Final Architecture

The overall architecture for TrojanNow is based on two major components: the server back-end and the Android application, which serves as the front-end. The front-end communicates with the back-end through REST API. Also, the communication itself is context-free.

The back-end utilizes MySQL as the storage back-end, which provides the persistent storage for all user data. I also choose the Laravel framework, which is easy to develop REST API. In addition, it also supports MVC, although in this case, it only returns JSON documents.

The front-end utilizes Android SDK as the framework. It is fully event-based, where every change of view is triggered by an event with messages. Also, it implements several connectors, which serve as the communication to the back-end. Those connectors are fully asynchronous, as mandated by the Android framework. It also implements several data repositories, which serve as indexing and caching for data retrieved from the back-end. The data repositories are mainly in-memory key-value pairs, which facilitates fast look-up for a certain abstraction object.

Architectural Style

The overall architectural style for TrojanNow is client-server. The client sends all the information to the server, which is also context-free, so that the server can execute the requests without keeping the state of the client.

The architectural style for the back-end is REST. It utilizes the HTTP protocol, with its limited verbs, to provide context-free APIs for users. Also, every resource on the server is named, mainly through their primary key in the database table. The caching and all other features of REST are naturally supported through the framework.

The architectural style for TrojanNow Android application is event-based, as this mechanism is naturally supported through Intents, which essentially serve as the messages between components. Through such event-based system, no variables between two views are shared, so each component will have full control over the messages they have received.

Implementation and Architecture

For the back-end, it utilizes MVC framework. Each controller is mapped to a controller in the code. The model in the architecture is also mapped to code, which serves as an abstraction for database tables. However, the view is combined with controller, as it only returns a JSON document, for all scenarios.

For the front-end, the architecture divides into seven major components: comments, database, login, messages, sensors, statuses, and user management. All components except the database contain one or more views in the Android application. Each component except database, has three sub-modules: abstraction, activities, and PostExecution.

An abstraction is a simple class with several getters and setters, which provides encapsulation of database objects. I also implement a PostExecution interface, which is not seen in the architecture. This is due to the Android AsyncTask, which has a method of PostExecution, as I want to separate all the

AsyncTasks, which communicate to the database, from the actual component UI update or control-flow update. Activities are naturally mapped to all activities in Android application, which serve as UI class in the architecture.

The database component serves as the client-side cache for the application, where application can retrieve data without contacting the remote server. However, in the actual implementation, I put the connector to the back-end also inside this component, since I enforce the rule that application should always use data from the local cache. Therefore, when I choose the change the implementation of back-end, the other part of the application will not need to change.

System Requirements

A back-end server that supports Laravel framework, which is PHP based. Also, a MySQL database is required, with a set of predefined schema in the provided migration SQL file. On the client side, an Android emulator, or an actual Android device, with accelerometer and GPS functionality. The minimum Android SDK version is 15, but 21 is recommended.

Implementation Details

I choose to implement my version of REST API, which serves as the back-end. Please note that the actual back-end should be listened on localhost, which in this case, a reverse proxy may be needed to facilitate the communication.

Also, since not all my message objects are suitable for serialization, and it will impact the performance, I choose to first put the message objects into a central repository. Then I only pass the key for that message object, and then the receiver will look up the table, and pick the actual object needed.

I also put the server address in the configuration file, which allows easy migration. Also, the SQL file for database migration serves the same functionality.