University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

MASTER THESIS

Pilsen, 2013 Petr Miko

University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

Master Thesis

Mobile system for management of EEG/ERP experiments

Pilsen, 2013 Petr Miko

I hereby declare that this diploma thesis is completely my own work and that I used only the sources cited.

Pilsen, May 16, 2013 Petr Miko, ...

Abstract

Gathering EEG Data Using RESTful Web Service and Android

The goal of the thesis is to present a problem and a solution of mobile gathering of EEG data. EEG researchers need to have their data available and synchronize them in manner of reading and writing meta-data and uploading data files. This thesis presents development of a RESTful web service on Java EE server and creation of an Android RESTful web service client, while respecting requirements and needs of EEG researchers.

Keywords: mobile, web service, Java, RESTful, Android, EEG, Spring, RestTemplate.

Contents

1 | Introduction

If we had to define aspects of modern application, mobile and service-oriented aspect would be certainly among the most mentioned ones. Being accessible from everywhere and anytime, while providing functionality equal to user's experience from computers, is a goal of application creators and most desired wish of many users.

Currently there are common expectations upon any of us that we are quick in our jobs and efficient at the same time. It requires having all the necessary data within our reach. Giving the fact that personal computers are not always available right away, focus of many companies and developers have shifted on devices, which are with us most of the time – our mobile phones and also tablets, whose market share increases literally every day.

Having a native application for mobile devices gives its creator advantage against common and in some cases obsolete ways to work with users data. User, who has just written an important note, does not want to manually copy the file, when he gets to the computer – he wants to save it and have it available in his/her computer by the moment, the computer is turned on. User, who searches for a specific piece of information, which is already available on a server, does not want to turn on the computer especially if he/she needs information right away – such user needs to have a possibility to access the information from his/her mobile phone, which is most definitely on the user and ready to be used. When an application provides to user such possibility, which extends and improves user's usecase, we speak of this possibility as of a service.

Modern application often provides many separate services which are joined as modules of an complex and sophisticated application. Every service provides its public interface, by which it is available for user, so a client application could access it and use it for its purpose. This approach of application design is recognized as SOA, service-oriented architecture, and application developed this way is, by definition, service-oriented.

This thesis is dedicated to provide a solution for a need of EEG researchers to have a quick, simple and accessible way to obtain basic information about their experiments and to store measured data, when measuring computer does not dispose of internet connection, while their mobile devices do.

The theoretical part of this thesis analyses requirements originated from such needs, submits solutions and explains the features of chosen solution. Specifics and basic description of required technologies are also presented in this part.

The following part of practical realization puts its focus on description of development process of an Android client application, service-oriented server side and also submits means to develop own application.

The last part of the thesis presents testing results, proposes ideas for future application enhancement and submits the thesis conclusion.

2 | EEG and ERP

In matter of recording brain activity, there are two possible approaches. A first approach is obtaining a record using an *electroencephalogram* in an diagnostic procedure called **Electroencephalography** (EEG). This method provides non-invasive and relatively cheap way to measure an active electric potential of brain cells. Even though this method is usable for diagnosing diseases like epilepsy or recognize brain death, its drawback is that the record is a mixture of hundreds different neural signals, in which might be difficult to recognize specific neuro-cognitive processes.

A second possibility is a derivative of EEG measurement – ERP measurement, in which we seek for **Event Related Potentials**. Event related potentials are measurable electric potentials created in reaction to a stimulus of sensory origin, like touch, smell or sight. Stimuli might be both of conscious or unconscious type, while the stimuli response might be usually more distinctive to observe in the recording, when a tested subject does not expect the stimuli to come. Such fact is often used when measuring stimuli response called *P3* or also *P300*. Such potential has positive voltage deviation and occurs right about 300 milliseconds after stimulus. Also, ERP potentials are measured locally at desired brain cortex, while EEG itself represents a global method of measuring the brain activity.

3 | EEG Base

EEG researchers produce large amount of recordings and related meta data during their experiments, like specific measuring scenario details, description of a measured subject etc. The EEG Base is a web portal created in order to provide a complex database for gathering such information, while extending researchers' possibilities in manner of sharing stored information between each other inside the scope of the EEG Base.

The functionality of the EEG Base has been incrementally improved and extended. Besides storing data and meta data the EEG Base now provides methods for processing stored experimental data and disposes of SOAP web services for third-party application access. Also, each user of the EEG Base is affected by rights that are assigned to his/her account – that ensures, user can access data, which are either public, or to which user participates.

Technologically, the EEG Base is a Java Enterprise Edition application, which uses frameworks as Spring, Hibernate, Apache CXF, etc. The portal is being developed under supervision of Roman Mouček, Petr Ježek and Petr Brůha in cooperation with students, who may gain useful knowledge of current software development technologies by participating on the development process.

The portal identifies itself with an INCF (*International Neuroinformatics Coordinating Facility*) effort in the development and standardization of neuroinformatical databases.

4 | Requirement Analysis

An experimenter uses a personal computer for his/her work with special hardware peripheries, like EEG caps or Brain Vision recorder. There must be clearly defined scenario and described a set of used hardware for experiment itself, in order to repeat such experiment. Also information about a tested subject is required – besides usual information of name and surname is relevant if the subject is left-handed, currently suffering from some disease etc. Environment, in which the experiment is conducted, is relevant too. Its description must be made and preserved.

All these pieces of information, and even more, must be stored as well as the experiment recording itself – EEG Base mentioned in the previous chapter serves for this purpose. Most of the time might be sufficient to use the very same personal computer as for conducting the experiment, but there are at least two valid situations, when using a computer is not a viable option.

1. **Computer is not connected to the Internet**
   Computer dedicated for conducting experiments may not dispose of internet connection. Assuming, that other members of research group are depending on experiment results and are not currently present, there arises a problem with delivering the data towards them.

2. **Need of quick access to existing information**
   Researchers may be out of reach to computers in need to quickly access some specific basic information about their experiments and scenarios (e.g. during a conference). If there is no viable alternative, experimenters might find themselves in situation, when they must work with incomplete information, or postpone their current activity till they can access the information required.

A solution must be presented for these unresolved needs of **data gathering** and **data storing**. Giving the fact that the EEG Base offers both the functionality and information required, an obvious solution is to use *services* of the EEG Base with a *mobile device* application.

Mobile devices are the perfect solution for filling the "unreachable computer" gap, because they are in present day almost always with us and do provide

internet connectivity. In addition, modern mobile operation systems and platforms built on them provide a solid and stable runtime for creating powerful client applications that can cover experimenters' needs.

Despite the facts that there are laptops with mobile internet connection and that the mobile phones might provide internet connection using tethering technology, the mobile phone by itself is more accessible in terms of device size and faster in terms of time required to start to use the application. Time to launch a native application in an already running mobile phone is shorter than time required to start a larger laptop, in which a web browser would have to be started. Mobile devices with right application provide better user experience than their possible alternatives.

Solution for data gathering and data storing itself must comply with a set of requirements, in order to cover the needs mentioned. The requirements are based upon the experimenters' needs and were discussed with EEG researchers of the University of West Bohemia.

- Provide access to experimenters' data from almost everywhere.
- Access itself must be simple, user-friendly and intuitive.
- Creation of new data file records is usually not needed nor possible without a computer, which would process them, unlike creation of meta data records. Meta data contain important information by themselves, so it is desired that they would be available any time for further operations. This fact results in a requirement that data files must be only stored to EEG Base, while meta data must be available any time, either for storing or for retrieving them once again.
- Experiments and scenarios must be creatable through the client application.

Moreover, these requirements originate with experimenters' use cases, which were also identified and are visible in Figure 4.1.

With services already existing inside the EEG Base application, a proper communication interface for the client-server contract must be defined.
The best solution are web services, which provide clean and simple abstraction with clear definition of communication usable across platforms. In this case RESTful web services are a perfect fit due to small communication overhead and simple implementation (details in the following chapter).

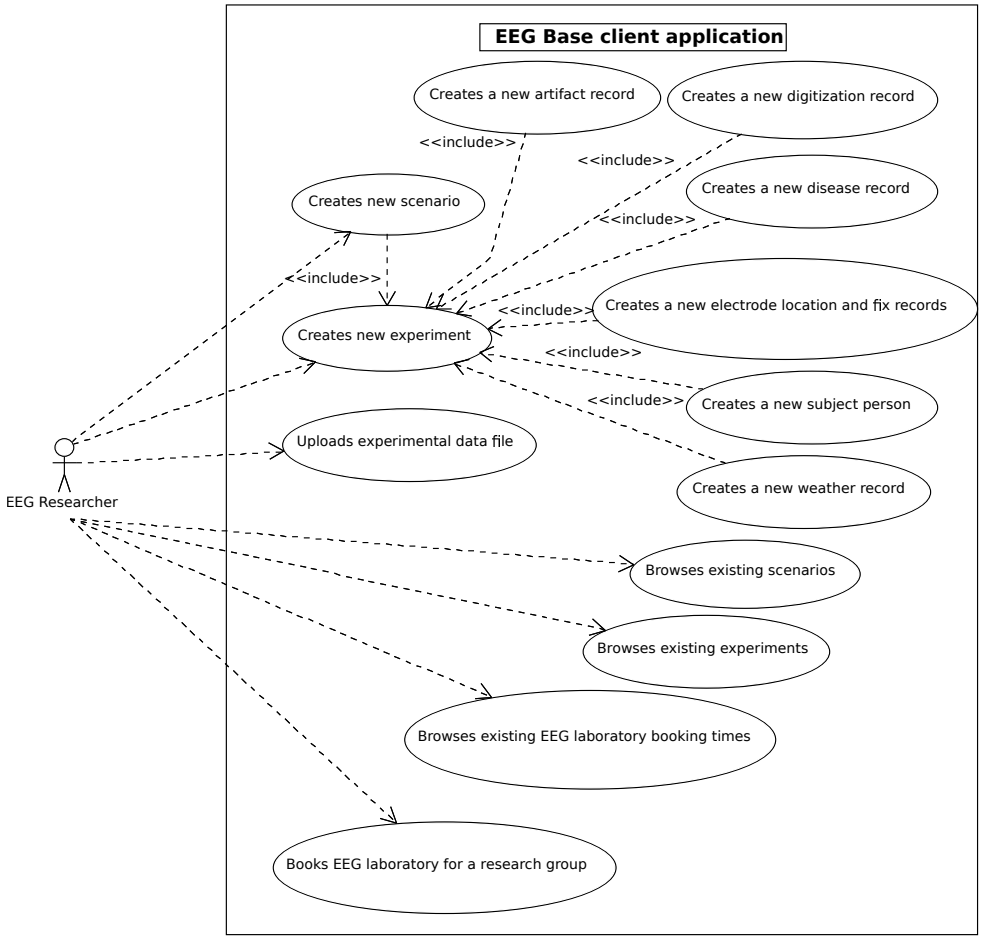


Figure 4.1: EEG Database client application use case diagram

5 | Web services: SOAP and REST

When a client application needs to use a service located on a server, using the remote procedure calls or remote method invocation can be difficult. Large pieces of shared code can be required and the solution will be probably platform dependent. But there is a better solution – to use a web service.

Every web service specifies its own communication protocol and provides a public interface for access; thus it is accessible equally from multiple platforms and keeps services modularity without a need of shared code. There are two separate web service technologies with different approach to the execution of remote procedure calls:

- SOAP web services
- RESTful web services

5.1 SOAP Web Services

Older, yet still vastly used, web services are SOAP web services. The SOAP represents name of a communication protocol, which these web services use – a **Simple Object Access Protocol**. This protocol is responsible for delivering messages usually via the HTTP protocol, which is used due to lower probability of being blocked by firewalls during their route.

Every SOAP web service provides public interface, which is accessible through specified URL. This service point of entry is called an *endpoint*. When a user wants to use capabilities of a SOAP web service, he/she needs to know the exact endpoint and an interface definition, which is contained in a **WSDL file**.

WSDL is a XML file written using the **Web Service Definition Language**, which states what service methods are available, which objects are used as return types and method parameters, and what is the structure of such objects. In terms of RPC, WSDL defines a server stub. Client stub is represented by source files, which are generated using the WSDL file. These files are used

within a client application afterwards.

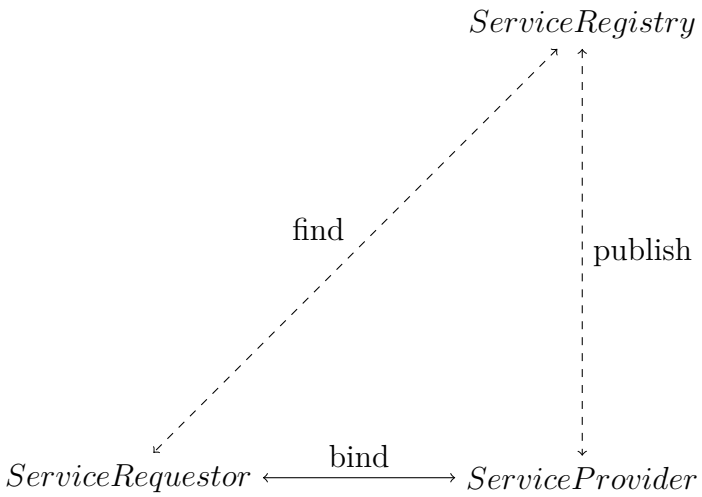


Figure 5.1: Communication diagram between web service elements [3]

If we want to relate to a common model of web services as in Figure 5.1, relations would be following:

- **Service Registry**: A location which contains information about available web services and their providers. These pieces of information are saved by service providers themselves →
  each web service is registered in a registry implemented using **Universal Description, Discovery and Integration protocol** (UDDI)[17] and a **WSDL** file, which represents the service's interface, is generated or manually created for each service.
- **Service Provider**: A hardware or software unit providing the web service itself → accessible by **endpoint** URL.
- **Service Requestor**: A subject requesting a specified web service → a **client application** which uses a code generated from WSDL.

5.2 RESTful Web Services

Representational state transfer is a architectural style for developing applications, which is specific by a set of constraints, as mentioned in [4].

- **Client-server separation of concerns** – a server does not need to handle GUI or an application state. In combination with a **uniform interface** this results in platform independence, allows better client scalability and server side itself is much simpler.
- **Stateless** – client-server communication is in REST stateless, so every request from client to server must contain all necessary information. Keeping of state belongs to the client side.
- **Cachable** – data within responses must be labeled whether they are cacheable or not, so client application could reuse them by saving them into a cache for further use. It lowers a necessary number of interactions and bandwidth usage overall.
- **Layered system** – a client does not need to obtain data precisely from the requested location, the requested location might be an intermediary, which works as a client itself upon other nodes in the network. This may, if well designed, improve scalability due to possible load balancing while using previously mentioned caches.
- **Code-On-Demand** – a server might provide to client a code, which executes on the client side and provides extra functionality. This principle is purely optional in the REST architectural style.

While SOAP protocol extends HTTP, REST reuses HTTP methods of GET, PUT, POST and DELETE to perform CRUD[1] operations over specified URL. For an information transfer it is used HTML, XML or, lately popular, JSON. To information transferred we refer as of *resources* and mentioned HTTP methods are used for analogical operations over them.

By mentioning a **RESTful** web service we mean an API[2], which complies with the REST specification and also communicates using XML or JSON via the HTTP protocol using its methods.

A RESTful web service's server side listens on specific URL, which usually contains a name of a domain object with which HTTP methods should work. Let us say, there is URL ending with suffix **/experiments**.
A common expectation for such URL is, that using HTTP methods we can:

- fetch all experiment records using a GET method

[1] Create, Read, Update, Delete – recognized as basic data operations
[2] Application Programming Interface

- remove all records by a DELETE request
- replace all records with a PUT method
- create a new experiment record using a POST HTTP method

For updating or deleting a single record we would need to know a resource identifier, which is usually put as a suffix to the original URL.

