Universitatea TRANSILVANIA din Braşov
Facultatea de Matematică şi Informatică
Specializarea Informatică Aplicată în Limba Germană

Lucrare de licenţă

Autor: **Micu Adriana**
Coordonator ştiinţific: **Conf. univ. dr. Dumitrescu Silviu**

Braşov

2013

Chapter 1

Introduction

Within the modern society, news is very important, because it keeps everyone informed about events that occur around them, which may or may not affect them. News can be used for staying informed about the new technologies, which is important in the Information Technology domain, because, everyone belonging to this field of knowledge wants to compete against their opponents and be the best. This is one of the reasons why being up to date is important. News is also used for entertainment. Everyone either reads news from the newspaper or from the Internet, or hears it on the radio or watches it on the television. The technology has evolved rapidly in the last years, so almost every person has a smart phone or a tablet. Because it is important to stay up to date in every way we can, applications which are designed for these devices have become very useful.

This is why this project implements such an application designed for mobile devices. What it needs is a connection to the Internet, which nowadays is not a problem.

News is also a reason why people socialize. It is always useful to talk to friends about anything that is happening. These messages can be called “private news”, so “private news” are only shared between friends, not like the other news, on the public service.

For this reason, because the purpose of this project is to keep a person informed and let the person inform others, the application also implements a chat option. In this way, the person who has this application installed on their

device can stay up to date with what their friends want to share with them, and they can share their private information with them.

A chat is a real-time, text-based communication between two users. This means that, when talking to someone in chat, when sending a message, this will be received by the other participant immediately. Having a written conversation with someone as if talking to that person and being able to send messages back and forth to each other is what makes the difference between chat and e-mails. Chat makes it easy for people around the whole world to keep in touch and talk faster and easier than through emails, and much cheaper than through the telephone, especially if the participants are not in the same country. The only thing which is used by the chat service is an Internet connection, no cost extra.

This documentation is structured in 6 parts, each of them describing different themes. In the first part, the introduction, there is presented the main purpose of the application and also some general facts about the news and about the chat service. In the second part, the most important concepts of Java are described and exemplified. The third part introduces the Android Operating System, its connection to Java and the components used to develop an application compatible with the mobile environment. In the fourth part, there is presented the server-based architecture with the database. The fifth part explains the steps of building a client-server mobile application for the Android OS and the connection between the client and the server. In the last part, the sixth, presents the application with its functionality.

The evolution of the possibilities to propagate the news

The newspapers were invented in the early 17th century, but before that, in centralized empires, official government bulletins and edicts existed. The first to use an organized courier service were the Pharaohs, in Egypt, 2400 BC. The next phase was born in the Ancient Rome, where government announcements spoken by Julius Caesar were carved in metal or stone and

posted to public places. China contributed afterwards with the production of handwritten news sheets read by government officials and their next step was to publish the first private newssheets in 1582. The "Relation aller Fürnemmen und gedenckwürdigen Historien", from 1605, is recognized as the world's first newspaper. The "Agence France-Presse" is the oldest news agency, which was founded in 1835.

In the modern society, there are news services which are available 24 hours a day using live satellite technology to bring current events to all consumers' homes as the event occurs. These events that used to take much time to get to every persons ears and eyes are now fed instantaneously to everyone around the whole world via radio, television, Internet and the mobile phone.

So that is how nowadays everyone can stay informed about everything that is happening around the world, but it is not enough, because, although to stay informed about what is happening around you is very important, everyone wants to know about their close ones all the time. That is why people communicate in different ways if they are not able to communicate face to face. The most popular ways to communicate are through telephone, through email, or through chat.

The history of chat

In the 1960's, while the Internet did not yet exist, there have been made the first references that a chat took place. There was developed a real time chat program that allowed users who were connected to the same computer to chat with each other. The next phase of the chat developing was in 1988, when Jarkko Oikarinen created IRC or Internet Relay Chat, by designing the first chat server at the University of Oulu, Finland. After this, other universities began to create their own servers too and then they connected their servers to create the first network.

The modern technology came with many ways to chat. Mostly, the Internet users chat through instant messaging online tools like Yahoo Messenger or social networking websites like MySpace.

Chat has nowadays become very helpful, that is why it is a very important part of communication. It can be used for entertainment, but also, in the business domain, it is used in their private network for workers to communicate with each other in large offices. Support staff or helpdesk use instant messaging to communicate easier with their customers and help solve their problems faster.

This means, that if chat would not exist, the lines of communication between everyone would not be as open as they are in the modern society.

Now that there have been presented some historical facts about the news and the chat service, and also the most important functionalities, facilities and uses of both, the presentation of the technical details can begin, starting with the following chapter: *2 Java: The most important concepts*.

Chapter 2

Java: The most important concepts

As the society and the technology evolved, the programming languages participated too. The tools for “communicating” to a machine what and how to do have evolved by looking less like machine-understood, and more like parts of our minds and like our type of expressing things. This is how OOP or object-oriented programming was born. This paradigm represents concepts as “objects” that have different characteristics or “attributes” that describe the object which are used to interact with one another to design a complex computer application.

In my project I used the Java 7 technologies, because it is the latest release for Java that contains new features, bug fixes and enhancements to improve the efficiency to develop and run Java programs. As everyone knows, in the past few years, Java has evolved and is now anywhere, from laptops to datacenters, game consoles to scientific computers and cell phones to the Internet. This was another reason to choose Java, and also because Java is the foundation for every type of networked application and is the global standard for developing and delivering mobile applications, games, Web-based content and enterprise software. Java enables the programmers to efficiently develop, deploy and use interesting applications and services, with more than 9 million

developers worldwide. So Java is the choice of so many programmers around the world, and it has been tested, refined, extended and proved by the community of Java developers. Java also makes applications available across heterogeneous environments, which is a very big advantage, because like this, software can be written on one platform and it can be run virtually on any other platform. Another reason why I chose Java is because I had to work with web services and with connections across the network to develop the server-side application, which Java can make possible in a very stable way. Java is also used to write powerful and efficient applications for mobile phones, remote processors, consumer products and practically any other electronic device.

2.1 Objects: Everything revolves around them

OOP is about abstract data typing, inheritance, and polymorphism. The level of abstraction is very important, and it has reached the point where a computer program is written to model the problem a person is trying to solve, and not modeling the machine, as it was before. The elements in the problem are the so called “objects” which can be read to express the problem and its solution. The connection to the machine still remains visible, because the objects have a state and they have operations which they can perform, but this can be found analog, in the real world, because objects have characteristics and behaviors.

To define objects that have the same characteristics, the “class” keyword was introduced. Each object can be represented with a unique entity in the computer program, and each object belongs to a particular class with its own characteristic and behaviors. Once a class has been created it can be reused, which is one of the greatest advantages of object-oriented programming.

For example, in my project, I used a class called “Appuser” to create the type of user that I want to have for the application. On the next page is a snippet of the class.

```java
package model;

import java.io.Serializable;


 * The persistent class for the appuser database table.
@XmlRootElement
@Entity
public class Appuser implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private int id;

    @Lob
    private byte[] avatar;

    private String email;

    private String firstname;

    private String lastname;

    private String password;

    private String phone;

    private String securityphrase;

    private String username;

```

Figure 2.1: Appuser class

Above we can see that the “Appuser” object has an id, an avatar, an email, a first name, a last name, a password, a phone and a security phrase. This object can now be used in every other class of the project to create a new unique user with its own characteristics. To set or get these properties, we need some “public” methods, because the attributes are declared “private”. That is why, by the convention, I created getter and setter methods for every attribute of this class. On the next page is a snippet with some of these methods.

```
40      public int getId() {
41          return this.id;
42      }
43
44      public void setId(int id) {
45          this.id = id;
46      }
47
48      public byte[] getAvatar() {
49          return this.avatar;
50      }
51
52      public void setAvatar(byte[] avatar) {
53          this.avatar = avatar;
54      }
55
56      public String getEmail() {
57          return this.email;
58      }
59
60      public void setEmail(String email) {
61          this.email = email;
62      }
63
64      public String getFirstname() {
65          return this.firstname;
66      }
67
68      public void setFirstname(String firstname) {
69          this.firstname = firstname;
70      }
```

Figure 2.2: Appuser class (continued)

The getter and the setter methods can be used outside this class to set and get the characteristics of specific users. This can happen thanks to the “public” access control modifier. This modifier allows the method, or the variable, or the class, depending on what it is referring to, to be visible to the world: all other classes in the project.

