FestivApp

Program Manager and Browser System for Large Events

Anna Kiss
Babeş-Bolyai University
Cluj-Napoca, Romania
annacs94@gmail.com

Károly Simon
Babeş-Bolyai University
Cluj-Napoca, Romania
simon.karoly@codespring.ro

Zoltán Szilágyi
Codespring
Cluj-Napoca, Romania
szilagyi.zoltan@codespring.ro

***Abstract*—Nowadays events consisting of hundreds of happenings are encountered more frequently. The FestivApp application aims to facilitate the navigation in events' program by providing useful functionalities for browsing them. It also provides more than the already existing applications have to offer by being able to handle multiple events simultaneously. Users can browse the program of the currently chosen event using the mobile application, while organizers can edit their programs' data through the web interface. Both client applications rely on data served by FestivApp's server.**

This paper aims to present the FestivApp project. Firstly, it mentions the important decisions concerning the architecture. Secondly, it describes the implementation of the components referring to the main technologies, tools and methods that were used during the development process. Lastly, the features of the application are demonstrated using some examples.

I. Introduction

There are many festivals, conferences or other cultural events with hundreds of happenings. Their program can become incomprehensible and hard to navigate in. For a participant the browsing of a printed program guide might be uncomfortable and time-consuming. Furthermore, in this way parallel events are presented only in a linear way, which is not transparent enough, thus the participant may miss important happenings. Some festivals offer mobile applications to facilitate the browsing of the program and to offer information about the locations and the presenters. But software development is costly and there are event organizers that could not afford an application. FestivApp offers a satisfying solution for them. In addition, it is more comfortable to access the program of multiple events from a single application than to use separate applications for each festival/conference. FestivApp can handle more events uniformly, this is the most important aspect that defines its functioning. Within the mobile application, users may choose among multiple events, organizers can edit the program of their events on the administration interface. Both client applications are served with data by the server. In accordance with this, the project includes an Android mobile application, a web interface and a central server.

II. The FestivApp project

A. Functionalities

In the Android application, events are shown on separate pages (tabs), each tab corresponding to a day. The user can easily change between the programs of two days (swipe between tabs). The events appear ordered by their start date and when the view is loaded, it scrolls automatically to the current date. It is possible to filter the program by categories and locations and to search based on the name of events or the name of presenters. Users can read detailed information about the events and they can mark them as favourite. Favourites appear in a separate list, and the user receives reminders before these events. Detailed description about a location is available on a separate page, where an offline map appears with the location's place pinned.

The aforementioned functionalities are also available without Internet connection. This is necessary because in the areas where large festivals take place the network is often overloaded, therefore accessing the Internet might be problematic. To ensure the proper functioning of the application, offline mode should be supported.

Operations that require Internet connection include the synchronization of an event's program with the server. Internet connection is required for navigation to specific locations too (*Take me there!* functionality), because this happens with the help of the Google Maps application. When they are connected to the Internet, users can also receive notifications from the organizers (push notification functionality, e.g. when the program of the event is changed).

Organizers and the system administrator can enter the web administration interface. Based on their privileges, they have access to different resources. Organizers are able to check and edit their festival's program. They can add new events, presenters, locations, categories; they can delete or modify the data of already existing events; they can upload the festival's map and pin the locations' coordinates on it. Push notification messages can also be broadcasted from this administration interface.

The system administrator is not authorized to modify festivals' data, he is able only to create new festivals (without any event), he can register organizers and he can assign organizers to festivals (grant privileges).

The server stores all the data in a single database, it is able to handle more festivals/conferences uniformly. It communicates with the clients through a RESTful API, answers their queries, serves them with data and executes the requested operations. If an error occurs, the corresponding error code will be included

in the answer's header.

The festivals' data can be queried by any type of user, but deleting and modifying is only allowed to organizers. Users' information can be accessed only by the system administrator. User authorization is done using the most recent security solutions.

B. Architecture

The main components of the system are: the server, the Android client and the web user interface (Fig. 1.). These modules communicate based on the *DTO* (Data Transfer Object) pattern. The DTO classes used by every component are placed in the Common module which in this way, becomes the fourth component of the system.

