Program Guide and Data Management Software System for Theaters

István Bege
Babes-Bolyai University
Cluj-Napoca, Romania
begeisti@yahoo.com

Melinda Tóth
Babes-Bolyai University
Cluj-Napoca, Romania
melindatoth24@yahoo.com

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

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

Levente Kintzel
Codespring
Cluj-Napoca, Romania
kintzel.levente@codespring.ro

***Abstract*—There is an increasing demand for applications in which users can easily browse different events organized by different institutions or organizations in a single place. Among these events are the following: theater or movie shows, cultural programs, presentations, meetups, etc.**

Most of the institutions (e.g. theaters) have their own web pages, where the programs along with the related news and pieces of information are published. These websites differ from one another and sometimes it may become difficult for the users to digest the required pieces of information from them. The aim of the software system is to offer a solution for this problem. With a single mobile application, using a uniform user interface they can browse the program of all those institutions, which uploaded their events into the system through the web administration interface. In the current version of the system mainly theaters are targeted but the solution will be generalized to support other institutions too.

The article presents the PlentyGO system, which contains two client applications and a central server.

I. Introduction

The current version of the PlentyGO software system can be used as a digital guide for theater shows, giving a whole new approach for theaters to publish their events. It also provides a solid advertisement possibility for the institutions, popularizing their shows to present them in a widely accessible manner.

Currently, if one is going on a trip to another city and decides to see a show in the city's theater, he/she has to search for its website to find its location and browse its program. Using the PlentyGO application this process becomes easier. In case the theater's repertoire is uploaded into the system, the tourist can simply open the PlentyGO Android application, where the shows can be easily filtered by city, by date and by institution. By tapping a show in the list view a full description is going to be displayed, and in case this is the right event, the application can navigate the user to the respective location.

The main advantage of the system is that the content can be internationalized, the data can be provided in multiple languages. The supported languages can be selected by the theaters. The users also have the possibility to setup their preferred language within the mobile application. The data will be downloaded by the Android application during the synchronization process, but only in the selected language, which by default is the device's main language.

There are several similar solutions on the market targeting festivals or conferences. There are white label solutions, like Greencopper or Appmiral, and some centralized platforms and general applications are also available. For example, FestivApp [1] is a system developed and published by a team consisting partially of the present thesis's authors. It is a general application, but it can be also used as a white label solution. Currently the system is already used by more than 50 festivals, thousands of users, and the team also received several requests from theaters and other institutions. However FestivApp was designed to handle large events (e.g. a festival or conference), with a relatively short duration (several days). Managing the periodic events organized by an institution is not the same problem. So, the team decided to build a new solution for these different requirements. The architecture of the PlentyGO system is similar to FestivApp, but the model and the features are different, and there are also differences in the selected technology stack.

II. The PlentyGO project

A. Main features

The theater shows can be browsed using the Android application. The selected language can be changed in the settings menu. The application can be used without registration, but some features will be available only for registered users. The registration could be done via Facebook or Google.

The main screen of the application contains a list view with the institutions (currently only theaters are supported), and a timeline view with the shows. From here the users can navigate to data sheets describing shows and theaters, and they can get detailed information about these entities. There is a possibility to filter the shows by city, theater and date. The shows and institutions can be marked as favorites and these favorites can be accessed easier within a separate list. Using Google Maps the application can navigate the users to the exact location of the shows.

The Android application can be used even if there is no data connection. The downloaded data is cached using the local storage of the mobile device, it can be accessed offline, and it can be synchronized later, once data connection is available.

There are two administrator roles: system and institution (theater) administrators. They can modify the content using a web user interface. There is a possibility for these users to select the supported languages (currently the system has support for Hungarian, Romanian and English). Institution administrators can see and update only those theaters and shows,

which are managed by them. For the system administrator all the institutions and shows are visible and editable. New institution administrator users can be registered only by the system administrator and institutions can be assigned to them.

The administrators can setup the supported languages for each theater and show, and the corresponding content has to be provided in these languages. A cover photo can be uploaded for each institution and a poster image for each show. Multiple start dates and times can be introduced for each show.

Using Google Maps the administrator users can set the location of a theater within the web application. It is possible to give the street address also in multiple languages , and assign it to a city. New cities can be introduced by system administrators.

The data used by the client applications is provided by the central server. Its main purpose is to authenticate the users and their requests, to handle these requests and to manage the persistent data.