Other modifiers in Java are: “private”, which is used to make methods, variables or classes visible only to the class they belong to, “protected”, which makes what it refers to visible to the package and all subclasses and the default

modifier, if before a method, variable, or class, no modifier name is written, the default one in set automatically, which means what it refers to is visible to the package.

In Java, there are also non access modifiers. The “static” modifier is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable will exist regardless of the number of instances of the class. If a method is static is created it is used independently of any instances of the class and it will be called different from a non-static method, using ClassName.methodName(parameters / no parameters), while a non-static method needs an instance of the class where it exists, and is called outside the class using: classInstance.methodName(parameters / no parameters).

The “final” modifier is used for variables which can be explicitly initialized only once. The state of the object, the data within the object can be changed, but not its reference. Often the “final” modifier is used together with the “static” modifier to make a constant into a so called “class variable”. Final methods are created so that they cannot be overridden by any subclasses, if the content of the method should not be modified. A class is declared “final” to prevent it from being sub-classed.

Another modifier is the “abstract” modifier, which is used for classes which should never be instantiated, only extended. The methods are declared abstract, without any implementation in the main class, but only in the sub-classed.

The “synchronized” modifier is used to indicate that a method can be accessed by only one thread at a time.

The “transient” modifier indicates the Java Virtual Machine to skip the variable with this modifier when serializing the object containing it.

The “volatile” modifier is used to let the JVM know that the thread accessing the variable must merge its own private copy if the variable with the master copy in the memory.

Object serialization is explained below, after introducing the "interface" keyword and the concept of inheritance and polymorphism, and the annotations are presented in the next subchapter: *2.2 Annotations: Beyond the program itself*.

An interface has to establish what requests can be made from a particular object. It is declared with the keyword "interface" and it mostly contains the header of the methods of the object which will be implemented. A class implements an interface using the keyword "implements". An interface or a class can be reused, so that, if two classes have similar functionality, the programmer doesn't have to build an almost identical class, but he just has to make additions and modifications to it. This is possible with inheritance.

Inheritance is used when there has to be created a class that includes some of the characteristics of an already existing class, so the new class will be derived from the existing class by reusing fields and methods from the existing class without having to write them again. A derived class, or a subclass, inherits all members from its superclass. Each subclass has its one and only direct superclass, which can have more subclasses. This is called the single inheritance. So that the program decides correctly which method it should use, depending on what type of class the object has, the concept of polymorphism was invented to help resolve this problem.

Polymorphism, which etymologically means "many forms", gives the programmer the possibility to create a subclass and use it in the same way as if he would use its superclass. This makes the code in object-oriented programming extensible, because new types can extend a superclass and this new subclass will work with its new code, but also with the already existing code in the superclass. The advantage of polymorphism is that, although using the subclass object like the superclass object, Java knows to automatically invoke the right methods.

Advancing with the explaining of what can be seen in the example from which I started, the description of the object serialization follows.

Object serialization

In Java, object serialization allows to take an object, if it implement the Serializable interface and turn it into a sequence of bytes that can be later fully restored to regenerate the original object. This way, objects can be sent across the network, without having to worry about the fact that the receiver may not read a correct data representation.

In my project, I used object serialization on the server-side application to be able to correctly send the data around to the database I used, and to the client-side application. Serialization is a method for the persisting of objects and like this they can be saved into a database.

In Java, objects can also be so called “exceptions” which represent errors which occur when anything does not run correctly within the program. Since the beginning of programming languages, this has been one of the most difficult issues, because the more complex the written code is, the more different errors may occur. Among all programming languages, Java’s exception handling is different because it forces the programmer to use it, because otherwise, if the code needs exceptions handled and that does not happen, compile-time errors occur.

Exception handling

An exception is an object that stops the normal program execution if it is “thrown” and not “caught” by an appropriate exception handler. A checked exception must be caught in the “try-catch-statement” in one of the catch-blocks whose associated exception class matches the class or superclass of the thrown exception. The exception is thrown in the same thread as the application, so the thread will stop, if the catch-block does not prevent it. The finally-block, which

follows after a try-catch-statement, is executed regardless whether an exception is caught or not. Below there is a simple example of a try-catch-statement in Java.

```
try {
    file.createNewFile();
} catch (IOException e1) {
    Toast.makeText(
            FragmentChangeActivity.c,
            "File could not be created ! Conversation NOT saved !",
            Toast.LENGTH_SHORT).show();
    e1.printStackTrace();
}
```

Fig 2.3 Try-catch statement

I used the above code in my project to write a file which saves a conversation between two users. In the try-block, the file which will contain the conversation has to be created. If that goes wrong, the catch-block prints on the screen of the user trying to save the conversation, the text: “File could not be created ! Conversation NOT saved !”, and also, for the programmer very useful, it prints the stack trace in the console, so that it is easier for the programmer to identify the root problem of the exception.

Further, I will explain the Java annotations and exemplify where and why I used this form of syntactic metadata.

2.2 Annotations: Beyond the program itself

Annotations are a form of metadata which provide information about a program that is not part of the program itself.

Annotations can be applied to almost all declarations: declarations of classes, declarations of fields, methods, and also other program elements.

A very used example is the "@Override" annotation. This annotation is applied to methods, and indicates that a method declaration is intended to override another method declaration in a supertype, if the subclass method, annotated with "@Override" has a signature that is override-equivalent to that of any public method declared in the superclass.

In the example from figure 2.1 shown in the above pages, there are four different types of annotations used, which I also used in other classes for the same purposes as in the class from the example. Below I will explain the characteristics of those annotations and why I had to use them.

The "@XmlRootElement" allows the mapping of my class to an XML element. About XML and its parsing, and their uses in my project I will explain in the fourth part which is about the server-based application and the database.

The "@Entity" annotation specifies that the class is an entity, which means it is a lightweight persistence domain object which represents a table in the relational database I used, and each instance corresponds to a row in that table.

Another annotation specific for the entities and the database work, is the "@Id" annotation, which specifies the primary key of an entity. The "@Id" annotated attribute of the class is the name of the primary key of the table in the database.

The "@Lob" annotation is used to specify that a persistent property of the class should be persisted as a large object to a database supported large object type. I used this annotation to store pictures in the database, with the database type Longblob. More about the database follows in the fourth part of this documentation.

In my project, on the server-side application, I also used different annotations for the JPA or Java Persistence API, which will be presented and explained in the fourth chapter along with the other annotations specific to JAX-RS or the Java API for RESTful Services.

Another important concept that is used in Java is the concept of the nested classes, which were very useful in my project.

2.3 Nested classes: Elegant and clear code

A nested class refers to a class definition within another class definition. If the nested class is static it is simply called “static nested class” and if it is not static it is called “inner class”. The big difference between these two is that an inner class has access to the members of the enclosing class, even if they are declared private, but the static nested class does not have access to the members of the enclosing class. Nested classes are very useful for the logical grouping of classes that are only used once, their use increases encapsulation and the nested classes can lead to more readable and maintainable code.

A static nested class interacts just like any other top-level class with the instance member of the outer class.

