Technical Documentation

Beike 2015

Getting started

The repository is cloned by running:

```
git clone git@github.com:BeikeElectricity/ProjectX.git
```

The project needs has the following dependencies:

- Java 7 SDK
- Android SDK
- A (virtual) android device

Gradle

For gradle, and the project, to work properly you need to have the apikey for the api in a `gradle.properties` file. Either in the ProjectX/App folder or in your gradle home folder outside of the git root. The location of the second file is:

- Unix: `/home/username/.gradle/gradle.properties`
- Windows: `C:\Users\username\.gradle\gradle.properties`

The content of the file should be:

```
CYBERCOM_API_KEY=<insert key here>
```

You can add this at the end of the file if it already exists.

Modifying and building

The project is best modified in **Android Studio**. The building process uses Gradle and is automated by Android Studio.

Running / Building Release

Building the application can be done by running the gradle task assembleRelease either from within Android Studio or by running the command:

```
gradle assembleRelease
```

For this to work you need to have a `release.properties` file where the keystore for signing the app is configured. This file should be in the same folder as `build.grade`.

See release.properties.sample for reference on how it should be configured.

Generating a new keystore can be done in Android studio at `Generate Signed APK` under `Build` in the menu. Then pressing the button for `Create new` under keystore path in the dialog.
For further information on signing the application, see the android developer manual.

The result of the grade command is a apk named `app-release.apk` in the folder `App/app/build/outputs/apk`. If you build a signed named version, please move a copy of the apk to a new folder, named by the version, in `Releases`.

You can also make a release build by utilizing the Generate Signed APK mentioned earlier. The resulting apk will be named `app-release.apk` and will be placed in the `App/app/` folder.

Testing

Testing is easiest done by by making a new Android Test run configuration in Android Studio

Web Server

The server is intended to run on the LAMP stack,
so make sure you have that installed on your system.
For example using ubuntu run

```
sudo apt-get update
sudo apt-get install apache2 mysql-server php5 php5-mysql
```

When you have your webserver up and running you need to run the setup.sh script in the database folder, this creates a database ProjectX with the correct tables and a user beike with password beike which is granted all to said database.

Then the buses need to be added manually to the Bus table in the form

INSERT INTO Bus (vinNumber, motorType) VALUES (‘Ericsson$<vinnumber>,’MT’);

where the motor type is either ‘Electric’ or ‘Hybrid’.

After that you need to put the php scripts from the webservice directory in your document root and restart the apache2 service.

```
sudo cp webservice/* /var/www/html
sudo service apache2 restart
```

Server protocol

The server has three operations, registration of users, registration of scores and retrieving of high score. It responds to GET request by a json with two fields “success” and “message”. “success” is set to “1” if the operation was successful and “0” otherwise. The “message” fields is set differently for different operations.

Register: Register a user or update the name of an id by doing a request to

```
Register.php?id=<id>&name=<name>
```

“message” is simple a string stating how the registration wen.

AddScore: Adds a score to the database. It’s called by

```
AddScore.php?id=<id>&score=<score>&time=<time>&bus=<bus>
```

The user and the bus needs to be present in the database and the same user can’t have another score for the same time. The “message” is again simply a string stating if the score was recorded correctly.

GetHighscore: Get the top global top ten scores in the database.

```
GetHighscore.php
```

The “message” is a sorted json array with the highest score first. The field “name” holds the name, not id, of the player and the field “score” holds the number of points.

ElectriCity Innovation API

This application uses the ElectriCity Innovation API and without the api the application stops working. For more information see the relevant API documentation from ElectriCity.

System Architecture

Overview

The activities talk to the model which consists of the networking module and the Game Model module. All activities use the networking module directly, except for `GameActivity` who access the game model. The networking module talk either to the Cybercom API or to the web server which serves as a layer between the internet and the database. See appendix 1 for a graphical overview.

