České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jakub Špatný**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Mobilní aplikace pro podporu organizace Dne Otevřených Dveří**

Pokyny pro vypracování:

Navrhněte a implementujte systém pro synchronizaci skupin studentů v rámci dne otevřených dveří. Systém se bude skládat ze dvou aplikací, první bude aplikace pro mobilní telefony a tablety (preferujeme android), druhá aplikace bude webová.

Mobilní aplikace bude určena pro osoby provádějící studenty po budově, budou do ní zaznamenávat informace o aktuálním stavu skupiny (pozice, počet, atd.) a dále bude aplikace zobrazovat informace o ostatních skupinách.

Webová aplikace bude určena pro správu systému a pro online přehled stavu.

Při návrhu aplikace spolupracujte se skupinou lidí, která je zodpovědná za přípravu DOD akcí na Karlově náměstí.

Aplikaci otestujte s reálnými uživateli.

Seznam odborné literatury:

Professional Android 4 Application Development - Reto Meier, Wrox 2012.

Satya Komatineni, Dave MacLean, Pro Android 4, Apress, 2012.

Vedoucí: Ing. David Sedláček, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 26. 3. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor's Thesis

Mobilní aplikace pro podporu organizace Dne Otevřených Dveří

Application for team synchronization during Visitors' Day

Jakub Špatný

Supervisor: Ing. David Sedláček, Ph.D.

Study Programme: Software Technologies and Management

Field of Study: Software engineering

May 18, 2015

Aknowledgements

I would like to thank Ing. David Sedláček, Ph.D. for supervising this thesis, and my friends and family for their support during my studies.

Declaration

In Prague on May 18, 2015 ....

Abstract

This thesis describes the analysis, design, implementation and testing of a system which aims to solve problems concerning group synchronization during open days at the Czech Technical University in Prague. The implemented system provides web and Android mobile clients which display the current locations of selected groups in real-time. With this information, group guides should be able to avoid collisions with other groups on the same path.

Abstrakt

Předmětem této práce je analýza, návrh, implementace a testování systému, který má za úkol vyřešit problémy týkající se synchronizace prováděných skupinek v rámci dne otevřených dveří na Českém vysokém učení technickém v Praze. Implementovaný systém poskytuje webového klienta a mobilní Android aplikaci, umožnující sledování polohy jednotlivých skupinek v reálném čase. S touto informací by měli být průvodci schopni zabránit kolizím s ostatními skupinkami na stejné trase.

Contents

1 Introduction **1**
1.1 Goals and motivation . . . 1
1.2 Thesis structure . . . 1

2 Problem description and requirement analysis **3**
2.1 Open Day Organization . . . 3
2.1.1 Main Blocks . . . 3
2.1.2 Organization stages . . . 4
2.2 Requirements analysis . . . 5

3 Analysis and system design **9**
3.1 Domain Model . . . 9
3.1.1 User . . . 9
3.1.2 Event . . . 9
3.1.3 Route . . . 9
3.1.4 Station . . . 10
3.1.5 Group . . . 10
3.1.6 LocationUpdate . . . 10
3.2 Use Case Model . . . 11
3.2.1 Actors . . . 11
3.2.2 User Management and Authentication . . . 11
3.2.3 Event Management . . . 11
3.2.4 Position Synchronization . . . 12
3.3 System Architecture Design . . . 13
3.3.1 Client Tier . . . 13
3.3.2 Application Server Tier . . . 13
3.3.3 Database Tier . . . 14

4 Server Application Implementation **15**
4.1 Layered Architecture . . . 15
4.1.1 Data Access Layer . . . 16
4.1.2 Business Layer . . . 17
4.1.3 Presentation layer . . . 18
4.2 Dependency Injection . . . 21
4.3 REST APIs . . . 22

4.4 Security 22
4.4.1 Authentication 23
4.4.2 Authorization 24
4.4.3 Security checks 25

5 Android Client Implementation 29
5.1 Android Platform 29
5.2 Building blocks 30
5.3 The prototype 32
5.4 The Implementation 33
5.4.1 User Interface 33
5.4.2 Resources 37
5.4.3 Responsiveness 39
5.4.4 Data persistence 40
5.4.5 Data synchronization 42
5.4.6 Google Cloud Messaging 43
5.4.7 Notifications 43
5.4.8 Account authenticator 44

6 Testing 45
6.1 Server Application Testing 45
6.1.1 Unit testing 45
6.1.2 Browser compatibility 45
6.2 Android Application Testing 46
6.2.1 Component testing 46
6.2.2 Compatibility testing 46
6.3 User testing 46
6.3.1 Web Client 46
6.3.2 Android Client 47

7 Conclusion 49
7.1 Future work 49

Bibliography 51

A Used Libraries 55

B List of used abbreviations 57

C UML diagrams 59

D Deployment guide 61

E CD Content 65

List of Figures

3.1 Application domain model diagram 10
3.2 Use Case Diagram describing user management and authentication 12
3.3 Use Case Diagram describing groups' position synchronization 12
3.4 Deployment Diagram showing the designed system architecture 13

4.1 Component Model of the Server Application 15
4.2 Web user interface created using JSF 20
4.3 OAuth 2.0 protocol flow [9] 24

5.1 Android software stack [13] 30
5.2 Activity lifecycle [8] 31
5.3 Sketched UI prototype in 2 iterations 33
5.4 Various states of the Main Activity 35
5.5 Various states of the Route Activity 36
5.6 Group icons used to distinguish their state 37
5.7 Floating Action Button pattern 37

C.1 Diagram of use case actors 59
C.2 Use Case Diagram describing event management 60

Chapter 1

Introduction

Open Day at the Czech Technical University in Prague is an event where prospective students can explore the university campuses and get information about study programmes, accommodation, and school life from the university staff or current students themselves. However, from the staff point of view, organizing such event is an exacting task, where many problems can arise. These issues then affect the planned schedule for the whole event.

1.1 Goals and motivation

The goal of this thesis is to design and implement a well-tested client-server application with web and Android clients, which would help to overcome problems concerning group synchronization during guided campus tours and other issues described in the following chapter. This application should provide everyone involved in the organization of the open day event with the much-needed information about each group's current location on the route circuit, which is essential for prevention of group collisions at stations of the routes. Also, the system should help organizers to keep all the event information neatly in one place. However it is important for the application to be easy to use, so that people will benefit from its use, rather than be burdened with additional duties.

1.2 Thesis structure

In the following chapters, current problems happening during the open day events will be discussed (chapter 2) followed by their analysis and the design of a solution to these problems in chapter 3. In chapter 4, the implementation details of the server application will be described. The 5th chapter deals with the overview of the Android platform and the implementation of a client application for this operating system. Testing of both applications will be discussed in chapter 6 and at last chapter 7 concludes the thesis.

Chapter 2

Problem description and requirement analysis

In this chapter, I will describe how is the open day event currently organized, explain the main problems with this organization, and specify the requirements for the system to be designed.

2.1 Open Day Organization

The Open Day event currently takes place twice a year at each faculty of the Czech Technical University in Prague (CTU). However, the Faculty of Electrical Engineering has 2 campuses (one in Dejvice and the other in Karlovo Náměstí) and the open day takes place on the same date. To satisfy students who want to visit both campuses, all activities are schedules to take place twice during the day with a sufficient time gap left between them for students' relocation from one campus to the other. Therefore the event has a strict schedule, which has to be followed closely otherwise students could miss the program at one of campuses.

In this thesis I will focus on the organization of the event at Karlovo Náměstí, however the system is going to be designed as generally as possible so that other faculties could benefit from it as well.

2.1.1 Main Blocks

At Karlovo Náměstí the event consists of two main blocks - Presentation and guided tour - which usually repeat twice during one day, but it is possible that this number may change in the future.

Collective presentation

The first block is held in a lecture hall, which can accommodate hundreds of students. Here the representing faculty officials talk about their study programmes and discuss general information regarding studying at the CTU.

Guided tour

The second block is to help future students explore the university campuses and departments. Visitors are divided into groups of around 20 people and each group is appointed with a tour guide from the school staff. Together they then walk one of the preplanned routes.

Each route is planned as a circuit and contains several stops in various departments along the way. The route being a circuit makes it possible for as many groups as there are stops to walk around the route simultaneously. Before the start of the tour, each guide is assigned a starting point from which he/she will lead his/her group around the route in a given direction until they have visited all stops. At each stop a university representative has a certain period of time to showcase various technology in the room or talk about related subjects. The adequate time spent at each stop is said to be 10 minutes. However, each group of students has different interests, therefore each group asks diverse amount of questions. It follows that the times spent at a stop vary considerably for each group.

The differences in time are the main cause for group collisions at route stops, which had in previous years been the most frequent problem in the tour organization. Group collisions happen when either a group leaves too early and has to wait at the next stop for the following team to leave or when a group exceeds the given time and someone else has to wait for them to leave.

There are also other less common situations which can arise before or even during the event, therefore they have to be considered in the design of proposed system. First, the number of students in each group can change even while the tour had already started. This is caused by the fact that some students want to leave early or that some visitors come late and want to join a group. Secondly, a guide must sometimes skip a station, for instance to save time when a group is way behind schedule or when the station must be closed.

2.1.2 Organization stages

The organization activities can be divided into three parts: planning before the event, organizing during the event, and evaluation after the event has finished.

Before the event

Before the Open Day can take place, the organizers have to plan and schedule the whole event. After they have settled on the event details such as date, place or routes' stations and their positions, they have to contact the tour guides to give them some basic information about their route and assign them their starting positions. Currently it is done by sending an e-mail to each person or by handing them printed instructions personally, however when the information changes the process must be repeated.

During the event

During the event, organizers try to follow the schedule as closely as possible. However this in not easy to control once the teams have left to explore the school. Currently the teams don't know other teams' positions (unless they collided on their path) and when organizers want to send latecomers to a group on the route, they have to call them using mobile phones or a walkie-talkie.

After the event

After the event, organizers have the chance to give feedback about the event organization and suggest improvement for the following year. Nevertheless, they do not have any data as to what was the average time spent at each station to decide whether the estimates they used while creating the event schedule were correct or not.

2.2 Requirements analysis

This section contains functional and non-functional requirements for the designed system. These requirements were either directly proposed by the client, represented by Ing. Martin Samek, or emerged from a client interview, which helped to discover client's needs.

REQ_001 - Event Management

REQ_002 - Create Event The system must allow users to create a new event.
REQ_003 - Edit Event The system must allow users to edit created events.
REQ_041 - View Event The system must allow users to view their events.
REQ_004 - Delete Event The system must allow users to delete their event.

REQ_005 - Route Management

REQ_006 - Add Route The system must allow users to add/create a new route to selected event.
REQ_007 - Edit Route The system must allow users to edit existing route before the start of the event.
REQ_042 - View Route The system must allow users to view a route of selected event.
REQ_008 - Delete Route The system must allow users to delete an existing route.

REQ_009 - Station Management

REQ_010 - Create Station The system must allow users to create a new station and add it to a route.
REQ_011 - Edit Station The system must allow users to edit existing station before the start of the event.
REQ_043 - View Station The system must allow users to view existing station.
REQ_012 - Delete Station The system must allow users to delete existing station before the start of the event.

REQ_013 - Guide Management

REQ_014 - Add a guide The system must allow to add/assign a guide to a route.
REQ_015 - Remove a guide The system must allow user to remove an assigned guide from a route.
REQ_016 - Guide starting position The system must allow user to assign and change order of guides' starting positions.

REQ_017 - User Management

REQ_018 - User registration The system must allow users to register.
REQ_019 - Edit account The system must allow users to edit their account information.
REQ_020 - Account deactivation The system must allow users to deactivate their account.
REQ_021 - Guide registration The system must allow users to create fast registrations by inputting e-mail address to which the system must send login credentials.
REQ_039 - Guide Import The system will allow user to upload a file in CSV (Comma Separated Values) format with the emails of guides for an event.
REQ_022 - User login The system must allow users to login to application with their credentials.
REQ_040 - Reset Password The system must allow users to reset their user password in case they have forgotten it.

REQ_023 - Group Information Management

REQ_024 - Add number of people The system must allow user to add a number of people to user's group.
REQ_025 - Update number of people The system must allow user to update the number of students in user's group.
REQ_026 - Location updating The system must allow user to inform the system about his current location.
REQ_027 - Station check-in The system must allow user to check-in at a station from a list of stations in his/her route. The station the user is checking in to must be the station following the one he left in the ordered list of station or it must be his/her starting position.
REQ_028 - Station check-out The system must allow user to check-out from a station. The station the user is checking out from must be the station he/she is currently checked in.
REQ_029 - Skipping station The system must allow user to skip a station. The station the user is skipping must be the station following the one he left in the ordered list of station or it must be his/her starting position.

REQ_030 - Notifications The system must notify user in specified cases.

> **REQ_031 - Approaching time limit notification** The system must notify the user, who is checked in at a station, when the station's time limit is nearing.
> **REQ_032 - Push notifications** The system must be able to send push notifications to users notifying them about specified events.
> **REQ_033 - Notification on guide coming** The system must notify user A that user B has checked out from a station preceding the station the user A is currently checked in.
> **REQ_034 - Notification on guide leaving** The system must notify user A that user B has checked out from a station following the station the user A is currently checked in or is on the way to station user B just left.

REQ_033 - Data Synchronization The system must synchronize data periodically.

REQ_034 - Current guides' positions The system must allow user to view current guides' positions.

REQ_035 - View route with stations The system must allow user to view a list of stations of the route he/she is guiding.

REQ_036 - Event Statistics The system must allow user to view statistics of his/her event, after the event has ended.

> **REQ_037 - Average time at a station** The system must allow user to view the average time spend at a station of chosen route.
> **REQ_038 - Average number of students in a group** The system must allow user to view the average number of students in groups of chosen route.

Chapter 3

Analysis and system design

This chapter describes an analysis of the problems from previous chapters. First, the application domain model is explained, followed by the description of the main use-cases of the designed application. At last, the system architecture will be discussed.