The server contains the backend and the implementation of the RESTful API. The backend has a multilayer architecture: the Repository layer is responsible for maintaining the connection with the database and manipulating the data; the application's main logic is placed in the Service layer; data is represented by model classes that can be found in the Model package. The API module communicates with the Service layer, making service calls when data needs to be queried or operations need to be performed in the database; the API module parses its responses into DTOs and forwards them to the client in JSON format through the HTTP protocol.

The Android application's architecture follows the *MVC* (Model-View-Controller) pattern: the AndroidClientModel module contains the model classes that represent the data; the AndroidClientView module includes view classes that determine how data is presented on the screen of the device; the AndroidClientController module provides controller classes containing the main logic of the client application. The Android client communicates with the server with the help of the ApiClient module that is able to send requests, to receive responses and to deserialize them into Java objects. The local storage of the queried data in the device's memory is solved by the Cache module.

Similarly to the Android application's architecture, the MVC pattern is followed by the web interface's architecture too. Beside the modules corresponding to the model, view and controller components, it contains a Service module responsible for the API requests and for receiving the answers from the server.

III. The Server

The most complex component of the system is the server, that has been written in Java language. Its implementation was challenging mainly because of the complexity of the application logic and the abundance of applied technologies. This section presents the server component by analysing its architectural layers separately.

A. Data model

The central entities of the system are *JPA* (Java Persistence API) entity classes. Every class is the descendant of the `AbstractModel` that has one single field, the

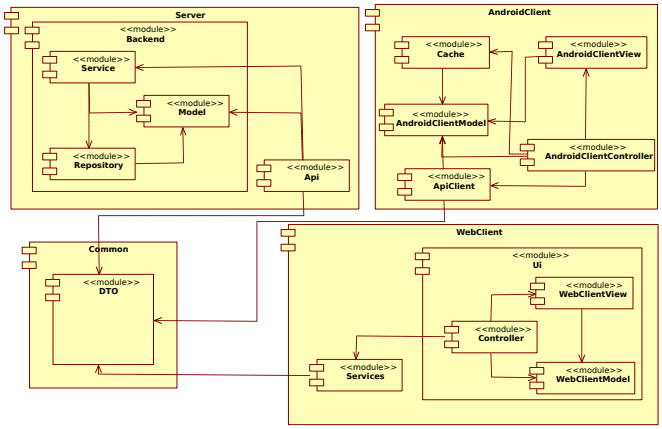

Fig. 1. The components of the server

`UUID`. This uniquely identifies the objects in the system. The `BaseEntity` class can be found on the next level of the hierarchy. Its single field `id` corresponds to the primary key in the relational database table corresponding to the class.

To reduce data transfer between the server and the client applications, it is possible to query only the data that has been modified after a specified date. The `SynchronizableEntity` abstract class is important because of this synchronization mechanism: its `lastModified` field stores the timestamp corresponding to the date when the entity was last modified, its `enabled` field marks whether the entity is enabled or not. Changing the value of the `enabled` field to false represents the deletion of the entity. Its introduction was necessary because otherwise the client applications would not be notified about data removals.

The `Program` class stores the attributes of a specific festival (name, start date, time zone). The other classes that store festival specific data can be enlisted below the `Program` entity: events, presenters, locations, categories, etc. The association between these classes and the `Program` class is not expressed via aggregation; rather it is represented by the `tenant id` that is equivalent with the `id` field of the `Program` class.

B. Component Management and Project Configuration

The most important technology used for developing the server module is the Spring framework. Spring is a lightweight framework that facilitates the development of Java applications. Its core is the *IoC* (Inversion of Control) container that manages the life cycle of the components and ensures dependency injection [1]. Other Spring Projects were also used during the development process, like the Spring Boot framework that helped to create and configure the project. Spring Boot takes over when it comes to initial project configuration, providing default values and properties specific to similar applications. In this way, an initial version of an operational application emerges really fast.

C. Data Access Layer

Spring provides plausible solutions for achieving data persistence: the Spring Data Access/Integration module enables to easily switch between technologies realising data access/manipulation, and with the Spring Data project the implementation of the data access layer becomes really easy. Especially by the predefined `CrudRepository` interfaces that make it possible to write queries by only declaring the corresponding methods. In the data access layer of the FestivApp project there are `Repository` interfaces that correspond to the main entity classes (`EventRepository`, `LocationRepository`, etc.).