B. The structure of the system

The central component of the system is a Java server, which stores the data in a MySQL relational database. These data will be queried by the client applications using a RESTful API. In order to support offline usage the data is also persisted by the Android client application after the synchronization process, using a SQLite database.

C. Architecture

The system is composed by three main components: a central server, an Android and a web client application. The communication between these components is implemented based on the Data Transfer Object (DTO) design pattern[2], within the *commons* module.

The server is composed by a backend and by an API layer. In the backend a *Model* module is responsible for representing the main entities of the system. The *Repository* module contains a JPA repository which is responsible for providing the Data Access Layer. The last module in the backend is the *Service* module, which contains the business logic.

The API layer contains a *Resource* module, where the REST requests are handled. It calls the *Service* module to execute the given request and to build the response. Before the response will be sent, the entities are converted into DTOs by the *Assembler* module.

The web client is composed by two layers. A *Service* layer is responsible for the communication with the server's API and it provides the data to other web components. The *UI* module contains the controllers and the corresponding HTML views for the web pages. The controller contains the view logic and it fills up the view with data.

The Android client has its own *Model* module for data representation and its own *Repository* module for persisting the models using a SQLite database. The conversion between the DTO and model objects is performed by the *Assembler* module. The *API* is responsible for the communication with the server's API. The UI layer contains the components needed by the view. The *Controller* provides the connections between these layers.

III. The Server

Both client applications are supplied with data by the central server. This component is the most complex part of the system. It is written in Java programming language using Spring technologies.

A. Technologies

Spring[3] is an open source application development framework for the Java platform. It supports the development of complex enterprise software systems and web applications. The life cycle of the components is managed by the Inversion of Control (IoC) container and the dependencies between them are also automatically resolved based on the Dependency Injection (DI) design pattern.

The Spring Boot[4] module made the project configuration easier and provided the embedded web server. The module favors the convention over configuration design paradigm. The application can be configured using simple Java classes without complex XML files. The central properties are stored in an external *YAML* file.

The client applications are served with data by the RESTful API. The API is implemented using the Spring Web MVC module which also provides the serialization and deserialization process for JSON objects. It also gives a possibility to upload multi-part files.

To secure the API, the server uses the Spring Security[5] framework. Its main task is to authenticate the users and to verify their roles for each request. This is done by the Spring Security OAuth2 module, which provides limited access to HTTP services for external applications.

B. Data Model

The main entities of the system are simple POJO classes annotated with JPA annotations (@MappedSuperClass, @Entity). On top of the class hierarchy stays the *AbstractModel*, which contains a Universally Unique Identifier (UUID) attribute, supporting the exact identification of the entities within the distributed system. Its child class is the *BaseEntity* class, which has a *Long* identifier corresponding to the primary key in the database. Those models which are transferred to the mobile application are derived from the *SynchronizableEntity* superclass. This class contains a *deleted* flag, which by default is false, and a *lastModified* timestamp, which stores the date of the last modification in UTC. The flag indicates the validity of the entity. If its value is true, the entity has been deleted on server side, so it should be deleted from the client database too. The timestamp is used for the mobile synchronization process. Only those entities will be returned by the API, which have been updated after the given timestamp. The timestamp will be the date-time combination of the last synchronization.

There is another superclass in the hierarchy for multilingual entities like institutions or shows. This is the *MultilingualEntity* class, which extends the *SynchronizableEntity* superclass.