3.1 Domain Model

Domain Model is an object model describing the problem entities, their attributes and relationships to other objects in the domain [6]. Based on the requirements analysis, such domain model (figure 3.1) describing the main entities of the future system was designed.

3.1.1 User

User is the base entity in the system on which all other entities are directly or indirectly dependent. A user can organize up to N events (see 3.1.2), be a station manager at up to N routes (see 3.1.3), and guide up to N groups (see 3.1.5). User can also have several user roles, which describe user permissions in the system. Each user is identified by a user name and an email address, which are unique among all instances of User class.

3.1.2 Event

Event entity represents a real-life event, which takes place on a certain day. It also serves as a container for its child Route (see 3.1.3) objects. Besides `name` and `date` attributes, an event contains an `information` attribute, which can store basic event instructions for guides (see 3.2.1) and station managers (see 3.2.1).

3.1.3 Route

Route entity represents a single planned route, which is used for guided tours, at a given time. Route can have up to N groups (see 3.1.5) and up to N route stations (see 3.1.4).

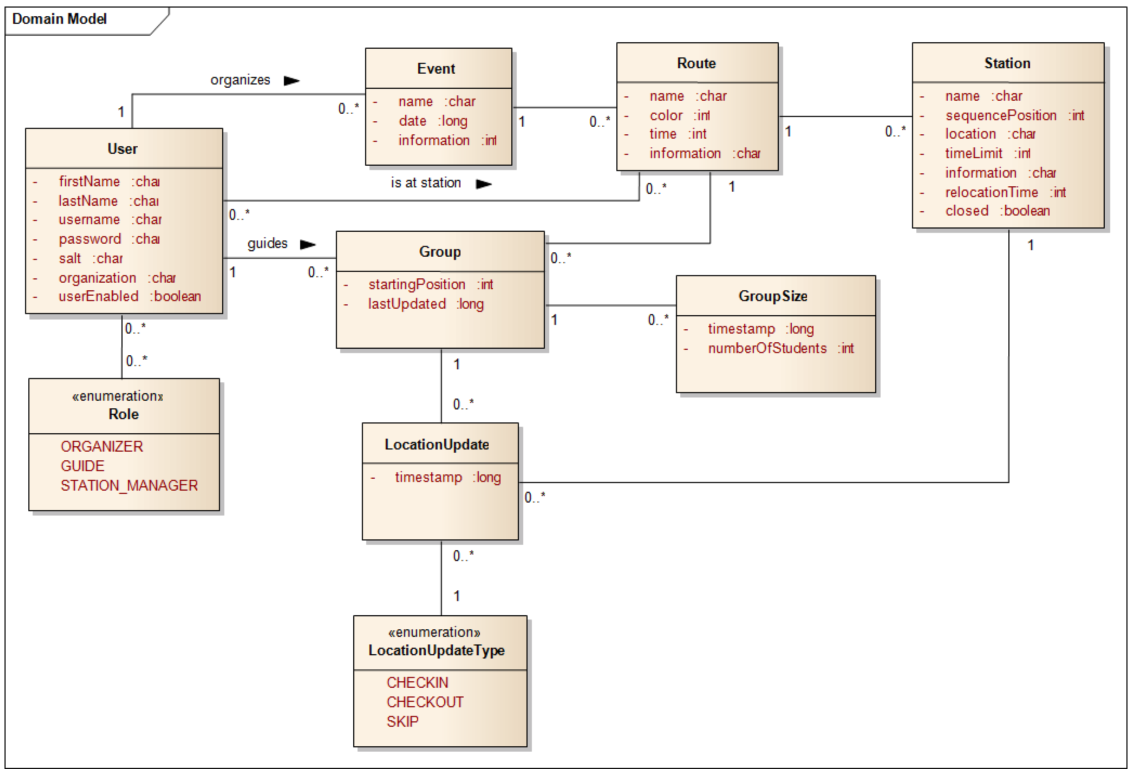


Figure 3.1: Application domain model diagram

3.1.4 Station

Station entity represents a stop on a route in a certain school department where its station manager (see 3.2.1) gives a short presentation about his/her subject or field of study. As in real life, each station has a `name`, `location` in the building (usually defined by a room number) and `information` about the presentation topic. Attribute `sequencePosition` describes the station's position in the ordered sequence of stations on the route circuit.

3.1.5 Group

A group of students is guided by one guide (see 3.2.1) along the stations of one route. The group starts its tour at the station with `sequencePosition` equal to group's `startingPosition` and move sequentially station by station until they have visited all route's stops. As the group moves along the route, it sends location updates (see 3.1.6) about its current position. The size of a group can change over time, therefore a group contains a list of up to N `GroupSize` instances, each containing information about group's size at a certain point of time.

3.1.6 LocationUpdate

LocationUpdate represents a group's location at a certain point of time. The location is specified by a particular station and an update type, which can either mean checking-in at a station, checking-out from a station, or skipping a station.

3.2 Use Case Model

Use Case (UC) Model is a form of requirement specification, which complements the "tradional" way of system requirement description as seen in section 2.2. UC Model helps to discover the system boundaries, actors, use cases (activities actors can do within the system) and the relationships between those use cases and actors [1].

3.2.1 Actors

Figure C.1 shows the actors that can operate with the system and their *is-a* relationships. The actors are:

User Someone who uses the system unauthenticated.

Authenticated User Someone who uses the system and is authenticated by his username and password.

Organizer An authenticated user who organizes and manages events.

Guide An authenticated user who guides groups of students at events.

Station Manager An authenticated user who gives short presentations at route stations of events.

Time An actor which can trigger time-bound actions.

3.2.2 User Management and Authentication

A `User` can create a user account through registration. After registration, `user` can sign in to the system or reset his user password in case he has forgotten it. An `Authenticated User` can edit his account, delete his account or sign out. An `Organizer` can register a guide or a station manager, whose email can be imported from a CSV file. This action will result in sending an email with user credentials to the specified email address, as shown in figure 3.2.

3.2.3 Event Management

Figure C.2 defines use cases concerning event management. An `Organizer` can create, view, edit and delete events, routes, and stations. Creating events can include adding routes, which in turn can contain creating stations or adding a group with a guide. While editing an event, `Organizer` can edit or delete the routes as well. Editing a route, can include the actions of adding or deleting stations, and adding or removing groups. Deleting an event will result in deleting all its routes, which will remove all its stations. `Guide`s and `Station Manager`s can view events, routes, and stations, however they don't have the rights to create, edit or delete them.

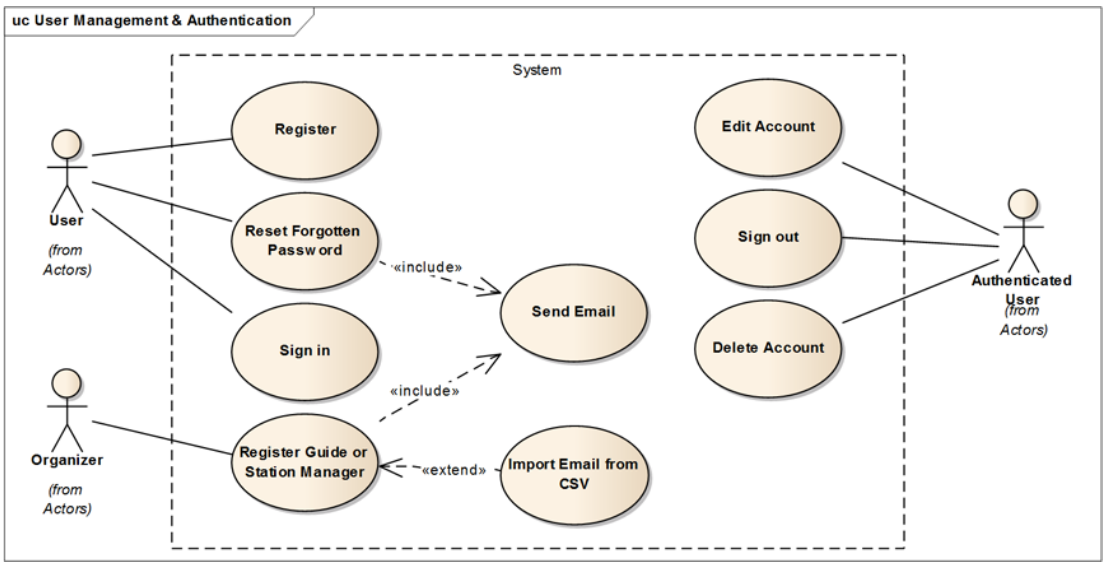


Figure 3.2: Use Case Diagram describing user management and authentication

3.2.4 Position Synchronization

As shown in figure 3.3, a `Guide` sends location updates, which results in sending a notification to other mobile clients. `Guide`s can also update the number of people in their group. `Guide`s, `Organizer`s, and `Station Manager`s can all see current locations of all groups.

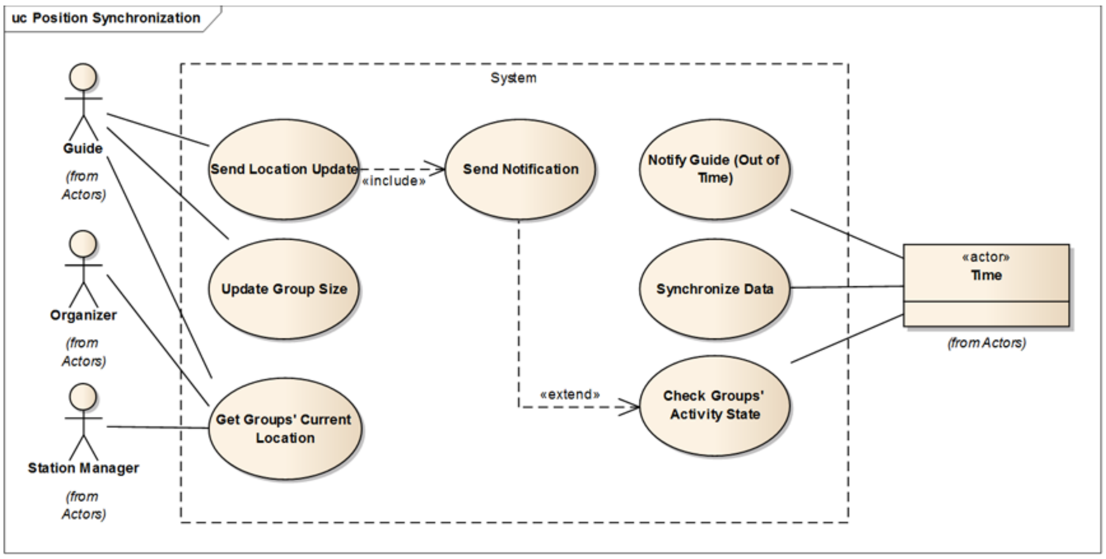


Figure 3.3: Use Case Diagram describing groups' position synchronization

An actor `Time` triggers notifications to guides based on the time they spend at a station, periodically synchronizes groups' positions data and checks whether all mobile clients are still active.

3.3 System Architecture Design

The application design, shown in figure 3.4, can be described as a 3-tier architecture. A tier is a logical or physical component. Each component can consume services from its adjacent tiers and provide services to its neighboring components [3].

3.3.1 Client Tier

In 3-tier architecture the User Interface (UI) runs on the client machines. In comparison, in 2-tier architecture, usually referred to as client-server, the client takes care of both the UI and Business logic [15].

The designed application contains 2 types of clients: Client PC and a mobile device with Android OS. These clients communicate with the application server (see 3.3.2).

Client PC The UI which is rendered in a web browser on the Client PC is provided by the Application Server. This type of client is called "thin" [10] and is to be used for event administration and real-time overview by its organizers.

Android Device The UI on the Android Device is part of a native application, which also contains part of the system's business logic, but is dependent on data transfer from/to the application server. This type of client is called "thick" [10] and is to be used by guides who lead groups of prospective students around the school to update their groups' information.

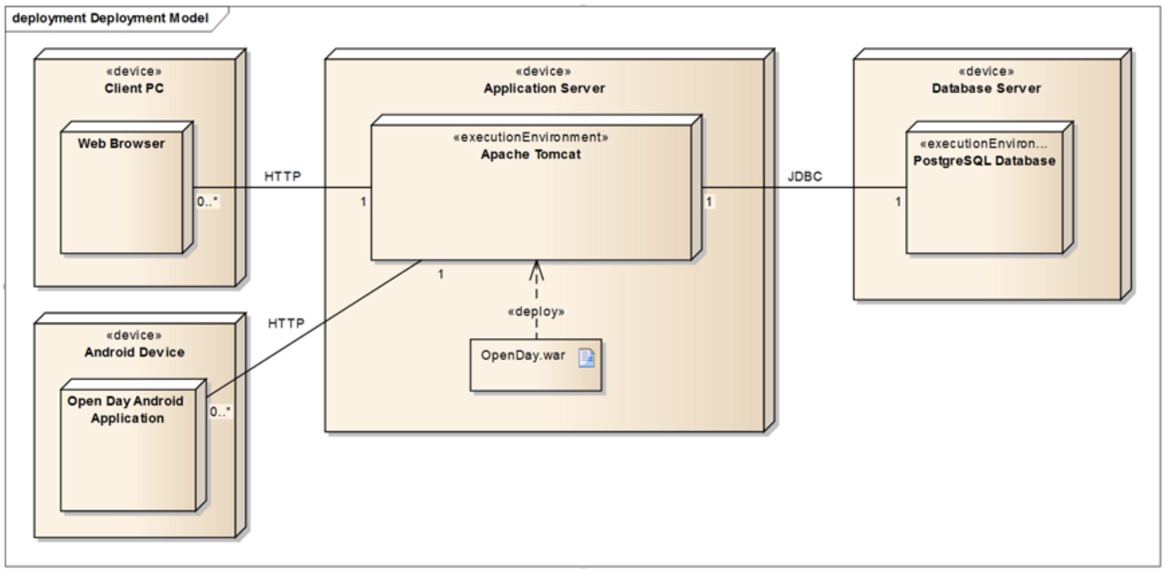


Figure 3.4: Deployment Diagram showing the designed system architecture

3.3.2 Application Server Tier

The Application Server Tier contains 3 layers: Presentation layer, Business layer and Data Access layer. Presentation layer provides clients with data needed to produce the UI. Business

Layer contains the application logic in form of business methods that are used to control the application flow. Data Access Layer transfers data between the Application server and Database server. For a detailed description of the layered architecture see section 4.1.