An inner class has direct access to the methods and fields of the instance of the enclosing class, with which it is associated. Below is a snippet from my project:

```
public class LoginActivity extends Activity {
                    ...
    public void handleResponse(String response) {
                    ...
    private class WebServiceConnection extends AsyncTask<String, Integer, String> {
                        ...
        @Override
        protected void onPostExecute(String response) {

            handleResponse(response);
        }
```

```
...
    }
}
```

Figure 2.4: Inner class

I used an inner class for the connection of the client application to the server by sending information with the help of that class. As it can be observed, the name of my inner class is "WebServiceConnection" and it has access to the "handleResponse" method declared and implemented in its outer class. For the connection to the web service, I had to implement this class which doesn't run on the same thread as the main activity of the client application, so that, the client application will not be blocked because of the connection to the server application.

About the "Activity" class, I will talk in the next chapter, and the "AsyncTask" will be presented in the fifth part of the documentation, as a very important part of the connection between the client application and the server application.

And now, one more subchapter, and then I will present Android and examples from my project, but before, about threads in Java.

2.4 Threads: Independently running subtasks

We already know that objects represent the way to separate a program into independent sections, but there are some cases when we need to turn a program into separate, independently running subtasks, which are called "threads". Threads are programmed as if each one runs by itself and has the CPU to itself. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

The system has to know when a thread should start, if there is more than one, it has to have a starting order which is defined by the threads priority. A

thread continues to run until it finishes its “run” method or an error occurs while running.

There are two ways to create e new thread of execution. One way is to declare a class that implements the “Runnable” interface. That class then implements the “run” method. The other way, which I used in my project and will be exemplified in the snippet below, is to declare a subclass of Thread, which should override the “run” method of class “Thread”.

```
private class NotificationThread extends Thread {

    public NotificationThread() {
        super("Notification-Service");
    }

    public void run() {

        NotificationService serv = NotificationService.this;
        while (serv.runFlag) {
            try {

                        ...

                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                serv.runFlag = false;
            }
        }
    }

}
```

Figure 2.5: Thread class

This thread will be started as follows by creating a new instance of the “NotificationThread” class and by calling the “start” method. I used this thread, as it name says, to check if the user receives any notifications, after a specified delay time.

Multithreaded execution is an essential feature of the Java platform. The so called “main” thread is created for every application. Along with this thread, every application is able to have other threads which run independently.

For the controlling of threads there are some defined methods: the “start” method, which brings the thread to life and can be called only once in a life of the thread, the “stop” method, which complements the “start” method by destroying the thread and can also be called only once in the life of a thread, the “suspend” and the “resume” methods which can be used to arbitrarily pause and restart the execution of a thread. Another very important and used ability of a thread is for it to sleep. We can put a thread to sleep for some period of time by calling the “sleep” method. Also, the thread class contains an “interrupt” method which allows one thread to interrupt another thread.

With the help of threads, the performance related problems, like processing that takes a lot of time, background processing or I/O work, can be resolved very easy and with reusable and more understandable code.

In my project, in the above example, I used the thread to do background processing, so that the main thread is not blocked or interrupted while waiting for any notification, because this task needs to execute continuously in the background.

So threads simplify various performance and usability problems.

The other important Java concepts which I used in my project will be presented in the next chapters, along with explanations and examples from Android, the server, and the connection between them.

Chapter 3

Android: Everything goes mobile

Android is an open source platform designed for mobile devices, whose owners goal is to “accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience”. For developers, Android provides all the tools and frameworks for developing mobile applications, by providing the Android SDK, which is everything a programmer needs to start developing for Android, not even needing a physical Android-running device, because Android also provides an emulator which simulates the functionalities of a real device. For the most part, running an application on the emulator is identical to running it on a physical phone, excepting the speed, which is much slower on the emulator. Some notable exceptions are that things which are hard to virtualize, such as sensors or Bluetooth.

The core of Android is designed to be portable, so this operating system can run on all sorts of physical devices, and also, for Android, a device’s screen size, resolution, chipset and other technical characteristics.

I will also present some historical facts about Android, because they are interesting and they offer some perspective on what the future might bring. Android was founded in 2003 in California, and in 2005, Google bought Android and nothing is known until 2007 when Android becomes officially

open sourced. The first Android SDK: 1.0 was released in 2008 and also the first Android running device was released, manufactured by HTC. In 2009 Android goes through an enormous, very fast growth releasing new versions of the operating system and having 20 different types of devices running Android. In 2010 already, Android reaches the second place as the best-selling smart phone platform and there are more than 60 different devices running this operating system. Now, worldwide Android has reached the first place as the best-selling platform for mobile devices, and the numbers keep on growing. The latest version released is Android 4.2.2, in February 2013 and the number of different devices running Android is above 300.

The next step presents how the whole Android system works. This operating system is built on top of Linux and some of the mail reasons for choosing Linux as the base of the Android stack are portability, security and features. Linux is a portable platform which is easy to compile on various hardware architectures, which brings to Android a level of hardware abstractions. Linux is also a highly secure system and that is why Android heavily relies on Linux for security, so that every Android application runs on a separate Linux process with permissions set by the Linux system and Android passes many security concerns to the underlying Linux system. Android uses many very useful features from Linux, such as support for memory management, power management, and networking.

So now that the introduction has been made, and the big image of the Android operating system is shaped, I can start with the technical details which I also used in my project, beginning with: *3.1 Android and Java: The Dalvik VM*.

3.1 Android and Java: The Dalvik VM

Dalvik is a virtual machine, designed specifically for Android, that runs applications and code written in Java. In Java, we write a Java source file,

compile it into a Java byte code using the Java compiler, and then we run this byte code on the Java VM. In Android, on the other hand, things are different, because although we still write the Java source file, and we still compile it to Java byte code using the same Java compiler, this compiled file is recompiled once again using the Dalvik compiler to Dalvik byte code which is then executed on the Dalvik VM. These compilations steps are luckily automated by tools such as Eclipse or Ant. For my project I used the Eclipse IDE or Integrated Development Environment. The plus of using Android Java is that although Java-specific user interface libraries, AWT and Swing, have been taken out, they were replaced with Android-specific user interface libraries, and also Android adds a lot of new features to standard Java, while still supporting most of Java's standard features.

To develop a mobile application, all that there is needed is the Android SDK, or Software Development Kit which comes with a set of tools as well as a platform to run it and see it all work.

I chose the Eclipse IDE because this tool has a lot of time-saving features and also the setting up of the ADT or Android Development Tools which is very intuitively, gives Eclipse a plus.

All this being said, further on I will present some of the most important Android-specific components and their use in my project with examples.

3.2 Application Components: Essential building blocks

Application components are the conceptual items which are put together to create a mobile Android application. The application has to be designed in terms of screens, features and the interactions between them. This is how the building of an Android application works. All these application components are loosely coupled by the applications manifest file, which is a one and only for an

Android application, and it describes each component of the application and how they interact. But more on the manifest file, whose name is "AndroidManifest.xml" in the subchapter dedicated to this indispensable component.

Activities

An activity represents a single screen that the user sees on the device at one time. Typically, an application has multiple activities that are loosely bound to each other, and the user navigates between them. This means the activities are the most visible part of the application. Mostly, an activity in an application is specified as the "main" activity, which is presented to the user, at the beginning, when launching the application.

Because starting an Activity may involve creating a new Linux process, allocating memory for all the user interface objects, inflating all objects from XML layouts (XML layouts will be presented in the next chapter: *3.3 User Interface: What the user experiences*) and setting up the whole screen requires a lot of processing, the Activity Manager was created to take care of the switching between the screens. For example, when a user switches from activity A to activity B, the activity A is set by the Activity Manager in the "back stack", so that, when the user want to switch back from activity A to activity B, this happens very quickly.

To create an activity, there must be created a subclass of the "Activity" class or of another existing subclass of it. So that the Activity Manager knows what to do with an activity when switching it in the foreground or in the background, methods like "onCreate" which is called when the activity is created, "onStop" which is called when the activity is stopped, "onResume" which is called when the activity resumes, or "onDestroy" which is called after the activity is stopped, so that there will remain no trace of that activity, must be implemented.