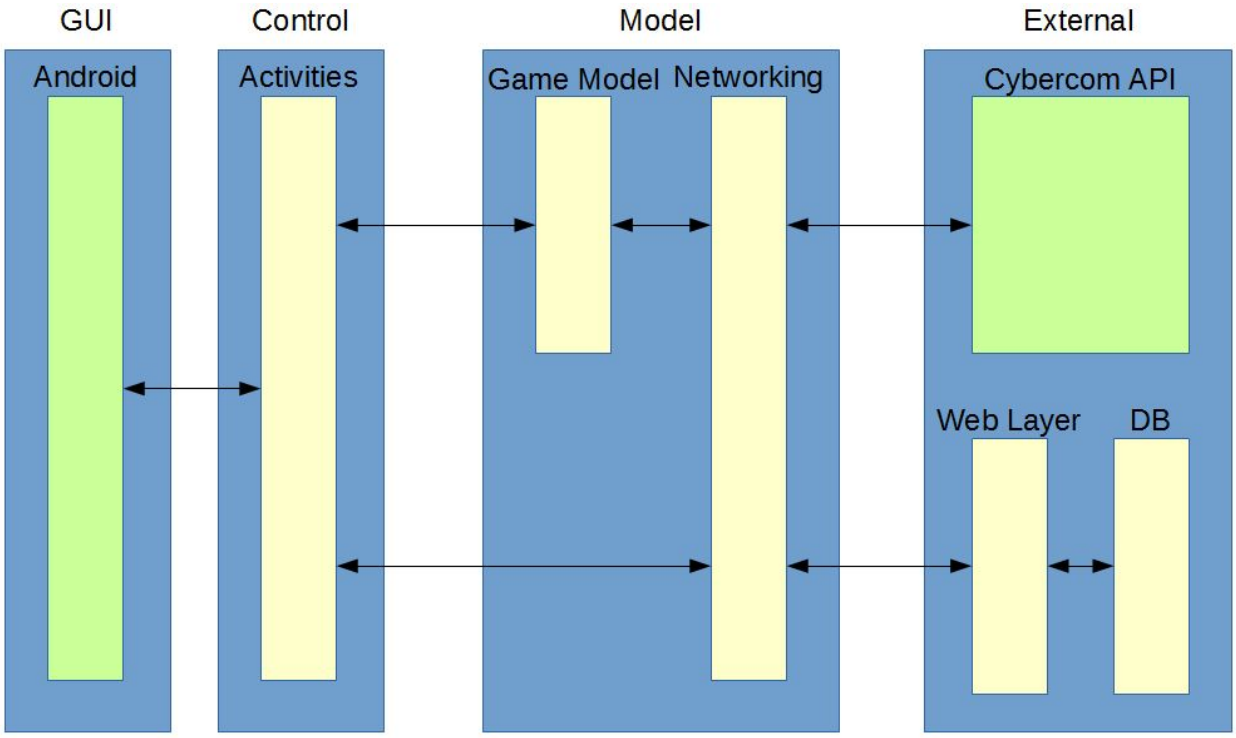
In depth

The app is divided in three different packages. Model, Android with Android Activities. Model contains a Network module that handles all communication with Electricity Innovation Challenge API and to our database. As well as the Model for the game which controls the behavior of our Game. Android contains all Android specific functions as well a subpackage with all the activities that handles interaction with the GUI, with some extra classes to handle Android's specific way to organise threads, e.g. android's handler class. See appendix 2 for a graphical explanation.