3.3.3 Database Tier

The Database Server runs a PostgreSQL Database instance, which is used for data storage. The Application Server communicates with the database using Java Database Connectivity (JDBC) application programming interface (API), which is a standard for database-independent connectivity [23].

Chapter 4

Server Application Implementation

This chapter describes non-trivial parts of the implementation of the server application designed in the previous chapters. First, the application's layered architecture will be described in more detail than it was in section 3.3.2. Secondly, the used Dependency Injection pattern will be discussed, followed by the description of APIs provided for mobile clients. The last part of this chapter covers the security of the server application.

4.1 Layered Architecture

The Application Server Tier structure decomposes into 3 substitutable layers, which organize classes by concerns such as presentation, business logic and data persistence. As shown in figure 4.1, neighboring layers communicate using clearly defined interfaces. In layered architectures it is common practice that each layer is dependent solely on the layer directly below it and it's also the only layer each layer is aware of [2].

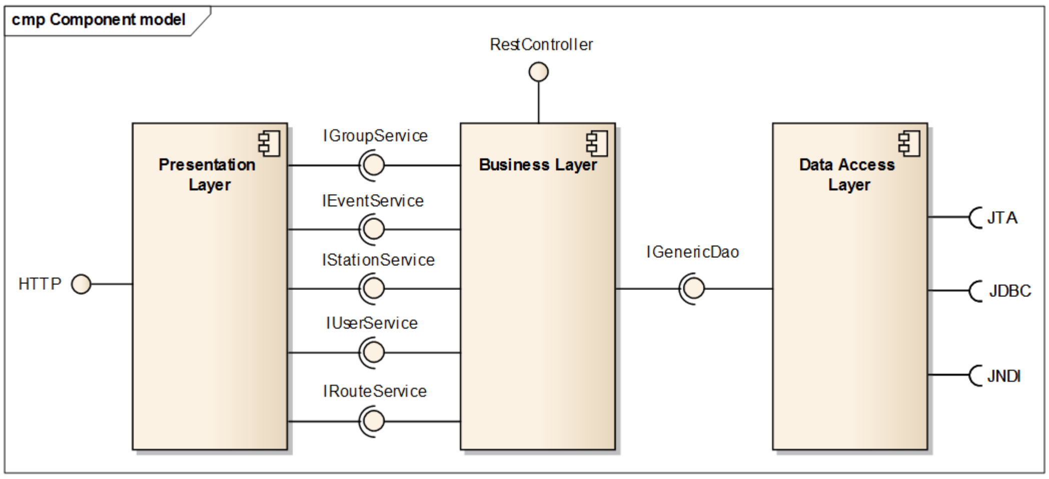


Figure 4.1: Component Model of the Server Application

4.1.1 Data Access Layer

Data Access Layer (DAL), sometimes also called Persistence layer, consists of a group of classes and other components which are responsible for storing and retrieving data to or from a database [2].

Persistence

Data Persistence is a process of saving state of a Java object in way that allows to recreate the object's state anytime in the future, even after termination of the program. Not all objects in an application have to be persistent, object whose state is lost after the end of the process which instantiated it is called *transient*. In object-oriented applications, relational databases offering a structured representation of persisted data are often used. Database management systems also ensure data integrity and concurrent access using transactions [2].

Object/Relational Mapping

Object/relational mapping (ORM) is a technique for persisting Java objects to the tables of relational database. In this application Hibernate's ORM[1] implementation was used.

DAL utilizes classes that match the domain model defined in section 3.1. These objects are necessary to be able to use the ORM as the Hibernate framework uses metadata defined in form of code annotations (such as `@Id` or `@ManyToOne`) in these classes to convert plain Java objects to or from their respective database representation.

```
@Entity
public class Event implements Serializable {

  @Id
  @GeneratedValue(generator="system-sequence")
  @GenericGenerator(name="system-sequence", strategy = "sequence")
  protected Long id;

  @Column(nullable = false)
  private String name;

  @ManyToOne
  private User organizer;

  @ElementCollection
  private Set<String> emailList;

  ...

}
```

Listing 4.1: Code excerpt showing Hibernate annotations

[1]http://hibernate.org/orm/

Data Access Object

To create a nice interface and hide complex data storage and retrieval code from the business layer, a widely-used design pattern called Data Access Object (DAO) was used. At the same time, the DAO pattern complies with the adapter pattern, acting as a bridge between two objects.

The business layer relies on the interface exposed by the DAO, which completely encapsulates the data source implementation. This approach makes it possible to change the underlaying data source or its implementation without affecting the exposed interfaces [6].

4.1.2 Business Layer

Business Layer, also called the Business Logic Layer, consists of components that provide the core logic of an application.

Data entities

The first type of data entity used in the business layer are business objects (BOs), which strictly represent the domain model and are used in communication with the underlaying data access layer. Besides the metadata to be used by the ORM framework as discussed in the previous section, these objects also contain part of the logic, which ensures the relationship constraints defined by the application's domain model.

Secondly, Data transfer objects (DTOs) are used for data transport from business to presentation layer in order to reduce the number of method calls to remote interface. To achieve this goal, DTOs do not strictly match the domain model like BOs, but can transfer more data with each call. DTOs do not contain any logic, on the other hand, they are easily serializable, unlike the BOs that are interconnected in a complex web of relationships that's difficult, if not impossible, to serialize [6].

Service layer

Service layer, which utilizes the 2 kinds of data entities, is a set of classes that implement the application's business logic. This layer defines an application boundary and acts as a façade, by providing a client API that's easier to use as it's oriented around use cases rather than CRUD operations [6]. Besides providing a clear API, the service layer methods are a place where transactional support and security checks are performed.

A transaction is a sequence of operations performed in a single unit of work that must meet four criteria. First, a transaction must be *atomic*; either all changes are performed, or no change is performed at all. Secondly, after the transaction has completed, all data must be in a *consistent* state. Third, a transaction must be *isolated* from all changes made by any another transaction, which isn't yet complete. Last *durability* is expected, so that when completed, transaction's effects are permanently persisted even in the event of a system failure [32]. As shown in code excerpt 4.2, `@Transactional` annotation enables Spring framework transactional support, which triggers a rollback in case of a runtime exception. This ensures the ACID criteria even if the run of a method was ended prematurely.

Code 4.2 also shows security check in form of `@PostAuthorize` annotation, which is performed after the method has completed. The security of the implemented application will be described in greater detail in section 4.4.

```
@Transactional(readOnly = true)
@PostAuthorize("hasPermission(obj.id, obj.ACLObjectIdentityClass, 'READ')")
public EventDto getEvent(Long id) throws DataAccessException;
```

Listing 4.2: Code excerpt showing Service method with transactional and security annotations

4.1.3 Presentation layer

The presentation layer is the topmost layer of the application, which displays graphical user interface that a user can easily understand. In the implemented application, the presentation layer was created using a Java Server Faces framework.

Java Server Faces (JSF)

JSF is a server-side component-based web framework for Java. The first JSF 1.0 specification and reference implementation was released in 2004 by the JSF Expert Group. Although the specification was very general and not enough thought was put into the API design, JSF managed to attract framework developers thanks to its high extensibility. In 2009, JSF 2.0 was released. Gaining from the experience of the various frameworks, version 2.0 improved nearly every aspect of the original specification, making it much easier for developers to use [7].

Managed Beans

To achieve a separation of presentation and business logic of the web application, JSF uses beans. A Java *bean* is a POJO (Plain Old Java Object) that exposes its properties using a standard get/set naming convention and a set of other methods that perform tasks for components. A *managed bean* is a Java bean that is managed by the JSF framework, making the object accessible from JSF pages. Every managed bean, which can be easily registered with JSF using a `@ManagedBean` annotation, must have a *name* and *scope*. When the bean's name is referenced in a JSF page, the framework either supplies an existing object or constructs a new instance with such name and scope if it does not yet exist.

```
@ManagedBean(name="newEventBean")
@RequestScoped
public class CreateEventBean implements Serializable { ... }
```

Listing 4.3: Code excerpt showing managed bean annotations

The scope defines the range of bean's lifetime. As of JSF 2.0, four bean scopes are defined.

Session scope A session scoped bean lives from the time that a session is established until its termination. Such bean is accessible during the entire session, however an overly large session state can become a performance bottleneck.

Request scope A request scoped bean is created with the submission of a HTTP request and disposed after the response was sent back to the client.

View scope A view scoped bean lives as long as the user is interacting with the same JSF view (page). The bean is destroyed when the user navigates to a different page.

Application scope An application scoped bean persists as long as the web application runs. This scope is shared among all sessions, which means that a single instance is provided to all users of the application.

JSF pages

In the original JSF 1.x specification JavaServer Pages (JSP) were used to create JSF Pages. However, JSP was found to be cumbersome to develop with and was later deprecated. Due to the framework's high extensibility, programmers could replace the default view handler with other view definition languages such as Facelets, which became a part of the JSF 2.0, and it's the technology used in the implemented application.

Facelets allows developers to create JSF pages by adding custom JSF tags into a XHTML document, which is simply a well-formed HTML file. Unlike web browsers, JSF won't be able to parse the file if it contains syntax errors. To use additional tag libraries (such as PrimeFaces) Facelets only require the library's xml namespace to be declared.

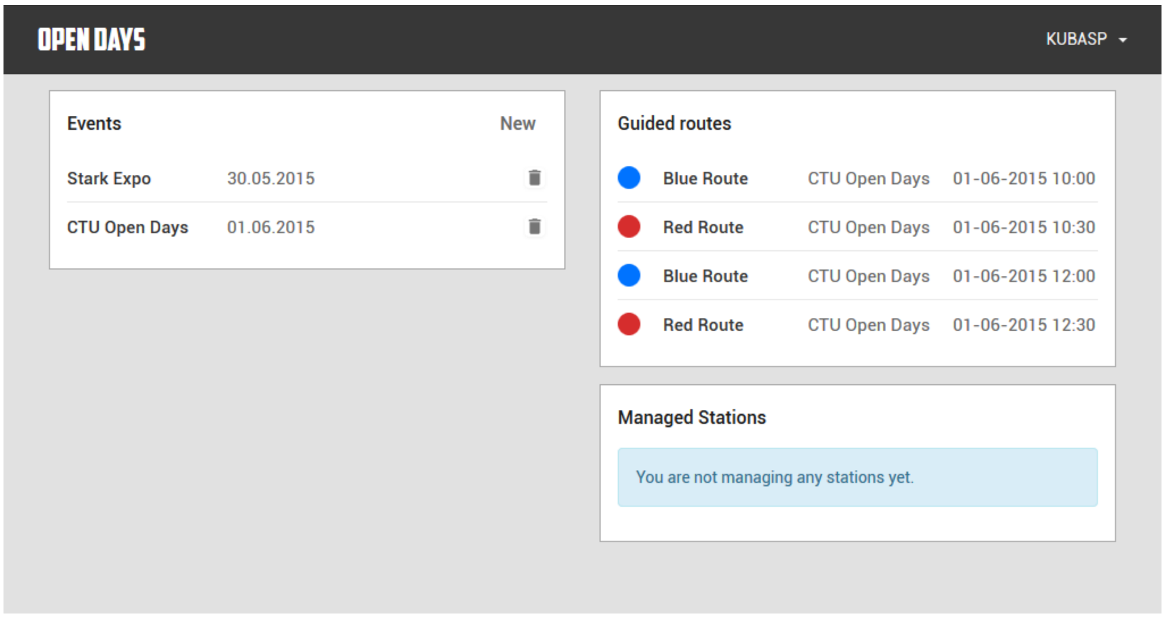
Based on the JSF Expression Language (EL) support, Facelets use EL expressions to reference properties and methods of managed beans. This allows to bind values or events of a JSF view component to bean's properties or methods [19].

Another great advantage of using Facelets is a reduction of code duplication by the usage of templating and composite components. This feature for instance allows programmers to create a base template, which can be then reused for other pages of the application. Besides the obvious code reuse, templating also helps to maintain a unified look and feel throughout the entire application.

Message bundles

Internationalization is a process of software design, which makes the application adaptable to different languages and locales without further engineering changes. Such program doesn't contain hardcoded values for user interface components, instead they are stored outside the source code and supplied dynamically [22].

In JSF, internationalization is achieved through message (resource) bundles. For each supported locale, a bundle properties file with an appropriate locale suffix to its name must be created. A default bundle, which doesn't need a locale suffix, is used when a required bundle isn't available. Each bundle consists of key-value pairs that are accessible from JSF pages using EL expressions.



Events	
Stark Expo	30.05.2015
CTU Open Days	01.06.2015

Guided routes		
Blue Route	CTU Open Days	01-06-2015 10:00
Red Route	CTU Open Days	01-06-2015 10:30
Blue Route	CTU Open Days	01-06-2015 12:00
Red Route	CTU Open Days	01-06-2015 12:30

You are not managing any stations yet.

Figure 4.2: Web user interface created using JSF

Validation

To ensure data correctness in the business logic, the JSF container can perform input validations before updating the model. JSF provides tags for basic validations, which can be added to the body of a component tag. In case of a validation error, the framework offers an easy way of displaying an error message to the user, as shown in listing 4.4.

```
<h:inputText id="username" value="#{registrationBB.username}">
    <f:validateLength minimum="4" />
</h:inputText>
<h:message for="username"/>
```

Listing 4.4: JSF validation that ensures input has at least 4 characters

More complex validations can be achieved through custom validator classes that implement the `Validator` interface. If the validation fails, a `ValidatorException` with an appropriate `FacesMessage` describing the problem should be thrown.

```
public class UsernameValidator implements Validator {

  @Override
  public void validate(FacesContext context, UIComponent component, ...
      Object value) throws ValidatorException {
    String username = (String) value;

    if(!userService.isUsernameFree(username)){
        FacesMessage msg = new FacesMessage("Username is already taken!");
        msg.setSeverity(FacesMessage.SEVERITY_ERROR);
```

```
            throw new ValidatorException(msg);
        }
    }

}
```

Listing 4.5: Custom JSF validator

4.2 Dependency Injection

As was already described in the previous sections, to achieve loose coupling, the application was divided into several substitutable modules, which are dependent on each other though a set of defined interfaces. In such applications, dependency injection (DI) is often used to wire these modules together.