5.3 Basic Comparison

The fact, that for implementing of a RESTful web service client we require only simple HTTP communication and using protocol's methods, vastly helped to using RESTful web services as a primary solution for many simple client-server applications. It applies even more in case of mobile device software. With no need to generate a client stub, we only need to know service URLs and how to operate over them. Moreover, messages from SOAP web service are larger that the RESTful ones, due to necessity of complying with SOAP standards.

On the other hand, SOAP services can rely on WSDL files, while RESTful web services might provide a web service description written in the WADL – Web Application Description Language. However WADL is not standardized and is not mandatory – user must rely on developers that they provide WADL or sufficient documentation of endpoint URLs. Also, RESTful web services should be stateless unlike SOAP ones, which implicates that SOAP web services are more suitable for other cases then RESTful web services, e.g. state-dependent sequential data processing.

6 | Java technologies

6.1 Annotations vs. XML Definitions

When creating a well-designed modern application, a certain level of code abstraction, modularity and separation of semantics and implementation is mandatory. What was created mainly in order to reasonably maintain large enterprise applications it became almost a part of clean code best practices.

This level of abstraction, modularity and semantics separation was at first achieved using XML files. XML files are suitable thanks to their flexibility of writing a required schema, which we need to contain specific information without loss of descriptiveness. This approach is still widely used, but it has a drawback in necessity of keeping both implemented code and its XML description.

Annotations represent other approach, which allows us to define code specifics right in the code implementation. Thus there is no further need for searching proper component configuration, because all of its semantics is defined right next to the code itself. Annotations are in its raw form a special form of interfaces, which are load and separated using Java reflection technology. Also, annotation definitions are usually simple and do require much less text written for proper settings of component's behaviour than the XML alternative [21].

Using annotations becomes vastly used approach in modern applications and is recommended, while there are still cases, when XML definitions are either easier or provide other benefits over the annotations, due to the fact that annotations are still a rising technology. As examples of frameworks, where XML definitions coexists with annotations, we can point out Spring framework, Hibernate or JAXB.

Bearing in mind the benefits of annotations simplicity, while keeping the same information and behaviour as the XML definitions, is the annotation way of implementation used in this thesis, unless the XML variant would significantly benefit us over the annotations.

6.2 JAXB

XML manipulation is one of the most performed operations nowadays, more importantly due to web services, whose communication is basically built on XML technology. **Java Architecture for XML Binding** is a framework for simple manipulation with XML in Java and can be simply described as "ORM for XML". The framework does most of the XML heavy-lifting for user in a way that the user does not need to know anything about differences between SAX and DOM parsing [19].

With JAXB technology the user defines or specifies a XML schema, by which is performed so called **binding**. It means that for specified XML schema are generated Java classes, which are afterwards used for reading existing XML files in a process called *unmarshalling*. The opposite process, when we store initialized objects of binded classes is called *marshalling*.

Basic usage assumes a XML schema defined in a XSD file, for which are generated Java classes using JAXB libraries. The libraries are available either in JDK 6 and higher, or stand-alone for separate download and usage. Another option is to define the XML schema in a Java code by a developer using annotations.

Marshalling and unmarshalling is by default handled by a developer's code, but with technologies like Spring it is simple to make the framework handle the marshalling/unmarshalling process by itself. With Spring framework, annotations have become a powerful tool, which gives us option to handle XML files easily, while keeping the code clean with all the semantics in one place.

An example of creating and reading a XML file using JAXB is available in appendix A.

6.3 Apache Maven

Maintaining a software project is a significant part of the development process. A developer needs to create a project structure, take care of required library dependencies, deploy a created application and so on. Taking care of these operations manually could be difficult, but fortunately there is a way to automate them, like with Apache Maven [5].

Apache Maven depends on its **Project Object Model** (POM) stored in a pom.xml configuration file. It contains projects meta-information, like

a name and parent POM reference, library dependency definitions, plugin definitions, report settings, profile definitions and many more. Giving the fact that Maven is fully extensible using plugins, POM.xml content, ergo Maven build behaviour, depends only on used plugins and their configuration.

In addition to Maven's extensibility, it uses repositories for storing plugins and dependencies. It allows companies or individuals to create their own repository, in order to keep their own files private, or simply to prevent issues related to unavailability of a public repository.

A basic set of plugins required for Maven to run and a repository reference for downloading other dependencies and plugins are located in Maven's **Super POM**. Every user creating POM file inherits this POM, either directly or indirectly through a parent POM.

Moreover, Maven defines a clear on-disc directory structure. A developer can rely on Maven that he/she finds expected files inside of an expected directory location.

Apache Maven provides more features than mentioned directory structure and extensibility – another one worth to mention might be creating tags and branches in repository, uploading build application to a production server, etc. This technology has some drawbacks. The most annoying one is possible dependency collision, when Maven cannot decide which library from which dependency should be used. In such situation Maven build results in an error and user must resolve it. This is a known issue, which is continuously attempted to resolve – Maven 3 has better dependency resolving than Maven 2, or lately using an **Aether** library for dependency management, which should bring better behaviour than native implementation.

But even with its occasional dependency issues, Maven usage spreads in Java projects world-wide as it gradually becomes a standard in definition of keeping and maintaining larger Java projects.

6.3.1 Directory Structure

Every Maven-based project has a directory structure as it is visible in Figure 6.1, unless it is defined otherwise. Either IDE creates the directory structure, when we choose that we are going to create a new Maven-based project, or we must create the directories manually.

Source files are located in the **src** directory, resource files in the **res** directory and files used or generated during build are located in the **target** directory. The **pom.xml** file contains all the information required for performing build

and also contains meta-data about project.

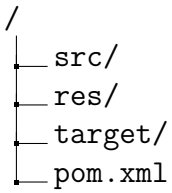


Figure 6.1: Maven Directory Structure

7 | Mobile Platforms

Before creating a mobile device application a platform must be chosen. While in computer software we can usually choose programming language, the mobile platforms are usually limited by a specific programming language or a framework, in which developers create their applications.

In this chapter all major platforms for mobile devices will be described with their specifics and drawbacks.

7.1 Windows Phone

Windows Phone is a Microsoft's proprietary mobile platform, which is based on Windows CE (Windows Phone 7.x, first release in 2010) and on modified Windows NT (Windows 8.x, first release in 2012) kernels. This platform provides truly distinguished user experience against other platforms due to its user interface based on *Modern Design Language* (previously called Metro), which is based on a tiles driven interface. Each tile contains a specific set of information, which is related to the application to which the tile belongs.

In terms of programming languages and frameworks, Windows Phone is built upon with Microsoft's .NET framework. User interface graphics is defined in special XML files called **XAML** files [15]. XAML is a language based upon XML and defines UI elements locations and their behaviour, using events and data bindings. This design language is a part of **Windows Presentation Foundation** module of .NET framework.

While the UI and UX basics lies in *Modern Design Language* and *Windows Presentation Foundation* as a rendering framework, in terms of programming Windows Phone platform uses .NET framework and C# as a programming language. C# allows to developers fully object oriented programming, while UI design definitions are located in separate XAML files.

Strengths:

- Free optimized IDE for WP application development (Visual Studio Express 2012 for Windows Phone)
- C# as a simple and powerful OOP language

- WP devices must comply with Microsoft HW specifics, ergo there are no weak devices and almost unified HW environment

Drawbacks:

- market share (currently below 2% of global market)
- proprietary – requirement of MS Windows for development (for Windows Phone 8.x development are required Windows 8)

7.2 iOS

All devices created by Apple Inc. are powered by iOS operating system. As well as Windows Phone the iOS is not an open source, but unlike WP is used only on Apple's devices – no other hardware manufacturer but Apple creates iOS powered mobile devices. Although iOS devices do not belong among the cheapest ones, thanks to them Apple nowadays competes with Google's Android operation system for majority in mobile devices market.

iOS is an operation system derived from OS X, which is an Apple's operation system for their computers and laptops. As a primary programming language for application development an **Objective-C** is used, in which is also created **Cocoa Touch** API [14] – an UI framework for building software applications for iOS platform.

Current devices are equipped with iOS 5 (released in 2011) and iOS 6 (released in 2012).

Strengths:

- Loyal user base – almost no piracy, users tend to buy applications
- Clear application guidelines – a developer can learn best practices from a reliable source
- Support – Apple provides thorough support in return to annual payments

Drawbacks:

- Necessity to comply to the application guidelines – if not conforming to them, application is refused to App store
- Annual payments – developer not willing to pay is not able to develop applications

7.3 BlackBerry

BlackBerry devices were for a long time a clear choice for a corporate segment of market. Their creator the BlackBerry company (former *Research in Motion, RIM*) have lost in past four years [20] many of its former customers due to far too long development process of new BlackBerry 10 OS. BlackBerry 10 OS was finally released in early 2013 and hopes to regain its slice of mobile device market.

A new BlackBerry platform provides the largest scale of available development languages and frameworks. A developer can use native C/C++ SDK with Qt for a user interface, or he/she can develop applications using Adobe AIR, HTML5 or even repackage existing Android application [1].

While BlackBerry now seems to be as a perfect choice for power-users and developers who want to try a new technology, its market share is so unsignificant that probably only the corporate segment can make Blackberry significant once more.

Strengths:

- Many ways to develop an application – C/C++ with Qt or QML, HTML5, Adobe AIR, ...
- Corporate customers – based on BlackBerry's past and sofisticated security could be expected, that the BlackBerry platform will be vastly used in the corporate segment once again. This strength could be applied to applications primarily developed for corporate environment.

Drawbacks:

- Market share – BlackBerry had significantly dropped its market share in past few years and BlackBerry 10 OS is a brand new platform
- Number of devices – only few devices using new BlackBerry OS so far

7.4 Android

Android is an open-sourced platform using Linux operation system, inside of which a Dalvik process virtual machine, a Java-based sandbox for Android applications, is ran. Currently it is with iOS one of the most used platforms, but while iOS devices are quite expensive and manufactured exclusively by

Apple, Android devices have multiple different manufacturers and cover price span from very cheap ones to as expensive as best Apple device.

Due to the fact of many manufacturers, Android stands and falls on support of various hardware and screen resolutions and depths [2]. In some cases, poorly HW equipped devices are equipped with new and more performance demanding Android versions, which may result in overall user's frustration over slow or even unusable device. It is caused by a fact that Google does not require minimum HW configuration for specific version of Android and it is only on manufacturer, how well-equipped device creates. Moreover, some manufacturers create their own user interface changes in order to distinguish their devices from other ones.

Since the version 4.0 (codename *Ice Cream Sandwich*) Android presented a new reworked API and design guidelines, which stands upon **Holo** theme. With new devices having Android 4.0 or newer and ongoing updating of older ones, there is decreasing necessity of supporting older versions of Android in order to benefit of features, which new API presents.

Similarly to Windows Phone, Android holds UI definitions separately to implementation in XML files. Android applications are written in Java (unless using C++ for native linux development) and then are compiled into class files transformed into **Dalvik executables** (dex files). Like Java packages compiled code into executable JAR files, Dalvik executables are packaged into APK files, which are used for distribution of Android applications.

Strengths:

- Independence of development machine OS – Android SDK is available for Windows, Mac OS and Linux equally.
- Documentation and community tutorials – Whole Android ecosystem is heavily maintained both by Google and developers community, which results in thorough documentation and countless How-To articles.
- Java – Android development is based on Java, one of the most spread programming languages.

Drawbacks:

- Fragmentation – API version fragmentation and HW fragmentation due to various devices; a programmer must test functionality and behaviour over larger span of devices or use an emulator.

- Android plugin for Eclipse IDE – while IntelliJ Idea in its new version offers an ideal development environment for Android, Eclipse IDE with its Android plugin is still recommended for development of Android applications. The problem is that an Android plugin does not work all the time as it should work.
- Cross-device inconsistency in user experience – due to different changes in the user interface per manufacturer, the user can be confused when using an other Android device, unlike in an other mobile platform.

7.5 Selection Decision

Following platform selection criteria were established:

- well documented API
- free SDK
- development platform independence
- large market share of mobile platform

Although iOS provides free SDK for universities, Objective-C is not a common programming language and considering the current price of any Apple mobile device it is not expected that an EEG researcher would be equipped with it.

For both Windows Phone and BlackBerry applies that although they represent an interesting piece of technology, their market share is not large enough to pay attention to them as a primary application platform. However, that may change in time, yet for Windows Phone there is still a limitation to the Microsoft Windows operation system.

Respectively for aforementioned issues of other platforms, *Android* was selected as a development platform. In addition, Android has the following benefits:

- Free – everyone can develop their application without payments
- No registration needed – the developer does not need to create any account to access the SDK
- Multiplatform development – Android SDK is available for Windows, Mac and Linux equally

- Market share – Android's market share is one of the largest and it still keeps growing
- Java-based applications – it is fairly simple to write an application,
- Documentation – many tutorials are available on Android Developers web as well as common JavaDoc documentation
- Community – There are numerous forums dedicated to Android
- Open source with support from Google – While Android is developed as an open source platform, Google organizes developers meet-ups and even maintains a YouTube channel dedicated to development on Android: Android Developers.

Moreover, Android of SDK v15 (4.0.x) was chosen for development since this version and higher ones represent majority in Android platform (see Figure 7.1 [6]).

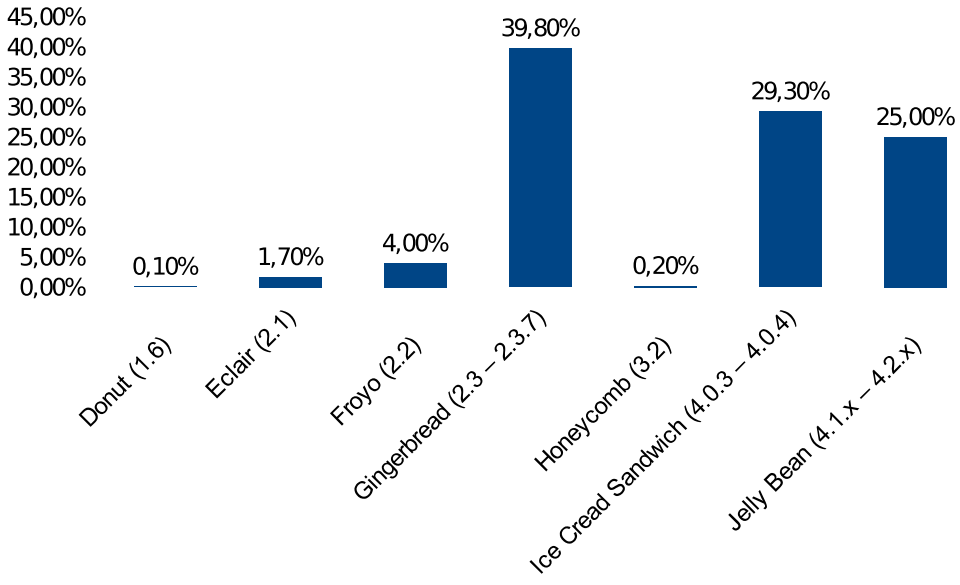


Figure 7.1: Android OS versions market share in April 4th, 2013

8 | Android Specifics

8.1 Resource Handling

The Android platform has a specific way of resolving resources in matter of which one will be used. The resources are separated into different resource directories by their purpose. The list of basic resource directories and their contents is following [11]:

- animator – XML files of property animations [10],
- anim – XML files of tween animations [13],
- color – specific colour behaviour for assigned view components,
- drawable – pictures,
- layout – layout definitions,
- menu – options/context/sub menu definitions,
- raw – files which are accessed using streams, usually binaries,
- values – files containing values used in an application. There is an existing naming convention for following files:
  - arrays.xml – value arrays,
  - colors.xml – colours values,
  - dimens.xml – dimension values,
  - styles.xml – styles definitions.
- xml – files, which are accessed using *Resources.getXML()* method.