The FestivApp project handles the program of different events uniformly, which means that the structure, the storage and the querying of the diverse data happens in the same way. This becomes possible with the help of the *multitenancy* mechanism: entities with similar structure are stored in the same table and are distinguished by a special identifier, the *tenant id*. With an *ORM* (Object-relational mapping) framework that supports multitenancy, queries stay simple because the constraint introduced by the tenant id does not appear in the query written at the level of the framework. The EclipseLink framework (which is the reference implementation of the JPA specification) has a really good multitenancy support, this is the reason why it is used instead of Hibernate, the ORM framework considered as default by the Spring Boot configuration. In order for multitenancy to work, one must ensure to set the value of the tenant id before every database operation. In the FestivApp project, the tenant id corresponds with the identifier of the program (`programId`). This appears in every API call tied to the event program, it is stored in a request scope bean, and when the API module forwards the request to the service layer, there is no need for the `programId` to appear in the parameter list. When the service layer would call the methods of the data access layer, the request is intercepted before the actual method invocation, with an aspect named `TenancyResolverAspect`. In the advice of the aspect the value of the `programId` is taken from the request scope bean and it is assigned to the tenant id.

D. Service Layer

The service layer contains the main logic of the application. It includes methods that check the parameters of the received requests, preprocess them, call the data access layer if necessary and forward the answer to the calling component; it logs and handles the exceptions and throws layer specific exceptions toward the upper layers (`ServiceException`).

In the FestivApp project the aforementioned logic is implemented in Spring beans annotated with the `@Service` keyword. The Service classes are structured in a simple hierarchy with a `BaseService` on the top. This contains methods that every other `Service` class would implement. Among these classes the `AccountService` has an important role - that is it contains the operations concerning user management, for example the `registerNewUserAccount` method. This ensures that no other user is registered with the specified identifier, hashes the password with the SHA-1 algorithm, and calls the corresponding method of the data access layer to persist the new information. Another example for a Service class is the `NotificationService` that is responsible for sending push notifications, using Google's Cloud Messaging service.

E. API

The FestivApp server publishes its resources through RESTful web services, which support messages in JSON format. In order to implement the REST API the Spring Web MVC framework has been used. The handlers that would be executed when a request arrives are defined in the Resource classes, annotated with the `@Controller` keyword. The URIs they map correspond with the REST standard. For example a GET request to the `/api/programs/{programId}/events` endpoint would return the list of the events that appear in the program of the event identified by the `programId`. A POST request to the same URI would insert a new event entity into the program. When the `eventId` is also concatenated to the URI, a PUT request would update the entity identified with `eventId` with the data encoded in the body of the message, whereas a DELETE request would delete the entity.

Most of the Resource classes correspond with the central entities of the application (`ProgramResource`, `EventResource`, `LocationResource`, etc.), but other classes also appear, such as the `FavouriteResource` that makes it possible to store and query users' favourite events and the `SocialResource` that contains methods enabling authentication through social networks.

Being used over the HTTP protocol, the RESTful API should support the HTTP status codes. The Spring Web MVC defaults to 200 (OK) in the case of success and 500 (Internal Server Error) in the case of any error or uncaught exception. This was completed with the 201 (Created), 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden) and 404 (Not Found) status codes. To add these error codes the exception handling mechanism of the Spring Web MVC framework has been used: it was necessary to declare special exception classes marked with the `ResponseStatus` annotation and to provide the status code corresponding to the exception as a parameter to the annotation. When an error occurs in the Resource classes, an aforementioned special exception is thrown and Spring sets the corresponding error code when building the answer.

F. Security

Web services usually restrict the access to some of their resources. In FestivApp's case secured resources are published through a RESTful API, which is based on a stateless communication model, therefore, security solutions that maintain this stateless behaviour had to be found. The team decided to use the Spring Security Framework along with the OAuth 2 standard [2]. The OAuth2 offers four types of authorization flows (authorization code, implicit, password, client credentials) from which the password grant type has

been chosen. This makes it possible to exchange the username and the password for an access token after authentication. FestivApp's server generates *JWT* tokens (JSON Web Token, IETF standard [3]). The access token is valid only for a limited period of time, so when it expires, the client requests a new one with the help of the refresh token. The latter is received in the first answer of the server along with the first access token. HTTPS is used for securing the communication.

IV. The Android Application

Android [4] is the most widespread mobile operating system, this is why it has been decided to develop the first FestivApp mobile client for this platform (the development of an iOS version is also in progress).