Dependency Injection is a software pattern, where an object is provided with objects it depends on, rather than constructing them itself. There are 3 common types of DI: Constructor injection (CI), Setter injection (SI) and Property injection (PI) [5].

Constructor injection CI requires the class to have a constructor which accepts all of its dependencies. This way, the dependencies will be passed to the object at the moment of its creation.

Setter injection SI requires the class to have a setter method for each of its dependencies, thus setting the dependencies *after* object's creation.

Property injection PI requires the class to have its dependencies declared as `public` fields, which provides the DI framework direct access to those properties.

In general, DI helps to achieve higher class cohesion, as classes are not responsible for creating objects they depend on. Furthermore, DI is useful for component testing, thanks to the possibility of easy dependency mocking or stubbing.

Dependency Injection with Spring framework

In this thesis, Spring framework was used to bind the layers into a cohesive application using dependency injection. The Spring container supports constructor-based and setter-based DI, however in this application, only setter injection was used.

Spring provides DI inside its context only for objects it manages. To mark an object for management, XML configuration or annotations can be used. First, a base package must be set, as shown in listing 4.6, to tell the framework where to look for user defined components [27]. For component configuration, class annotations `@Component`, `@Service` and `@Controller` were used.

```
<context:component-scan base-package="cz.kubaspatny.opendayapp"/>
```

Listing 4.6: XML configuration of Spring's base package

To request a dependency injection of a field, the instance variable or its setter method must bear an `@Autowired` annotation. From personal feeling of better clarity, implemented application prefers variable annotations to method annotations as shown in code excerpt 4.7.

```
@Component("userService")
public class UserService implements IUserService {

    @Autowired
    private IEmailService emailService;

}
```

Listing 4.7: Autowired annotation on a field

4.3 REST APIs

The implemented application provides a HTTP accessible interface to be consumed by mobile clients for reading and uploading information. The interface was designed and implemented according to the REST constraints.

REST (Representational State Transfer) is a set of architectural principles used for designing distributed systems. The architectural style suggests to follows four basic design principles [26].

Stateless Web server should store no client context between requests, instead the client holds its session state. This principle induces reliability, and scalability of the system [4].

Transfer XML/JSON For data transfers, the resources should be represented in JSON or XML formats [26]. The implemented interface makes use solely of the JSON representation.

Resources The system should expose directory structure-like URIs for its resources [26]. For instance, the following URI is used to access stations of a route in the implemented application.

```
dod.felk.cvut.cz/v1/route/{routeId}/stations
```

Explicit use of HTTP methods The system should map its CRUD (create, read, update, delete) operations to corresponding HTTP methods (`POST`, `GET`, `PUT`, `DELETE`) [26].

4.4 Security

This section describes the security of the implemented application, which was mostly achieved by using another part of the extensive Spring framework, a subproject called Spring Security.

Spring Security handles two main areas of protection: Authentication and Authorization. *Authentication* is the process of establishing whether a principal is really who they claim to be. *Principal* usually refers to a user, but it can also mean a device or other system. *Authorization* is a process of allowing (or denying) access to a particular recourse or action within application [12].

4.4.1 Authentication

The application uses two ways of user authentication. The website users authenticate themselves using their username and password combination, whereas the mobile client uses an OAuth 2.0 access token instead.

Password authentication

The website application provides a login form, which prompts the user for his username and password. These credentials are then passed on to a custom implementation of Spring's `AuthenticationProvider`, which either returns an `Authentication` object that represent an access token for the authenticated principal, or throws a `BadCredentialsException` for incorrect username-password pair.

In order to protect users' passwords in case of a database breach, the application uses salted password hashing. *Hash function* is an algorithm that transforms input of arbitrary length into fixed-length result. A hashed password hides the plain-text form, however mere hashing is not sufficient protection. For two identical passwords, the function returns the same result, which facilitate the use of brute-force attacks or other more sophisticated methods such as Lookup or Rainbow tables. To randomize the results, a random string of characters (*salt*) is appended to user's password before hashing. This way, two identical passwords will render different results. During authentication, the provided plain-text password is appended with the salt that is saved in the database, and hashed using the same function. At last, the hashed results are compared.

Access token authentication

Spring Security's implementation of the OAuth 2.0 authorization framework, which enables client applications to gain access to a service with a user's permission, was used for access token authentication.

Roles The framework specification defines four roles. *Client* is an application accessing protected resources on behalf of a user. *Resource owner* is the end-user, who grants the client access to his data. *Resource server* hosts the protected resources, to which it either grants or denies access based on access tokens, which are issued by the *authorization server* [24]. The implemented server application serves as both the resource and authorization server, while the Android application acts as a client.

Flow The OAuth2 flow, as pictured in figure 4.3, describes the communication among the four roles needed for client to access a protected resource. First, the client requests an authorization from the user, who responds with an authorization grant. Client then uses the grant to obtain an access token from the authorization server. To access a protected resource from the resource server, client uses the previously acquired token.

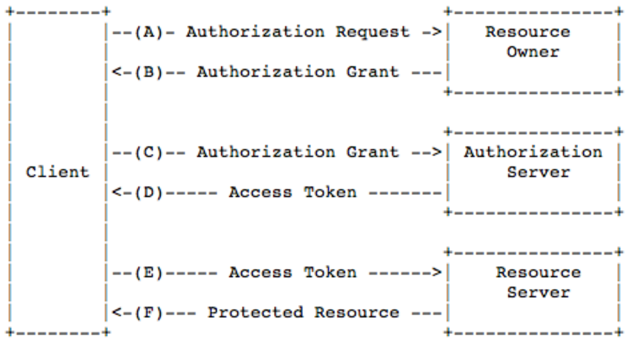

Client
--(A)- Authorization Request ->
Resource Owner
<-(B)-- Authorization Grant ---
--(C)-- Authorization Grant -->
Authorization Server
<-(D)----- Access Token -------
--(E)----- Access Token ------>
Resource Server
<-(F)--- Protected Resource ---

Figure 4.3: OAuth 2.0 protocol flow [9]

Grant types OAuth2 defines four grant types: Authorization Code, Implicit, Resource Owner Password and Client Credentials.

The Android client application uses the *resource owner password* grand type, which is used to directly exchange username and password for an access token. This grand type is not suitable for 3rd-party applications, as the user must provides his password directly to client. However, it's often used by trusted clients, such as those created by the service owner itself [25].

Access Token The access token is a credential representing an authorization given to client to access protected resources. Tokens usually have an expiration date, after which the client must either use a refresh token (if it was issued one) to obtain a new access token or re-prompt the user for authorization.

4.4.2 Authorization

The application users are granted authorization based on two criteria: user roles and access control lists (ACLs).

Role-based access control

For basic system access restriction, the resource server implements a role-based security model, which restraints access depending on a particular group membership. Necessary user roles for the application were discovered during system analysis in chapter 3 and their relationships can be seen in figure C.1.

There are 4 custom roles defined: `ROLE_USER`, `ROLE_ORGANIZER`, `ROLE_GUIDE`, and `ROLE_STATIONMANAGER`. Every authenticated user is given the `ROLE_USER` authority. Users, who registered on the website using the registration form are granted the `ROLE_ORGANIZER`, which allows them to create new events and register new users for given email addresses. Automatically created users are given both `ROLE_GUIDE` and `ROLE_STATIONMANAGER` roles. As the names suggest, these roles give permission to guide groups and to manage stations of a route.

The Spring Security framework also internally grants other than the user-defined roles, which can however still be used to define access restrictions. For instance, yet unauthenticated users bear the `IS_AUTHENTICATED_ANONYMOUSLY` role, which is used to deny access to areas for logged-in users only.

Access Control Lists

Role-based access control wasn't sufficient for the application's security needs, therefore ACL-based restrictions were implemented to achieve a fine-grained access control. Using ACLs enables permission granting in respect to an action for a particular domain object instance, rather than an action only. For example, in the application an event organizer should be able to view *and edit* the event, while the event's guides and station managers are only permitted to view its details. Other users are completely banned from both editing and viewing this event's information.

Spring Security implements an ACL module for domain object instance security, which was designed with respect to high ACL retrieval performance and independence from ORM frameworks accomplished by using JDBC for direct database connection [28]. To record user permissions, the implementation uses the following 4 database tables:

ACL_SID Secret Identity is used to uniquely identify any principal within the application.

ACL_CLASS Used for object instance class identification. Table contains a single row for each class we wish to use ACLs for.

ACL_OBJECT_IDENTITY Stores information about particular object instance. Besides the object ID, reference to object's class (ACL_CLASS) and a reference to object's owner (ACL_SID) are stored.

ACL_ENTRY Each row stores particular permission of a user (ACL_SID) to a given object instance (ACL_OBJECT_IDENTITY).

The framework also defines correspondingly named interfaces, which were used in the service layer (see 4.1.2) to manage user permissions.

4.4.3 Security checks

The previously described methods of authorization were used within the application at 3 different levels: URL, Method, and View.

URL access

The application enforces URL access restrictions using role-based security. Except for the login and registration pages, all website and API endpoint URLs are set to require fully authenticated users. The XML settings, as shown in figure 4.8, uses the `ROLE_USER` to signify an authenticated user, whereas `IS_AUTHENTICATED_ANONYMOUSLY` for unverified one. Restrictions based on other user roles are only used at view level.

```
<sec:http authentication-manager-ref="userAuthenticationManager">
   <sec:intercept-url pattern="/login.xhtml" ...
       access="IS_AUTHENTICATED_ANONYMOUSLY" />
   <sec:intercept-url pattern="/register.xhtml" ...
       access="IS_AUTHENTICATED_ANONYMOUSLY" />
   <sec:intercept-url pattern="/**" access="ROLE_USER" />
</sec:http>
```

Listing 4.8: URL restriction settings

View security

To hide actions from unauthorized users, JSF `rendered` tag was used. The tag accepts EL expressions that evaluate to boolean values (`true`, `false`).

To enforce role-based security, JSF context provides a method `isUserInRole()` to easily check user's authority. However, to be able to use ACL-based security in the view context, a custom managed bean must had been implemented.

```
<p:commandLink
   rendered="#{acl.hasPermission(route.id, route.ACLOIdClass, 'WRITE')}">
```

Listing 4.9: ACL-based view security using custom `acl` managed bean

Method security

In case of security failure at URL or view level, method invocation security ensures that only rightfully authorized users are able to proceed. Spring Security 3.0 introduced four new expression-based method annotations, which are used at the service layer of the implemented application. The annotations accept SpEL (Spring Expression Language) expressions, which support operations for both role-based and ACL-based security. SpEL also enables method arguments, their fields and methods to be referenced from the expressions, which was broadly used for the ACL security definitions [29].

Of the four, the most often used annotation in the application is `@PreAuthorize`, which is used to decide whether a method invocation is actually permitted or not.

```
@PreAuthorize("hasPermission(#st.id, #st.ACLObjectIdentityClass, 'WRITE')")
public void updateStation(StationDto st) throws DataAccessException;
```

Listing 4.10: @PreAuthorize method annotation using ACL security

On the other hand, the `@PostAuthorize` annotation validation is performed *after* the method had already been invoked, which could result in damage in case of unauthorized, possibly malicious, user. Nevertheless, this annotation was used on read-only methods to determine whether a user had been granted necessary permission to view the object returned by said method.

```
@Transactional(readOnly = true)
@PostAuthorize("hasPermission(returnObject.id, ...
    returnObject.ACLObjectIdentityClass, 'READ')")
public StationDto getStation(Long id) throws DataAccessException;
```

Listing 4.11: @PostAuthorize method annotation checking requested READ permission

`@PostFilter` annotation can be used on methods that return a collection of objects to remove those instances for which the authenticated user does not have the required rights.

```
@PostFilter("hasPermission(filterObject.id, ...
    filterObject.ACLObjectIdentityClass, 'READ')")
public List<RouteDto> getRoutes(Long eventId) throws DataAccessException;
```

Listing 4.12: @PostFilter annotation filters out objects from returned collection

The last annotation, `@PreFilter`, enables to filter objects from collections passed as method arguments, however it was not used in the implemented application [30].

Chapter 5

Android Client Implementation

This chapter describes the implementation of the mobile client, which is used by station managers and group guides during ongoing events. First, the Android platform and its parts used in this thesis will be discussed. Secondly, the design process from a prototype to the final product will we explained. Last, implementation of the client application itself will be described.

5.1 Android Platform

Android is an open platform, which includes operating system, graphic user interface, and applications. In 2008, the first commercially available version was introduced as a system solely for mobile phones, however the platform has since expanded to support much wider range of devices, such as tablets, televisions or wearables [13].

Market Share

At the beginning of 2009, only 1.6% of all smartphones sold worldwide were using the Android operating system. However, within the following years the Android global market share rose quickly. As of the end of 2011, Android claimed 52.5% of the global market. The highest portion of worldwide sales was reached in the third quarter of 2013. At that time, Android dominated the global market with a 81.9% share, while its biggest competitor, Apple iOS, managed to ship only 12.1% of all units [17]. Thus, by implementing an Android client the highest portion of users can be addressed.

Android Software Stack

The Android platform components are composed as a stack. *Linux kernel*, which forms the lowest stack layer, handles core services such as power, process, and memory management. Moreover, it acts as an abstraction layer between hardware and the rest of the components. On top of Linux kernel is a set of various C/C++ *libraries*, which are available to application developers through the application framework layer. *Android Run Time (ART)* consists of core Android-specific libraries and a register-based virtual machine Dalvik, which had been specifically optimized to achieve the capability of running several instances efficiently at the same time. *Application Framework* layer exposes classes to be used in Android applications. All Android applications run on the topmost *application layer*, which in turn runs within ART using the classes and services provided by application framework layer [13].