To start an activity from another one, or from another class, passing an "Intent" that describes the activity to start, to the "startActivity" method makes it possible. The activity can be shut down by calling its "finish" method.

For example, in the figure below is a snippet from my main activity, the "LoginActivity" which is the first activity that is shown to the user when the application is accessed.

```
public class LoginActivity extends Activity {

    Button buttonLogin;
    Button buttonRegister;
    EditText username;
    EditText password;
    EditText security;
    TextView attempts;
    private static int nrAttempts = 3;
    public static String usernameS;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);

        initialize();
    }

    private void initialize() {

        buttonLogin = (Button) findViewById(R.id.buttonLogin);
        buttonRegister = (Button) findViewById(R.id.buttonRegister);
        username = (EditText) findViewById(R.id.editTextLoginUsername);
        password = (EditText) findViewById(R.id.editTextLoginPassword);
        security = (EditText) findViewById(R.id.editTextLoginSecurity);
        security.setVisibility(View.INVISIBLE);
        attempts = (TextView) findViewById(R.id.textViewAttempts);
        attempts.setVisibility(View.INVISIBLE);
    }
```

Figure 3.1: Example of Activity

As I did, as it can be seen in the above example and as it is a good approach, I chose to "do the set up" in the "onCreate" method. Firstly, in the "onCreate" method, the content view of the activity is set. The login layout is

created in the layouts folder of the project, but more on layouts in the next chapter. Next follow the initializations of the components which exist in the login layout. The activity does not know which components of the inflated layout to use, if they are not initialized. The "Inflater" is used to instantiate layout XML files into its corresponding View objects to define the user interface of that activity-screen using the "setContentView" method. The "findViewById" method retrieves the widgets in that user interface, so that the programmer can interact with them programmatically.

A subdivision of Androids activity is the "**Fragment**", which represents a behavior or a portion of the user interface in an Activity. Fragments can be combined in a single activity to build a multi-pane user interface and reuse a fragment in multiple activities. A fragment has its own lifecycle, receives its own input events and works like a "sub activity" which can be reused in different activities. Fragments were introduced in the Android API with the Android 3.0 version, primarily to support more dynamic and flexible user interface designs on large screens such as tablets.

For example, in my project I used fragments to create tabs, where the user can see the sent and the received friend requests. This works much faster than a "TabActivity", which would not be composed of fragments, and which would run slower because the tabs would be loaded separately. Like this I can also reuse my two fragments which are shown in the tabs.

Another example from my project, where I used Fragments, is to change the user interface from the news categories view to the chatting conversations. How I implemented this is: at the beginning the user sees the fragment with the categories of news. If the user wants to talk to a friend, he simply slides the screen to the right, and a Fragment slides until the middle of the screen, where the user can see his friends. By clicking on a friend, he can start the conversation with that friend, and the fragment with the news categories is replaced with the fragment for the conversation. So here, there are three

fragments in a single activity, which can be comfortably reused in other activities, because every fragment is designed as a modular and reusable activity component.

Services

A service does not provide a user interface, and it can be used as an application component which performs long-running tasks in the background. The service can be started from another application component and it will continue to run in the background until it is stopped, even if the user switches to another application. Services are useful if we have actions that we want to be performed for a while, regardless of what is on the screen. The lifecycle of a service is much simpler than the lifecycle of an activity, because the service can either be started or stopped. A service runs in the background, but this does not mean that it runs on a separate thread. If a service is doing some processing that takes a while to complete, such as performing network calls, then the service has to run on a separate thread, because, otherwise, the user interface will run noticeably slower, because if not programmed to run on different threads, services run on the same main application thread as the activities, which is called the user interface thread.

To create a service, there must be created a subclass of the Android "Service" class, or of one of its existing subclasses. In the implementation of a service, the callback methods that handle the key aspects of the service lifecycle of the "Service" class must be overridden, if appropriate, to do the wanted tasks. One of the most important methods to override is the "onStartCommand" method which is called when another application component, such as an activity, requests that the service should be started by calling the "startService" method. Once this method executes, the service starts and runs in the background until the work is done and the "stopSelf" or the "stopService" method is called. Another important method of the service to override is the

"onCreate" method, which is called before the "onStartCommand" method, when the service is first created, to perform the one-time setup procedures. The last call the service receives is the "onDestroy" method, which is called when the service is no longer used and is being destroyed. This method should be implemented to do the cleanup of the resources used in the services.

In my project I used two services. One service starts when the user starts a conversation with a friend, and it checks if the user receives messages from the friend he is chatting with. When a message comes up, it is shown on the user interface in the conversation screen. This service stops as soon as the user closes the conversation. The other service is started when the user minimizes the application, because it verifies whether the logged in user receives any messages from any friend of his. If so, the user receives on his device a notification, seeing who wrote him, and what that friend wrote. By taping the notification, the service stops, and on the users screen the conversation screen appears, where he can respond to the friend.

For both of the presented services I used separate threads, because, as said above, it is the best approach for not blocking the user interface, because both services connect through the network to the server, and sometimes that may take a long time and the user interface would be blocked.

Below I chose to exemplify and explain my "NotificationService" class, which does the work for the Android notifications. Here is a snip of the class:

```
public class NotificationService extends Service {

    static final int DELAY = 3000;
    private boolean runFlag = false;
    private NotificationThread mNotificationThread;

    public static boolean notify = false;
    public static String username;
```

Figure 3.2: Service example

```
public void onCreate() {
    super.onCreate();

    notify = false;
    username = "";

    this.mNotificationThread = new NotificationThread();
}

public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    this.runFlag = true;

    this.mNotificationThread.start();

    return START_STICKY;
}

public void onDestroy() {
    super.onDestroy();

    this.runFlag = false;
    this.mNotificationThread.interrupt();
    this.mNotificationThread = null;
}

private class NotificationThread extends Thread {

    public NotificationThread() {
        super("Notification-Service");
    }

    public void run() {

        NotificationService serv = NotificationService.this;
        while (serv.runFlag) {
            try {

                        ...

                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                serv.runFlag = false;
            }
        }
    }

}
```

Figure 3.3: Service example (continued)

As it can be seen, firstly, in the “onCreate” method, the setups are made and the thread used for the work of the service, is initialized. Next, in the “onStartCommand” method, the run flag is set to true and the thread is started. The “onDestroy” method the run flag is set to false and the thread is interrupted so that everything stops and it is also made empty.

This service is called, as I said before, when the application is minimized by the user. This happens in the following way:

```
case R.id.menu_minimize:
    startService(new Intent(this, ServiceNotification.class));
```

So the service is passed to an “Intent” and then it is started. To understand what is with the intents, follow the next paragraphs.

Intents

Activities, services and broadcast receivers, three of the core components of an application, are activated through messages called “intents”, which are sent among the major building blocks triggering an activity to start up, telling a service to start or stop or being simply broadcasts. An intent object is a bundle of information, which presents information of interest for the component that receives the intent, such as the action to be taken and the data to act on, and also for the Android system, such as the category of component that should handle the intent and instructions on how to launch the target activity.

Processes and Threads

The Android system starts a new Linux process for an application with a single thread of execution, every time when an application component starts, and the application does not have any other components running. By default, all components of the same application run in the same process and thread, called the “main” thread. Else, if the application component starts and there already

exists a process for that application, because another component from the application exists, then the component is started within that process and uses the same thread of execution. Application components can be arranged to run in separate processes, and there can be creates additional threads for any process.