A. Data Model

The data model applied within the Android client resembles the server's data model. Every class is the descendent of the `AbstractModel`. Beside this there is another abstract class named `SynchronizableEntiy`. Its `enabled` field makes it possible for the client to get notified about the deletions that occur on server side. If its value is `false`, then the corresponding record needs to be deleted from the database. Every class inherits the `id` field from the `BaseEntity`. The main entities are the `Program`, the `Event`, the `Category`, the `Presenter` and the `Location`. In the `Event` class a `favourite` field can be found, the value of which is set to true if the event was marked favourite. The `Event` has also an `asciiTitle` attribute that is necessary because of the search functionality: it ensures that the search does not depend on letters with accents that may appear in the text.

B. Data Access Layer

In the data access layer the methods connected to data manipulation are declared in `Repository` interfaces, and the implementations are technology specific (*OrmLite*) beans. OrmLite is a lightweight framework for the persistence of Java objects in relational databases [5]. The `DatabaseHelper` class has the most important role in the data access layer. It can be considered a Factory, as its main task is to create the *DAO*s (Data Access Objects) and serve them to other classes ensuring that only one entity exists of them. The `Repository` classes in their constructor request DAO objects from the `DatabaseHelper` and in their methods perform the database operations with their help. DAO classes may throw `SQLException`. After logging these exceptions, a layer specific `RepositoryException` is thrown toward the upper layers.

C. The API Client

The Android application connects to the server through the RESTful API. The network communication happens in the API Client module. The creation and sending of the API calls, the reception of answers, along with the parsing of the request and response messages (from JSON to Java objects and vice versa) is performed by the *Retrofit* library [6] and the *OkHttp* HTTP client. *Retrofit*'s main benefit is that the programmer only has to declare the required methods that would be translated into API calls in `RetrofitApi` interfaces, since their definition is not required. The `ApiManager` class represents the module's core. Its main task is to create objects corresponding to the `RetrofitApi` interfaces (`RetrofitProgramApi`, `RetrofitEventApi`, etc.). When instantiating the `RetrofitApi` classes, it is necessary to define a `RequestInterceptor`, which ensures that the access token is set in the header of the requests that require authorization. The `ApiManager` defines an `Authenticator`, too which handles the *Unauthorized* error messages by exchanging the refresh token for a new access token.

D. The Synchronization Mechanism

A program may contain hundreds of events, therefore the download of the entire program happens only once, when the program is selected by the user. After that only those entities are retrieved that were changed after the last synchronization. In this way, the volume of data to be transported between the server and the clients is reduced.

The `SyncService` class holds the synchronization logic. It is a service that runs in the background and when it calls the methods of the API client, it passes the date of the last synchronization as a parameter. When the answer arrives, it invokes the methods of the data access layer to save the newly acquired data.

E. The User Interface

The main view of the FestivApp application is the `MainActivity`. This contains more fragments, between these the user can switch with the help of the navigation drawer. The most important fragment is the `PagerFragment` that holds more `DayFragment`s, each of them corresponding to a day in the program of the event. These can be switched/swiped with the help of a ViewPager. An important feature of the application is the search functionality. To implement this, a `ContentProvider` was necessary that queries the actual results from the database and displays them as suggestions. When choosing a suggestion or pressing the search button, the results are displayed. The `EventActivity` contains detailed information about an event, the `LocationDetailsActivity` holds the description of a location and a map with the coordinates of the location pinned on it. This map is a static image, accessible without Internet connection.

An important element of the application is the list of the events. Its loading, scrolling and redrawing should happen fast and smoothly. For accelerating these processes a RecyclerView [7] has been used. This component recycles the views that exited the screen during a scroll operation and updates them with the information that should enter. The dataset could be so huge that its storing in the memory could reduce performance. Therefore, it was necessary to implement a cursor-based

adapter that queries data from the database only when it is indeed needed.

In order to create a user-friendly interface, Google's Material Design Guidelines [8] have been followed. The components/elements that are implemented based on the guidelines include the floating action button (its appearance and behaviour), the navigation drawer, the tab layout, the animations, the colors and the layout. In order to provide an appearance that corresponds with the selected festival's brand, the colors had to be changed dynamically, from code.

F. Dependency Injection

When developing Android applications, an IoC container is not available, but there are libraries that make dependency injection possible. Within the FestivApp application the Butterknife framework is used for injecting UI elements and the Dagger library for injecting other components (DAO classes, classes of the API client, etc.).