Resources can be further distinguished using suffixes in resource directory name. The suffixes, which are separated by dash from one another, represent device's parameters or abilities – **qualifiers**. Using them we can create multiple resource directories, in which we put resource files which belong only to devices with a specific set of attributes. However, if no qualifier is used or

no resource is found in a directory with specific qualifier, a default directory with no qualifier defined is used as a fallback. A complete process of selecting proper resource is observable in Figure 8.1 and the list of available qualifiers is available on Android Developers site [12].

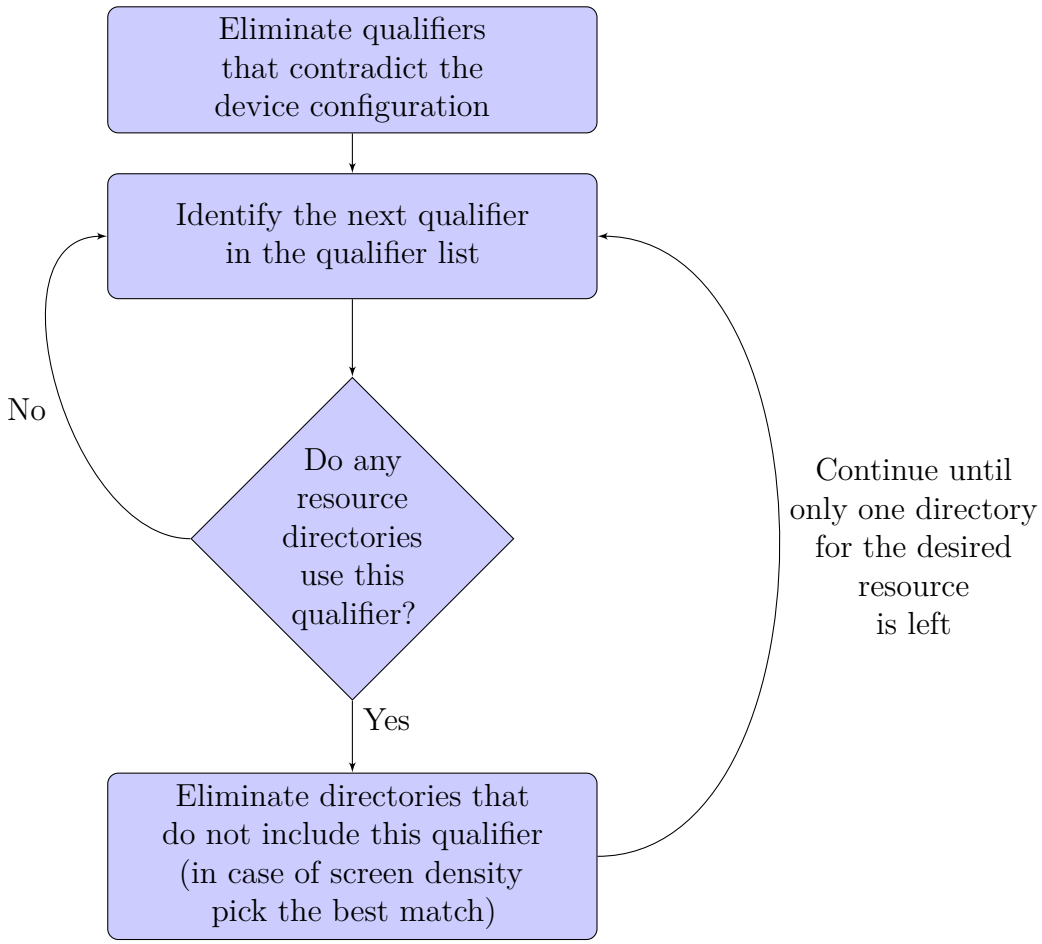


Figure 8.1: Resource selection flowchart [12]

8.1.1 Application Localization

When writing an Android application, it is most recommended to avoid using hard-coded strings and use Android's references to string resources instead. These resources are located in **values** directories in **strings.xml** files. The plural is used intentionally, because it can be easily distinguished which strings should be used by specifying locale qualifier in values directory name .

The locale is distinguished by a two-letter language code (ISO 639-1), which might be optionally followed by a dash and two-letter region code (ISO 3166-1-alpha-2), like *de* for German, *ja* for Japanese, etc. It means that texts in German should be located in resource file **values-de/strings.xml**. The same logic applies to other languages.

Each string in strings XML resource file is an application string resource. String of an application name could be stored like in Figure 8.2 and then referenced in a code by its name *app_name*.

```
1 <resources>
2     <string name="app_name">EEG Base</string>
3 </resources>
```

Figure 8.2: Example of strings.xml structure

8.1.2 Application Layouts

With Android devices supporting a large scale of devices with various screen resolutions and densities, developers must adapt their applications so they work correctly on all of them. Obviously, an adapting application layout is inevitable in order to provide acceptable user experience over such large amount of devices.

As well as with localization strings, there may be multiple **layout** directories with proper qualifiers, which allow us to provide desired user experience across various devices.

8.2 The Life Cycle

The foundation stone of Android applications are **Activities**, **Services** and **AsyncTasks**. Activities are primarily meant to display GUI and to interact with a user. Services and AsyncTasks provide parallel operations to a GUI thread (i.e. Activities).

Important is that all the components have their life cycles. The life cycle clearly defines, where a specific logic should be putted. Because of in this thesis we will not meet Services any further, let us focus on a life cycle of an Activity.

8.2.1 Activity

An Activity life cycle can be simply described as a phase of creation, productive life and its destruction. The creation and destruction phases consist of several methods, where each method has special purpose. All of the life cycle methods have their default behaviour, which can be altered. In order to alter the activity's behaviour, we need to override the default implementation – yet in most cases it is mandatory to call *super* equivalent of a given method, in order to all internally required operation would be performed.

The creation process is the life cycle part, where layout is inflated on a screen and filled with data, or simply reused from a previous run, when its instance is still available. The methods in this phase are:

- onCreate – Activity is newly created here. The layout is inflated and available data are filled into view elements. If there is a known previous state of this activity type, available information from the state can be reused,
- onRestart – This method is called upon an Activity, which was in the *stopped* state and now should be started again. Not used as often as *onCreate* method for overriding,
- onStart – the activity becomes visible, yet not interactive,
- onResume – the activity becomes interactive → we are in the state of *productive life*.

None of these methods can be interrupted, the logic inside of them is always performed. It is different from the destruction phase. Before further explanation, let us present a list of this phase's methods.

- onPause – another activity becomes interactive (resumed), a current activity must be paused, yet still visible. The current activity can become resumed again, or it can be stopped.
- onStop – by now the activity is not visible any more. The activity can be restarted or destroyed.
- onDestroy – the last step before the activity is destroyed.

While knowing the methods, let us go back to the matter of interruption. While only the method *onDestroy* explicitly destroys an Activity instance

and all of the methods should be performed, it may happen, that the activity must be killed, e.g. due to a sudden need of freeing memory. In such case only *onPause* method run is guaranteed. If there are any critical data to be stored, it should be done here. However, the activity is still visible in *onPause* method, therefore, in order to prevent slowness of user interface, no large data should be processed here.

Fragments

Fragments are components subsidiary to activities – they represent reusable components, which are used within activities. As Activities they have own life cycle, which is dependent on the parent activity. The life cycle has its methods, which can be overridden, as well as in activities.

Although Fragments are also used in this thesis application, it is not necessary to describe each method – their functionality is similar to activities' functionality and further details are available on Android Developers website[7].

8.2.2 AsyncTask

AsyncTasks are Android components for performing operations in parallel to a GUI thread. They require a reference to a visible activity, both for context reference and their ability to write into the GUI thread before and after their background processing.

The available methods of AsyncTask are:

- onPreExecute – operations which should be performed on GUI, before background processing starts,
- onProgressUpdate – operations performed, when progress is updated from *doInBackground* method,
- doInBackground – all the heavy lifting, which should be performed in parallel to the GUI thread, should be done here,
- onPostExecute – final steps before AsyncTasks ends. Method runs on GUI thread in order to reflect results into it.

The AsyncTask components are more suitable for short operations, which directly affect flow of application GUI. As an example of the situation suitable for an AsyncTask is fetching data, which should be displayed right after they

are available. The data fetching would be performed in parallel to the GUI thread so it would not freeze and would be updated by AsyncTask when the job is done.

Only drawback worth mention is that due to its reference towards currently visible activity, the AsyncTask must be handled when the activity is being destroyed, otherwise there will occur an error once the AsyncTask attempts to access the GUI thread.

8.2.3 Service

Services, like activities, hold own context, but they are running outside the GUI thread. They are primarily meant for performing operations independent of the rest of application's flow, like downloading of large files, which can run independently of the rest of the application.

Because Services are not used in this thesis application, its methods will not be further discussed.

8.3 Manifest File

An Android manifest is a file gathering essential information about an application through its properties and components. It is a specifically formatted XML file, into which developer declares an Android SDK version, permissions, used activities, services, content providers, etc.

It is necessary that developers define all possible information into the manifest. Using the Android Manifest devices check, whether they are capable to run the application and request proper rights from user. Also, without records about used Android application components (i.e. Activities, Services, Resource providers and Broadcast Receivers) application cannot run properly. An error is raised if application tries to access a component not mentioned in Manifest file.

9 | Architecture

In the following chapters we will focus on a server side and a client side implementation of the RESTful web service. But first, let us take a look on architecture of such implementation.

Data, which are needed on client side, are already available in the *EEG Base*; they only need to be handed over to the client on request. Therefore we implement controllers, which handle HTTP requests (Spring Controllers), and a component, which will fetch data from the existing data layer (Data Access Objects) into a transferable format and which will be available through the controllers (Spring Services).

All components on the server must use existing technologies. Using Spring IoC and MVC is an ideal choice, which also define its architecture, due to dependency injection.

On the client side we implement logic for client-server communication. Such logic will be present in *CommonServices* and will be used in *CommonActivities*, component representing user interface – both components will be described further in Chapter 15.

Communication between the server and client must be secured. Since we use RESTful web service, the communication channel will be the HTTPS protocol with SSL encryption.

A basic overview of components discussed is visible in Figure 9.1.

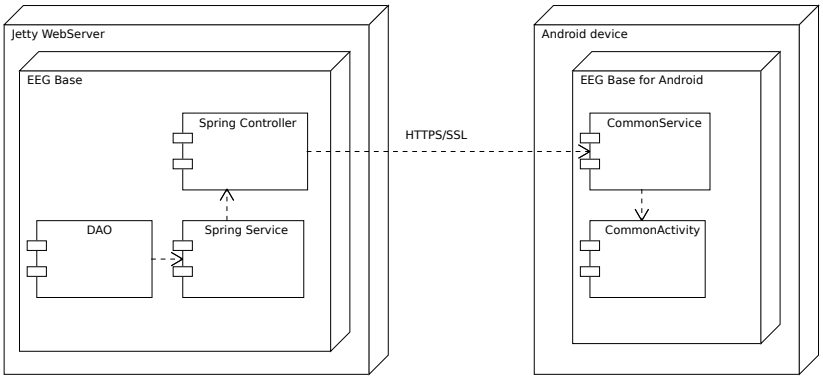


Figure 9.1: Deployment diagram

10 | Server Side: RESTful WS

Before we start any work on a client side, the server side of a web service must be implemented. Creation of a RESTful web service for Java EE application using Spring MVC and JAXB will be discussed in this chapter. A reader must be aware of Java EE basics in order to understand the following text.

For handling of a HTTP request we use a Spring MVC component of Spring framework.

10.1 Messages

Only rarely is returning or obtaining of primitive data types sufficient for a whole RESTful web service, unlike the complex types handling. In order to provide complex types from the server to the client and vice versa, data containers have to be transformed into XML or JSON format and be sent to the client.

In our application are used XML messages. The JAXB framework is used for *Java object* $\rightarrow$ *XML message* transformation. While we could reuse existing data containers of current implementation, we assume that the internal data containers might change in time in matter of adding or removing its properties.

In order to prevent the necessity of editing a client code every time this internal change occurs, we create border data containers, which contain only required properties and which are used specifically for our RESTful web service. Our JAXB annotated data containers are automatically marshalled and unmarshalled in Spring MVC's controllers every time a data container is received in request or sent via response.

In the EEG base, such data containers take place in the *wrappers* subpackage of its parent RESTful endpoint and their name ends with a **Data** suffix.

10.2 Requests and Responses

As in a Java Enterprise web application, we need to define a servlet, which handles incoming requests and outgoing responses.

For creating a web service endpoint itself is recommended to create a controller class, service interface and service implementation. Using controller we handle HTTP requests using service, which is autowired into the controller using dependency injection by its interface. Separation of service implementation and its controller is recommended due to better scope handling of transactions, when operating with the data layer.

10.2.1 Servlet Definition

A servlet name and its mapping must be defined in **web.xml** at the beginning. Let us create a servlet named *rest*.

```
<servlet>
  <servlet-name>rest</servlet-name>
  <servlet-class>
   org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>rest</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Listing 1: Servlet definition in web.xml

Having this servlet definition we create a new servlet file in the *WEB-INF* directory called **rest-servlet.xml**. Inside of it we create the following beans:

- *org.springframework.web.servlet.view.ContentNegotiatingViewResolver*
- *org.springframework.web.servlet.view.xml.MarshallingView*
- *org.springframework.oxm.jaxb.Jaxb2Marshaller*
- *org.springframework.web.multipart.commons.CommonsMultipartResolver*

ContentNegotiatingViewResolver resolves view requests by its content type. In bean's body we can define multiple view implementations into **defaultViews** property list. In our case we define xml view implementation, which we also set as default by setting application/xml as default content type. It means that unless the content type is different from aplication/xml, a XML response is returned.

View implementation for XML is represented by **MarshallingView**. It uses internally a JAXB marshaller **Jaxb2Marshaller**. Inside the JAXB marshaller bean we define all marshallable data containers into a list of property named **classesToBeBound**. By enumerating the data containers we tell Spring, that instances of such classes should be automatically marshalled and unmarshalled without the need of using **@ResponseBody** annotations.

By now, we need to allow Spring MVC annotations and define package, in which are present annotated classes. We achieve it using the following lines.

```
1 <mvc:annotation-driven/>
2 <context:component-scan
3         base-package="cz.zcu.kiv.eegdatabase.webservices.rest"/>
```

Listing 2: Enabling MVC annotations in cz.zcu.kiv.eegdatabase.webservices.rest package

10.2.2 Controller

Spring MVC provides controllers, special components meant for handling HTTP requests. Such components can be created either using XML definitions and by extending a proper abstract class, or using **@Controller** annotations. Controlling HTTP requests we can cover all RESTful web service requirements.

In order to control HTTP requests, we need to create mapping over specific URL and handle it by controller's method. The **@RequestMapping** annotation over a method makes the method listen on the URL specified in annotations *value* property. By default such mapping applies to GET HTTP requests, if we desire to handle other HTTP methods, we need to specify it by *method* property of the annotation.

Mapping's URL is specified by a string which appends to previous mappings. Imagine that we request mapping over the whole class with value **/experiments** and then again we request mapping **/mine** over method, which returns a marshallable container of user's experiments. A GET HTTP request over **/experiments/mine** would trigger the method and a container of user's experiments would be marshalled into a response.

URL mapping string can also contain parameters, which serve as input parameters for a linked method. Every parameter must be named the same as the method's input parameter a must be wrapped inside **{}** brackets. If we

had method annotated as in Listing 3 and a GET request upon URL ending with */experiments/public/15* suffix, the method **getPublicExperiments** would process such request and would pass integer 15 as input parameter.