Some tasks require a lot of work, which sometimes takes a lot of time, that is why, in Android, there is the so called "AsyncTask" which allows the developer to perform asynchronous work on the user interface. It performs the blocking operations in a worker thread and then the result is published on the UI thread, without requiring other threads and or handlers in the program. To use this, there must be created a class that extends the "AsyncTask" class and there has to be implemented the "doInBackground" callback method, which runs in a pool of background threads. To update the UI, there has to be used the "onPostExecute" method, which delivers the result from "doInBackground" and runs in the UI thread, so that the UI will be safely updated. The "AsyncTask" class can be called using the "execute" method. This task can be programmatically canceled an any time, from any thread. More about this in the fifth chapter: *5. A client-server mobile application: The connections and how they work together.*

Permissions

Android is a privilege-separated operating system, which means that every application runs with a distinct system identity: Linux user ID and group ID. Androids security architecture is build so that no application, by default, has permissions to perform any operations that would adversely impact other applications, the operating system, or the user, which includes reading or writing the user's private data such as contacts or emails, reading or writing another application's files, performing network access, knowing the location of the user, etc. Android separates all its applications from each other and that is why applications must explicitly share resources and data. In order to do this,

they declare the "permissions" they need for additional capabilities not provided otherwise. Applications declare the permissions they require statically, and the Android system prompts the user for consent at the time the application is installed.

Permissions are declared in the "AndroidManifest.xml" file, which is presented in the next subchapter. At application install time, permissions requested by the application are granted to it by the package installer based on checks against the signatures of the applications declaring those permissions and / or interaction with the user.

For example, in my project I needed several permissions. I used the permission for Internet, because it is a client-server application. The permission for writing to the external storage was used so that a user can save the conversations he wants. Another permission used is for the GPS provider, so that the application can have access to the location of the users.

Android Manifest

The most important part of an Android application is in the "Android.xml" file, because this file put everything together and with the help of this file, the Android system will know what the application consists of, what its main building blocks are, and what permissions it requires. Because of this highly importance of the file, it must be present in every application's root directory, so that the Android system will find it and look into it firstly, and then to the rest of the application code, which would not be possible without the manifest file.

All application's activities, services, permissions must be declared in the manifest file, otherwise, if an activity is not declared in the manifest file, that activity will not be found by the Android system when running the application, so it will not be started and the whole application will be forced to close. If a service is not declared in the manifest file, it will not start with the application.

If permissions needed are not written in the manifest file, those privileges will not be permitted to the application, for example if a file has to be written in the storage of the application, it will not be possible without a permission to do that. Below is the Android Manifest file I used in my project:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="bv.unitbv.chatnews.app"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="14" android:targetSdkVersion="17" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <application android:label="@string/app_name"
        android:icon="@drawable/app_icon"
        android:theme="@style/CustomBlackStripesTheme"
        android:allowBackup="true" >

        <activity
            android:name=".LoginActivity"
            android:label="@string/app_name"
            android:screenOrientation="portrait"
            android:configChanges="keyboardHidden|orientation" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".EditProfile"
           android:label="Edit Profile"
           android:screenOrientation="portrait"
           android:configChanges="keyboardHidden|orientation"></activity>
         <activity android:name=".news.PullToRefreshNewsActivity"
           android:label="News"
           android:screenOrientation="portrait"
           android:configChanges="keyboardHidden|orientation"></activity>

                              ...
         <service android:name=".ChatService"></service>
         <service android:name=".NotificationService"></service>

    </application>
</manifest>
```

Figure 3.4: AndroidManifest.xml

In the second line, I named the package for the application, which serves as a unique identifier for the application.

I used in my project, the latest Android releases, as it can be seen the minimum API level is 14 which is for the Android 4.0 version, and the targeted API level is 17, which is for the Android 4.2 version. This is declared in the "uses-sdk" tag, and this means that the application will run on a device which has the Android OS version 4.0 or higher.

Next I declared the permissions I used for my project in the "uses-permission" tag.

In the "application" tag, there are declared each of the application's components and this tag has attributes that can affect all the components. Firstly, the main activity is declared, which is mentioned in the "intent-filter". The intent-filer describes the different capabilities of a component. Next are declared the other activities. I wanted my application to run only in the "portrait" mode and that is why I set the attribute "android:screenOrientation" to "portrait".

As I said before, if a service is not declared in the manifest file, it does not start, so, after I declared the activities, I also declared the two services I use.

In this subchapter I chose to present the applications components technically, and now, after the "underground" is built, I start to present the "foreground": the user interface.

3.3 User Interface: What the user experiences

The UI is everything the user sees and interacts with. In Android, there are a variety of pre-built UI components, such as structured layout objects and UI controls that allow the developer to build the graphical user interface for an

application. Dialogs, notifications and menus are also modules provided by Android.

There are two ways to create a UI in Android: declarative or programmatic, but mostly, they are used together. The declarative way is to use the XML to declare how the UI will look like, by writing specific tags, but this approach, although is more intuitive, it does not provide a good way of handling user input, and that is where the programmatic approach comes in. The programmatic UI involves writing Java code to develop the UI, which is similar to Java AWT or Swing. The best approach is to use both, because an XML file should contain everything that is static: the layout of the screen, the widgets, etc. and programmatically, there can be declared everything that is non-static and their actions when something happens on the UI, for example a button is clicked, a list item is selected, etc.

A screen which a user sees is firstly built out of a layout which can contain other layouts and the screens components. That is why, the first user interface component I will present and exemplify with code from my project is the layout.

Layouts

Android organizes its UI elements into layouts and views, and everything the user sees, such as a button, label, or text box is a view.

The XML layout file, which has to be saved in the res/layout folder of the Android application, compiles into a “View” resource which can be loaded from the application code in the activities “onCreate” callback method implementation by calling the “setContentView” method and by passing to this method the reference to the layout resource in the form of: R.layout.layout_file_name. If the other method is chosen, and the XML file is

not used, or is just partially used, the elements can be created programmatically by creating an instance of that element and initializing it.

Below I want to explain how I made the layout for a list item using XML and Java code for the same layout.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/username"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_toRightOf="@+id/text"
        android:text="TextView"
        android:textSize="15sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@+id/avatar"
        android:text="  " />

    <ImageView
        android:id="@+id/avatar"
        android:layout_width="60sp"
        android:layout_height="60sp"
        android:layout_marginLeft="0dp"
        android:scaleType="fitCenter" />

    <ImageView
        android:id="@+id/cancelRequest"
        android:layout_width="15sp"
        android:layout_height="15sp"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:src="@drawable/cancel_friend_request" />

</RelativeLayout>
```

Figure 3.5: Layout example

As there can be observed, the layout is a “RelativeLayout”, which enabled me to specify the location of the child objects relative to each other: the “username” text view is to the right of the “avatar” image view. This is one of the main layouts (linear layout, relative layout, web view) that are used more frequently than others. A linear layout organizes its children into a single horizontal or vertical row and it creates a scrollbar if the length of the window exceeds the length of the screen. A web view displays web pages. The relative layout, which I also used, is very powerful, because it does not require to nest unnecessary layouts to achieve a certain look, and like this, the total number of widgets that need to be drawn is minimized and this improves the overall performance of my application.

The attributes can be set by means of setter methods. According to the type of the widget, in the XML file there can also be set different attributes.

To access a child programmatically, it needs to have the “id” attribute set. This is how the children of this layout are accessed and instantiated:

```
username = (TextView) convertView.findViewById(R.id.username);
avatar = (ImageView) convertView.findViewById(R.id.avatar);
cancel = (ImageView) convertView.findViewById(R.id.cancelRequest);
cancel.setOnClickListener(new OnClickListener() {

      public void onClick(View v) {

                  ...
    }
});
```

Figure 3.6: Accessing and instantiating the widgets

For example, above, there can be seen, that the cancel “ImageView” has an “onClickListener”, so that, when the user taps on that image, the “onClick” method is called and the application performs the actions written in the method.

Input controls

The input controls are the interactive components in the application's user interface: buttons, text fields, seek bars, checkboxes etc. The input controls can be added to the UI through the XML file, or programmatically.

Input events

The View class provides the ability for the developer to know how to programmatically access the View object with which the user interacted. Thanks to this, there can be implemented methods such as "onClick", "onLongClick", "onTouch", "onKey", and other event-specific methods.