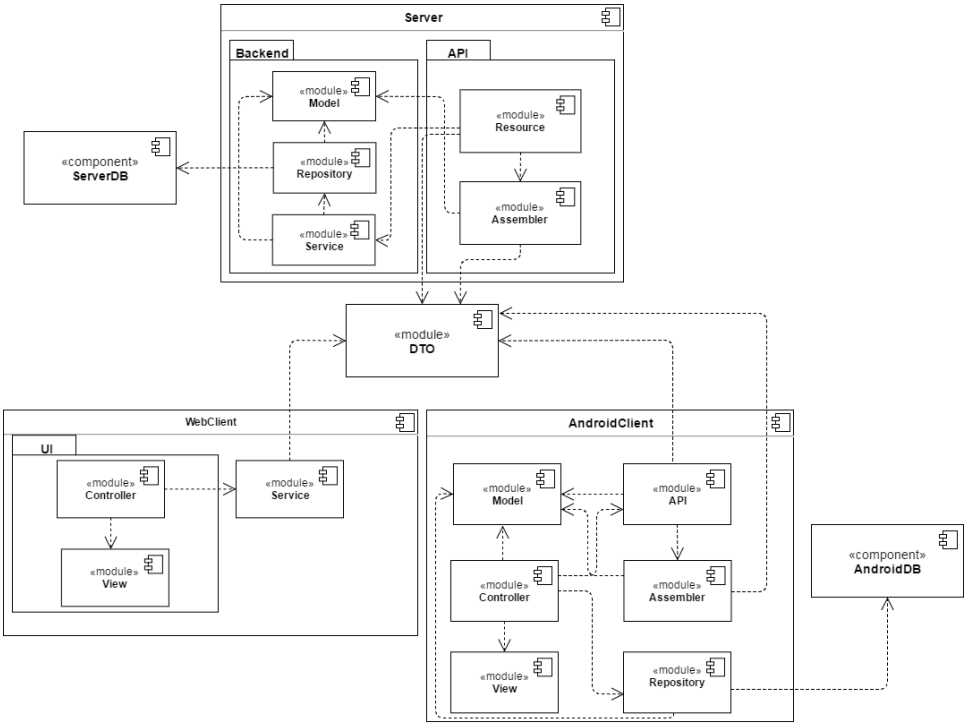

Fig. 1: The main components in the PlentyGO software system

It no longer uses the *@MappedSuperClass* annotation like its ancestors. The *@Entity*, *@Table*, *@Inheritance(strategy = InheritenceType.JOINED)* triplet is used, which means that it has its own database table and this is the base table for all the multilingual entities. Only the specific attributes are stored in the tables corresponding to the entities derived from this base class (city, location, institute, etc.) and join operations are performed at select statements. The *MultilingualEntity* class holds a default locale and a set of available locales which are supported by the system. The collection holds only those locales in which the given entity's multilingual data is filled. This becomes very handy during the synchronization process.

C. Content internationalization

Internationalization the content was one of the most significant challenges of the development process, because it affects the server and the client applications too.

The Repository layer on the server must know how to handle multilingual data. This layer is provided by the Spring Data JPA[6] module and behind that the Hibernate ORM framework is used as JPA implementation. The first task was to configure the framework correctly by adding JPA annotations to the model classes and attributes.

At the data representation level each multilingual entity is represented by the *MultilingualString* class. It is extended from the *BaseEntity* and has a single Map attribute. The collection holds the multilingual data, where the key object is a locale supported by the system. The collection's value is a *MultilingualData* object, which has a single string attribute. This is the translated text corresponding to the given locale. The class uses the *@Embeddable* annotation instead of *@Entity*, because there is no need to identify these strings inside the application. Their use is meaningless without the corresponding key locale.

The relationship between a multilingual data and its translations is one-to-many, so two tables are required to store these in the database. The Map collection is described by three annotations: *@ElementCollection*, *@MapKeyJoinColumn* and *@CollectionTable*. The first one indicates the type of the relationship, the second one gives the name of the key object in the collection table defined by the third annotation. The column for the JOIN operation is also given in the last annotation. Each entity which holds a multilingual data declares a *MultilingualString* object at the class level.

While the data is synchronized only in a specified language by the Android application, the data can be managed in all the available languages within the web user interface by the administrators. Because of this, two DTO classes have been implemented for each multilingual entity in the *commons* module. Those with *Localized* prefix contain data in a single language, while those with *Multilingual* prefix contain data in all supported languages.

Some of the DTOs need more than one *Assembler* class. Because of this, each model with multilingual attribute has two *Assembler* classes. Those that extend the *MultilingualBaseAssembler* class convert the models to multilingual DTOs and vice versa, while those that implement the *LocalizedBaseAssembler* interface convert the models to localized DTOs. The localized DTOs are never converted to models, because it would lead to data loss.