```
@RequestParam("/experiments/public/{count}")
public ExperimentDataList getPublicExperiments(int count)
{ // method implementation }
```

Listing 3: Request mapping with an input parameter in a request URL

All method input parameters might not be a part of request URL string, they may be a part of a request body or parameter. It applies mostly to POST requests, where we extract parameters using **@RequestParam** annotation before an input variable of corresponding data type. For transforming whole request body into a complex data type we must use **@RequestBody** annotation.

During web service processing an error may occur, an exception is thrown. Every exception should be intercepted and handled in a way, that user is notified in a proper way. **@ExceptionHandler** annotation serves for exception interception and its value is set to an exception type class. This annotation is being placed over the method which should handle the exception in some way. In our web service we send error message through a servlet response. The error message has its HTTP Status set to a proper value and an exception string as a message body.

```
@ExceptionHandler(RestServiceException.class)
public void handleRSException(RestServiceException ex,
  HttpServletResponse response) throws IOException {
    response.sendError(
  HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
  ex.getMessage());
}
```

Listing 4: Exception handling in web service controller

10.2.3 Service

Service is another Spring component, which is created inside the container. It provides a specific set of functionality, which is provided to other components which use it.

In our case, we define service's methods in an interface, using which we recover implementation bean for autowiring, and an interface implementation, which will be annotated using **@Service**, so that Spring would create its bean.

The services contain logic of web service, which is not related to HTTP request handling, but to the behaviour itself. It means that we create new records in it, fill marshallable data containers with proper data, return them, etc. Another point of view is that our service is an additional layer between controller and data layer, which only returns data in a marshallable format.

With data access separated from controller we can easily set data transaction scopes. These are set using **@Transactional** annotations. With them we wrap method execution with database session, so that data which yet have not been fetched due to lazy loading would be fetched right in time. Transactional session can be either with possibility to edit database content, or for read only. It is set using annotation's *readOnly* parameter. It also is specific for Transactional annotation, that it must annotate a method in implementation and not in an interface, as it is usual.

For being able to use Transaction annotations, a transaction manager must be already created and reference to it must be set in the servlet definition like in the following line:

```
1 <tx:annotation-driven transaction-manager="txManager"/>
```

Listing 5: Allowing transactions annotations

10.3 Security

When creating a web service, two security issues occur. The first issue is that a user should be able to access only data to which he/she has access rights. The second issue is a communication channel – data are sent over not secured http channel by default and there is a risk that someone would read the communication and obtain private information.

10.3.1 User Verification

A perfect solution for the first issue is the Spring Security framework. It allows to create a semi-session for each incoming request and destroys it when the request is handled. It would be probably the most needed feature

for RESTful web services, where is no need to create and keep a classic session due to web service stateless architecture. For user authentication itself it is only needed that incoming request had an Authentication HTTP request header set.

Another benefit of Spring security is its support of annotations. Applying restrictions in accordance to user's rights over a method or class was never easier – all that is necessary is to put **@Secured** annotation over the class or method and put the following line into servlet definition, otherwise annotations won't be used:

```
1 <security:global-method-security secured-annotations="enabled"/>
```

Listing 6: Allowing Spring Security annotations

The Spring Security framework itself must be configured for the application to run properly, but its configuration is not a matter of this thesis.

10.3.2 HTTPS and SSL

A Java EE application can automatically switch communication to a secured channel, if server allows it. For the Tomcat server it means to create a certificate (or use an existing one) and assign it to a secured socket which is used for HTTPS communication afterwards – the setup process is described in the Tomcat server manual.

In web application, it is necessary to define a security constraint over a URL pattern within the web.xml as in Listing 7:

```
1 <security-constraint>
2   <web-resource-collection>
3     <web-resource-name>Spring REST SSL</web-resource-name>
4     <url-pattern>/rest/*</url-pattern>
5   </web-resource-collection>
6   <user-data-constraint>
7     <transport-guarantee>CONFIDENTIAL</transport-guarantee>
8   </user-data-constraint>
9 </security-constraint>
```

Listing 7: Forcing secure channel for RESTful web services in web.xml

With these settings, each time a request to *http://app-domain/rest/** is created, it will be redirected to *https://app-domain:secured-port/rest/**. It is thanks to a **transport-guarantee** parameter, which enforces usage of protected transport layer connection (HTTPS), and its *CONFIDENTIAL* parameter, which requires communication encryption [18].

11 | Client side I: Creating a New Project

Throughout this chapter we will focus on creating an Android application and on implementation parts, which are generally applicable to other applications. However, the examples will be applied to the **EEG Base for Android**, which is official name for the Android-based client application (on device named as the *EEG Database*), which was created for resolving needs mentioned in Chapter 4.

11.1 Creating Android Application Using Apache Maven

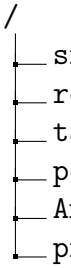


Figure 11.1: Android project on-disc file structure

Using Maven for defining a project provides us many features like mentioned in Section 6.3, while the most beneficial ones for us are *clear directory structure*, *automated library dependency resolving* and *automated deployment* of application itself.

With Maven we need an on-disc file structure like displayed in Figure 11.1. Such structure can be achieved either manually or using IDE, which is recommended.

The next important step is to clarify the following aspects of the current application:

- **Minimal SDK version** – a minimal required version of Android SDK to run is **15**, which represents Android 4.0.x,
- **Application package**, in which are located source classes. This should not be changed, even more when the application is available on Google

Play, where the package name serves as an application identifier,

- **Required libraries** – in our application we will use the **RestTemplate** library from the *Spring for Android* framework, in order to simplify RESTful web service creation. Also a library for XML marshalling will be necessary – because Android does not provide JAXB as in Java, the **Simple XML** framework is used, since it provides almost identical behaviour while using annotations like in JAXB (see Appendix B).

11.2 Project's Maven Settings

Let us set the project using Maven, so it would resolve dependencies, build and deploy an android application into device. Such settings are defined in **pom.xml** file.

A **groupId** will be identical to aforementioned *application package*, **artifactId** will be application name in lower case with dots as word separators, e.g. *eeg.base*. Since we are creating an Android application, the *packaging* will be an Android application package **apk**.

Now we must define framework dependencies. On the Maven repository website[1] we search for required libraries and copy a dependency snippet between *dependencies* pair tags. Like this we find and define Spring for Android RestTemplate and Simple XML framework. Moreover we must define Android SDK dependency, so Maven would connect the SDK with a compiler. It is done in a similar way as other dependencies, but this one has parameter **scope** which is set to **provided** and it looks for a system environment variable of *ANDROID_HOME*. The *ANDROID_HOME* variable should be set as a path to the location of Android SDK installation.

The last step will be setting up Maven build plugins, which will compile and package our application. For compiling the Java code we use **Maven Compiler Plugin** and in order to compile the Android code and package it into an APK, we must define plugin dependency to **Android Maven Plugin**. This plugin also allows project to link other APK files as library dependencies, which would be impossible without it. The only drawback of the Android Maven Plugin is that it requires Maven at least in version 3.

With all these items set, we can build our application and deploy it directly to our connected device with the following command:

[1] `http://mvnrepository.com/`

```
mvn clean package android:deploy android:run
```

The *clear* command clears the workspace from previously compiled files, the *package* will create the APK file, the *android:deploy* installs the application into a connected device and finally the *android:run* starts the application in the device.

A prerequisite to successful application deploy into an Android device is to have enabled the *USB debugging* in the *Developer Options* section of system settings on device. Also note that Android currently supports only Java 1.6 (which should be set in Maven Compiler Plugin) and that our application will require SDK at version higher than 15 inclusive (belongs to Android dependency settings).

As an example may serve POM file of *EEG Base for Android*, which is on attached DVD.

11.3 Android Manifest

In the Android Manifest we need to set the *package name* and *application version* as first ones of many parameters. The package name is identical to package, where the source classes are located in, application version is set as pleased.

The next important thing is to set SDK reference. The reference is set through a *minimum SDK version* value of 15, with possible higher value as *target SDK version*. It is generally recommended to set target version up to the newest version, but it is not mandatory.

Since our application will server as a client for a RESTful web service, we need a connection to the Internet. In order to provide such connection to our application, we must define the following *user permissions*:

- android.permission.INTERNET – provides us internet connection in general, if it is available,
- android.permission.ACCESS_NETWORK_STATE – allows us to check current connection state, which is good to know before initiating an web service request (there is no point in creating a request if there is no existing network connection).

If we would need other permissions as well, the Android manifest is the place, where all permissions are defined.

Another manifest's block, the *application* block, describes the application with its icon, label, global theme and its components. In our application we use **Holo** theme, which is a new standard theme since Android 4.0.x (ICS). Also, since our application does not use content providers, services or broadcast receivers, only activities will be placed into the application's block body.

The application itself must have an entry point – an activity, which is started as a first component of launching application. It is set using an *intent-filter* settings block of an activity. In this block are specified an *action* (MAIN) and a *category* of such intent (LAUNCHER).

While all components inherit a theme and UI options settings from its parent, i.e. the application block, these settings may be changed in each component. That is important for components, whose behaviour should be different from other components. Let us present some examples, which are also used in the *EEG Base for Android*:

- android:theme="@android:style/Theme.NoDisplay"
  Activity with this parameter will not display any layout. Such behaviour is desirable in an application entry point, where we check, whether the user is logged in (so we let him/her further into application) or we enforce user's log in.

- android:theme="@android:style/Theme.Holo.NoActionBar"
  ActionBar is a new central place for putting application's controls since Android ICS. But in some cases it is required to the ActionBar would not be displayed. This parameter provides such behaviour.

- android:uiOptions="splitActionBarWhenNarrow"
  There is a difference between layouts for hand-held devices and tablets. Even though custom layouts provide most of the customization, sometimes is desirable that ActionBar's controls display in separated menu bar in the bottom of the screen. Such behaviour is achieved using this parameter.

For further implementation details of manifest used in the *EEG Base for Android* head for the manifest file located in project's directory on attached DVD.

12 | Client side II: Working with Data

12.1 SharedPreferences: Storing User Data

In many applications, there is a need of storing basic information like username or password. While such information might be stored in a database, we do not need a database at all. The *Shared Preferences* are useful way to store such information without implementing additional logic, which is required for working with database.

In our application we use Shared Preferences for aforementioned storing of credentials and information required to connecting to the server. There may be multiple Shared Preferences, which are distinguished by a string identifier/name and can be accessed with different modes of Android context, through which the Preferences are available.

The obtained *SharedPreferences* are a decorated Map with own Editor used for changing its content and with a change listener, used for an automatic update of dependent components.

12.2 ListView and Complex Data Types

Android provides a basic set of View elements, which can be used. They can be both created and manipulated in a source code and defined in layout XML and accessed in a code afterwards [8]. But what happens if default component's behaviour does not provide functionality we require? Sometimes it is needed to write own component, but often there is a simpler way.

A **ListView** is the most common representative of such components. When we have data in collection, we would like to display them in a component meant for such purposes – a list view. A ListView by default displays only a string per view item, which is not enough for reasonable display of a complex type record. In order to change the way the ListView displays related data, we must implement own *Adapter*.

There are multiple provided implementations of Adapters and in our case we use an **ArrayAdapter** implementation. We must create a custom class, which uses the ArrayAdapter as a parent and inside of class implementation we override desired methods, which affect the Adapters behaviour.

The most likely methods to be overridden are methods *getView* and *getDropDownView*. The first one is used for usual displaying all items into view elements visible at activity start, like the ListView, the second one is used in view elements using drop down behaviour, like Spinners. Since both methods have identical parameters, their behaviour can be covered by a common method, which initializes view element with requested data and returns it.

The initializing method reads data from a provided data collection by requested position, then initializes a new element's view, sets the data to it and returns the prepared element view. The common approach for creating an Adapter is creating a reference to a proper row layout, which is inflated and then set with data in an initializing method.

Any Adapter in the *EEG Base for Android* application can serve as an example. The Adapter classes are located in a package named *cz.zcu.kiv.eeg.mobile.base.data.adapter*. In case we also want our Adapter implementation to be filterable, we must implement a **Filterable** interface. This interface is more usually used over **CursorAdapter**, which uses DB, but it is possible to use it over ArrayAdapter too – only it is necessary to override more methods, keep reference to original data and properly notify about changes. For more details see **ExperimentAdapter** in the *EEG Base for Android*.

12.3 Providing Data Between Activities

Passing a primitive data types to another activity is fairly simple – we obtain/create a **Bundle** (a container for data, working like maps), into which we put the data, set new Intent extras and create the Intent. A new Activity in its *onCreate* method extracts the extras Bundle from saved instance Bundle and uses them. Retrieving data from an ending activity, which was started by the method *startActivityForResult*, is similar. There we override the method *onActivityResult*, where we retrieve the extra Bundle, which is provided by the ending activity.

There is a issue with the complex data types – the Bundles are the only efficient way for passing data, but they do require a **Serializable** or a **Parcelable** interface implemented.

Even though the *Serializable* implementation is more closer to many Java developers, we will present here an implementation example for the *Parcelable* interface. The *Parcelable* interface was designed specially for Android and should be more efficient.

In an parcelable class a developer must implement its own serializing, during which we work with a **Parcel** instance, which works similar to a Bundle. The important thing is to keep the same order of reading from a Parcel instance as it was in case of writing to the Parcel; otherwise the serializing will not work.

Also, in order to keep Android designs, we must create a public static **Parcelable.Creator**, which is used for recreation of the data contained in other parcelable containers.

An example can be seen in Appendix B.

13 | Client Side III: RestTemplate

13.1 Communication Using HTTP/SSL

In order to provide secure connection with server, the HTTPS protocol must be used with SSL encryption. But once such combination is used, a client application must be edited, so that it would be capable to communicate with the server over the secured communication channel.

Our application will discuss the case, when the application will accept all certificates used for communication. The reason for such implementation might be various – from creating a functional prototype, to simply not having an official certificate for communication encryption, while this should be a temporary solution, like it is in the case of the EEG Base.

While many examples use in RestTemplate for the communication with the server a *CommonsClientHttpRequestFactory* class, this class has a huge drawback – it cannot send a file without buffering it into memory first. In case of large files it leads to out of memory errors, which makes this class not usable for our web service client.

Instead we create own HTTP request factory class, which will accept all SSL certificates and will be also useful for RestTemplate. It will extend a **SimpleClientHttpRequestFactory** class from the Spring Framework http client package[1]. An instance of this class can disable the content buffering prior send by setting a *false* value into *setBufferRequestBody* method. This class should be available in project since we have a dependency of the Rest Template defined in project's POM file.

In this new class we have to override the method **openConnection**, which will decorate the parent's method by setting the HostnameVerifier instance, which accepts all hosts (simple interface implementation, which returns *true* in its only method), and setting a custom SSL Socket factory to a HttpUrlConnection instance, which was first casted to a HttpsUrlConnection instance.

The custom SSL Socket factory is obtained by a getter from a SSLContext TLS instance, which was initialized with parameters:

[1] cz.zcu.kiv.eeg.mobile.base.ws.ssl.SSLSimpleClientHttpRequestFactory

- KeyManager[] – null,
- TrustManager[] – implementation returning a **X509TrustManager** instance. The methods of such trust manager are empty, only the **X509Certificate** array getter creates a new empty array of such data type,
- SecureRandom – null.

The RestTemplate using this custom client http request factory will accept all SSL certificates over HTTPS protocol.

13.2 Direct Usage of RestTemplate

As well as in the EEG Base for Android, we will presume, that the RESTful web service on the server authorizes requests, which do have set a HTTP header parameter of HTTP Basic Authentication. It is sufficient in case, that the server uses Spring Security. Let us present an example of a remote creation of new user, which will represent the approach for client-server communication in general – other cases are very similar.

First we need to define a request's HTTP header. The header is represented by **HttpHeaders** instance, and in it we must define the following HTTP header parameters:

- *Authentication*: Here we use HTTP Basic Authentication by creating a **HttpBasicAuthentication** instance. Into this instance are set username and password,
- *Acceptable response content types*: A collection of acceptable response types belongs here. Since our web service communicates only using XML messages, we put here a list with only item of *APPLICATION_XML* string,
- *Request content type*: As well as for response we must define of what content type is our request. In this example we create a new user, which is represented by XML message. *APPLICATION_XML* string belongs here.

Second, we create the whole HTTP message entity. The **HttpEntity** object constructor (the object is here typed as a *Person* instance) consists of a Person instance itself and previously created HTTP header instance.

With existing HTTP entity, we only need to create a new **RestTemplate** instance and use it for performing a request. A constructor parameter for the RestTemplate would be our custom client HTTP request factory from Section 13.1. And in order to RestTemplate marshall a Person instance into a XML message, we must add a XML message converter – let us use a **SimpleXmlHttpMessageConverter** class for this purpose.

The last step is client-server interaction itself. The RestTemplate offers many HTTP methods – in this case we need a POST method, which is commonly used for a new record creation. Moreover, we expected except OK message information about newly created user in response, therefore we use a *postForObject* method, which has parameters of URL (where to perform a request), HTTP entity (with previously set header fields) and a expected response object class (in our case Person.class).

The whole example looks in code like in the following snippet:

```
1  //assume initialized instance of Person class named "person"
2  HttpAuthentication authHeader
3           = new HttpBasicAuthentication(username, password);
4  HttpHeaders requestHeaders = new HttpHeaders();
5  requestHeaders.setAuthorization(authHeader);
6  requestHeaders.setAccept(
7          Collections.singletonList(MediaType.APPLICATION_XML));
8  requestHeaders.setContentType(MediaType.APPLICATION_XML);
9
10 SSLSimpleClientHttpRequestFactory factory
11         = new SSLSimpleClientHttpRequestFactory();
12 RestTemplate restTemplate = new RestTemplate(factory);
13 restTemplate.getMessageConverters().add(
14         new SimpleXmlHttpMessageConverter());
15
16 HttpEntity<Person> entity
17         = new HttpEntity<Person>(person, requestHeaders);
18 Person response =
19         restTemplate.postForObject(url, entity, Person.class);
```

Listing 8: RestTemplate example

14 | Client side IV: Navigation

Using a proper navigation is a crucial part of every application. In previous versions of Android were key navigation elements located in options menu or they were present as view elements like buttons and hardware keys. Our application, as well as *EEG Base for Android*, is built upon Android SDK newer than Ice Cream Sandwich, so our focus must be targeted on the ActionBar, which now handles most of navigation within the application.

14.1 Option Menus

There are two possible ways to invoke options menu – by a hardware button, or by clicking on menu icon on the ActionBar. While the menu icon may not be visible on devices with hardware menu button, on devices without it is visible every time a menu item is set into options menu.

The options menu can be defined in two ways: directly in a code, or by creating a *menu* resource XML file, with menu items defined in it [9]. Both activities and fragments can present own options menu items, which are merged into one common menu.

All items are hidden in menu by default. If we want to see them on the ActionBar, we must set the item's *showAsAction* attribute. Possible values for this attribute are

- *always*: will be visible under all circumstances, recommended max. for 2 menu items,
- *ifRoom*: visible on the ActionBar only if there is a free space,
- *withText*: shows item's title even if there is an icon set,
- *never*: item will be always inside the options menu and not in the ActionBar,
- *collapseActionView*: when a menu item is clicked on, its view expands all over the ActionBar. This behaviour is quite useful for implementing a search text box, like in **ListAllExperimentsFragment** class in the *EEG Base for Android*.

Now let us see, how implement option menus in activities and fragments.

14.1.1 Activities

Implementing option menus in activities is fairly simple: the only things need to do are inflating the menu and handling item selection events.

The menu is being inflated in the method *onCreateOptionsMenu*. Here we obtain a *MenuInflater* instance and inflate proper menu XML resource into provided menu instance. It is crucial to return *true* after the job is done, otherwise activity will ignore our menu settings.

```
1 @Override
2 public boolean onCreateOptionsMenu(Menu menu) {
3     super.onCreateOptionsMenu(menu);
4     MenuInflater inflater = getMenuInflater();
5     inflater.inflate(R.menu.exp_menu, menu);
6     return true;
7 }
```

Listing 9: Example of inflating menu resource in ExperimentActivity

Now we have an inflated menu, which has its menu items. Each menu item has its identifier, which is used in event handling method for event source recognition. The event handling method is called *onOptionsItemSelected*. In order to handle events over our menu resource, we must override it and alter its behaviour.

```
1  @Override
2  public boolean onOptionsItemSelected(MenuItem item) {
3      switch (item.getItemId()) {
4          case android.R.id.home:
5              finish();
6              break;
7          //other cases distinguished by identifiers
8      }
9      return super.onOptionsItemSelected(item);
10 }
```

Listing 10: Example of handling menu items' events

Note the *android.R.id.home* identifier. This is an identifier for a special

button, a *home* button, located instead of activity's icon, which has the icon as its contents but right next to it is a "<" character. This button is meant for going up in the logical hierarchy of screens. Imagine the situation, when you were in a list view of records and you clicked on record. A details activity was displayed, but you clicked on a button, which took you to "page 2". The back button would take you to "page 1", but the home button, with a proper implementation, would take you back to the list view of records.

In order to use home button, you must edit behaviour of the ActionBar in activity's *onCreate* method, as in Listing 11.

```
1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3    super.onCreate(savedInstanceState);
4    getActionBar().setDisplayHomeAsUpEnabled(true);
5    }
6 }
```

Listing 11: Turning icon into Home/Up button

14.1.2 Fragments

The situation is a bit different in fragments' implementation, since an activity is superior to a fragment. Fragments have no options menu by default, so we must inform a parent activity, that our fragment provides it. It is performed in the fragment's *onCreate* method, by passing a *true* value to *setHasOptionsMenu* method.

MenuItem events are handled in the same way as in activities, but do not forget to invoke a super method, since the event must be passed to an activity if the menu item identifier was not recognized!

Inflating menu is also a bit different from the approach used in activities. The *onCreateOptionsMenu* method already provides a menu inflater instance as well the menu instance itself, so the only thing necessary is to inflate a menu resource and to invoke the super method.

14.2 Tabs Navigation

Tabs are one of the most common navigation elements in Android applications. They are used to separate content by their categories and to make orientation easier. The tabs are a part of an ActionBar and are displayed either inside of it, if the screen is large enough, or right below it.

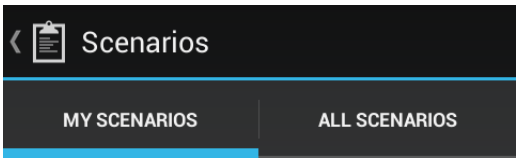


Figure 14.1: Tabs as a navigation menu in the EEG Base for Android

In order to provide a navigation with tabs, we need to alter ActionBars' behaviour and create an implementation of an **ActionBar.TabListener** interface. The custom TabListener resolves what happens, when a tab is selected (usually a fragment is changed), unselected (fragment is detached) or reselected (usually nothing happens)[1].

The ActionBar behaviour change is quite straightforward – we only need to set another navigation mode, create tabs and add them to the ActionBar.

```
1  @Override
2      public void onCreate(Bundle savedInstanceState) {
3          super.onCreate(savedInstanceState);
4          ActionBar actionBar = getActionBar();
5          actionBar.setNavigationMode(
6                  ActionBar.NAVIGATION_MODE_TABS);
7          ActionBar.Tab tab = actionBar.newTab()
8                  .setText(R.string.string_identifier)
9                  .setTabListener(tabListenerInstance);
10         actionBar.addTab(tab);
11     }
12 }
```

Listing 12: Single tab example of ActionBar with tabs navigation mode

[1] see *cz.zcu.kiv.eeg.mobile.base.ui.TabListener*

14.3 Spinner Navigation

Another choice of navigation might be a spinner instead of an activity name. An implementation of such navigation is a bit more difficult than the tabs mode, but on the other hand, it is more useful for referencing activities as well as fragments. Unfortunately, currently there is no straightforward way to support both spinner and tabs mode in one activity at once.

If we want to have a spinner as a menu in an ActionBar, we must set it to a *list navigation mode*. Also we must implement a **OnNavigationListener**, so we could handle menu item selection events. The last thing we need to do is to create a **SpinnerAdapter**, which we set together with the navigation listener to the ActionBar.

A working example of such implementation is used in the *EEG Base for Android*, specifically in the **NavigationActivity**[2] class. A custom array adapter called **MenuAdapter** is created in the NavigationActivity and it needs a reference to an array of strings, array of icon references and a list item layout identifier. The result is visible in Figure 14.2.

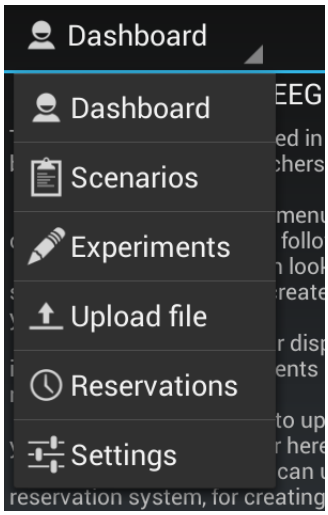


Figure 14.2: Spinner navigation menu in EEG Base for Android

[2] cz.zcu.kiv.eeg.mobile.base.ui.NavigationActivity

15 | Client Side V: EEG Base for Android Specifics

Implementation details are too specific to the *EEG Base for Android* since this chapter to be described as a generally recommended approach to develop a RESTful web service client. This chapter is dedicated to implementation specifics of the *EEG Base for Android*.

Parts of the final application in relation to their logic and packages will be presented in following sections.

15.1 Resources

The *EEG Base for Android* uses default a resource structure as it is discussed in Section 8.1.

15.1.1 Application Localization

The application supports English and Czech languages. There are two *values* directories, one without a language qualifier and the second one with a *cs* qualifier. *strings.xml* files, which contain string values, are in both of them as well as *arrays.xml*, which contains string arrays in proper language.

15.1.2 Application Layouts

Tablets and mobile phones have different layouts of different sizes in order to keep information readable and the user experience intact. As a threshold for distinguishing where to use mobile or tablet layout a minimal width qualifier of 600dp was used. It is usually a width of 7" tablets, like Nexus 7.

Portrait and landscape alternatives are provided as well. In the case of large layouts, like layout for adding a new experiment, layouts were separated into partial files, which are included into main layout using the *include* tag in layout's XML file. Including sub-layouts benefits us of possibility to present

different user experience across various devices by creating different sub-layout per different device, like in mentioned phones or tablets.

Icons of different size for various screen densities are also provided, since it is a standard in an Android application development. A website tool commonly recognized among Android Developers — an **Android Asset Studio**[1] — was used for their creation.

15.2 Archetypes: Project Specifics

In the *EEG Base for Androids* two main archetypes are used:
CommonActivity and **CommonService**. These two classes are decorated *Activity* and *AsyncTask*. The CommonService is used for parallel jobs of CommonActivity and both archetypes were primarily created for easy status change from a AsyncTask to a related Activity. Also, by the time Activity gets destroyed (e.g. screen orientation change) existing AsyncTasks may still be running. AsyncTasks usually have its ProgressDialog, using which is user informed that the application is currently working. In order to prevent window leaks, the alert dialog must be destroyed and recreated with a reference to a new Activity.

The CommonActivity and CommonService resolve this issue by using
the **ServiceReference** class, which works as a FIFO queue of references to CommonServices and their messages to ProgressDialog. Each time an instance of CommonActivity is created, it checks if there is any reference in the FIFO. If there are any references, all CommonServices in FIFO are set with new CommonActivity reference for context and a new ProgressDialog is created with a proper message.

Dismissing a progress dialog is also handled. If a CommonActivity gets paused, the dialog is dismissed. If CommonService informs the Activity that the job is done, the reference is removed and the dialog is closed.

The CommonService has in also a methods for simple message passing to
a CommonActivity and a method, for passing an Throwable instance. Such throwable is analysed, and if it is of known type related to common issues, only a text message is passed forward. This is used for handling errors during client-server communication, where exceptions provide only codes or hardly readable messages.

There is only one limitation to CommonServices – they must be executed

[1] `http://j.mp/androidassetstudio`

serially in order to prevent issues with progress dialog. It is achieved by executing on an Executor pool with a *AsyncTask.SERIAL_EXECUTOR* parameter.

Both CommonActivities and ComonServices decorate their parent classes by simple access to shared preferences which contain user's credentials.

The last archetype is a **SaveDiscardActivity**. This activity inherits from a CommonActivity and is meant for views, where accepting and storing of displayed data or dismissing them are the only options available. This activity and its style are based on **Done+Discard** Android UI design pattern[16].

The package of archetypes is *cz.zcu.kiv.eeg.mobile.base.archetypes*.

15.3 Data Layer

All classes which are meant for containing and providing data are placed in the *cz.zcu.kiv.eeg.mobile.base.data* package. Only two classes are in the package root – **ServiceState**, which holds enumeration of possible CommonService states, and **Values**, which serves as a container of various constants and values, like RESTful web service endpoints, flags and keys for activity results etc. Other classes are in the following sub-packages.

15.3.1 Container

Simple data containers were created in order to contain available data.
The containers are *Parcelable* and they can be used for XML marshalling and unmarshalling in RestTemplate thanks to Simple Framework annotations. The matter of Parcelable was discussed in Section 12.3 and an example of such parcelable and marshallable container can be seen in Appendix B. Such containers are created for all required data types, e.g. artifacts, experiments, electrode locations, etc.

The mentioned containers are placed into a *xml* sub-package, since there is an exception in **FileInfo** container. This class is a simple decoration of a Java File class, which provides file's size string in human readable form, and it is used in application's file system browser. The FileInfo class is not marshallable nor parcelable, since it is limited to local files and the objects are not passed to Bundles.

15.3.2 Adapter

Since the *EEG Base for Android* requires a custom behaviour of spinners, list views and other similar components, there must be created custom ArrayAdapters as well. These adapters are located in the *adapter* data sub-package and provide view elements in accordance to provided data containers and related row layouts, as discussed in Section 12.2.

15.4 Web Service Communication

All the knowledge about RestTemplate and SSL connection from Chapter 13 is used in *CommonService* implementations, which are located in the *cz.zcu.kiv.eeg.mobile.base.ws.asynctask* package.

Each CommonService implementation's name consists of an action prefix, which tells us what a service does (like *create* or *fetch*), and the related data container. It means, that **CreateWeather** class serves for creating a new Weather record on the server, while the **FetchArtifacts** class retrieves from the server a list of all existing artifacts.

As every AsyncTask (CommonService is an AsyncTask with decorations), these classes operate both with the GUI thread and on background without it (see Section 8.2.2). All the client–server interaction takes place in the *doInBackground* method, where we initialize a RestTemplate instance and perform a required operation, like shown in Listing 8.

Almost all CommonServices are fairly simple to understand to, since they exchange only one instance of marshallable container at the time. This does not apply to the **CreateScenario** and **UploadDataFile** classes, which must upload an existing file from device's file system as well as required meta-data.

Since we use Spring MVC on the server and Spring's RestTemplate on the client, the solution is simple as well. On the client we create an instance of **MultiValueMap**, into which we set all meta-data under proper keys, and a single **FileSystemResource** reference to file in device's memory with a "file" map key. With these settings RestTemplate sends file as a MultiPart object, which is by the key names binded to proper Controller method's parameters. An example of sending scenario file is visible in Listing 13.