Menus

The options menu is a group of menu items which are specific for an activity, not for the whole application. Each activity must implement its own menu. These items, from the options menu are presented in a so called "action bar" which is at the top of the screen, and it can be shown or hidden, depending on what every activity specifies. The action bar provides a combination of action items which are shown on top the screen and action items which are presented in an overflow menu. The menu, like the other application components, can be created from an xml file, specially create for the menu, or it can be created programmatically.

There follows an explained example, on the next page, of how I created the options menu for my application. I defined the menu in XML, but I also created items and added them to the existing ones, programmatically. Android provides a standard XML format to define menu items. To have an activity create its menu as the programmer wishes, the "onCreateOptionsMenu" method has to be overridden. So that something happens when an option from a menu is

selected, the “onOptionsItemSelected” method has to be overridden to do the specific work for the taped item. The item is also found by its id, so again, the “id” attribute is a very important one.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@+id/menu_showRequets"
          android:icon="@drawable/megaphone_small"
          android:showAsAction="ifRoom"
          android:title="See Requests" />

    <item android:id="@+id/menu_addFriends"
          android:icon="@drawable/friends_add_small"
          android:showAsAction="ifRoom"
          android:title="Add Friends" />

    <item android:id="@+id/menu_minimize"
          android:icon="@drawable/mini_minimize"
          android:showAsAction="ifRoom"
          android:title="Minimize App and receive notifications" />

    <item android:id="@+id/menu_editProfile"
          android:title="Edit profile" />

    <item android:id="@+id/menu_news"
          android:icon="@drawable/rss"
          android:showAsAction="ifRoom"
          android:title="News" />

    <item android:id="@+id/menu_notify_sound"
          android:title="Choose notification sound" />


</menu>
```

Figure: 3.6: XML Menu

In the above XML I have defined a menu using the “menu” tag, and 6 items using the “item” tag. For some items, besides the “id” attribute, I also set the “icon” attribute, because I want those items to be shown in the action bar with that icon. The “showAsAction” attribute has to be set in order to show an item on the action bar. I have set this attribute to “ifRoom”, so that the item is shown on the action bar only if there is enough room, because on a smaller

device, there would not fit so many items on the action bar. The “title” attribute has to be set for the items which will not be shown in the action bar, so that in the menu, they have a title. This title also has to be set for the items which will be shown in the action bar if there is room, because if there is not enough room, they have to have a title in the menu. These menu is created using the “onCreateOptionsMenu” method in an activity:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater menuInflater = getMenuInflater();
    menuInflater.inflate(R.layout.prefmenu, menu);
    return true;
}
```

Figure 3.7: Create Menu

The XML with the menu is called “prefmenu” and it is inflated to Java code with the “MenuInflater”. The items presented above can be identified using the ids in the implementation of the “onOptionsItemClickListener” method:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.menu_addFriends:
        startActivity(new Intent(this, AllUsersActivity.class));
        return true;
    case R.id.menu_showRequets:
        startActivity(new Intent(this, FragmentTabsPager.class));
        return true;
    case R.id.menu_editProfile:
        startActivity(new Intent(this, EditProfile.class));
        return true;
    case R.id.menu_minimize:
        Intent startMain = new Intent(Intent.ACTION_MAIN);
        startMain.addCategory(Intent.CATEGORY_HOME);
        startMain.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(startMain);
        startService(new Intent(this, ServiceNotification.class));
```

Figure 3.8: onOptionsItemSelected

As I said, I also added items to the menu programmatically:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    if (tv.getVisibility() == View.VISIBLE) {
        MenuItem item = menu.add("Save Conversation");
        item.setOnMenuItemClickListener(new OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {

                            ...


            }
        });


        MenuItemCompat.setShowAsAction(item,
                MenuItemCompat.SHOW_AS_ACTION_NEVER);
```

Figure 3.8: Adding menu item programmatically

I chose to add programmatically an item to the menu, because, using fragments, in the fragment for the news, I did not need the “Save conversation” menu item, but when chatting, I needed it, so I created a new “MenuItem” and set its title in the constructor. This menu item also has to do some action when taped, so I implemented its “OnMenuItemClickListener” to save the conversation to the device.

Dialogs

Dialogs are small windows that prompt the user to make a decision or enter additional information. A dialog’s window does not fill the whole screen and is used to require the user an action before they proceed.

Dialogs can be created programmatically with any custom design the user wants. In my project I implemented a custom dialog when the user wants to send a friend request to another user. From the list of users, if he taps on a list item, the dialog pops up and shows details about the user and also a positive button, which sends the friend request, and a negative button which closes the dialog and returns the user to the list of all users.

[eventual, screenshot cu dialogul]

Notifications

The notification is a message, which can be displayed to the user outside of the application's normal user interface. When the system is told to send a notification, it is firstly shown in the notification area of the mobile device, and to see the notification's details, the user has to open the notification drawer.

The "Notification" class can already be found in the Android API, and extended from there. I used the notification so that, if the user is still logged in, although he is doing something else on his mobile device, or he is doing nothing with it, he will still be able to receive messages from his friends, which will be shown in the notification area. By sliding the notification drawer the user can see who wrote him and what the message is.

A notification can also have different actions to perform when taped. In my project, when taped, the notification sends the user to the UI fragment where he can chat with the friend who wrote him. This is called a "PendingIntent".

This is how I prepared the pending intent:

```
public PendingIntent getPendingIntent() {
    return PendingIntent.getActivity(this, 0, new Intent(this,
            FragmentChatNewsActivity.class), 0);
}
```

Figure 3.8: PendingIntent

And this is how I attached the above pending intent to my notification:

```
PendingIntent pi = getPendingIntent();
notification.setLatestEventInfo(this, username, message, pi);
```

Figure 3.9: Attach PendingIntent to Notification

Toasts

A “Toast” looks like a small pop up, which is used to provide information to the user interface. The information can only be in the format of a “String”. The Toast is shown in the screen on the activity which it belongs to, and it only wraps its content, it is either bigger, nor smaller than the amount of text is has to display. The activity on top of which the Toast is shown remains visible and interactive.

For example, I chose to use the Toast to show a message that the conversation has been successfully saved to the device or not, when the user performs this action.

```
Toast.makeText(FragmentChangeActivity.c,
        "Conversation SAVED successfully !",
        Toast.LENGTH_SHORT).show();
Toast.makeText(FragmentChangeActivity.c,
        "File could not be created ! Conversation NOT saved !",
        Toast.LENGTH_SHORT).show();
```

Figure 3.10: Toast save conversation

3.4 Media Playback: The multimedia framework

Android provides an API also for the multimedia framework. Using the "MediaPlayer" class and the "AudioManager" class, sounds can be programmatically set to play whenever the developer desires. The files that can be played with the help of the methods of the above mentioned classes are the ones located in the raw resources folder.

In my project the user can chose between the sounds provided by my application, by listening to them, which should play when a notification comes up. The sound is played when the notification comes up as follows:

```
MediaPlayer mPlayer;
mPlayer = MediaPlayer.create(this, R.raw.sound1);
mPlayer.start();
```

Figure 3.11: Media Player

The "create" method is used to create a sound from the raw resource of the application, and with the "start" method, the sound starts to play.

3.5 Location API: To know where you are

Another functionality which my application provides is the ability for the users to see where their friends are located. The location service which runs on every mobile device automatically tracks down the location of the user through the internet or through the incorporated GPS. The specific permissions, as mentioned in the "Permissions" subchapter have to be set, so that the application becomes location aware. After this, all there remains to be done is to instantiate the "LocationManager" class and its characteristics.