V. The Web Interface

FestivApp's web interface was written in HTML language, the handling of dynamic content being done by JavaScript. The AngularJS framework [9] is responsible for maintaining the components, for defining the navigation logic and for ensuring the asynchronous communication with the server. To apply object oriented approaches, the TypeScript language was used. The user interface was built using the Bootstrap framework, which ensures a proper appearance in the majority of browsers and makes it possible for the web page to easily adapt to different screen sizes.

The data model of the web module is represented by TypeScript interfaces and with the exception of some entities and fields it corresponds to the data model of the server. The entities that store program specific information are the `IProgram`, `IEvent`, `IPresenter`, `ILocation`. The `INotification` entity represents a push notification, the `IStyle` entity contains graphical elements corresponding to an event's brand.

The Restangular module is responsible for the communication with the server and for the processing of the resources. It sends asynchronous requests, waits for the answers and processes the responses, then it sends feedback to the Controller. The feedback happens through `Promises`, with the help of callback functions. The framework is able to automatically process embedded resources, for example the `IStyle` entity is embedded in the `IProgram` and it is processed without any further operation.

Some resources of the FestivApp REST API are restricted, so it was necessary to find a simple solution to authenticate the HTTP requests. This solution was served by the HTTP interceptors. Every HTTP request is preprocessed by an interceptor that sets the access token acquired after the initial authentication in the header of the request. The same mechanism is responsible for forwarding the user to the login page if the server's answer is 401 (Unauthorized).

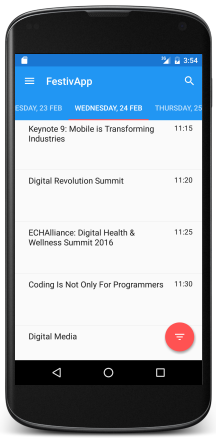

Fig. 2. Main View

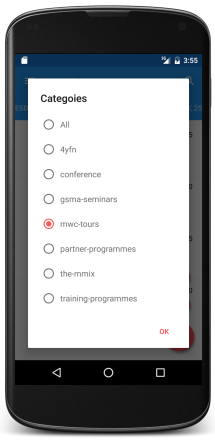

Fig. 3. Filter

HTTP requests are stateless, but user specific session information has to be stored somewhere. A good solution for this is the Local Storage introduced in HTML5. The FestivApp web interface stores here the access token, the id of the selected program and the state of the menu. The information about the current state of the view is persisted to be restored after an eventual logout.

VI. Development Tools and Methodologies

The development process of FestivApp followed the Scrum method and it was facilitated by many popular and efficient tools. In order to manage the project and to record tasks, the Trello system was used. For version control the team used Mercurial while RhodeCode served as a central repository management system. The development of the new functionalities happened in separate branches that were closed and merged to the default branch if the pull request was approved.

The Gradle system was used as a build and dependency management tool. The build process of the web module was supported by Gulp. Jenkins was used for Continuous Integration, linked with RhodeCode, Artifactory and the SonarQube static code analyzer platform.

VII. The Functioning of FestivApp

When launching the Android application for the first time, one should choose a festival, the program of which would appear on the screen. After this the main view emerges with the day by day program (Fig. 2). This could be filtered by categories or by locations. To access the filter functionality, the floating button in the bottom right corner should be pressed (Fig. 3). In order to search one should click the corresponding icon in the top right corner of the screen. If at least two characters are entered in the text box, suggestions appear in a drop-down list. The results of the search appear in a list, ordered by their start date. To read detailed information about an event, the corresponding element of the list should be clicked. If so, a new page appears with the detailed information about the event. This page contains a button, by which the