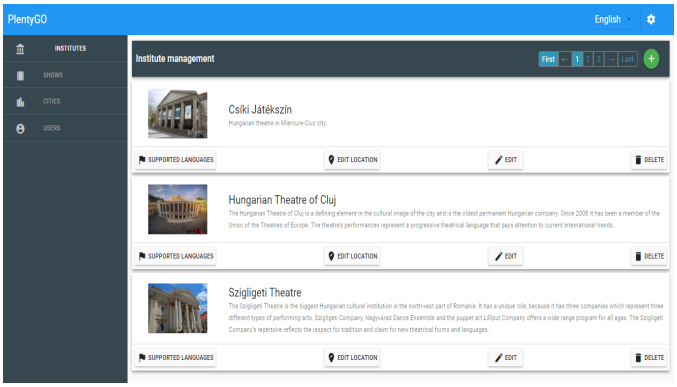

Fig. 2: The institutions menu on the web administration interface

D. Integration of Social Networks

The Android application gives a possibility for users to login via social networks (Facebook and Google). The selected provider returns an access token for the application, however the user cannot be identified on the server with this access token, because the authentication has happened against a third-party server. To resolve this issue, an authentication process has been implemented on the server, the so-called "Reverse OAuth authentication". The process essentially obtains an external access token from a given provider, then send it to the server along with the provider's name. The server then connects to the given provider's authorization server and verifies the user's credentials using the received access token. In case the user does not yet exist in the database, the server creates it, sets its role and its type according to the provider. Then saves it into the database. After the user is saved, the server simulates an authentication request. The result is an access and a refresh token, and these tokens are returned to the mobile client. The Android application now can use these tokens to identify the user during the interactions with the server.

IV. The Web administration interface

The administration user interface is a stand-alone web application, which has no physical connection with the central server. It has its own web server, which serves its static files. As soon as these files are served by the server to the browser, the web page behaves as a single-page application and it generates dynamic content using JavaScript.

The application is developed using the *AngularJS*[7] front-tend framework. The framework is responsible for managing the web components, it resolves the dependencies between these components based on the DI design pattern and provides a data binding function, which automatically resolves the DOM manipulation when the corresponding model is updated in the background. TypeScript is used as script language, because it supports the object-oriented programming paradigm and it gives the possibility to use basic types as well. This script language is not directly supported by the browsers, so a Gulp task compiles it to plain JavaScript. The user interface is developed using the AngularJS Material Design[8] framework, which is the reference implementation for the Google's Material Design specification. The design is based on Material Design guidelines.

The data is represented by TypeScript classes. With some differences, these classes correspond to the DTO objects from the *commons* module. The main entities are the *Institute*, *Show*, *City*, *Location* and *User* classes.

The web application uses the *Restangular*[9] module to send the REST requests to the server. The module provides these functionalities by just a few lines of code and all HTTP methods are supported by the framework. The requests are executed asynchronously, so there is no guarantee when the actual answer arrives. To handle this, asynchronous functions are returning promises, to which callback functions might be assigned, whether the promise was fulfilled or rejected.

Both system and institution administrators are able to select their most appropriate language on the web page. Currently they can select English, Hungarian and Romanian. The application uses the *angular-gettext* module to display the multilingual web labels on the selected language. Its use is quite simple, it is enough to annotate the multilingual labels in the source code with a corresponding HTML tag. Having every label annotated, a Gulp task processes the source files and generates a template file, which contains all the translatable strings. With the help of the Poedit editor, these strings can be translated to any language. The translations are saved into files with .po extension. The file names correspond to the language of the translation. In order to use these translations in runtime, a Gulp task converts the *po* files to JSON format and adds them to the *classpath*. In this way the internationalized data can be downloaded by the clients. When the page language is changed, the module loads the corresponding JSON file and all the multilingual labels are automatically changed on the UI.

Each institution administrator has the possibility to manage the data, but only for the assigned institutions. The server returns only those institutions and shows, which are managed by the current user. For implementing this mechanism, it was essential to identify the web users. After an email address and a password is entered by the user on the login form, the *AuthService* web component sends an authentication request to the server. If the given credentials are correct, the server generates an access token together with a refresh token and sends it back to the client. These tokens are *JSON Web Tokens*, in which the user's name and his/her roles are also encoded. The storage of the tokens is provided by the HTML5 *Locale storage* feature, which saves the given data into the user's browser. The access token is added to every HTTP request header by a *HTTP interceptor*. In this way the server always knows who is the sender.

Each token has its own expiration date. If the server signals with the corresponding HTTP status code that the token has expired, the interceptor calls the *AuthService* to renew that with the refresh token. If this token is expired as well, the

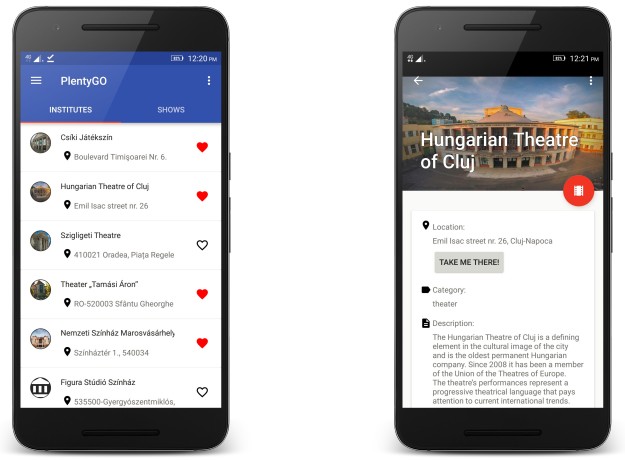

Fig. 3: Theater list and detailed view within the Android application.

application requires a new login from the user.