On the next page is the snipet with the example of how I used this in my project, to identify, as a user loges in, what his location is, and send it to the

database, from where it is the retrieved in the views of the friends, so that they now each other's location. But more on the connection to the database follows in the fifth part of this documentation.

```
private LocationManager locationManager;
private static LocationListener mLocationListener;
private static Criteria criteria;
locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_FINE);

mLocationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
        if (location != null) {
            double lat = location.getLatitude();

            double lng = location.getLongitude();

ConnectivityManager cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo netInfo = cm.getActiveNetworkInfo();
if (netInfo != null && netInfo.isConnected()) {
    locationManager.requestLocationUpdates(
            LocationManager.NETWORK_PROVIDER, 1000, 100,
            mLocationListener, getMainLooper());
} else {
    locationManager.requestLocationUpdates(
            locationManager.getBestProvider(criteria, true), 1000, 100,
            mLocationListener, getMainLooper());
}
```

Figure 3.12: Location awareness

First, I initialized the location manager with the location service of the device, and next I set the "Criteria" to be at a very good accuracy: "ACCURACY_FINE". With the help of the "LocationListner" class I see if the location of the user has changed, so that it can be updated. The "ConnectivityManager" class helps me to check if the device has the network connection available. If so, the location manager will take the location's coordinates of the user from the Internet, otherwise, it will look for the GPS on the device and it will use it to get the location of the user.

Now that every aspect of the front-end part of my application has been presented, further comes, the presentation of back-end technologies and their implementation in my project.

App signing
App distribution

Chapter 4

Back-end technologies: Strong server-side for a stable client-side

Java Enterprise Edition was developed, in the first place, to be used on central servers, where it brings a variety of services for the clients, which communicate through TCP/IP or HTTP protocols.

Client-server model applications can be built waving the Java EE technologies as the back-end solution. A client-server model application is a distributed application, in which the client is the service that makes requests for information to the server, and waits for the response from it so that it can move on with the processing, and the server is the service which receives the request from the client and processes it by gathering the requested information and send it to the client that requested it. It is important that the client needs to be aware of the address of the server and its existence, while the server does not need to know either about the existence of the client, nor about its address so that it can accept a connection. Once the connection is successful, the client and the server can send and receive messages and handle the sent and received information.

A server can be connected to a database and can communicate with it, so that the database sends answers to the server and the server send it to the client.

My project uses this client-server architecture model.

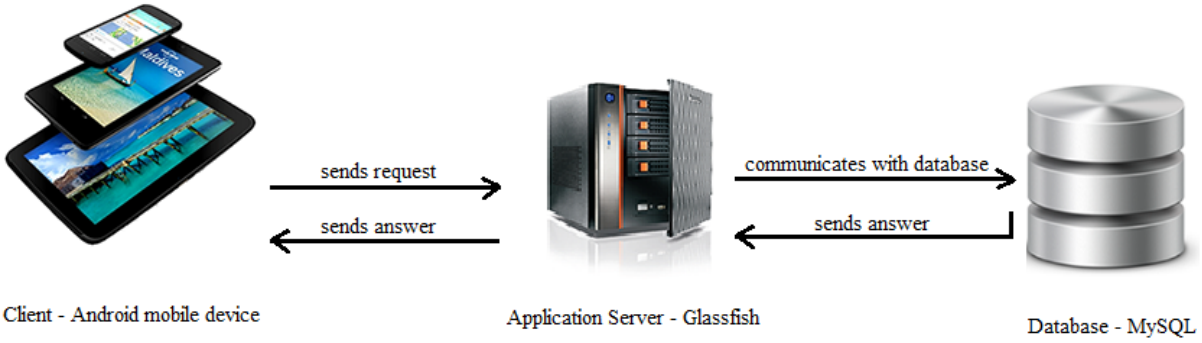


Figure 4.1: Client-server architecture in my project

In the above figure we can see how my application works: the client, a mobile device, sends a request to the server, for example, when a user wants to log in, the username and the password are sent to the server, which communicates with the database to see if the received data is valid, and then the database sends the answer to the server, which processes it and sends it further to the client which also processes it and shows it as a result on the user interface.

4.1 The Glassfish application server

The Glassfish application server is an open-source project started by Sun Microsystems for the Java Enterprise Edition platform and now sponsored by Oracle Corporation. Because the Glassfish server was implemented for Java EE, it supports Enterprise JavaBeans, Java Persistence API, Java Server Faces, Java Messaging System, Remote Method Invocation, etc. so that enterprise applications can be created.

Further I will present the Glassfish server supported technologies which I used in my server-sided application.

Enterprise JavaBeans

EJB is the part that manages the server-sided application logic, which is divided in components. It is the server-side model which manages the business logic of the application. In enterprise applications, the EJB specification has become a standard way to implement the business logic of the enterprise applications. EJB's are mostly used to handle persistence in a standard way. This is the approach I also used in my application to implement its business logic.

EJBs can be stateless, stateful, singleton session beans or message driven beans. Stateful session beans, as their name says, are business objects that have a state, and they keep track of the identity of the client along a session, to know with which they have to deal and make operations for. A stateless session bean has no state associated with it, and is thread-safe, because if concurrent access to the same bean is attempted, the container automatically routes each request to a different instance. Singleton session beans have a global shared state within a JVM. Message driven beans are triggered by messages instead of method calls.

In my project, I used stateless beans to manage the methods invocated by the client for a user, for a friend, for an invite and for a message.

My back-end logic works combined with the JPA persistence context, which provides the persistence context, the queries to the database and the entity management.

Java Persistence API

O/RM or object/relational mapping is a programming technique used to convert data stored in two different environments : relational database systems like MySQL, Oracle Database, etc. and object oriented programming languages like Java, C++, etc. This correlation results into a virtual database of objects

which can be accessed from the programming language. Using this technique objects can be translated into structures that can be stored in the database and can be recovered from there easily, by keeping their properties and associations. This kind of objects is called persistent objects. An O/RM reduces most of the time, the volume of code necessary to be written and increases the robustness of the program.

The Java Persistence API provides such an object/relational mapping facility for managing relational data in Java. In order to do this, JPA also provides Persistence Entities which are each mapped to a single table in the database. For example, in my project I used 6 entities which are mapped on 6 tables in the database: “Appuser”, “Friend”, “Request”, “Message”, “Rssfeed” and “Rssfeedlist”. The database to map to is known from the persistence file which the project must have:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" >
    <persistence-unit name="ChatNewsRestWebService">
        <jta-data-source>jdbc/licentaRes</jta-data-source>
        <class>model.Appuser</class>
        <class>model.Invite</class>
        <class>model.Friend</class>
        <class>model.Message</class>
        <class>model.Status</class>
        <class>model.Rssfeed</class>
        <class>model.Rssfeedlist</class>
      <properties>
        <property name="eclipselink.jdbc.cache-statements" value="false"/>
        <property name="eclipselink.jdbc.native-sql" value="false"/>
      </properties>
    </persistence-unit>
</persistence>
```

Figure 4.2: persistence.xml

The “persistence.xml” file is a standard configuration file for the JPA. This file defines a persistence-unit with a unique name, which in this case is my projects name. Next the Java Transaction API or JTA data source has to be set. This data source has to be already declared in the JDBC resources of Glassfish,

so that it is found there, and the application can communicate with the database set in the resource's properties. After this, the classes which are entities and have to be managed by the persistence context are declared in the "class" tag.

The persistence context is referenced using the "@PersistenceContext" annotation, when declaring the "EntityManager" interface which will be used to interact with the persistence context. The EntityManager API is used to persist and create entity instances and also for removing them. The persistence unit in the "persistence.xml" file defines the entities that can be managed by the EntityManager.