```
1 FileSystemResource toBeUploadedFile =
2         new FileSystemResource(scenario.getFilePath());
3
```

```
4  MultiValueMap<String, Object> form =
5           new LinkedMultiValueMap<String, Object>();
6  form.add("scenarioName", scenario.getScenarioName());
7  form.add("researchGroupId",
8          Integer.toString(scenario.getResearchGroupId()));
9  form.add("description", scenario.getDescription());
10 form.add("mimeType", scenario.getMimeType());
11 form.add("private", Boolean.toString(scenario.isPrivate()));
12 form.add("file", toBeUploadedFile);
13
14 HttpEntity<Object> entity =
15          new HttpEntity<Object>(form, requestHeaders);
16 ResponseEntity<Scenario> response =
17   restTemplate.postForEntity(url, entity, Scenario.class);
```

Listing 13: Example of uploading a Scenario file with its meta-data

Since uploaded files may be large, do not forget to disable file buffering prior its sending, otherwise the application will run out of memory. It is managed by setting a *false* into a request factory method *setBufferRequestBody*.

15.5 User Interface

The user interface classes are located in *cz.zcu.kiv.eeg.mobile.base.ui* package and its sub-packages. The package root contains the **NavigationActivity** class and the **TabListener** class, which was mentioned in Section 14.1.

The main package contains sub-packages with their own classes. These sub-packages are named either by a data type their classes work with (*datafile, experiment, person, scenario, reservation*), or by logic or a phase their classes belong to (*dashboard, filechooser, settings, startup*).

15.5.1 Startup

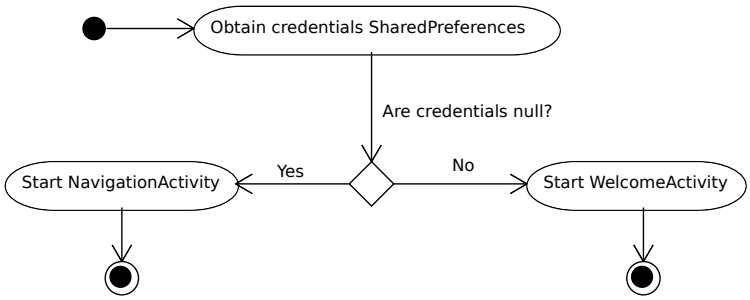


Figure 15.1: Activity diagram of application startup

On start up in **StartUpActivity** application checks, if there are user's credentials stored in shared preferences. If so, a **NavigationActivity** is started, otherwise a **WelcomeActivity** is launched, like in Figure 15.1

In WelcomeActivity (Figure 15.2), user provides and confirms credentials, which are first offline validated, whether they contain a credentials data or not. After the validation phase starts a **TextCredentials** CommonService, which tries to connect to server and obtain user details. If succeeds, the credentials are saved and user is let to the NavigationActivity, otherwise he/she is informed by an error message and must either wait till the server is available (if there was a connection issue), or must correct the credentials.

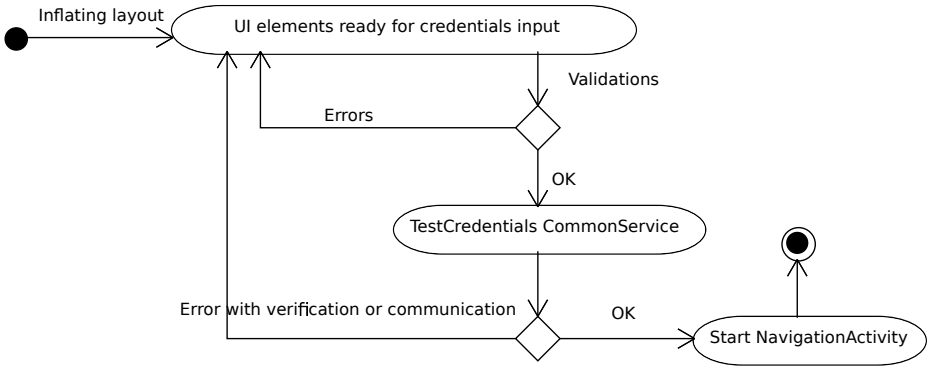


Figure 15.2: Activity diagram of WelcomeActivity

15.5.2 NavigationActivity

When the user is in **NavigationActivity**, all sections of an application are accessible. Simple sections are resolved as fragments, which are replaced inside the activity. Other sections, which demand further navigation elements (tabs), are resolved as activities, which are started from the NavigationActivity (see Figure 15.3).

The *onStop* and *onRestart* actions relate to Activity's life cycle, when another Activity takes place on the top of the GUI stack. The *onDestroy* can be possibly called only by the garbage collector, when the application is inactive (i.e. "turned off"). Such behaviour is correct, since the StartUpActivity creates a new instance during a new application run.

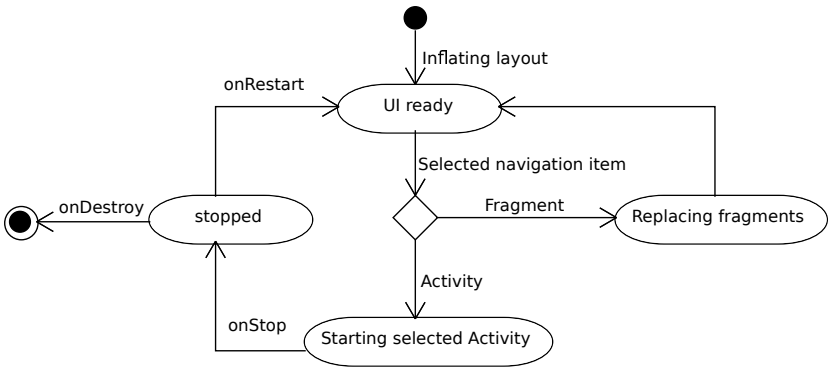


Figure 15.3: Activity diagram of NavigationActivity

Browsing through application sections is resolved using a spinner-like list menu, as discussed in Section 14.3. This activity on start up fills its body with an instance of **DashboardFragment**, which contains only a welcome message.

15.5.3 Record Browsing

There are three stand-alone sections for browsing data in the *EEG Base for Android* .

- Experiments – located in **ExperimentActivity**,
- Scenarios – located in **ScenarioActivity**,

- EEG Lab reservation times – located in **ReservationFragment**.

While ExperimentActivity and ScenarioActivity are separate activities created from the NavigationActivity, the ReservationFragment is a fragment, which replaces NavigationActivity's content, so its activity diagram is slightly different from Figure 15.4, since there is no Up button.

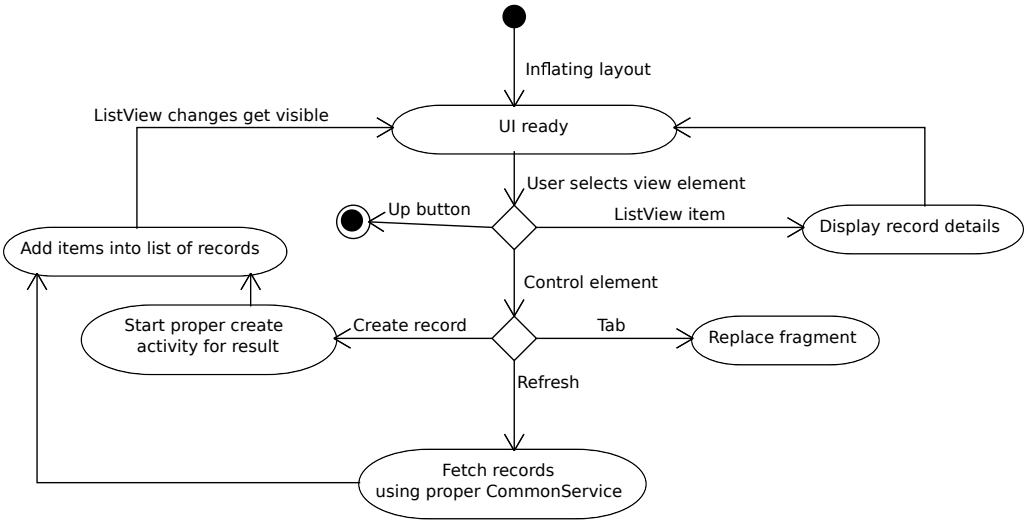


Figure 15.4: Activity diagram of activities with tabs for record browsing

In all three sections you can fetch existing records from the server, create new ones in a separate dedicated activity (**AddRecordActivity** for reservations, **ExperimentAddActivity** for experiments and **ScenarioAddActivity** for scenarios).

There is a need of distinguishing user's records from all public records in experiments and scenarios sections, therefore these activities dispose of tabs in navigation menu. When selecting a tab, related fragment replaces body of an activity. Each fragment has its own array adapter, so fetched data are independent across fragments.

When selecting an item from fragment's list view, details of the specific record are displayed. Information is displayed either in its fragment (which applies to tablets), or in a separate activity (phones). This behaviour was chosen in order to preserve amount of displayed information on screen, while keeping layout clear and readable. The application chooses which way to display data upon fragment's creation, when it retrieves a layout resource. If there is a details frame in the layout, the data are displayed

inside this frame, otherwise a details activity is created. Details class naming convention in this application is *<Type>Details<Activity|Fragment>*, e.g. **ExperimentDetailsFragment**.

The only issue in displaying details is with a collection of records, which we need to display in scrollable component. In such case we cannot use another listview or a similar scrollable component, but we must create a frame layout, into which we programmatically create and inflate row items one by one. Such solution may be seen in **ExperimentDetailLists** class. This class contains methods methods into which we provide only frame layout and a collection with data – the rest is handled by the class's methods.

15.5.4 Creating a New Record

In all aforementioned sections (reservations, experiments, scenarios) we can create new records. In general, there is a SaveDiscardActivity, whose layout contains all required input fields. When they are filled, user confirms the data, proper CommonService related to the data type is triggered and the record is created. These activities also provide created record as a result to previous activity, in order to add it into its array adapter.

From **ExperimentAddActivity** or **ScenarioAddActivity** we can also create records of data types, which do not have its section for browsing, but which are available for purposes of a new complex type creation. Such data types are people, artifacts, digitizations, diseases, electrode fixes, locations and types and weather types. All data types have its container, which is used in the name of an activity for creation of such record. The naming convention for these activities is *<Container>AddActivity*.

The last creatable record is Reservation, which represents booking time to the EEG Lab at the University of West Bohemia. The difference against other data types is, that these records can be removed from the application, if they are owned by user or the user is an administrator.

15.5.5 Choosing File for Upload

Since EEG researchers gather files, which need to be uploaded, we must be abble to provide such logic. File upload takes place in two places: in **ScenarioAddActivity** class and **DataFileUploadFragment** class. In case of scenarios we upload files only when creating a new scenario record. In case of data files, we can upload them only to existing experiments from

the NavigationActivity.

In both cases we need to select a file to be uploaded. Because Android does not provide any file-browsing component, a **FileChooserActivity** was created for these purposes. It is a list view, which fills **FileInfo** instances as its rows. On an item click event we return to caller activity. The process of uploading a data file is visible in Figure 15.5.

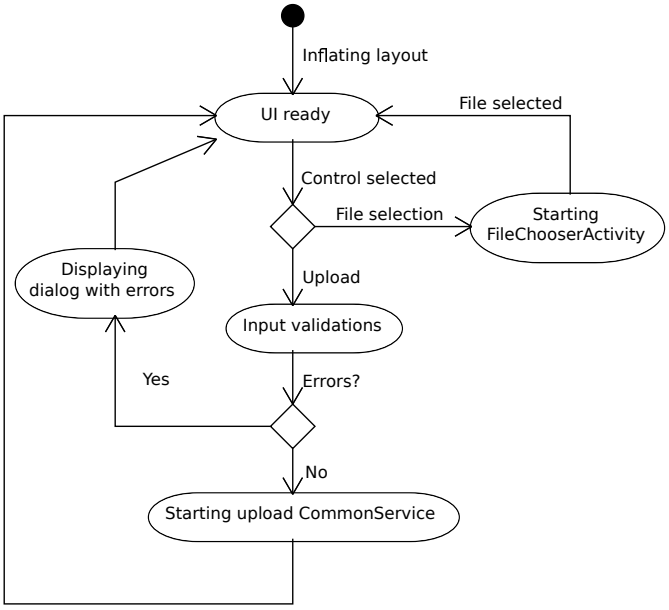


Figure 15.5: Activity diagram of DataFileUploadFragment

15.5.6 Settings

Settings are simple custom implementation of SaveDiscardActivity, which serves for making changes in credentials shared preferences. It provides means to edit username, password and web service URL. Moreover, there is a feature that enables to set Monday as a first day of week. It is useful for choosing a date in DatePicker component on tablets, where a calendar is displayed, because Sunday as a first day is confusing for some users.

15.6 Utilities

Utility classes are located in the package *cz.zcu.kiv.eeg.mobile.base.utils*. These classes gather useful methods used across the application.

- **ConnectionUtils** provides the method *isOnline*, which checks, if there is an existing network connection.
- **FileUtils** contains methods for obtaining file's MIME type and its file size in a human readable format.
- **LimitedTextWatcher** is an interface implementation used over *EditText* components, which have a maximal text length set. It provides means to update another *TextView* component with a message of characters left till a maximum text length limit is reached.
- **ValidationUtils** class gathers methods used for various commonly used validation operations, like checking validity of email string, URL string or just checking emptiness of a string.

16 | Testing

Server side functionality was split into two sections – a service core functionality and a URL mapping functionality. The service core functionality was verified using the unit testing framework JUnit 4, while the URL mapping was tested by the *EEG Base for Android* client application.

Unit tests are located in *cz.zcu.kiv.eegdatabase.webservices.rest* test package. The transactions and dependency injection work similar as in the tested components, only a special Spring context file is required. This context file initializes database connection and other vital Spring components (beans, transaction manager, ...). The context loading is managed by code shown in Listing 14, which is placed above the class body.

```
1 @RunWith(SpringJUnit4ClassRunner.class)
2 @ContextConfiguration(locations = {"classpath:test-context.xml"})
```

Listing 14: Class annotations for loading Spring context

Creating a new record was tested in unit test methods as well as checking its presence in the output of getter methods (CR). In case of getter methods was checked, if content of service core method matches the data available through DAO objects (R) (Table 16.1).

Data type	**Test cases**	**Results**
Artifact	CR	OK
Data File	CR	OK
Digitization	CR	OK
Disease	CR	OK
Electrode Location	CR	OK
Electrode Fix	CR	OK
Experiment	CR, R public records, R user's records	OK, OK, OK
Person	R login, R all records	OK, OK
Research groups	R public records, R user's records	OK, OK
Scenario	CR, R public records, R user's records	OK, OK, OK

Table 16.1: Unit tests in accordance to data types and their results

Table 16.1 represents a brief summary of performed unit tests and their results. Tests themselves are represented by methods annotated with the **@Test** annotation over them.

However, in some test cases a logged user was required. It was resolved by manual logging in *setUp* test stage, as shown in Listing 15. Although there is a security risk in putting user credentials available in a plain text, this is the only solution, which was found and resolves the issue. A possible solution might be reading credentials from a properties file, where they would be set only in case of testing, or to create an account for testing purposes.