Call Chain Example

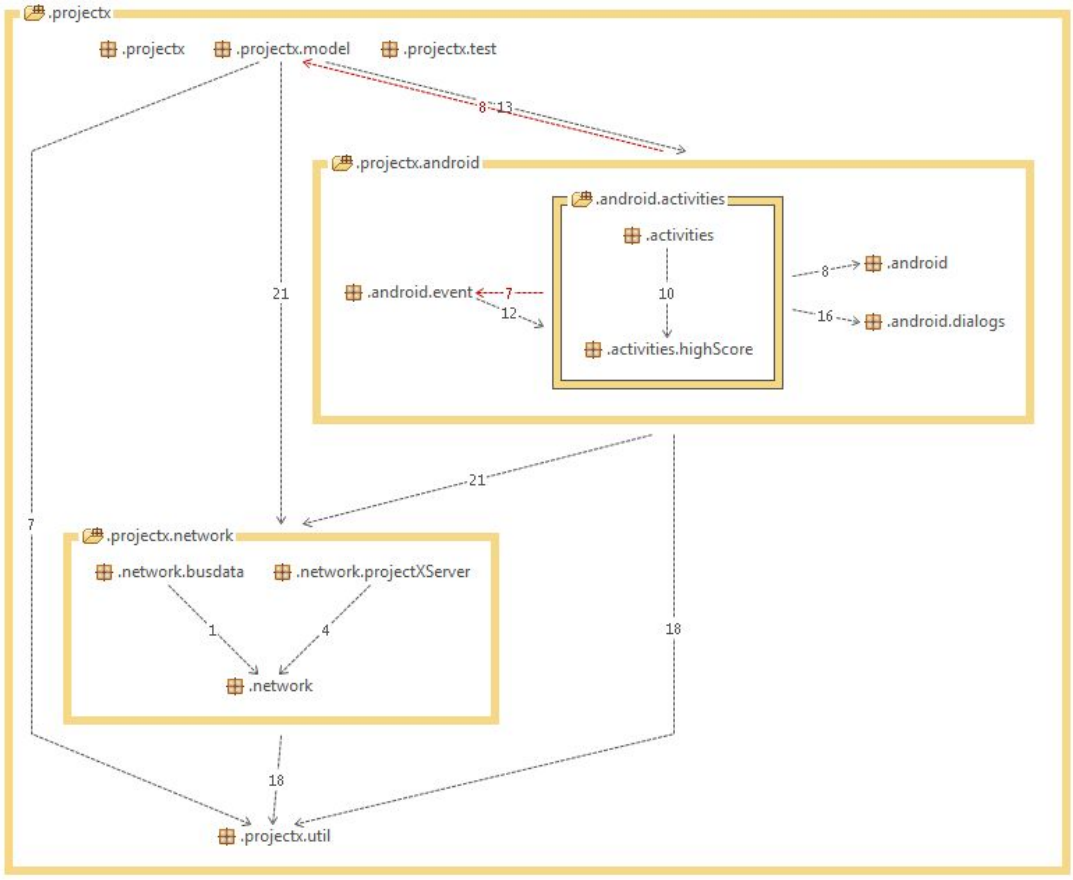
When the bonus button is clicked `GameActivity` runs a method that asks the `GameModel` to get the last time the stop button was pressed and return a new factor. To do this `GameModel` calls `Count` that makes a new thread to calculate the factor from the current time along with the time when the stop was pressed on the bus. `Count` asks `SimpleBusCollector` who then use `RetrieverReader` to get data from the Electricity API. `SimpleBusCollector` returns a `BusData` object to `Count`. Who then extrapolates the timestamp of when the stop button was pressed and then use the time difference of the stop button and the time the user pressed to calculate a bonus factor. Then makes a callback to `GameModel` to update the bonus factor. `GameModel` then updates the `GameActivity` via the `GameEventTrigger` who uses `GameEventHandler` to handle the updating back to the UI thread. See appendix 3 for a sequence diagram.

Appendix 1



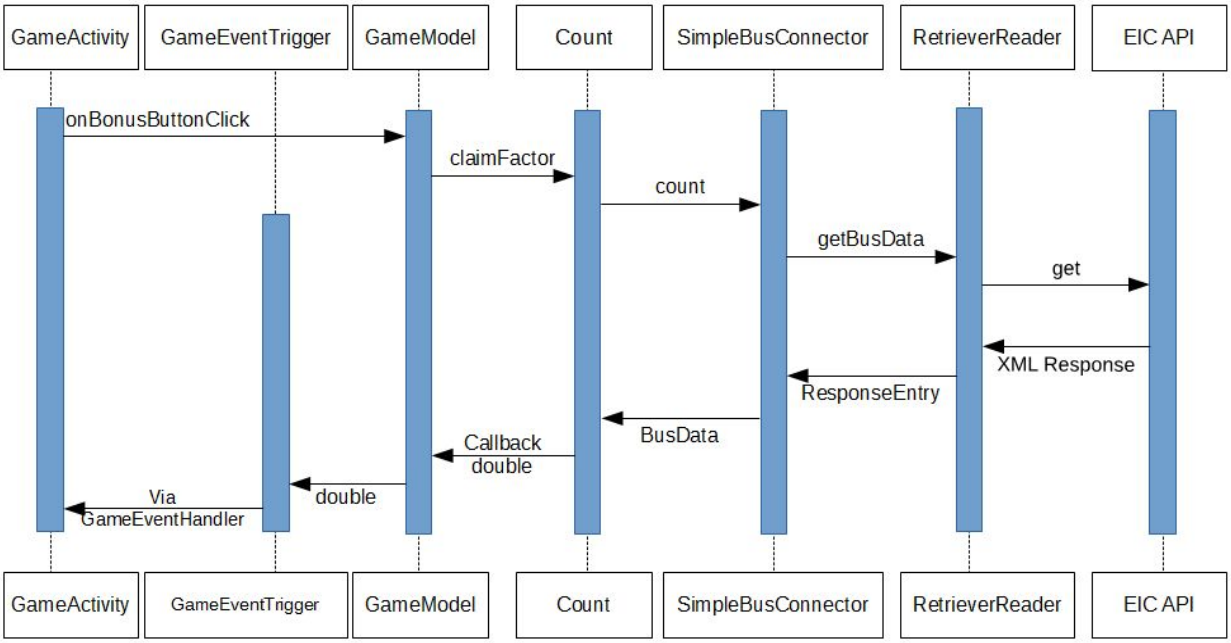
The general overview of the application.

Appendix 2



Packet overview

Appendix 3



Example of the call chain when pressing the bonus button