information about the position, such as longitude-, latitude-, altitude-values and a few more. When the position is set, the Västtrafik api is used to check nearby stop locations and see if any of the stop locations is matching with the current bus stop locations. When a match is found the start-location is set. The position is later on updating every 15 seconds while on the bus. When the user exits the bus, it checks for the nearest matching bus location (same as above) and sets that as the end location. The corresponding travel length between these two stops with car are then calculated using Google Maps Directions api.

Design decisions

Api-level

The minimum and target API levels were chosen when the project was created in Android Studio and we did not really know what the different levels implicated. The minimum level, 15, was chosen because it was supposedly supported by a big percentage of the world's andorid phones and the target level, 23, was chosen because it was the latest and would therefore have the most updated functions. In hindsight we might not have needed the latest API level as target level since we have not really used the changes from previous API levels.

MVC

The MVC-pattern has been applied to this project to get a good structure. MVC stands for Model-View-Controller, Model keeps all the data, View is where the data is shown and Controller handles user inputs and create changes in the model and the view.

There seems to be a lot of different perceptions about what MVC really is in Android. Some people mean that the XML-files are the View, the Fragment-/Activity-classes are the Controller and Model is some kind of database or api. And some others mean that Fragment-/Activity-classes are the View, another class applied to the View are the Controller and the Model also here a database or api. After some research it seems like there is no strictly correct way to do it and that it varies in different projects. Therefore the MVC-type that we have chosen is XML-files as View and Fragment-/Activity-classes as Controller because it seems like the most logical.

The fragment-/activity-class (controller) has a XML-file (view) and can always access the database (model) because it is a Singleton-object.

Model

The models that we are using are a database containing all the users and companies with all their information, such as carbon dioxide saved etc. and the api’s that were explained above. There are also two classes with hardcoded values that we use as models, the Electricity

busses and the Bus 55’s bus stops. This is not a solution that we are happy with, but it made work a lot easier in this project and it works perfectly fine when there is just a limited number of busses (one route).

View

There is at least one XML-file for every layout in the application. The XML contains all of the components that are shown when the application is running.

Controller

The controller classes are, as written above, the Fragment- and Activity-classes. As root we are using an Activity-class that is a Drawer-activity. When the view is switched you do not change the activity-class, instead a fragment-class is called, which contains a view that will be shown. All adapters and medal classes are controller classes as well.

If the XML-file contains buttons or editable text fields a listener to those are applied in the fragment class. It handles the user's input and does what the user wants to. For example in the top list view; the view loads by default with the top list for this month’s top people. If the user later on presses the button “year” the fragment class gets new data from the database with the year-top list instead of month-top list and applies that to the view.

Adapters

When you look into the code, you will see a lot of adapters. There are pager adapters and list adapters. The pager adapters are extending the built in Android class PagerAdapter, and the list adapters the Android class BaseAdapter.

All the list adapters pretty much look the same. Same goes for the pager adapters, but with important differences. Our vision was to create a listadapter and a pageradapter that will work in all cases, and not have one for each case, but we could not manage to do that. This is mostly because there was not so much time, and it was not prioritated.

External dependencies

Gson

The response that comes from the api after a api-request is a Json-object. Json was something new for all of our group members and we did not quite know how to handle it. After some research Google's Gson-library was found and it was easy to apply. Gson is a library that you can use to transform a Json-object into a Java-object and vice versa.

JUnit

Testing is a very important part of a project if you want a well-functioning application. In this project the JUnit framework has been used for testing. The reason why this framework was chosen is primarily because it is easy to use and is the most common way to test Java-code (Wikipedia). Tests with JUnit has been made for the most important parts of the model, but not all of it. The DatabaseCompany class has not been tested with unit tests in the required extent since doing so would have required connecting to the database. In order to connect to the database an Android context is needed, but when running unit tests there is no context. Instead of using JUnit for this some acceptance tests has been made. These are different test cases that should be performed manually to ensure the application works as intended. These tests cover the important parts of the model that is not covered with JUnit test, as well as the user stories.

Firebase

One of our main goals with this application were to enable multiple users, and make it possible to log in and get information about the user from anywhere. In other words a database with some server of some kind was needed. After some survey Firebase was found. Firebase is a database in the cloud where you can store and get data. It also includes an authorization tool for user login, it matches the email with the password. When a user register the password is encrypted, which makes the management of users more professional. The email is also checked to see if it is a valid email.

MPAndroidChart

To visualize how much the user has traveled by bus different bar charts are drawn. To do this an open-source library MPAndroidChart was used. This library is a good tool for drawing different types of charts.

GUI design

To make an application inviting for users it must be good looking, and a crucial part of that are its colors. Our goal were to make a stylish application, with a matted green as the main color. According to Designing Interfaces a good way to accomplish this is to choose one or two major colors as hues, and then use palettes to make different levels of brightness of these colors. This is called *Many Hues, Few Values.*

Navigation

A very important part in a well-functioning application is that it should be easy to navigate through. After looking into a number of applications with different navigation types, the group found its two favourites; drawer navigation and a navigation-bar stuck to the button of the screen. Specifically these two because of them being very common, and therefore the user will feel at home when starting to use the app. This is called *Habituation* in design terms. The user do not have to think about how to navigate, it is in her reflexes.

