Developer documentation

Getting started

Drop David an email at stigdavidbergstrom@gmail.com with your GitHub username and he will add you as collaborator. When you've been added, just clone the repo:

```
git clone https://github.com/davidbergstrom/arsenal.git
```

Dependencies

To run and contribute to the project the following is needed:

- JDK 7+
- Android SDK 19+
- AGA SDK
- Gradle 1.8+
- Git
- An Android device or Emulator

We recommend that you use the latest version of IntelliJ IDEA or Android Studio with the supplied Gradle plugin (version 0.12+).

Building and installing

Gradle is used to automate the process of some common tasks, such as building, testing, running and packaging the application. To run these tasks either use the Gradle plugin mentioned in the above section, or use the Gradle Command Line tool, documentation found here (http://www.gradle.org/documentation).

A top-level build file called **build.gradle** is located in the root directory. Make sure your environment matches the configuration found here, as it applies to all modules in the project.

Another **build.gradle** file is located in the **app/** directory. It contains the build configuration (library dependencies etc.) specific for the **app module**, which currently is the only existing module.

All external libraries used in the **app module** are located as JAR files in the **app/libs/** directory. As Gradle recursively compiles all files found here, libraries can easily be kept synchronized by just adding or deleting files in this folder.

Testing

Automatic JUnit tests are included in a separate folder within the **app** module, called **androidTests**. These tests can be run using the recommended Gradle plugin or by using the Gradle Command Line tool. Note that JUnit is required to run the tests.

Contribute to the GitHub repository

To contribute with code to the project you're required to follow the Git guidelines outlined below.

Git Guidelines

These guidelines are inspired by the Gitflow worklow (http://nvie.com/posts/a-successful-git-branching-model/). In short it defines a strict branching model designed around app releases.

Master branches

The central repository, origin, holds two parallel main branches, both with an infinite lifetime.

1. Master
   The source code of HEAD must always reflect a production-ready state.
2. Develop
   The source code of HEAD must always reflect a state with the latest, fully tested, development changes for the next release.

How to use them

When the develop branch reaches a stable point and is ready to be released, it should be merged back into the master branch and then tagged with a release number. Therefore, each time develop is merged into master, there's a new production release by definition.

Supporting branches

Next to the main branches there's also a few supporting branches. These branches always have a limited lifetime, since they will be removed eventually.

The different types of branches that may be used are:

- Feature branches
- Release branches
- Hotfix branches

Common conventions for supporting branches:

Naming:

- Prefix accordingly (read more in each branch section)
- Small caps only

- Separate words using hyphens.

Branch & Merge:

- No fast forward in merges (use --no-ff flag)

Feature branches

Feature branches are used to develop new features for upcoming releases. A feature branch exists as long as the feature is in development. As soon as it's completed it will be merged back into the **develop** branch and then deleted.

Conventions:

- Prefix with *feature/*
- Branch off: **develop**

```
$ git checkout -b feature/new-feature develop
```

- Merge into: **develop**

```
$ git checkout develop
$ git merge --no-ff feature/new-feature
```

Release branches

Once the **develop** branch is ready to be released, a new **release** branch can be forked off of it. Any bug fixes and documentation related to the release should be added to it. While development of new features can proceed as usual, they can't be added until next release.

Once it's ready to be released the branch gets merged into the **master** branch and tagged with a version number. It should also be merged back into **develop** and then deleted.

Conventions:

- Prefix with *release/*
- Branch off: **develop**

```
$ git checkout -b release/new-release develop
```

- Merge into: **master** & **develop**

```
$ git checkout master
$ git merge --no-ff release/new-release
$ git checkout develop
$ git merge --no-ff release/new-release
```

Hotfix branches

Hotfix branches are used to fix any bugs in a released version of the app. Therefore it is also the only branch that should fork directly off of **master**.

As soon as the bux fix is complete, the branch should be merged into both **master** and **develop**. Master should be tagged with an updated version number and the **hotfix** branch can be deleted.
Conventions:

- Prefix with *hotfix/*
- Branch off: **master**
  ```
  $ git checkout -b hotfix/new-hotfix hotfix
  ```
- Merge into: **master** & **develop**
  ```
  $ git checkout master
  ```
  ```
  $ git merge --no-ff hotfix/new-hotfix hotfix
  ```
  ```
  $ git checkout develop
  ```
  ```
  $ git merge --no-ff hotfix/new-hotfix hotfix
  ```

Architecture

The source code resides in the following packages, ordered from the closest-to-backend to the closest-to-the-user:

- system
- constants
- models
- utils
- activities
- fragments
- views

HTTP requests using Remote

Package
system

API endpoints
World Trucker: https://api.worldtrucker.com/v1/
Google Maps: http://maps.googleapis.com/maps/api/

Response format
JSON

Usage
All HTTP GET requests are performed asynchronously by passing an arbitrary URL and a receiver object of type **ResponseHandler** to the **Remote** class**.** The **Remote** creates an instance of HttpURLConnection and uses it to perform a GET request. If the server responds with status 200 it passes the JSON response to the **ResponseHandler**.

Milestones from World Trucker

All milestones shown in the application are handled by the **Ranking** model, which uses the **Remote** to get available milestones within a specific area and of a specific type from the World Trucker API.

Manipulating the Map

Package
model

Usage
The **Map** contains an instance of a **GoogleMap** which is used throughout the class to manipulate the view of the map in an activity. The **Map** contains a **Route** which has coordinates and other information relevant for the **Map**. A **Route** is created by providing coordinates or addresses which will be used to retrieve the whole **Route** from the Google APIs. The **Route** will then create **Legs** and **Steps** that will represent parts of the **Route**. The **Route** also contains **Pauses** which has a location on the **Route**. The **Map** will listen to the **Route** with **RouteListener** for changes and alert its observers through the observer-pattern.

AGA and vehicle signals

Package
system

Usage
Communication with a vehicle is done using the AGA Jar files. In the class **VehicleSystem** an **AutomotiveListener** is implemented. The **AutomotiveListener** receives AGA signals sent from a vehicle. To add a new signal, simply implement a new case statement in the receive method in **AutomotiveListener** with the signals id and register the signal id in the class constructor.

The class **VehicleSystem** is run on a separate worker thread. For thread safety **Atomic** instance variables are used. On non primitive types synchronisation is used.

The class **VehicleSystem** implements an observable pattern. Other classes can register to listen to updates from this class. The **VehicleSystem** sends updates to the observer when vehicle states change significantly. The class listening can decide whether to to get values from the **VehicleSystem** class. This is (in this implementation) done from the class **NavigationModel**.

Communication with user interface

Package:
model

NavigationModel is a class containing the primary communication with the user interface. The **NavigationModel** contains references to both the **Map** class and the **VehicleSystem** class. It observers changes in both these classes and decides what to do with this information. It also contains an algorithm used to find pauses on the route during a drive session.

If information needs to be presented to the user interface a message with that information is sent to the main thread running the user interface. This is done using thread-safe handlers.

Most data processing in this class is done via a pipeline thread to which methods in this class can push their task to. Some method calls to the **Map** and **Route** class can NOT be pushed to run on the pipeline thread. This i because some methods in **Map** and **Route** manipulate the map view. Things like moving the camera angle for the Google map can only be done on the main thread.