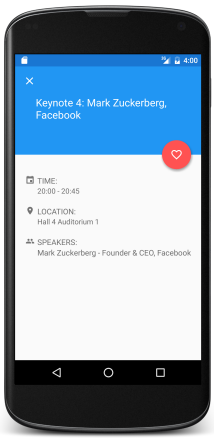

Fig. 4. Event View

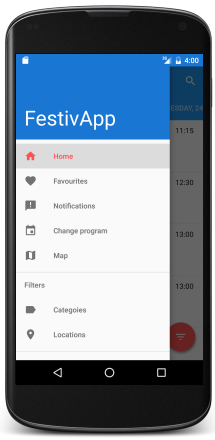

Fig. 5. Navigation Drawer

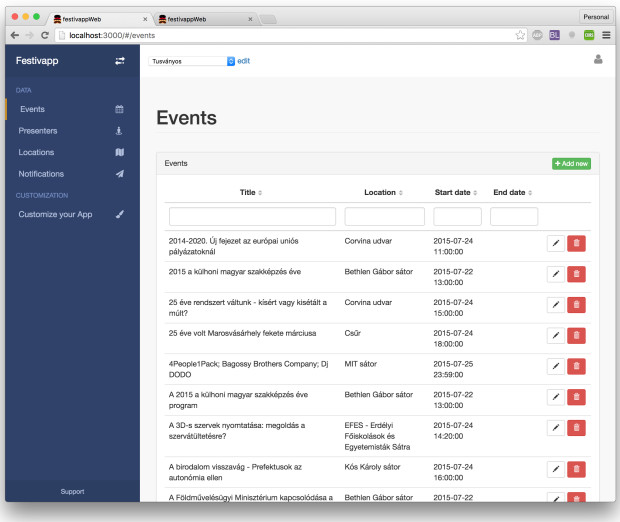

Fig. 6. Events

event can be marked as favourite (Fig. 4). An important element of the application is the menu (Fig. 5). From here, one can reach other pages and functionalities. For example, there is a separate view for the received push notifications and for the events marked as favourite, but the map of the event is also accessible and so are the pages that contain detailed information about the locations and the categories. To change between programs, the *Change program* menu item should be pressed. The web interface can be divided into three main parts: the menu, from where operations concerning the program and the mobile application can be reached; the top bar with the possibility to change between the event programs, and the main part that shows the content of the selected page. After signing in, the organizer sees the data of the selected program. The events, presenters and locations appear in a table that offers the possibility to filter, order, delete, modify and insert data (Fig. 6). For every program one can define its name, its time zone, its start and end date. A map can also be uploaded, and markers can be placed on it when editing the locations' data. Furthermore, the real geographical coordinates can be defined, too, so the client application could provide navigation with the help of Google Maps. On the web interface the selected program's appearance (colors, logo etc.) can also be defined.

VIII. Conclusions and Plans for the Future

In accordance with the objectives the team succeeded to build a general software system that offers the possibility to manage and browse the program of large events (also without Internet connection). An efficient synchronization mechanism emerged, it has been achieved to present the application in the selected event's brand/colors, and an API has been developed that could be used also by external systems if they want to access data stored in FestivApp's database.

The application is already published, and helped the participants of many events to navigate in the program. From their feedbacks, reviews and also from discussions with event organizers many new ideas have arisen, including but not limited to: a news feed in the application; weather forecast for open air events; internationalization of the program, providing the possibility for the organizers to upload the event program in different languages; localizing friends within a festival; carpool functionality; etc.

When developing the mobile application it has been an important objective to guarantee proper offline functioning. This is necessary because in the area where a festival takes place the Internet connection is usually not ensured and the mobile network may also be overloaded. This is the reason behind the idea of creating an internal chat, which would be able to use alternative channels like BlueTooth or WiFi direct when there is no connection to the Internet.

There exist plans for the further development of the web interface too. For example, the manual upload is useful if the data does not already exist on the Internet, but if other databases or static web pages already contain it, an automated import would be a better alternative.

References

[1] C. Ho, R. Harrop, and C. Schaefer, *Pro Spring 3*. Apress, 2012.
[2] E. D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749. [Online]. Available: http://tools.ietf.org/html/rfc6749
[3] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519. [Online]. Available: https://tools.ietf.org/html/rfc7519
[4] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura, *Programming Android*, 2nd ed. O'Reilly Media, 2012.
[5] G. Watson, *OrmLite - Lightweight Object Relational Mapping (ORM) Java Package*. [Online]. Available: http://ormlite.com/
[6] Retrofit: A type-safe HTTP client for Android and Java. Square. [Online]. Available: http://square.github.io/retrofit/
[7] RecyclerView. Android API Reference. [Online]. Available: http://developer.android.com/reference/android/support/v7/widget/RecyclerView.html
[8] *Material Design Guidelines*, Google. [Online]. Available: https://www.google.com/design/spec/material-design
[9] *Angular JS API Reference*, Google. [Online]. Available: https://docs.angularjs.org/api