```
@Before
public void setUp() throws Exception {
    Person user =
        personDao.getPerson("petrmiko@students.zcu.cz");
    List<SimpleGrantedAuthority> authorities =
         new ArrayList<SimpleGrantedAuthority>();
    authorities.add(new SimpleGrantedAuthority(
        user.getAuthority()));
    Authentication auth =
         new UsernamePasswordAuthenticationToken(
                user, "heslo", authorities);
    SecurityContextHolder.getContext().setAuthentication(auth);
}
```

Listing 15: Manual logging in within Junit 4 test

Client side communication was either tested continuously during development and during user testing. Following scenarios were created for user testing:

1. Log into application
2. Display details of public scenario
3. Create experiment with a new electrode location
4. Upload file to a newly created experiment

These scenarios were performed by the following users, while their overall satisfaction marks are noted in Table 16.2. Scaled on range of 1 as the lowest to 10 as the highest satisfaction with the application.

Gender	**Age**	**Errors found**	**Mark**	**Reason to granted mark**
Male	24	1	7/10	Error found, could not find menu without manual
Male	23	0	8/10	Dark layout, tabs cannot be changed by swipe gesture
Female	23	0	9/10	Would prefer larger font size
Female	17	0	8/10	Could not find menu without manual, preference of Windows Phone UI

Table 16.2: User testers and their marks to the application

Only one error was found during user testing. This error was fixed and did not affect other testers. The first tester was also an EEG researcher. No further errors were found and all test scenarios were performed successfully.

A 1.19kB large file was uploaded during user testing , but a file of 250MB size was uploaded in a separate test .

17 | Future enhancements

Capabilities of discussed RESTful web service server side and its client side can be further improved. Following improvements are outside the scope of needs discussed in this thesis, but user experience would be improved if the improvements are implemented .

17.1 WADL Support

Currently there are no other files the server side controller classes, where information about existing endpoints of RESTful web service would be collected in. Implementing automatic WADL file generation on server start up, may be a viable solution for developers, who would like to implement an own client application for EEG Base. This WADL file would be available at main endpoint URL */rest/* afterwards.

17.2 Changing Tabs by Swipe Gesture

A swipe-driven tab changing is becoming a new navigation standard in modern applications. However, for providing such feature an Android's **Support Library** would have had to be used, which would require additional comprehension of the library and its new components.

17.3 File Downloading

In the current state endpoints for downloading user's data files by an identifier in GET request[1] are prepared. In response to such request a binary file is returned. A proposed enhancement lies in implementing a Service (not a CommonService, which is a decorated AsyncTask!) or using an Android component for background downloading file, if there exists such component.

[1] *getFile* methods in **DataFileServiceController** and **ScenarioServiceController**

17.4 Visualizing Data File

An additional step to file downloading might be allowing a user to open them. In case of a plain text files should be easy to display file's content, but in case of EEG data files it would be required to implement a graphic visualization of data file's content (signals, epochs, ...).

18 | Conclusion

The main goal of this thesis was to present a solution for gathering EEG/ERP data using mobile devices. After an analysis of needs and requirements the Android mobile platform was chosen for mobile application development. This application can be ran on mobile phones and tablets with Android v4.0.x and newer.

In order to provide requested data synchronization it was analysed, what type of data is needed to transfer. As a result was developed a RESTful web service server side and its client counterpart will be developed. Both allow to save and read meta-data, since they are needed the most, and to save larger data files, which are needed only to be stored.

The implementation process of such RESTful web service provided interesting insight into modern Java frameworks and technologies. Even more it was interesting that since Android platform is based on Java, many libraries and frameworks may be used in Android too, like the RestTemplate library. On the other hand, the Android platform has its specifics in design and expression possibilities. A lot of energy has to be put into finding solutions to issues like designing navigation, internal data transfer, screen orientation change issues, etc. Also, security has a significant role in presented solution. Therefore a custom implementation of HTTP client factory for RestTemplate was created. This factory implementation supports all SSL certificates (EEG Base does not have own certificate yet) and resolves issue with file buffering into memory, which is present in default implementation.

As a result of effort put into thesis's solution, a fully functional RESTful web service was implemented into the EEG Base. An Android client application *EEG Base for Android* was created as well. The application uses the web service for client-server communication and provides required data synchronization.

While creating an application, the *EEG Base for Android*, a lot of effort was focused on simplicity of software project management. Therefore a Apache Maven was used for clear definition of a directory structure, simplicity of dependency management and building the client application itself.

Both the client side and the server side can be easily extended for further logic in future, as suggested in Chapter 17.

List of Figures

List of listings

List of Abbreviations

API — Application Programming Interface
APK — Android Application Package
CRUD — Create, Read, Update, Delete
DAO — Data Access Object
DOM — Document Object Model
EEG — Electroencephalogram
ERP — Event Related Potentials
GUI — Graphical User Interface
HTTP — Hypertext Transfer Protocol
IDE — Integrated Development Environment
INCF — International Neuroinformatics Coordinating Facility
IoC — Inversion of Control
ISO — International Organization for Standardization
JAXB — Java Architecture for XML Binding
JDK — Java Development Kit
JSON — JavaScript Object Notation
MVC — Model-View-Control design pattern
ORM — Object-relational Mapping
POM — Project Object Model
REST — Representational State Transfer
RPC — Remote Procedure Call
SAX — Simple API for XML
SDK — Software Development Kit
SOA — Service Oriented Architecture
SOAP — Simple Object Access Protocol
SSL — Secure Sockets Layer
TLS — Transfer Layer Security
UDDI — Universal Description, Discovery and Integration
UI — User Interface
URI — Uniform Resource Identifier
URL — Uniform Resource Locator
UX — User Experience
WADL — Web Application Description Language
WSDL — Web Services Description Language
XAML — Extensible Application Markup Language

XML — Extensible Markup Language
XSD — XML Schema Definition

Bibliography

[1] BlackBerry. Develop with blackberry. http://developer.blackberry.com/develop/, 2013. [Online; accessed 28-April-2013].

[2] Dieter Bohn. Android device diversity and fragmentation charted in minute detail. http://www.theverge.com/2012/5/15/3023119/android-device-diversity-fragmentation-chart, May 2012. [Online; accessed 29-April-2013].

[3] Peter Brittenham. An overview of the web services inspection language. http://www.ibm.com/developerworks/webservices/library/ws-wsilover/, June 2002. [Online; accessed 16-April-2013].

[4] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000. [Online; accessed 16-April-2013].

[5] The Apache Software Foundation. Apache maven website. http://maven.apache.org, unknown. [Online; accessed 25-April-2013].

[6] Google. Platform versions. http://developer.android.com/about/dashboards/index.html, April 2013. [Online; accessed 7-April-2013].

[7] Google. Fragment documentation. http://developer.android.com/reference/android/app/Fragment.html, unknown. [Online; accessed 27-April-2013].

[8] Google. Layouts. http://developer.android.com/guide/topics/ui/declaring-layout.html, unknown. [Online; accessed 20-April-2013].

[9] Google. Menus. http://developer.android.com/guide/topics/ui/menus.html, unknown. [Online; accessed 25-April-2013].

[10] Google. Property animation. http://developer.android.com/guide/topics/graphics/prop-animation.html, unknown. [Online; accessed 25-April-2013].

[11] Google. Providing resources: Grouping resource types. http://developer.android.com/guide/topics/resources/providing-resources.html\#ResourceTypes, unknown. [Online; accessed 16-April-2013].

[12] Google. Providing resources: How android finds the best-matching resource. http://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch, unknown. [Online; accessed 7-April-2013].

[13] Google. View animation. http://developer.android.com/guide/topics/graphics/view-animation.html, unknown. [Online; accessed 25-April-2013].

[14] Apple Inc. Cocoa touch. https://developer.apple.com/technologies/ios/cocoa-touch.html, unknown. [Online; accessed 28-April-2013].

[15] Microsoft. Xaml overview. http://msdn.microsoft.com/en-US/library/ms752059.aspx, unknown. [Online; accessed 28-April-2013].

[16] Roman Nurik. Done+discard design pattern. http://www.androiduipatterns.com/2012/08/roman-nurik-on-done-discard.html, August 2012. [Online; accessed 26-April-2013].

[17] OASIS. Uddi xml.org. http://uddi.xml.org/uddi-101, 2006. [Online; accessed 5-April-2013].

[18] Oracle. Establishing a secure connection using ssl. http://docs.oracle.com/javaee/5/tutorial/doc/bnbxw.html, 2010. [Online; accessed 29-April-2013].

[19] Edward Ort. Java architecture for xml binding (jaxb). http://www.oracle.com/technetwork/articles/javase/index-140168.html, March 2003. [Online; accessed 22-April-2013].

[20] Matt Silverman. The rise and fall of rim. http://mashable.com/2012/06/25/rim-decline-chart/, June 2012. [Online; accessed 20-April-2013].

[21] Craig Walls. *Spring in Action*. Manning Publications, 2011. ISBN: 1935182358.

A | JAXB Annotations Example

First we need to create a data container, which will be annotated by JAXB annotations. We use a container for simplified scenario information from the EEG Base as visible in Listing 16. The *propOrder* parameter in the *XmlType* annotation enforces order of elements in result XML file.

```
1  package data;
2
3  import javax.xml.bind.annotation.XmlElement;
4  import javax.xml.bind.annotation.XmlRootElement;
5  import javax.xml.bind.annotation.XmlType;
6
7  @XmlType(propOrder = {"scenarioId", "scenarioName"})
8  @XmlRootElement(name = "scenario")
9  public class ScenarioSimpleData {
10     @XmlElement
11     public int scenarioId;
12     @XmlElement
13     public String scenarioName;
14 }
```

Listing 16: ScenarioSimpleData.java

The next step is to create an instance of the ScenarioSimpleData class, fill it with data and unmarshall it into *scenario.xml* (Listing 17).

```java
package logic;

import data.ScenarioSimpleData;
import javax.xml.bind.*;
import java.io.File;

public class Writer {
  public static void main(String[] args) {

    ScenarioSimpleData scenario = new ScenarioSimpleData();
    scenario.scenarioId = 12;
    scenario.scenarioName = "Test JAXB scenario";

    try {
      File file = new File("scenario.xml");
      JAXBContext context =
        JAXBContext.newInstance(ScenarioSimpleData.class);
      Marshaller marshaller = context.createMarshaller();
      // output pretty printed
      marshaller.setProperty(
        Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
      marshaller.marshal(scenario, System.out);
      marshaller.marshal(scenario, file);
    } catch (JAXBException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 17: Writer.java

Output stored in scenario.xml is the following:

```xml
<scenario>
    <scenarioId>12</scenarioId>
    <scenarioName>Test JAXB scenario</scenarioName>
</scenario>
```

Listing 18: scenario.xml

In order to recreate the scenario instance, code from Listing 19 must be run.

```
1  package logic;
2
3  import data.ScenarioSimpleData;
4  import javax.xml.bind.*;
5  import java.io.File;
6
7  public class Reader {
8    public static void main(String[] args) {
9      try {
10       File file = new File("scenario.xml");
11       JAXBContext context =
12         JAXBContext.newInstance(ScenarioSimpleData.class);
13       Unmarshaller unmarshaller = context.createUnmarshaller();
14       ScenarioSimpleData scenario =
15         (ScenarioSimpleData) unmarshaller.unmarshal(file);
16       System.out.println("Scenario ID: "
17         + scenario.scenarioId);
18       System.out.println("Scenario name: "
19         + scenario.scenarioName);
20     } catch (JAXBException e) {
21       e.printStackTrace();
22     }
23   }
24 }
```

Listing 19: Reader.java

Standard text output from such code is the following:

```
Scenario ID: 12
Scenario name: Test JAXB scenario
```

B | Parcelable Data Container

```java
import android.os.Parcel;
import android.os.Parcelable;
import org.simpleframework.xml.Element;
import org.simpleframework.xml.Root;

@Root(name = "owner")
public class Owner implements Parcelable {
    public static final Parcelable.Creator<Owner> CREATOR
            = new Parcelable.Creator<Owner>() {
        public Owner createFromParcel(Parcel in) {
            return new Owner(in);
        }

        public Owner[] newArray(int size) {
            return new Owner[size];
        }
    };
    @Element
    private int id;
    @Element
    private String name,  surname;
    @Element(required = false)
    private String mailUsername, mailDomain;

    public Owner() {
    }

    public Owner(Parcel in) {
        id = in.readInt();
        name = in.readString();
        surname = in.readString();
        mailUsername = in.readString();
        mailDomain = in.readString();
    }
```

```
35
36      public int getId() {
37          return id;
38      }
39
40      public void setId(int id) {
41          this.id = id;
42      }
43
44      public String getName() {
45          return name;
46      }
47
48      public void setName(String name) {
49          this.name = name;
50      }
51
52      public String getSurname() {
53          return surname;
54      }
55
56      public void setSurname(String surname) {
57          this.surname = surname;
58      }
59
60      public String getMailUsername() {
61          return mailUsername;
62      }
63
64      public void setMailUsername(String mailUsername) {
65          this.mailUsername = mailUsername;
66      }
67
68      public String getMailDomain() {
69          return mailDomain;
70      }
71
72      public void setMailDomain(String mailDomain) {
73          this.mailDomain = mailDomain;
74      }
75
```

```
76      @Override
77      public int describeContents() {
78         //default value
79          return 0;
80      }
81
82      @Override
83      public void writeToParcel(Parcel dest, int flags) {
84          dest.writeInt(id);
85          dest.writeString(name);
86          dest.writeString(surname);
87          dest.writeString(mailUsername);
88          dest.writeString(mailDomain);
89      }
90  }
```

Listing 20: Example for Parcelable data container, which handles XML marshalling

C | Installing Android Application

C.1 Maven Build and Install

In order to build and install an application from a source Maven 3 is required. Open console and enter directory with **pom.xml** file. Inside of it, build project.

```
mvn clean package
```

If the build finishes successfully, an installation package **eeg.base.apk** should be present in the *target* directory. Now you can either install the application as described in Section C.2, or the install application using Maven. In order to install the application using Maven, enable *USB debugging* in *System settings > Developer options*. When set, type and confirm following snippet in console. It installs the application and starts it once it is installed.

```
mvn android:deploy android:run
```

C.2 Install from APK

In order to the install application directly from the APK file we need to enable the *Unknown sources* item in *System settings > Security*. Now there are two possible ways to install the file. Either copy APK file into device (MTP protocol, mount phone as USB storage or just send it to yourself by a mail) and start it, or enable USB debugging and use ADB application from Android SDK, if you have it on your computer:

```
adb install eeg.base.apk
```

D | EEG Base for Android: User Manual

The application is started by clicking on *EEG Database* icon in device's main menu. If the application is started for the first time, a log in activity screen will be presented to user (Figure D.1).