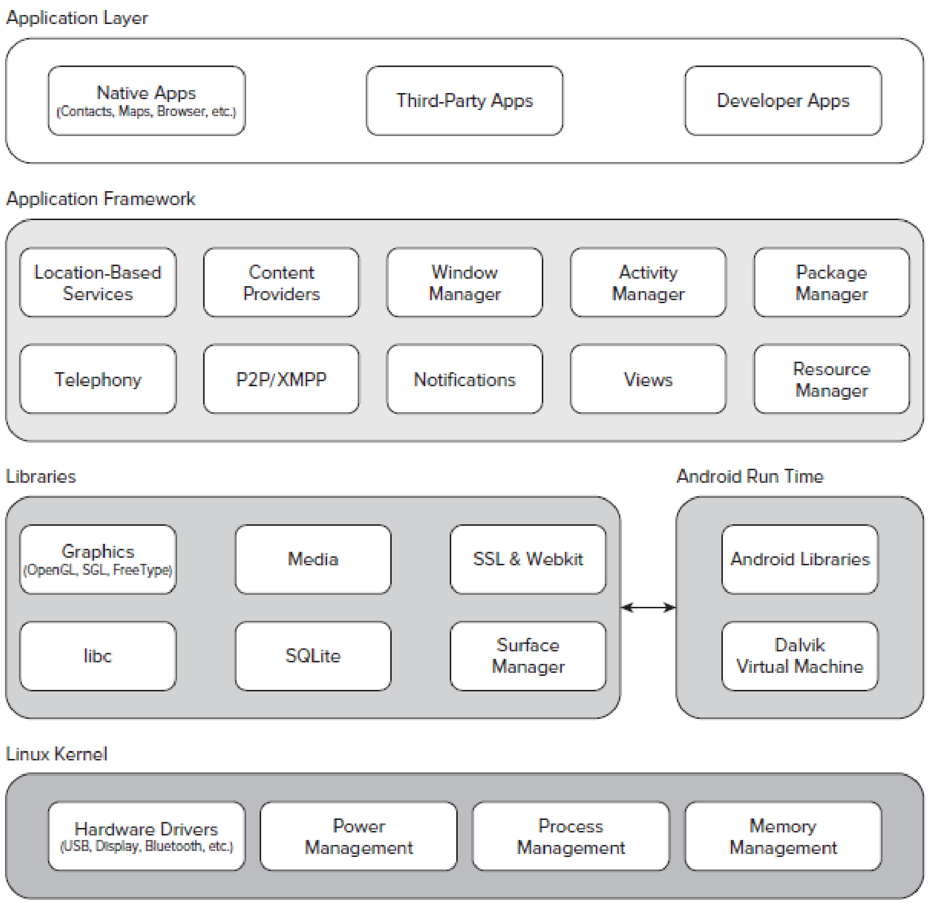


Figure 5.1: Android software stack [13]

5.2 Building blocks

All Android applications comprise of many loosely coupled components that are wired together through the application manifest file, which describes each component and their relationships. The following application components represent the building blocks used by developers to create Android apps [13].

Activities

Activities are used to create the presentation layer of an Android application. To display information and handle user interactions, activities make use of Views and Fragments. Each application consists of one or more subclasses derived from the `Activity` class.

During the run of an application, Activity instances transition between several states of their lifecycle, as shown in the state diagram in figure 5.2. The application doesn't control the lifecycle of its activities, instead it is handled exclusively by the Android run time. In order to react to state changes, the activity class provides a series of callback methods, which are called by the system prior activity's state transition.

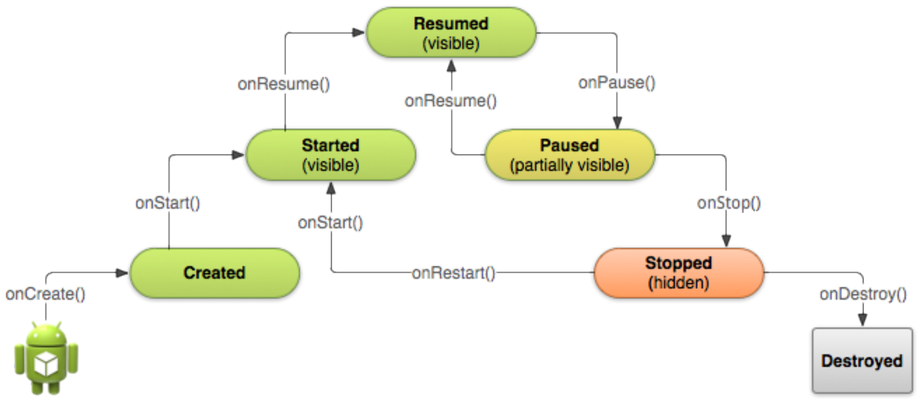


Figure 5.2: Activity lifecycle [8]

Fragments

A Fragment represents a portion of the user interface or functionality of the application. Like Activities, Fragments have a UI layout to display information and they receive user input events. However, fragments are not self-contained and must be embedded inside an Activity to be displayed [13].

Services

All components of a single application run on the same main thread, which performs interactions with UI components, therefore it's often called the UI thread. Service is a component that can be used to perform long-running operations without blocking the UI thread [13]. Unlike activities, services run in the background without providing user interface.

Content Providers

Content provider is a component used for application data management and persistence. Typically, content providers provide access to a central data repository, such as an underlaying SQLite database. Moreover, the component enables a way of sharing data to other applications [13].

Intents

An intent is an object sent within a system message-passing framework to request an action from an application component [13]. Three common actions are: starting an activity, starting a service, and broadcasting a message.

Broadcast receivers

Broadcast receiver is a component, which enables the application to listen for broadcast Intents. Upon receiving a broadcast message, the application is started in order to handle the received Intent [13].

Notifications

Notification is a technique allowing to alert the user while the application is not in an active foreground state, such as from a background service or broadcast receiver [13]. Notifications use lights, sounds, and vibrations to gain user's attention.

Android Manifest

Every Android project must contain an Android manifest file, which defines the structure of the application. The manifest contains descriptions of all Activities, Services, Content Providers and Broadcast Receivers used within the application. Moreover, the file describes application meta-data (application name and version, theme, icon, etc.), required permissions, and hardware or platform requirements [13].

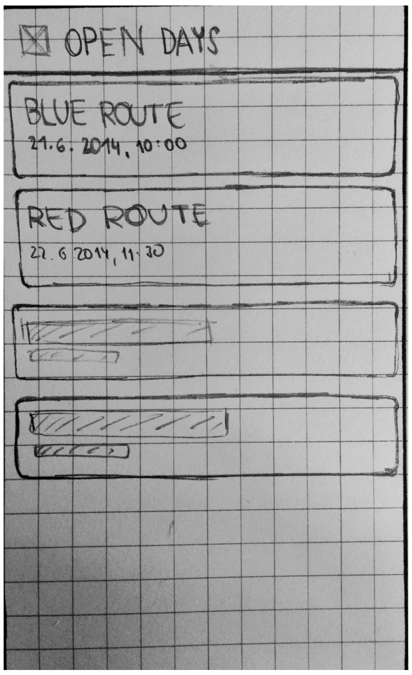
5.3 The prototype

Before implementing the Android client, a low-fidelity prototype was created to be able to see and interact with the application before it was actually built.

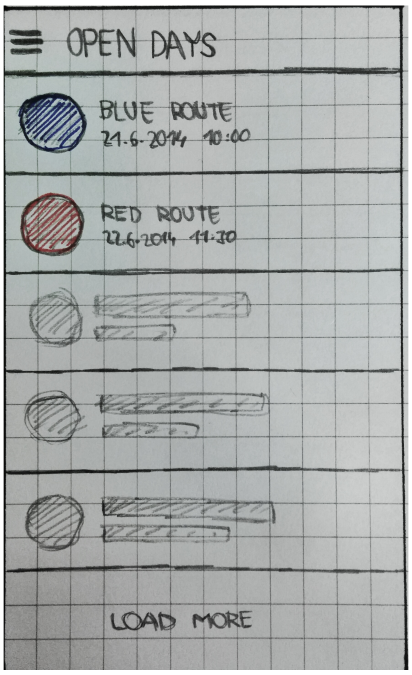
From personal experience, basic tools such as paper and pencil were used to create UI sketches, rather than using software prototyping tools like balsamiq[1]. To me, paper prototyping has proven to be the fastest method of creating user interfaces sketches, as creativity is not restricted by the tool's limits. However, to create a click-able prototype, which can be run on an Android device like a native application, a designated tool called POP[2] (Prototyping On Paper) was used. POP enables to take photos of the sketches, mark user interaction areas in the UI, and finally create links between individual screens of the designed application.

The prototyping process was iterative. First step was to sketch the UI and create the click-able prototype. Then the application was analyzed and discussed with a small test group. Using the gathered feedback, the process repeated from step one. Figure 5.3 shows two iterations of the process for main (launcher) activity of the client application. Sketches for login, main, and route activities in their various states were created and can be found on the attached CD together with screenshots from the implemented application for comparison.

[1]https://balsamiq.com/
[2]https://popapp.in/



(a) 1st iteration



(b) 2nd iteration

Figure 5.3: Sketched UI prototype in 2 iterations

5.4 The Implementation

Based on the prototype from the previous section, a real application was implemented. This section describes the main components of which the client application comprises, various implementation problems and their solutions, and certain best-practices used during development.

5.4.1 User Interface

In Android, all user interface components descend from the `View` class, therefore generally referred to as *Views.* A View is responsible for drawing itself on the screen and for user interaction event handling.

Interactive UI components, such as buttons, text fields and other, are ofter called *widgets.* `View` is also a base class of `ViewGroup`, which is designed to act as a container for multiple Views. View Groups are commonly referred to as *layouts*, however they can be used to create reusable composite components as well.

Base Activity

The user interface is implemented using 3 activities, all of which are descendants of a custom `BaseActivity` class, which carries out a few common tasks.

First, the activity checks whether a user is correctly logged in. In case no account is found, the user is redirected to `AuthenticatorActivity`, where he is prompted to authenticate. Activities that don't require authentication to be shown (e.g. the Authenticator activity) can simply override `requireLogin()` method to return false.

```
@Override
public void onAccountsUpdated(Account[] accounts) {
    for (Account account : accounts) {
        if (AuthConstants.ACCOUNT_TYPE.equals(account.type)) {
            return; // User logged in correctly
        }
    }

    // No account found -> start the authenticator activity
    Intent intent = new Intent(this, AuthenticatorActivity.class);
    startActivity(intent);
    finish();
}
```

Listing 5.1: Base activity authentication check

Next, the base activity checks the device to make sure it has the Google Play Services properly installed. If it doesn't, a dialog that allows users to download the installation package is shown. At last, Google Cloud Messaging (see 5.4.6) registration is verified. If the application is not yet registered to receive push notifications, asynchronous registration is performed and the newly obtained registration ID stored for later use.

Authenticator Activity

The login screen of the client application is implemented by the `AuthenticatorActitity` class, whose layout contains a logo, 2 text fields, and a button. In Android, layouts can be either created programmatically in activity's code, or they can be declared in a XML layout resource file that is loaded during the creation of an activity. As the Authenticator activity's layout file in listing 5.2 suggests, the latter option was preferred in the implemented application.

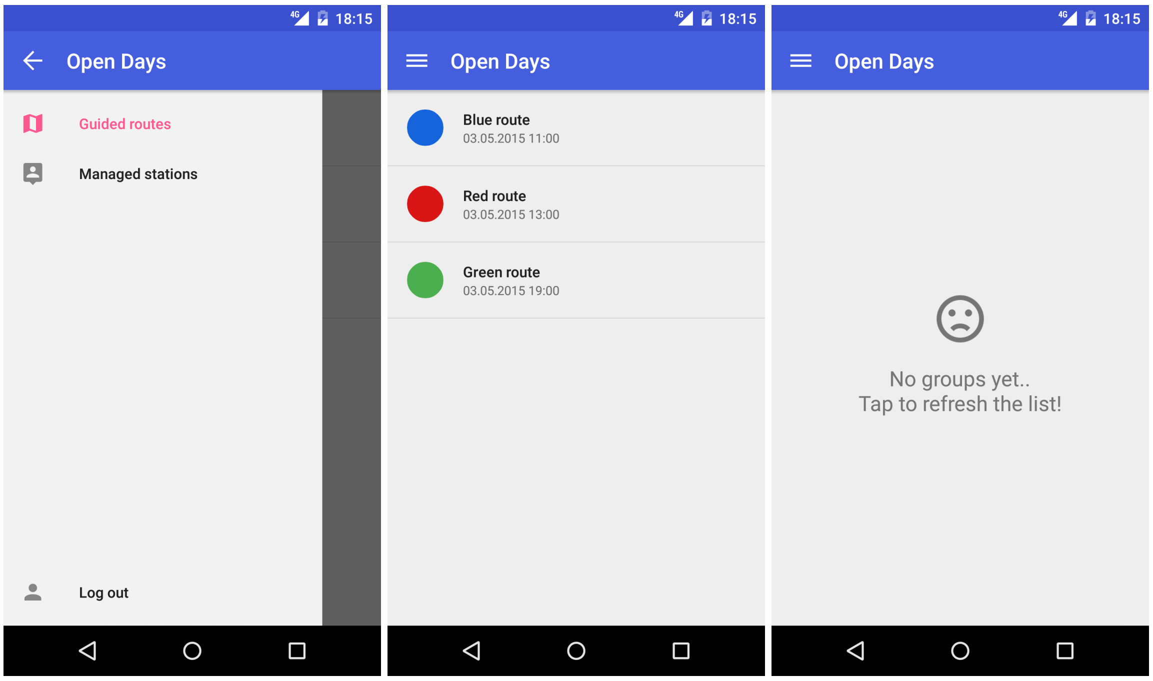
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <Button
        android:id="@+id/loginButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/login" />