V. Android client

The structure of the Android application follows the MVC design pattern. The model classes are responsible for data representation, the Activity classes act as controllers and the UI components are defined in XML files, together with the relationships between them.

The same way as on the server, the model classes are simple POJOs with the corresponding *OrmLite*[10] annotations. The main entities are the Show, ShowDateTime, Institute, Location, City and User. The FavoriteShow class represents the user's favorite shows. Each show has a genre (ShowGenre), each user has a type (UserType). These attributes are enums. The *AbstractModel* stays on the top of the class hierarchy, while the *SynchronizableEntity* abstract class stores the entities' state using the deleted flag.

In the Repository layer a *DatabaseConfigUtil* class generates the database schema according to the model classes. The CRUD operations are provided by Data Access Object (DAO) classes. Every model class has its own DAO instance with the necessary database operations. The instantiation of the DAO classes is done by a *DatabaseHelper* class, which operates as a Factory. It returns only a single instance for each DAO.

The views are provided by Activity classes, while the user interactions are handled by Fragments. When the application starts a *LoginActivity* appears for the user. From here, the application navigates to the main view, which contains two tabs. The first tab contains the theater list, which is provided by the *InstituteFragment*. The second tab is provided by the *ShowFragment*, and it contains a list with available theater shows. On the *ShowDetailsActivity* users can get detailed description about a show. The *InstituteDetailsActivity* provides this function for theaters. Tapping the corresponding menu icon in the main view, a *MenuDrawer* appears where users can select the content's language. This is done by the *SettingsActivity* class.

The communication with the server's RESTful API is implemented using the *Retrofit*[11] library and the *OkHttp* client. For each entity a *Retrofit* interface is implemented, which contains the REST requests addressed to the server. The library automatically converts the JSON response objects to the corresponding DTO objects. The Assembler module converts the DTOs to models.

There are two authentication mechanisms for the mobile users: they can login into the application using a social network account, or as a simple guest user. When the login is performed through a social network, the application requires an access token from the selected provider and sends it as a JSON object to the "/oauth/social" endpoint. The server verifies the validity of the token, and if the authentication was successful, it returns its own token pair. The application uses the received access token in order to be authenticated on the server by signing each request. In case that the device has no data connection, users can login with a simple guest user and can browse the content which was synchronized previously.

The synchronization mechanism is done by the *SyncService* class. It contains a function for each entity, that queries only those data from the server, which were updated after the last synchronization. The timestamp of the last synchronization is saved for each entity in the application's Shared Preferences storage. As a response, the server sends a list of *MultilingualEntityDTO* objects along with the actual timestamp in UTC. Every *MultilingualEntityDTO* contains the entity's id, a deleted flag, a default locale and a list of available locales. When this response arrives to the client application, each available entity is queried by its id, one by one in the application's language (if it is supported). In case the entity does not provide its content on the selected language, the API queries it corresponding to its default locale. Those entities whose deleted flag is true, will be deleted from the application's database. As a last step, the API updates the received timestamp for the entity in the Shared Preferences.

VI. The use of the PlentyGO system

A. Web user interface

After a successful login, the user might select the most suitable language for displaying the web page. The first menu is the theater management (Fig. 2), which contains a list with the available theaters. For system administrators all the theaters are displayed in this list, while institution administrators can see only the theaters assigned to them. For each of the theaters, administrators can set the list of supported languages in which the institution provides the content. There is a possibility to give the exact location of the theater using Google Maps, so that mobile users can navigate there easier. To upload a cover image for an institution, it is sufficient to click on the existing image and select a different one.