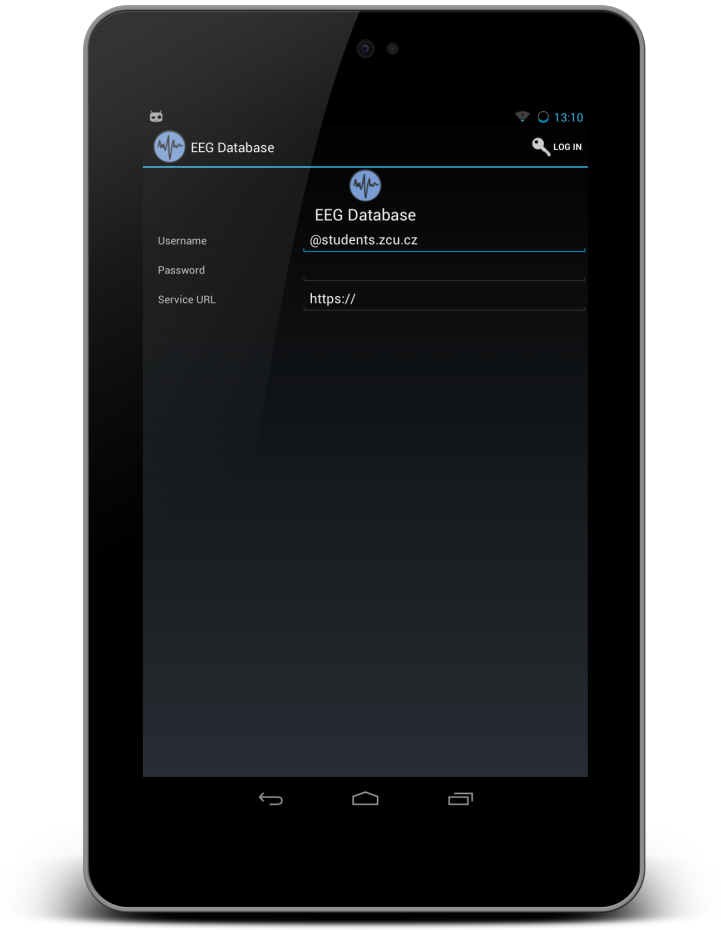


Figure D.1: Log in screen

When the user successfully logs into the application with own EEG Base

credentials and proper service URL (testing server — `https://147.228.64.172:8443` — was used during the thesis implementation), the welcome screen called **Dashboard** is displayed to the user.

(a) Welcome message (b) Menu in welcome screen

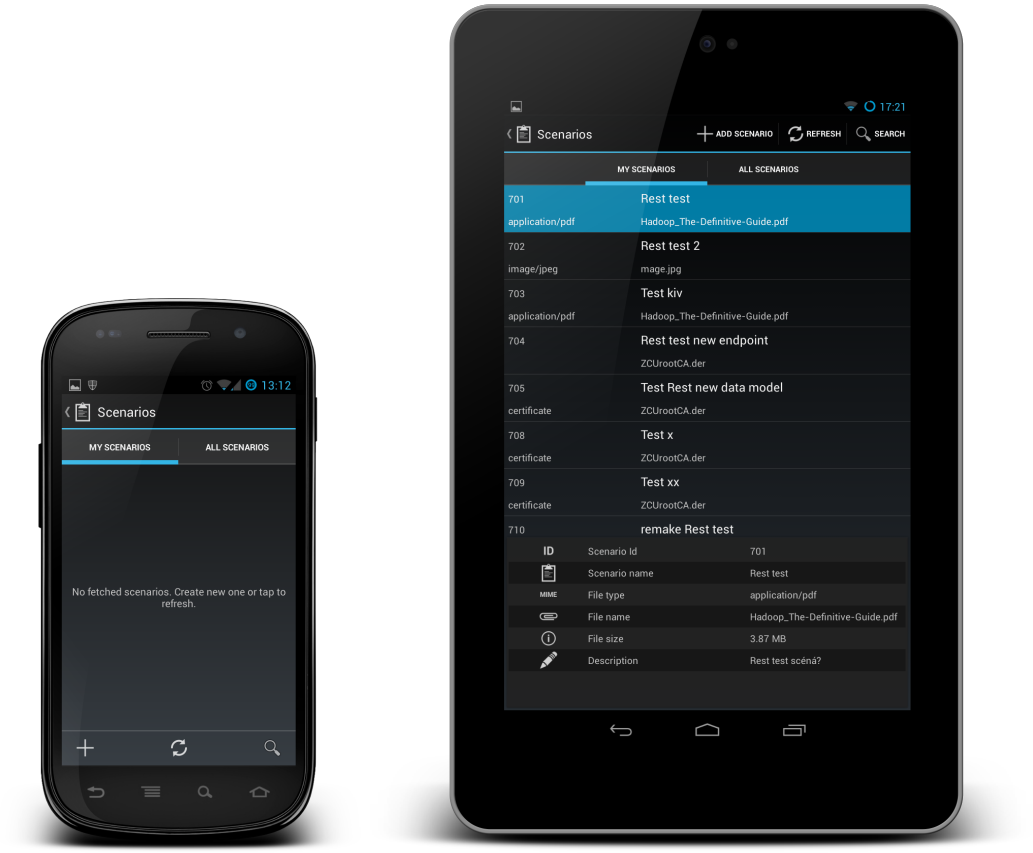
Figure D.2: Dashboard screen

The Dashboard shows a welcome message(Figure D.2a).

The welcome message contains basic application description and tells the user that in order to switch into another application section, he/she must use a menu located in the upper left corner of a display, as it can be seen in Figure D.2b.

Scenarios and **Experiments** sections look very similar, except for information they both provide. They both allow to fetch records from the server and display detail, filter fetched results, or create new records.

Records can be fetched either by clicking the *Refresh* button in the action bar, or by tapping the empty screen body.



(a) No records fetched yet (b) Fetched records with selected record for detail

Figure D.3: Scenarios section

Once records are fetched, any of them can be selected, in order to display details either in a new screen (mobile phones) or in a details screen section, like in Figure D.3b. Since records have been fetched, they can be filtered by title or identifier. It is done by clicking the magnifying glass icon of **Search**

button. The filter can be removed by entering blank filter string, otherwise filtered results will be displayed. An example of filtering results by "rest" keyword can be seen in Figure D.4.

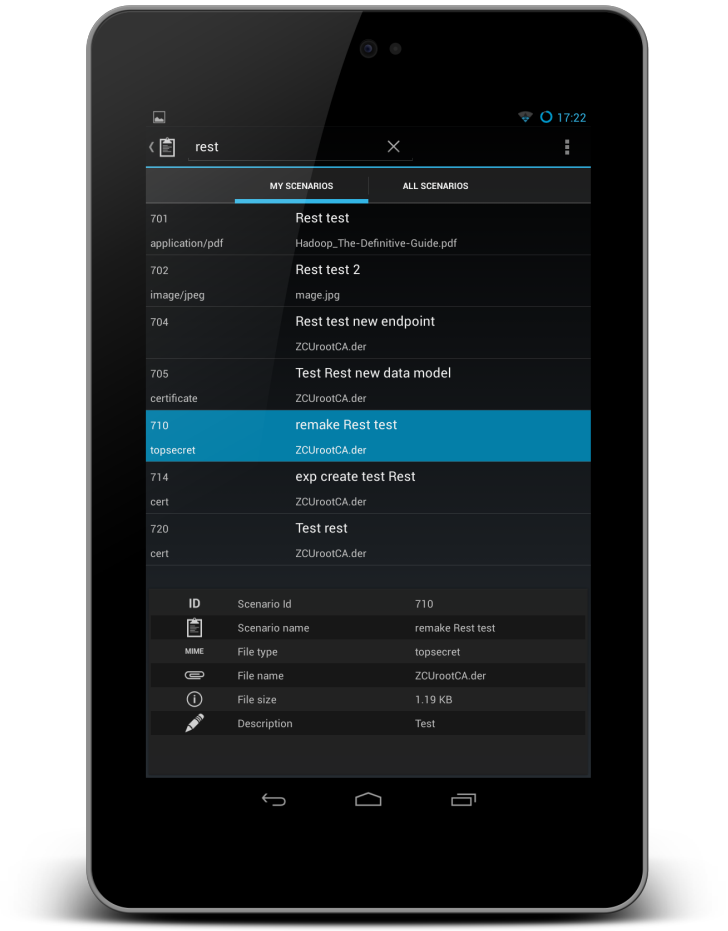


Figure D.4: Filtering fetched records

The record's details may contain list of secondary information, like a list of electrode locations. If there is such a list, any item can be clicked on, in order to display a dialog window with further information.

Now let us focus on creating new records. Both in scenarios and experiments section there is a related button in action bar for creating a new record. It has an icon with plus symbol and it opens a new activity screen. Inside this new screen we must fill all the required fields, otherwise application will show dialog with error message describing, what is missing.

In case of a scenario type, a file from the device's file system must be selected. In order to select the file we must click on a button with a directory icon; a file browser is opened (Figure D.5a). When a file is selected, the file browser is closed and user may save the record. An example of filled information with the selected file for upload can be seen in Figure D.5b. The record is saved by clicking the *Save* button or discarded by the *Discard* button.

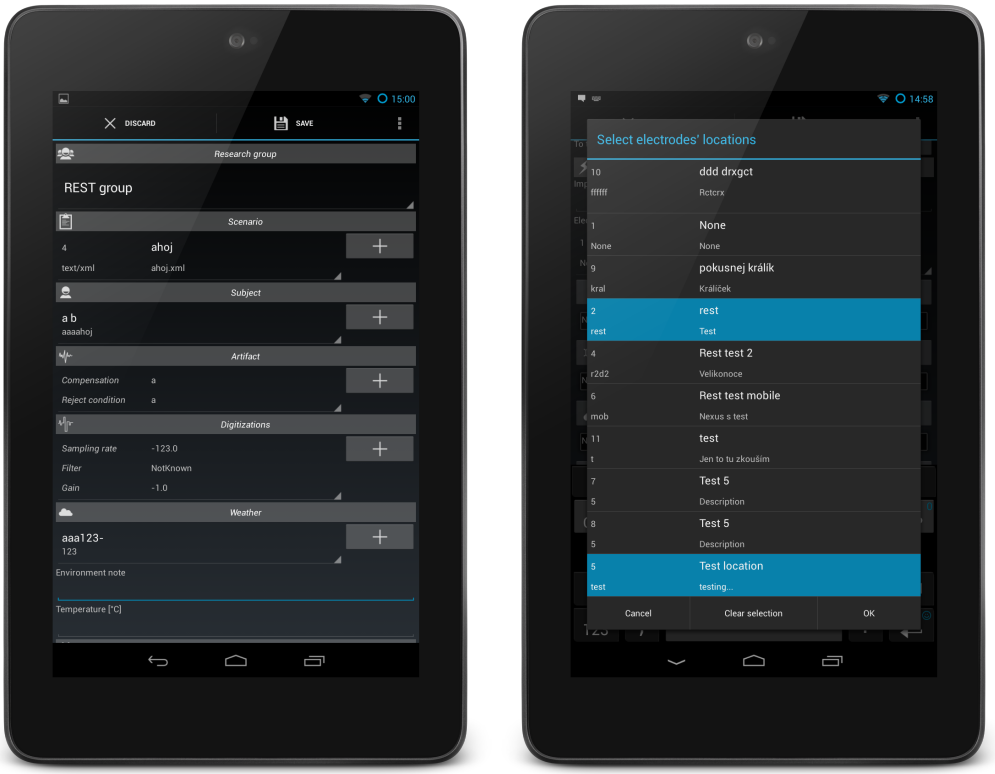
(a) File browser (b) New scenario creation screen

Figure D.5: Creating new scenario

In case of experiments we must first wait till required data are fetched from the server. Creating a new experiment is a complex operation, which requires a lot of additional information (Figure D.6a). When creating a new experiment, we must fill all the information required as in the case of a new scenario record.

Some of the supplementary information may be created during experiment creation, like new subject person, electrode location, etc. It is done using buttons with the plus icon, which opens a new screen activity and returns back, when the record is created.

A multi-selection list is specific for this application. It requires, unlike a classic list menu, a confirmation of selection. This applies to items with possibility of multiple records like electrode locations (Figure D.6b).



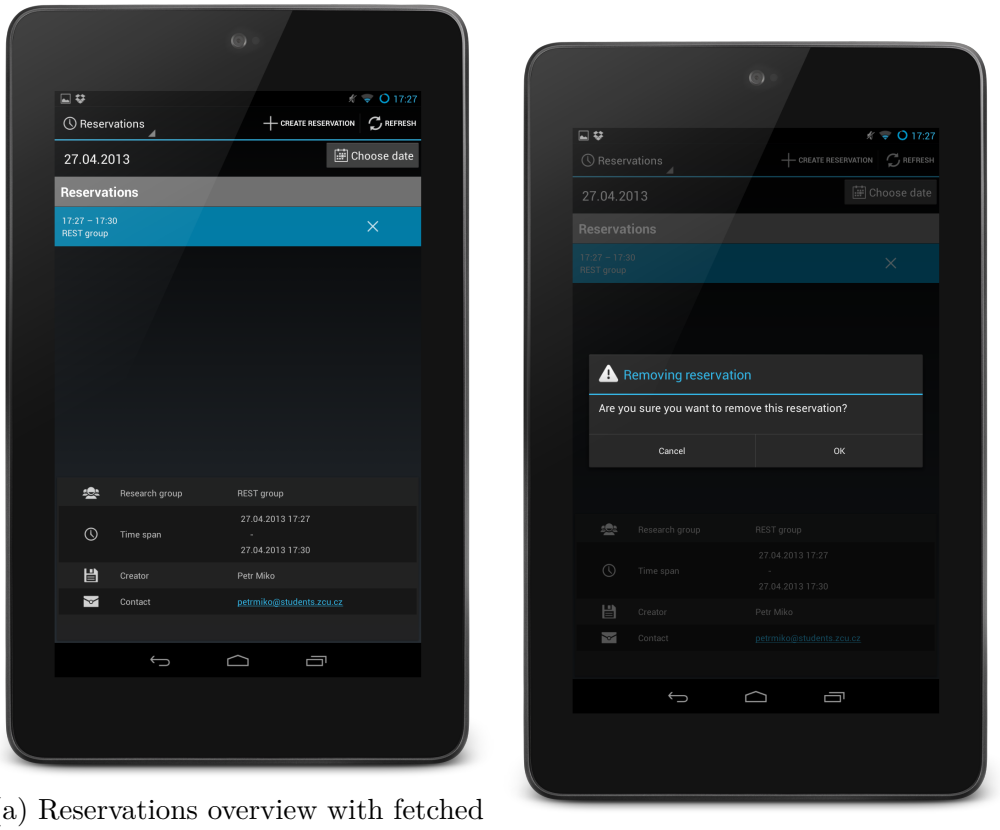
(a) New experiment screen (b) Multi-selection list

Figure D.6: Creating new experiment

In the case of experiments files are not uploaded during experiment creation process. File upload from section **Upload file** must be used, since there may be a continuous need of file uploading. A proper experiment (Experiment spinner) and a file (*Select file* button), using file browser like in Figure D.5a, must be selected in this section, and then confirmed by *Upload file* button (Figure D.7).

Figure D.7: Experiment data file upload

Another section of the application is a **Reservation** section. In it user may select a date using *Choose data* button and then fetch records to such date from the server, either by tapping the empty list body, or by clicking the *Refresh* button. There are details, which can be displayed by selecting a specific record, as well as in Experiments or Scenarios and there can be created a new record in activity under the *Create reservation* button (Figure D.8a). Unlike in other sections, reservations can be removed if they belong to the user, or the user has administrator rights in the EEG Base. Such record removal is performed by clicking on the X button in the record list item and the following dialog must be confirmed (Figure D.8b).



(a) Reservations overview with fetched removable record

(b) Removing reservation record

Figure D.8: Reservations

The last section of the application is **Settings** (Figure D.9). User can change the EEG Base credentials or service URL here. Unlike login screen there is important to keep */rest* URL suffix, which was not present in log in. There is also a possibility to set Monday as a first day of week, which might be useful for users with tablets (so there would be extended date pick dialog), which are used to such calendar layout.

Figure D.9: Settings