</LinearLayout>
```

Listing 5.2: Authenticator activity XML layout declaration

Main Activity

`MainActivity` is declared as the application's launcher activity, which is started by the system when a user selects the application from the Android system's home screen.

The activity makes use of two fragments (`GroupListFragment` and `ManagedStationsListFragment`), which are switched based on the selection from the main menu. The menu conforms to a Navigation Drawer pattern[3], which is defined as a panel containing the application's main navigation options that slides out from the left edge of the screen.



(a) Navigation Drawer (b) List of routes (c) Empty list state

Figure 5.4: Various states of the Main Activity

Both of the fragments contain a list of routes. While `GroupListFragment` shows routes, where the current user guides a group, the `ManagedStationsListFragment` lists those that

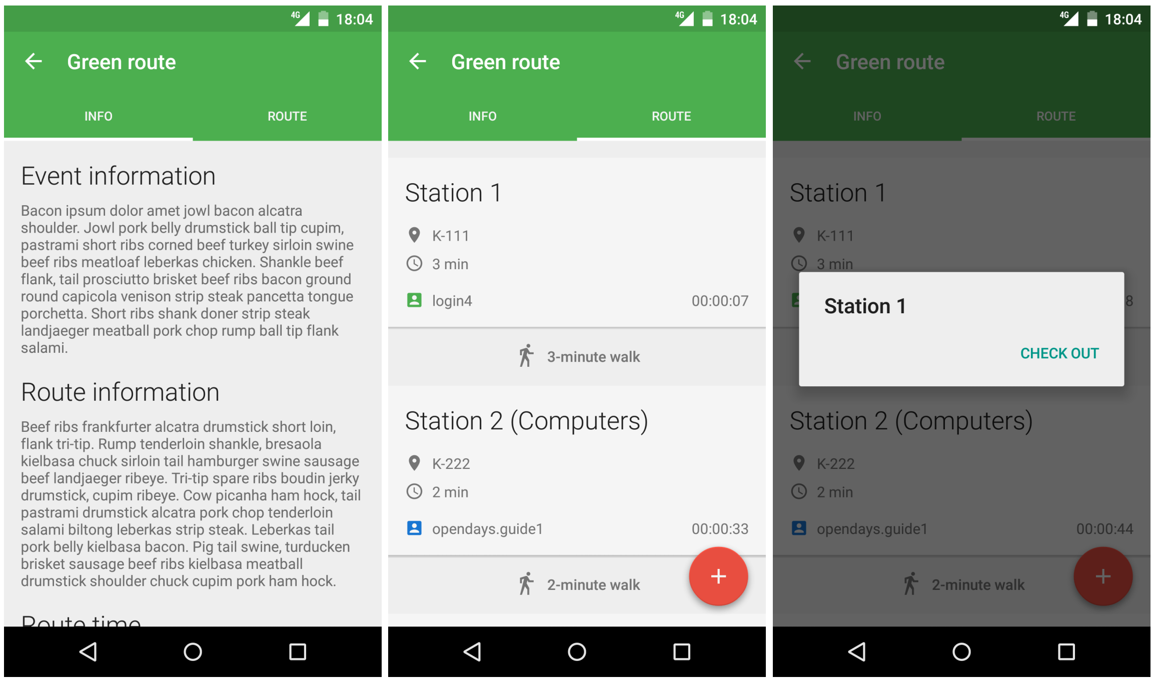
[3]https://developer.android.com/design/patterns/navigation-drawer.html

contain a station managed by the user. When the user clicks on a route in the list, a new Intent is created to start the `RouteActivity`. For routes from `ManagedStationsListFragment`, an argument signifying a read-only access is passed.

Route Activity

From the functional point of view, `RouteActivity` is the most important activity of the application. The goal was to create a compact UI, where all information is at hand. The solution makes use of a `ViewPager` component, which allows users to flip left and right through a list of fragments. In this case, the list contains only two fragments: `RouteInfo` and `RouteGuide`.

As shown in figure 5.5a, the `RouteInfo` fragment provides information about the selected route, its event, and the time it takes place. The route's color isn't shown in the info fragment itself, instead the application's toolbar is colored based on the current route.



Event information

Bacon ipsum dolor amet jowl bacon alcatra shoulder. Jowl pork belly drumstick ball tip cupim, pastrami short ribs corned beef turkey sirloin swine beef ribs meatloaf leberkas chicken. Shankle beef flank, tail prosciutto brisket beef ribs bacon ground round capicola venison strip steak pancetta tongue porchetta. Short ribs shank doner strip steak landjaeger meatball pork chop rump ball tip flank salami.

Route information

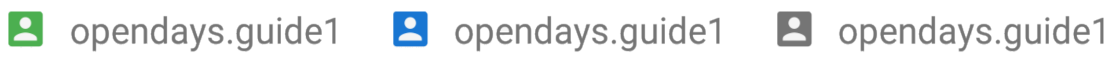
Beef ribs frankfurter alcatra drumstick short loin, flank tri-tip. Rump tenderloin shankle, bresaola kielbasa chuck sirloin tail hamburger swine sausage beef landjaeger ribeye. Tri-tip spare ribs boudin jerky drumstick, cupim ribeye. Cow picanha ham hock, tail pastrami drumstick alcatra pork chop tenderloin salami biltong leberkas strip steak. Leberkas tail pork belly kielbasa bacon. Pig tail swine, turducken brisket sausage beef ribs kielbasa meatball drumstick shoulder chuck cupim pork ham hock.

(a) Information Fragment (b) Route Fragment (c) Location Update Dialog

Figure 5.5: Various states of the Route Activity

The `RouteGuide` fragment, shows a list of route's stations ordered based on the current guide's starting position. Stations that are marked as closed by the organizer are left in the list, but their entries are grayed-out.

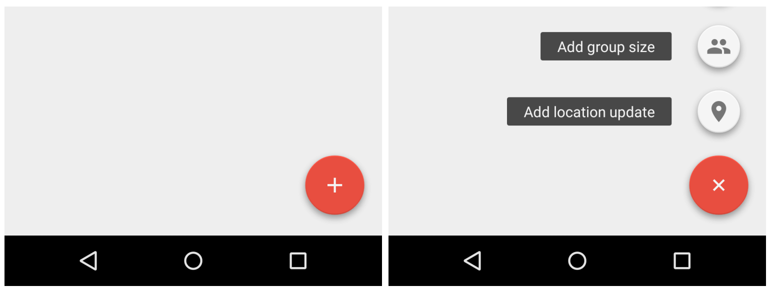
The list also displays the current location of all groups on the route along with the information about time elapsed from the moment of their last location update. For quick assessment, the group icons are color coded as shown in figure 5.6. A group belonging to currently logged in user is painted green. Groups which have synchronized their state with the remote server in the past 10 minutes are considered *active* and are rendered blue. Inactive groups are marked gray.



(a) Current user (b) Active group (c) Inactive group

Figure 5.6: Group icons used to distinguish their state

At last, `RouteGuide` fragment allows guides to change their starting position, update the size of their group, and most importantly send location updates. As emerged from the prototyping process, updating group's location must be intuitive, but it shouldn't be performed accidentally. To achieve these requirements, a combination of a Floating Action Button (FAB) pattern[4] and a confirmation dialog was used. The Google Material design guidelines describe FAB as a circular button, which should represent only the most common action on given screen, but which can also display related actions upon touch [21]. Figure 5.7 shows the selected functionality used in the implemented application.



(a) Closed state (b) Opened state

Figure 5.7: Floating Action Button pattern

Android Guidelines

The client application's user interface was developed with respect to Google Material Design Specification[5], which was defined to achieve a consistent user experience across all Android applications and devices. Besides other things, the extensive specification provides style recommendations for the use of color and typography, discusses layout principles, key lines and metrics, and defines a number of components to be used in Android applications.

5.4.2 Resources

Android provides a wide support for resource externalization to keep values, such as string constants, layouts, and images, away from the application's source code. The accepted resources range from simple text values to more complex, such as images, animations, themes,

[4]http://www.google.com/design/spec/components/buttons-floating-action-button.html
[5]http://www.google.com/design/spec/material-design/introduction.html

menus, and layouts. Different resource files can be used depending on the device's hardware specifications, such as screen size, resolution, device orientation or language [13].

Value resources

The implemented application contains several XML resource files, which store used colors, application styles, layout dimensions, and all user interface text labels. The default value files are stored in the `res/values` folder. Alternative resources are placed in folders with pattern `res/values-suffix`, where *suffix* specifies the required condition.

To achieve internationalization, language specific resources were placed in folders suffixed with a lowercase two-letter language code. Other used criteria include platform version using `v<API LEVEL>` pattern, or the screen orientation with 3 possible values: `port` (portrait), `land` (landscape) or `square` (square).

```
<resources>
    <color name="green_800">#056f00</color>
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <string name="no_internet">No internet connection!</string>
</resources>
```

Listing 5.3: XML resource file

Drawable resources

The application's image resources, such as icons, shapes, backgrounds and 9-patch graphics (stretchable component backgrounds), are stored in the appropriate `res/drawable` folders suffixed with screen pixel density qualifiers.

In Android, pixel density is measured in dots per inch (dpi). The platforms currently groups all possible screen densities into 6 categories: low (ldpi), medium (mdpi), high (hdpi), extra-high (xhdpi), extra-extra-high (xxhdpi), and extra-extra-extra-high (xxxhdpi).

Using resources

When a user starts the application, the system automatically chooses the appropriate resources. Resource values were mainly referenced from layout files, as shown in listing 5.4.

```
<TextView
    ...
    android:text="@string/guided_groups_empty_state_label"
    android:textColor="@color/grey_600"/>
```

Listing 5.4: Using external resources in a layout

The system also provides methods for easy resource access from code using a static `R` class, which is generated from the external resources during project compilation. The class contains subclasses for all of the resource types defined in the project, for instance `R.string` comprises of the text resources [11].

```java
ToastUtil.error(getActivity(), getString(R.string.no_internet));
```

Listing 5.5: Using external resources in Java code

5.4.3 Responsiveness

By default, all Android components, such as Activities and Services, run on the main application thread, which is often referred to as the UI thread. The reason for this naming lies in that fact that all events modifying the user interface are passed to a message queue, which is processed by the main thread. When there is a long running operation performed on the main thread, the message queue is blocked, which results in an unresponsive application [14].

Activities, which are unable to process input event within 5 seconds are considered unresponsive by the system and `Application Not Responding` (ARN) dialog may be shown, giving the user the option to kill the process altogether. However in reality, users notice user interface delays exceeding a couple of hundred milliseconds [13]. Therefore, time consuming operations, such as network communication, I/O operations, and complex calculations, are performed on a background thread in the application.

To create a background thread, standard Java `Thread` class can be used, with one big limitation however - the user interface cannot be updated from a non-UI thread. Instead, the Android platform provides the `AsyncTask` class to be able to perform operation on the background thread while updating the UI from event handlers running on the main thread.

```java
private class LogOutTask extends AsyncTask<Void, Void, Void> {
        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            dialog = createAndShowDialog();
        }

        @Override
        protected Void doInBackground(Void... params) {
            SyncHelper.cancelSync(getActivity(), account);
            new DbHelper(getActivity()).clearUserData();
            return null;
        }

        @Override
        protected void onPostExecute(Void aVoid) {
            super.onPostExecute(aVoid);
            dialog.dismiss();
        }
}
```

Listing 5.6: AsyncTask used for possibly long running log out operation

As shown in listing 5.6, long running operations are performed in the overridden `doIn Background` method, which is executed on a non-UI thread. To interact with the user interface, two methods were usually used: `onPreExecute` and `onPostExecute`. As the names suggest, the method is called before, respectively after the run of `doInBackground` method.

5.4.4 Data persistence

In order for the Android client to work without constant Internet connection, it was necessary to store data persistently on the device memory.

First, data containing information about events, routes, stations and group positions downloaded from the remote server is locally cached. Secondly, location updates and group sizes created in the client application by the current user are initially stored, but later removed after successful upload to the application server. In the application, structured data persistence was achieved through a combination of a SQLite database and a Content Provider.

SQLite Database

SQLite is a single-tier, lightweight relational database management system focused on use in embedded devices and limited-memory environments [13]. As already described in section 5.1, the SQLite library is by default included in the Android software stack. Moreover, the Android system provides data security by making all databases fully encapsulated, accessible solely by the application which created them.

To manage the database instances, an abstract `SQLiteOpenHelper` class was implemented. The SQLite Open Helper makes decisions, whether a database needs to be created or upgraded before opening it and caches the instance after it had been successfully opened. The implemented subclass overrides `onCreate`, `onUpgrade`, and `onDowngrade` methods, which simply execute SQL scripts for table creation, upgrade, and downgrade respectively, on the database instance passed as a method argument.

```
private static final String SQL_CREATE_GROUPSIZES =
            "CREATE TABLE " + GroupSizes.TABLE_NAME + " (" +
             GroupSizes._ID + " INTEGER PRIMARY KEY," +
             GroupSizes.COLUMN_NAME_TIMESTAMP + TEXT_TYPE + COMMA_SEP +
             GroupSizes.COLUMN_NAME_GROUP_ID + INT_TYPE + COMMA_SEP +
             GroupSizes.COLUMN_NAME_GROUP_SIZE + INT_TYPE + " )";

public void onCreate(SQLiteDatabase db) {
    db.execSQL(SQL_CREATE_GUIDEDGROUPS);
    db.execSQL(SQL_CREATE_GROUPSIZES);
    ...
}
```

Listing 5.7: SQLiteOpenHelper method for creating database tables

To access the database, `SQLiteOpenHelper` provides `getWritableDatabase` and `get ReadableDatabase` methods returning writable or read-only database instance, which can be directly used for data querying. However to create a generic interface for data access, using a Content Provider was preferred.

Content Provider

To provide a data access interface to be consumed by Content Resolver, Content Providers make use of an URI addressing model with the `content://` schema.

A new Content Provider was created by implementing an abstract `ContentProvider` class. It was necessary to override the `query`, `insert`, `update`, `delete`, and `getType` methods, which are used by the Content Resolver. All of the methods internally use the previously implemented SQLite Open Helper to access the database and manage data.

The Content Provider supports batch operation processing, however by default it executes each of the operations in its own transaction. The implemented application needed to perform batch operations wrapped in a single transaction. In order to do that, the `applyBatch` method must had been overridden to utilize a transaction from the underlaying SQLite database, as shown in listing 5.8.