The second menu contains the management of the shows. For each show, the list of supported languages can be set

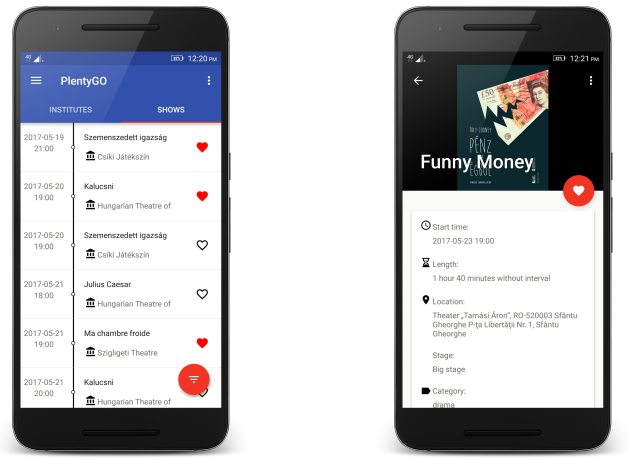

Fig. 4: Show timeline and detailed view.

separately. The users can add start date-time combinations for each performance using a modal window.

The third menu is reserved exclusively for system administrators, where they can add more cities to the system. These cities can be assigned to locations, in this way the shows can be filtered by locations within the mobile application.

The last menu is where the user management happens. The system administrator can add new users and assign theaters to them.

B. Android client

Users have to be authenticated by the server before using the application. This could be done when the application is launched for the first time. Performing this once is sufficient, as the application saves the settings and the login process is automatically repeated when the application is launched subsequently.

In the main menu of the application, users can switch between two tabs. One stands for the list of institutions (Fig. 3), whereas the other contains their shows. By tapping an institution on the first tab, a description view appears, where the user can get detailed information about the theater in the application's language. If the theater's location is given, a "Take me there" button appears on the screen. By pressing it, the Google Maps application starts and shows the exact location on the map and it also offers a navigation possibility for the user.

The second tab contains a list with available shows, which are displayed in a timeline view (Fig. 4). There is a floating button on the bottom right side of the screen, which provides filtering features. The displayed shows can be filtered by favorites, cities, theaters and dates. Pressing the heart icon on the right side of the show will mark it as a favorite. The favorite list can be accessed later more easily. Tapping a show element, a detailed description screen can be opened for the selected event. Here the user can find out the show's genre, the name of the director, the list of actors, etc.

On the left side menu users can set the preferred language for the content and also can log out from the application. After a logout operation a new login will be required from the user at next start up.

VII. Conclusions and Further Development Plans

PlentyGO is already a functional software system and it can be used by theaters for managing and publishing their data. The mobile application can be used by the users as a digital guide for theater shows. The shows published by different institutions can be browsed on a unified UI. For data synchronization data connection is required, but the mobile application can also be used in offline mode, displaying previously synchronized and cached content.

During the development process several new ideas have emerged as further development possibilities. For example:

- the possibility for assigning more media contents to theaters and shows;
- introducing a notification system to notify the users if a new show is added to the repertoire of a favorite theater or a favorite show will be scheduled again for presentation;
- the users could share their opinions and experiences about theaters and shows in comments and a rating system could also be implemented;
- extending the system by introducing new types of institutions and events (e.g. cinemas, museums and other cultural institutions, sport facilities, etc.);
- as a long term goal, a new module could be introduced, which is suitable for ticket purchasing and for seat reservation.

References

[1] A. Kiss, Z. Szilágyi and K. Simon, *FestivApp: Program manager and browser system for large events*, 2016 IEEE 14th Symposium on Intelligent Systems and Informatics (SISY), 2016.

[2] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2012

[3] R. Johnson, et al., *The Spring Framework - Reference Documentation*, [Online], Available: http://docs.spring.io/spring-framework/docs/2.0.x/reference/index.html

[4] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, S. Deleuze, M. Simons, *Spring Boot Reference Guide*, [Online], Available: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/

[5] B. Alex, L. Taylor, R. Winch, G. Hillert, *Spring Security Reference*, [Online], Available: https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/

[6] O. Gierke, T. Darimont, C. Strobl, M. Paluch, *Spring Data JPA - Reference Documentation*, [Online], Available: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[7] Brad Green, Shyam Seshadri, *AngularJS*, O'Reilly Media, 2013

[8] *AngularJS Material - Introduction*, Google, [Online], Available: https://material.angularjs.org/latest/

[9] *AngularJS service to handle Rest API Restful Resources properly and easily*, [Online], Available: https://github.com/mgonto/restangular

[10] Gray Watson, *OrmLite - Lightweight Object Relational Mappin (ORM) Java Package*, [Online], Available: http://ormlite.com/

[11] Lars Vogel, Simon Scholz, David Weiser, *Using Retrofit 2.X as REST client - Tutorial*, [Online], Available: http://www.vogella.com/tutorials/Retrofit/article.html