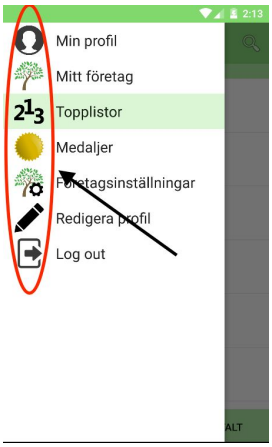
Eventually after some discussion the group decided to use drawer navigation because it seems to fit our application better. Primary because the drawer navigation will not reduces any space of the screen, and you can have more number of selections than in a navigation bar at the bottom. In android studio, there is a very good tool for achieving this. The drawer activity is shown by swiping from left to right, or pressing a button in the toolbar.

Toolbar

In almost all cases, applications have some kind of toolbar on the top of the screen. This is where you see the title of the current view that is showing, and it is where you can add tools, such as search or settings. In our case there is a tool-item for opening the drawer-navigation, and a tool-item for search (search-window where you can find other users).

Satisficing and Instant Gratification

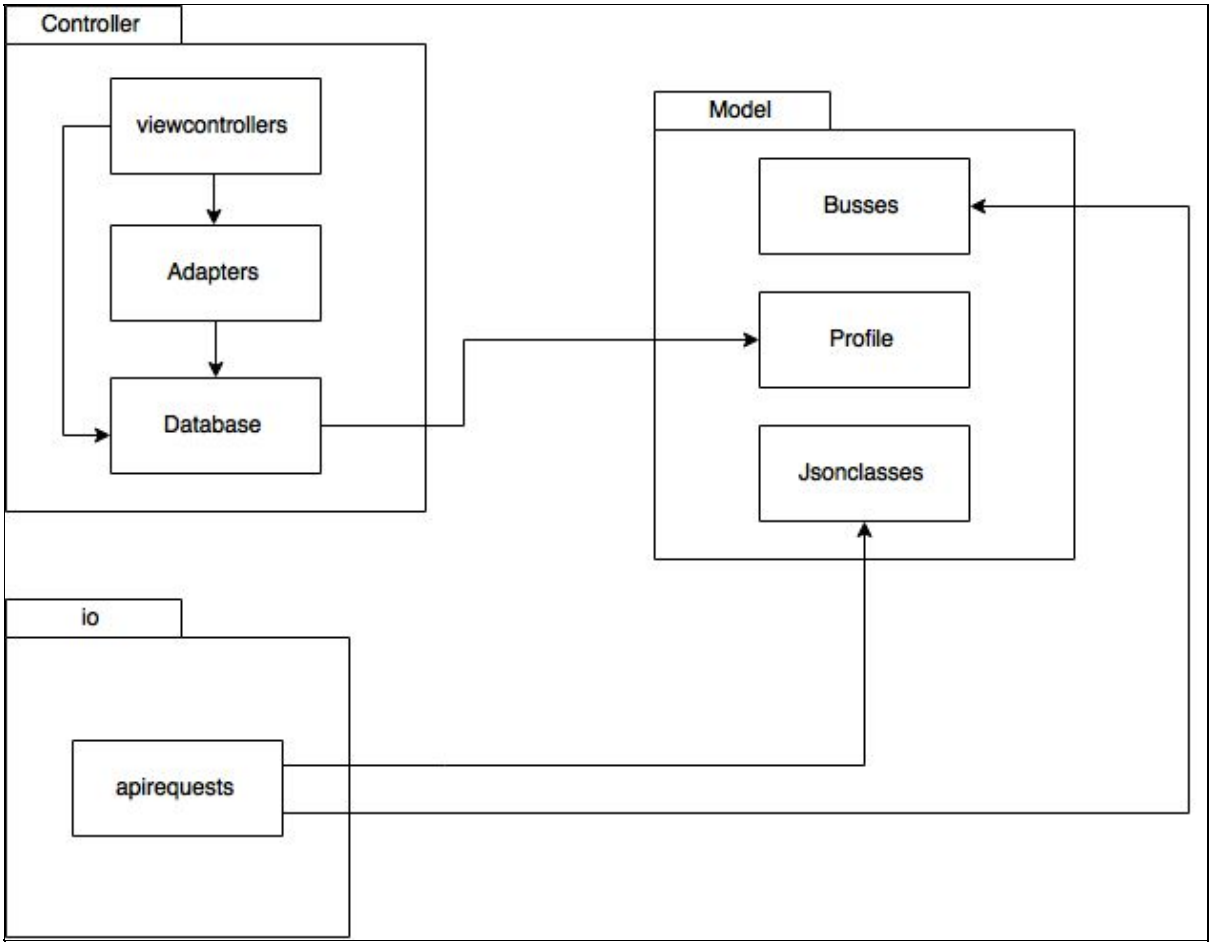
People who are using applications are often eager and want it to go smoothly, and they really do not want to waste time on, for example, finding the right button to do an action. A good way to decrease the searching-time for the user, is to apply icons to buttons and such. Using icons the user do not have to read the title of every action-item, and can find what she wants quicker.



After a user performs an action, she wants to see the result right-away. This is in other words called *Instant Gratification*. This is usually not a problem when the views load as they are supposed to. The problem is when something has gone wrong, for example if the user performs a bad action and nothing is supposed to happen but the user expects it to. In these cases so-called *Toasts* have been implemented. This is a feature in Android, and is often used in applications to give the user feedback when something has gone wrong.

UML

The UML diagram maybe look odd; no view-classes and almost no model classes. This is because, as written above, we see our XML-files as views and our models are the API:s that we are using and the database. These classes, whom are interacting with the api's and the database is in the io folder.



Källor

https://github.com/google/gson
https://www.firebase.com
https://en.wikipedia.org/wiki/JUnit
https://github.com/PhilJay/MPAndroidChart
http://designmodo.com/navigation-patterns-mobile/
Designing Interfaces 2nd Edition, Jenifer Tidwell