```
@Override
public ContentProviderResult[] ...
    applyBatch(ArrayList<ContentProviderOperation> operations){
    final SQLiteDatabase db = dbHelper.getWritableDatabase();
    db.beginTransaction();
    try {
        int numOperations = operations.size();
        ContentProviderResult[] results = new ...
            ContentProviderResult[numOperations];
        for (int i = 0; i < numOperations; i++) {
            results[i] = operations.get(i).apply(this, results, i);
        }
        db.setTransactionSuccessful();
        return results;
    } finally {
        db.endTransaction();
    }
}
```

Listing 5.8: Transational support for batch operations in Content Provider

Content Resolver

Each Android application contains a Content Resolver instance, which can be used to query Content Providers, as resolver includes data management methods corresponding to those in the provider [13].

Content Resolver was extensively used to insert, update and delete data, especially using batch operations. However, to query the database asynchronously, Cursor Loaders were used.

Cursor Loader

Android Loaders perform asynchronous data loading and actively monitor the underlaying data source for changes. When the data changes, the loader automatically returns the refreshed records, which can then be used to update the user interface. Cursor Loader is a loader, which queries a content resolver and returns a database cursor [18].

Cursor Loaders were used in `GroupList`, `ManagedStationsList`, `RouteInfo`, and `Route Guide` fragments by implementing the `LoaderCallbacks<Cursor>` interface methods. As shown in listing 5.9, it is necessary to create cursor loader instance with requested database projection and selection, which internally queries the content resolver, and passes the resulting Cursor to a `onLoadFinished` callback, where the requested data can be obtained.

```
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    String[] projection = {COLUMN_NAME_ROUTE_NAME};
    CursorLoader cursorLoader = new CursorLoader(getActivity(), uri, ...
        projection, null, null, null);
    return cursorLoader;
}
```

Listing 5.9: Creating a cursor loader instance

5.4.5 Data synchronization

Data synchronization between the Android client and the remote server is one of the most important functions of the implemented system. The client application should use the latest server data to accurately display current group positions. At the same time, it's necessary to upload locally cached location updates to the server, for the data to be redistributed to other client devices.

Instead of implementing the whole synchronization mechanism, the application uses Android's sync adapter framework, which helps to automate asynchronous data transfers. Using the sync adapter has several advantages over a custom solution. First, a data transfer can be triggered by a variety of events, such as a data change, elapsed time, or a certain time of a day. Secondly, the adapter automatically monitors network connection to perform synchronization only with an Internet connection. Next, the framework allows account management integration for seamless authentication during data transfers. Most importantly however, the framework aims to improve battery performance by synchronizing data transfers across other apps as well, which reduces the number of times the device must be waken by the system [31]. As shown in listing 5.10, the implementation of a sync adapter was very straightforward, the data transfer code was simply placed in an overriden `onPerformSync` method of the `AbstractThreadedSyncAdapter` class.

```
public void onPerformSync(Account account, Bundle extras, String ...
    authority, ContentProviderClient provider, SyncResult syncResult) {
    new SyncHelper(mContext).performSync(syncResult, account, extras);
}
```

Listing 5.10: Sync adapter implementation

In the application, four types of events cause the sync adapter to perform a synchronization: local data change, server data change, elapsed time period and user demand.

Local data change When the client user updates his/her position, a data change is registered by the framework, which immediately schedules an `UPLOAD` sync that uploads all locally saved data to the server. In case a transfer cannot be run or fails, the framework adds it to a queue, which is automatically processed again when possible.

Server data change When relevant data on the server side changes, the client application is notified to refresh its data through a push notification sent within the Google Cloud Messaging framework (see 5.4.6). However, as the message delivery may fail, a fallback solution in a form of periodic sync is used as well.

Time period To guarantee data synchronization even in case of update message delivery delay or failure, the sync adapter is set to run periodically at a fixed interval. The synchronization period is recalculated during each data transfer in order to save battery and reduce network data usage. If an active route has been found, the period is set to a minute, otherwise the adapter runs at one hour-long intervals.

On Demand The user can demand an immediate data refresh using a vertical swipe gesture over the list of routes on `MainActivity` screen. This type should only serve as the last resort, as the framework is forced to run a single synchronization by itself, which is bound to be an inefficient use of power and network resources [31].

5.4.6 Google Cloud Messaging

Google Cloud Messaging (GCM) is a free service, which allows sending messages with up to 4kb of data from a server to devices running the Android operating system. The service architecture consists of 3 main components: GCM Connection servers, client application server, and the client application itself.

First, the client application registers with GCM to obtain a registration ID, which is uploaded to the implemented application server, where it's stored in a database. Afterwards, the server can use the stored registration ID to send a message to the client application through the GCM Connection servers. At last, based on the ID, the framework delivers the message to the correct device and application [20].

In the implemented system, 2 kinds of messages are used. Synchronization message is sent by the server to notify the client application about remote data change. Location update message is sent in case a group, which started before or after the current user, changes it's position. The latter type contains a text payload that the client application displays in form of a status bar notification.

5.4.7 Notifications

The application uses notifications to gain user's attention even when no Activity is in an active foreground state. A notification is a message displayed within the notification tray, which is handled by the Notification Manager. The notification can also vibrate the device, play sounds or flash LEDs to alert the user [13].

As said in previous section, notifications are used to display payloads received directly from the server, most often however, they alert a user, whose group is checked in at a station, that the time limit is exceeded.

5.4.8 Account authenticator

The first step to avoid saving user's password on the client device was enforcing the use of OAuth2.0 authentication tokens instead, as already described in section 4.4.1. Yet to securely store the authentication tokens, an account authenticator component, which is used within the Android's account and authentication framework, was implemented.

At first, when there is no saved user account, the authenticator starts the `Authenticator Activity`, prompting the user for username and password. These credentials are used to obtain an access and a refresh token from the authorization server. Next, a new account is created for given username, however instead of a password, the access token is saved and the refresh token attached to the `Account` object as additional data. At last the component can be used to authenticate server requests.

The authenticator was implemented to automatically use the stored refresh token to obtain new access tokens with each request. If this action fails due to token expiration, the component immediately re-prompts the user for credentials to obtain a new valid refresh token. In case other application tried to access the account password created by the client, a security exception would be thrown by the Account Manager [16].

Chapter 6

Testing

This chapter describes testing methods used during and after development of the application to assure its quality. First, the testing of the server application will be discussed, followed by the description of Android application tests. At last, the user testing of the whole system will be covered.

6.1 Server Application Testing

The server application was tested in two different ways. The modules of data access and business layers were tested using programmatic Units test, while for the UI of the presentation layer browser compatibility tests were performed.

6.1.1 Unit testing

To separately test individual parts of the application, JUnit framework was used. The unit tests were written immediately after finishing a particular component, which helped to ensure its correctness and find software errors early in the development process. Once all test for given component passed, they were added to regression test suite, which was always run before committing new changes to a repository to make sure that all of the already tested components were still working properly.

6.1.2 Browser compatibility

Compatibility tests were performed to verify that the user interface is displayed correctly in different web browsers. All application screens and their states were tested on desktop in Mozilla Firefox 37.0.2, Google Chrome 42.0.2, Internet Explorer 11, and Safari 8. On Android devices, the default web browser and Google Chrome 42.0.2 were used.

The testing discovered only minor problems, such wrong custom font rendering and rounded edges not being supported in Internet Explorer.

6.2 Android Application Testing

In the client application, unit testing was also utilized for individual components. However being a native application, compatibility testing must had been performed on various system versions and hardware configurations.

6.2.1 Component testing

The Android platform supports unit testing with the JUnit framework. However, many methods in Android require a Context object from an `Activity` or `Application` to carry out its actions. To start an activity under test and be able to obtain its context inside a unit test, `ActivityTestRule`s were used.

6.2.2 Compatibility testing

The client application must function properly on different Android versions and hardware configurations. To test the application on most combinations, both real devices and software emulators were used.

The application was tested on the following Android versions: 4.0.4, 4.1.2, 4.2, 4.4, 5.0 and 5.1.1. Few major errors were discovered on older system versions due to API changes. These bugs were solved by using the Android Support Library package, which provides backward-compatibility of the newer APIs. Tested screen densities range from `mdpi` to `xxhdpi`, which helped to improve the user interface layouts to be displayed properly on most devices.

6.3 User testing

On the finished application, user testing was performed. The test group consisted of 4 students of the Czech Technical University in Prague, as they and the university staff are the expected target audience of the system.

6.3.1 Web Client

Each person had a list of actions to be performed using the web user interface. First goal was to create a new account and log in to the system. Then, a new event with blue and red route was to be created. Each route should have at least 3 stations, 2 groups, and 1 station manager. Other actions included adding a station to an existing route, reordering guides starting position, changing the user password, or changing the system language.

This test pointed out an user experience problem, which 2 of the 4 people in the test encountered. It is necessary to add email addresses of guides and station managers *before* creating a new route, if they are to be added in that step. However, it is possible to add groups and station managers to an already existing route, which the users later found out.

6.3.2 Android Client

The mobile client was tested by all of the 4 people at the same time. Each person logged in to the application using given credentials and was supposed to guide a group along the stations of a blue route at 3 o'clock, gradually updating their location and their group size.

During the test, all groups stayed active and no unexpected collisions occurred. However, the test group agreed that notifications informing guides about time limit expiration were sometimes hard to notice. Also, the application doesn't force guides to input their group size, which caused that one user completely forgot to add his group's size until the end of the test.

Chapter 7

Conclusion

In this thesis, a system for team synchronization during open days at the Czech Technical University in Prague was designed and implemented based on the requirements acquired from the current organizers. The web application allows event administration and real-time overview to its organizers, while the mobile application is to be used by persons who lead groups of prospective students around the school. The latter enables guides to update their groups' information, such as position or size, and at the same time shows the current location of other groups on the same route. Application components were continuously tested by programmatic unit tests and in the end the whole system was tested by real users.

7.1 Future work

The created application fully implements required functionality, however based on current and future feedback, the system might be further improved or extended to provide additional features.

For instance, more mobile clients could be implemented to support other operating systems such as Apple iOS or Windows Phone OS. This extensibility is allowed thanks to the REST API provided by the server application.

Other suggestions include adding a communication functionality between guided groups of the same route, so that users don't have to leave the application in order to pass messages to each other. Furthermore, new user types could be added - for example the prospective students could use the application to rate the event and get additional information about each station. Last, usability testing of the application could help to improve the overall user experience and ease of use.

Bibliography

[1] J. Arlow and I. Neustadt. *UML 2 a unifikovaný proces vývoje aplikací: Objektově orientovaná analýza a návrh prakticky.* Brno: Computer Press, 2nd edition, 2007.

[2] C. Bauer and G. King. *Hibernate in Action.* Greenwich, CT: Manning Publications, 2005.

[3] M. Cade and H. Sheil. *Sun Certified Enterprise Architect for Java™ EE Study Guide.* Prentice Hall, 2nd edition, 2010.

[4] R. T. Fielding. Representational state transfer (rest). `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, seen on 14. 5. 2015.

[5] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, seen on 24. 4. 2015.

[6] M. Fowler and D. Rice. *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley, 2003.

[7] D. Geary and C. Horstmann. *Core JavaServer Faces.* Prentice Hall, 3rd edition, 2010.

[8] Android activity lifecycle. [online]. `http://developer.android.com/images/training/basics/basic-lifecycle.png`.

[9] Oauth 2.0 protocol flow. [online]. `http://technogerms.com/wp-content/uploads/2013/10/Screen-Shot-2013-10-04-at-3.17.24-PM.png`.

[10] J. Kaplan and W. Crawford. *J2EE Design Patterns.* O'Reilly, 2003.

[11] S. Komatineni and D. MacLean. *Pro Android 4.* Apress, 2012.

[12] G. Mak, J. Long, and D. Rubio. *Spring Recipes: A Problem-Solution Approach.* Apress, 2rd edition, 2010.

[13] R. Meier. *Professional Android 4 Application Development.* Wiley/Wrox, 2012.

[14] M. L. Murphy. *The Busy Coder's Guide to Android Development.* CommonsWare, LLC, 2013.

[15] Simcrest - what is 3 tier architecture: and why do you need it.
`http://blog.simcrest.com/what-is-3-tier-architecture-and-why-do-you-need-it`, seen on 14. 12. 2014.

[16] Accountmanager.
`http://developer.android.com/reference/android/accounts/AccountManager.html#getPassword%28android.accounts.Account%29`, seen on 7. 5. 2015.

[17] Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 4th quarter 2013.
`http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/`, seen on 30. 4. 2015.

[18] Cursorloader.
`http://developer.android.com/reference/android/content/CursorLoader.html`, seen on 4. 5. 2015.

[19] The java ee 6 tutorial - what is facelets?
`http://docs.oracle.com/javaee/6/tutorial/doc/gijtu.html`, seen on 4. 5. 2015.

[20] Google cloud messaging for android.
`https://developer.android.com/google/gcm/index.html`, seen on 7. 5. 2015.

[21] Buttons: Floating action button.
`http://www.google.com/design/spec/components/buttons-floating-action-button.html`, seen on 30. 4. 2015.

[22] The java™ tutorials - internationalization.
`https://docs.oracle.com/javase/tutorial/i18n/intro/index.html`, seen on 4. 5. 2015.

[23] Java se technologies - database.
`http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html`, seen on 14. 12. 2014.

[24] The oauth 2.0 authorization framework.
`https://tools.ietf.org/html/draft-ietf-oauth-v2-31`, seen on 24. 4. 2015.

[25] An introduction to oauth 2.
`https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2`, seen on 24. 4. 2015.

[26] Understanding rest.
`https://spring.io/understanding/REST`, seen on 14. 5. 2015.

[27] The ioc container.
`http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html`, seen on 24. 4. 2015.

[28] Domain object security (acls).
`http://docs.spring.io/spring-security/site/docs/3.0.x/reference/domain-acls.html`, seen on 27. 4. 2015.

[29] Security namespace configuration.
`http://docs.spring.io/spring-security/site/docs/3.1.7.RELEASE/reference/ns-config.html`, seen on 28. 4. 2015.

[30] 16. expression-based access control.
`http://docs.spring.io/spring-security/site/docs/3.1.7.RELEASE/reference/el-access.html`, seen on 28. 4. 2015.

[31] Transferring data using sync adapters.
`https://developer.android.com/training/sync-adapters/index.html`, seen on 6. 5. 2015.

[32] Transactions (database engine).
`https://technet.microsoft.com/en-us/library/ms190612%28v=sql.105%29.aspx`, seen on 22. 3. 2015.

Appendix A

Used Libraries

Server Application

GCM Server https://github.com/google/gcm

GSON https://code.google.com/p/google-gson/

Hibernate http://hibernate.org/orm/

Joda-Time http://www.joda.org/joda-time/

Primefaces http://www.primefaces.org/

Spring Framework http://projects.spring.io/spring-framework/

Spring Security http://projects.spring.io/spring-security/

Android Applicaiton

AppCompat developer.android.com/tools/support-library/features.html#v7-appcompat

Espresso https://code.google.com/p/android-test-kit/wiki/Espresso

FloatingActionButton https://github.com/futuresimple/android-floating-action-button

GSON https://code.google.com/p/google-gson/

Joda-Time http://www.joda.org/joda-time/

OkHttp https://github.com/square/okhttp

Appendix B

List of used abbreviations

ACL	Access Control List
API	Application Programming Interface
ART	Android Run Time
BO	Business Object
CI	Constructor Injection
CSV	Comma Separated Value
CTU	Czech Technical University in Prague
DAL	Data Access Layer
DAO	Data Access Object
DI	Dependency Injection
DTO	Data Transfer Object
EL	Expression Language
FAB	Floating Action Button
GCM	Google Cloud Messaging
JDBC	Java Database Connectivity
JSF	Java Server Faces
JSP	Java Server Pages
NTP	Network Time Protocol
ORM	Object/Relational Mapping
PI	Property Injection

POJO Plain Old Java Object

REST Representational State Transfer

SI Setter Injection

SID Secret Identity

SpEL Spring Expression Language

UC Use Case

UI User Interface

Appendix C

UML diagrams

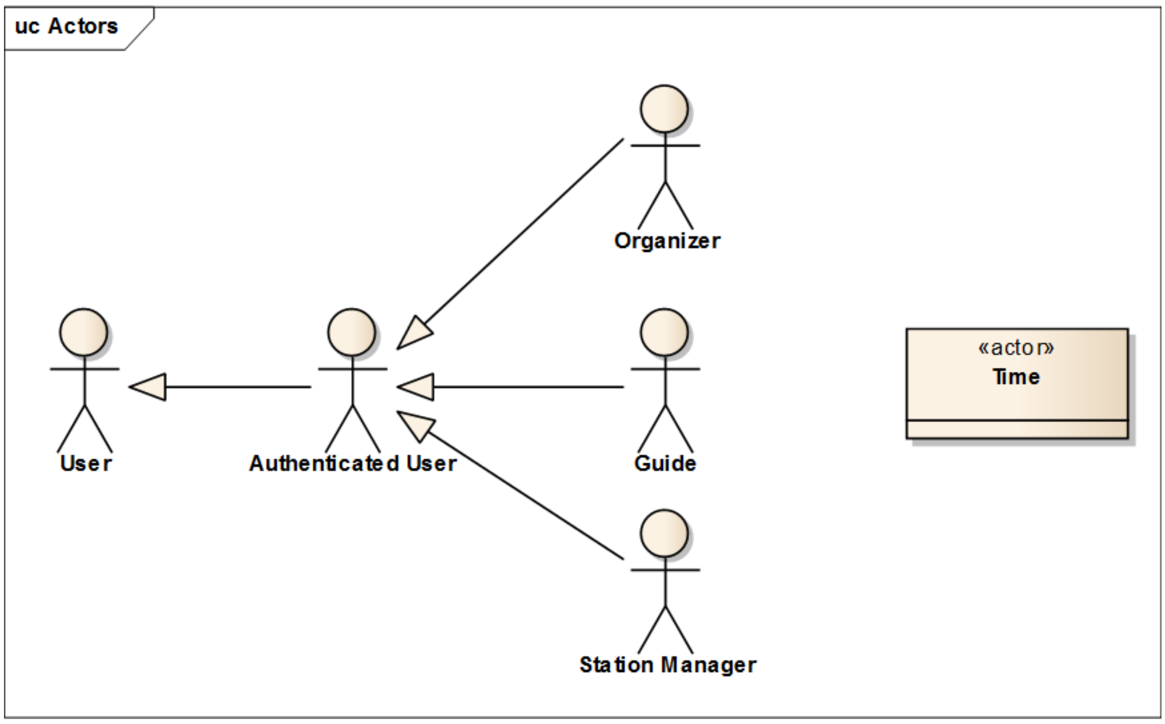


Figure C.1: Diagram of use case actors

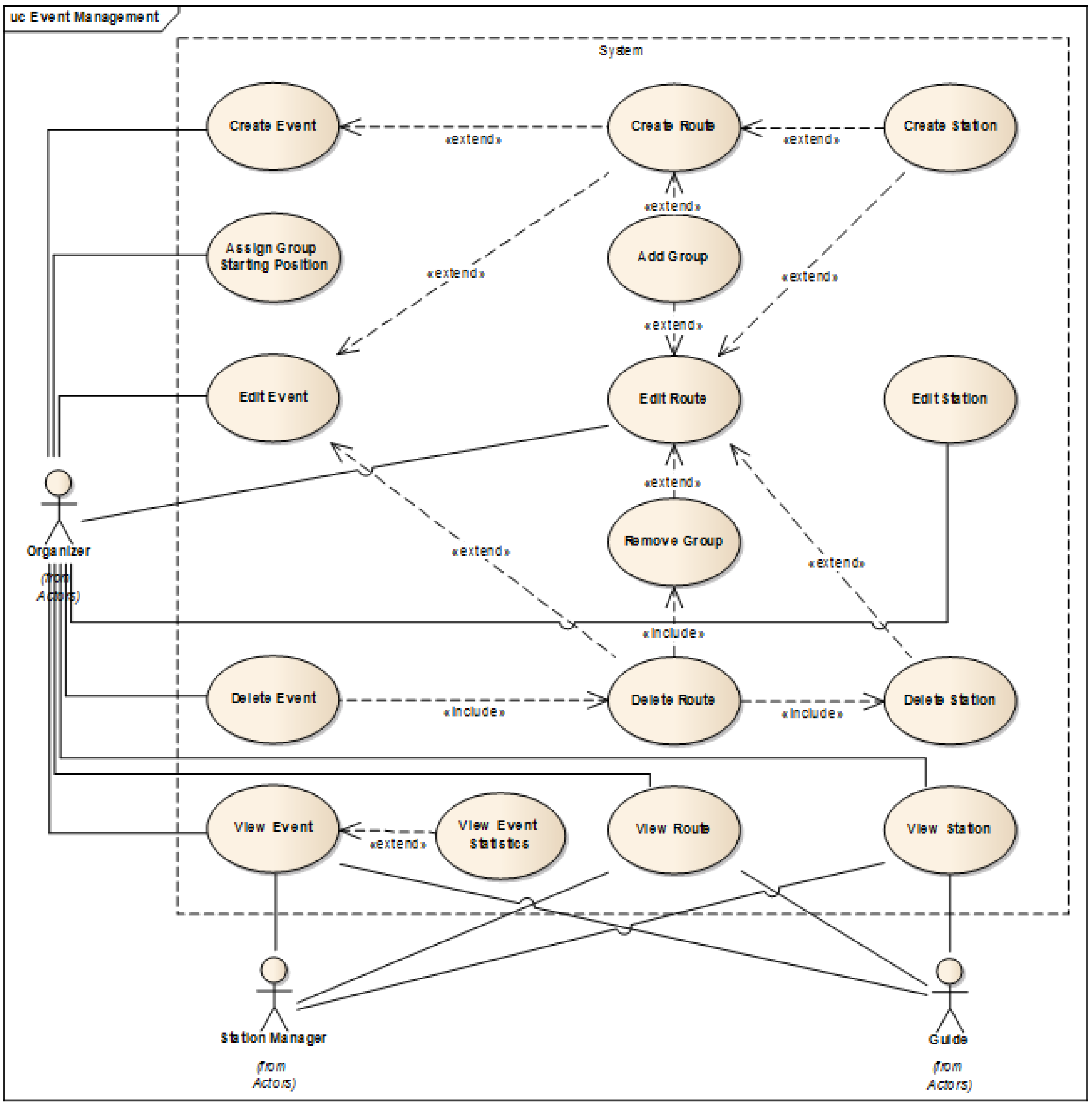


Figure C.2: Use Case Diagram describing event management

Appendix D

Deployment guide

Server

This guide describes how to deploy the implemented application on a server running Debian 7.0 "Wheezy" system.

Server time

Before we start installing programs needed for application deployment, we should set the server time. We will use the Network Time Protocol (NTP), which will automatically sync server's time.

Debian's default repositories contain `ntp` packages, which allows us to install it using the `apt` packaging system. First we update local `apt` repository and then install the required package using the following commands in the machine's terminal.

```
sudo apt-get update
sudo apt-get install ntp
```

Next edit file `/etc/ntp.conf` to use the following servers.

```
server 0.cz.pool.ntp.org
server 1.cz.pool.ntp.org
server 2.cz.pool.ntp.org
server 3.cz.pool.ntp.org
```

PostgreSQL

To install PostgreSQL database packages, run the following command in the terminal.

```
sudo apt-get install postgresql postgresql-contrib
```

A new linux user called `postgres` with a default Postgres role was created, to which we will now switch in order to create additional database accounts.

```
sudo -i -u postgres
```

To create a new user named `opendays` type the following command.

```
createuser opendays
```

The shell script will prompt 3 questions, which we will all decline.

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Still logged in as the `postgres` user, we can now create a new database with the same name as the newly created user.

```
createdb opendays
```

Now that a new user and a database have been created, we create a new linux user with the same name and switch to said account.

```
sudo adduser opendays
sudo -i -u opendays
```

Next we connect to our `opendays` database.

```
psql
```

To change the `opendays` user password, type:

```
\password opendays
```

Being connected to the database, we can run SQL commands. We need to copy and paste the contents of 2 files. First we run SQL script from file `initializeACLSchema.sql`, which will create tables necessary for Spring's ACL Security. To check if the tables have been correctly created run a `\d` command, which should render following output.

```
 Schema |            Name            |   Type   |  Owner
--------+----------------------------+----------+----------
 public | acl_class                  | table    | opendays
 public | acl_class_id_seq           | sequence | opendays
public | acl_entry                  | table    | opendays
public | acl_entry_id_seq           | sequence | opendays
public | acl_object_identity        | table    | opendays
public | acl_object_identity_id_seq | sequence | opendays
public | acl_sid                    | table    | opendays
public | acl_sid_id_seq             | sequence | opendays
```

Next we create tables needed for oAuth token persistence by pasting the contents of `initializeTokenSchema.sql`, after which the following tables should be added.

```
public | oauth_access_token         | table    | opendays
public | oauth_approvals            | table    | opendays
public | oauth_client_details       | table    | opendays
public | oauth_client_token         | table    | opendays
public | oauth_code                 | table    | opendays
public | oauth_refresh_token        | table    | opendays
```

Next we must supply the database connection information to `credentials_localhost.properties` file and build a new `.war` file by running `mvn package` in your console.

Tomcat 7

As well as PostgreSQL, servlet container Apache Tomcat 7 can also be found in the Debian's default repository. To install the container, run the following commands in the linux console.

```
sudo apt-get install tomcat7
```

After the installation has completed, you can check `localhost:8080` in your web browser to see whether it was configured correctly. If you see webpage with title `It works!`, then you can proceed to deployment of the `.war` file we created in the previous section. Rename the `.war` file to `ROOT.war` and place it in `/var/lib/tomcat7/webapps` folder.

Now edit `/var/lib/tomcat7/conf/server.xml` file to make tomcat listen on port `80` instead of `8080`.

```
<Connector port="80" protocol="HTTP/1.1" connectionTimeout="20000" ...
   redirectPort="8443" />
```

Next edit `/etc/default/tomcat7` file. First, uncomment and allow `AUTHBIND` to be able to use port lower than 1023 (in our case 80).

```
AUTHBIND=true
```

Secondly, increase Java heap space for the container from 128MB to 512MB.

```
JAVA_OPTS="-Djava.awt.headless=true -Xmx512m -XX:+UseConcMarkSweepGC"
```

Finally, restart Tomcat to deploy the application.

```
sudo service tomcat7 restart
```

Android

In class `cz.kubaspatny.opendays.app.AppConstants` change the `HOST` constant to the new domain.

```
static final String HOST = "http://dod.felk.cvut.cz/";
```

In terminal run the following command to build an `.apk` file, which can be submitted to Google Play Store.

```
./gradlew assembleRelease
```

Appendix E

CD Content

The following folders are to be found on the attached CD:

```
CD
|-- android
    |-- src            source files of the Android appplication
    |-- apk            contains release .apk files
|-- deployment guide
    \-- guide.pdf      PDF version of deployment guide
|-- screenshots
    |-- android        contains screenshots of the Android application
    |-- prototype      contains sketches of application prototype
    |-- web            contains screenshots of the web application
|-- server
    |-- src            source files of the server application
    |-- war            contains deployable archives
|-- thesis
    |-- src            source files of this thesis in Latex
    \-- thesis.pdf     PDF version of this thesis
```