```
@Stateless
public class AppuserEjb {

    @PersistenceContext
    EntityManager em;

    public Appuser getAppuser() {
        return (Appuser) em.createQuery("Select u from Appuser u")
                .getResultList().get(0);
    }
    public String register(Appuser appuser) {
        try {
            em.persist(appuser);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "User successfully registered !";
    }

    public Appuser find(String username, String password) {
        try {
            Query query = em
                    .createQuery("Select u from Appuser u where " +
                            "u.username = :username AND u.password = :password");
            query.setParameter("username", username);
            query.setParameter("password", password);
            return (Appuser) query.getSingleResult();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

Figure 4.3: EntityManager use

In the example on the page before there can be seen that the "EntityManager" interface is associated with the "@PersistenceContext" annotation. Like this, the interface does not need to be initialized with the "new" construct, because the initializations are already done by the annotations according to the "persistcence.xml" file. The "register" method receives an Appuser object as a parameter and to send this object and persist it to the database, the only thing that needs to be done with the entity manager is to call the "persist (object_to_persist)" method on the entity manager object. The entity manager can also be used to retrieve objects from the database. In the "find" method I try to search the user by its username and password by creating a "Query", which, if found, returns the object that corresponds to the sent username and password. This is how the communication with the database works.

4.2 MySQL Database

The MySQL database is the world's most popular open source database relational database management system. The reason why MySQL is the chosen one, for so many developers, is because it is cheap and it does not need a lot of resources. MySQL is one of the fastest relational database management systems and it presents very good performance, and is also very easy to use comparing to other RDBM systems. As it name says, MySQL uses the Structured Query Language or SQL, which is the standard that is the most used nowadays. Some of the capabilities are: MySQL is multi-threaded and it can be used in different ways: for graphical applications, web applications, along with different programming interfaces of C, C++, Java, PHP, etc. MySQL can also be used in the network, it is very portable, of small dimension, and it has a consistent documentation.

MySQL works in a network based environment and has a client-server architecture, so MySQL is actually composed out of more components. The

MySQL server is the component which represents the application that responds on the requests and manages the access to the disk or to the memory and is multi-threaded, which means that it supports multiple users simultaneously. Another component is represented by the client applications which were created so that the developers can access and process the data from the database. Other components with which we do not interact are independent programs which for example verify the tables and make corrections where there is needed.

Of course, the client application and the server application do not have to be on the same computer, and they do not even have to run on the same operating system.

These are the main reasons why I chose to work with the MySQL database.

4.3 Web Services: REST Web Services

The Java EE web services technologies include the Java API for XML Web Services (JAX-WS) and the Java API for RESTful Web Services (JAX-RS).

Like the authors of "The Java EE Tutorial" presented on the Oracle website define: *web services are client and server applications that communicate over the World Wide Web's (WWW) HyperText Transfer Protocol (HTTP). As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services*.

JAX-WS use XML messages, which are written in the Simple Object Access Protocol or SOAP standard, and include content about the abilities of the service, which is machine-readable, and written in the Web Service Description Language or WSDL.

JAX-RS provides the functionality for Representational State Transfer: RESTful web services. Different from the JAX-WS technology, REST is often better integrated with HTTP that SOAP and do not require XML messages or WSDL service-API definitions. In my project I used this type of technology, with the help of the Jersey project, which is the production-ready reference implementation for the JAX-RS specification and it implements the support for the annotations defined in the JAX-RS specification, to make it easier for developers to build RESTful web services with Java and the JVM. Developing RESTful web services uses the existing Internet Engineering Task Force or IETF standards with a lightweight infrastructure which allows the service to be built with minimal tooling. I chose this specification, because my web service is completely stateless, because the client's interaction with the server does not have to suffer if a reconnection to the server is required after restarting the server or losing the internet connection. The client and the server application, both need to be aware of the schemas used which are sent and received from each other. For example, from the server I send to the client a JSON object which has specific tags of my class, of which the client is aware and knows how to process meaningfully. On the other way, the client sends an URL to the server which is correctly parsed at the server. More about the connection follows in the next chapter. Also, REST is mostly useful for PDAs or mobile phones, and this is another reason why I chose to implement this kind of web service.

Using the REST architecture bring many advantages, like the one that the client application and the server application can be developed independently, they just need to have a uniform interface about which both are aware. Portability of the client code is improved, thanks to that fact that clients do not

keep track of the storage of the data, which remains internal to each server, and the server is not concerned about the user interface which brings scalability. For example, my Android client application is not aware of how the data is stored, it just sends requests to the server, which does not have to be always the same, thanks to the fact that the user interface is not concerned about the database. On the other side, the server receives requests, which does not matter from what kind of client application came, if the common uniform interface is understandable and can be parsed meaningfully. So, like this, my server application can easily be used, for example, for a desktop application, not only for a mobile client application. The service being stateless is also another advantage, because the server does not need to store any data between requests, because each request from any client contains all of the information needed for the server, so that it knows what answer to send back to the client. Like this, it was easy for me to log in from more devices and send different requests to the server without having any problems about the data being merged, or confused or mixed up, which means that the application is also very stable and reliable.

As it can be observed the unified interface between the client and the server application is very important. This aspect will be treated detailed in the fifth part of the documentation which follows.

Chapter 5

The connection: How everything works together

As I already presented until now, my whole project is built out of two applications: one on the server side, one on the client side. The client side is represented by the Android application and the server side by the RESTful web service application.

First, I will present what the client does to send requests, and process them for the server, next what the server does to receive these requests and process them so that it takes the correct actions with the received information. After that, I will also explain and exemplify the part where the server processes the answer and sends it to the client, where the client must again parse the received answer.

A REST web service uses the specific operations of the HTTP protocol and it can understand the methods like GET, POST, PUT, PATCH and DELETE. This is why the client also needed to know this type of protocol and its main methods. Hypertext Transfer Protocol or HTTP is an application protocol of communication between a web client and a web server. The HTTP communication starts from the client, which initiates a HTTP request to a

precise server. The server processes the request and sends an answer back to the client, which is called a HTTP answer. A HTTP request to a web server contains: the resource requested by the client, which is in the form of a path to a file, the type of the request: GET or POST, HTTP headers for example the host and the user-agent, and also information introduced by the user through html forms or through an Uniform Resource Locator or URL.

5.1 Client → Server, Server → Client

So that the client reaches the correct resources on the server with the URL it knows, on the on the server application there has to be created or edited the "web.xml" file, so that the server knows where to direct all the requests from the client. The Jersey REST Service, which I implemented in my server application must know where it finds the resources which are available to the client application. In the XML file, I defined the Jersey REST Service from the "com.sun.jersey.spi.container.ServletContainer" class, and then in the servlet-mapping section there had to be created and set a global URL pattern. Below is my "web.xml" file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.co
  <display-name>JerseyRESTServer</display-name>
  <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>bv.unitbv.chatnews.rest</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Figure 5.1: web.xml

The “init-param” allows us to let the servlet container know that, at the request from the client, it should use the resources found in the classes of the “bv.unitbv.chatnews.rest” package. Next, the “servlet-mapping” creates the global URL pattern, which, in my case, will consist of: the IP address of my computer, on which I have the server and the database, followed by the port on which my application is running, and by the name of the web application project, and after that comes the URL pattern set in the “web.xml” file, followed by the specification of path to the resource that has to be used in order to do the correct action with the information.

I will start by explaining how the client connects to the server based on an example about how the user can log in to the application. When the application is opened, the user is prompted to enter its username and password, or to register if the user does not have an account. After entering the username and the password, the user taps the “Log In” button. This is what happens on the user interface, but in the back, the method for the pressing of the “Log In” button is called and this triggers the connection to the server:

```
public void onClickButtonLogin(View view) {
    if (security.getVisibility() == View.INVISIBLE) {
        WebServiceConnection ws = new WebServiceConnection(WebServiceConnection.POST);

        ws.post("username", username.getText().toString());
        ws.post("password", md5(password.getText().toString()));

        ws.execute(new String[] { WebURLHelper.getMainURL() + "/login" });
```

Figure 5.2: Connection start

In the above figure we can see that firstly, a new instance of the “WebServiceConnection” class which takes the parameter that the method will be a “POST” method. This class is presented with details below. Next the class has a method called “post” which, with the help of the “BasicNaveValuePair”

class binds the name, which is the first parameter, with its value, which is the second parameter. Next, the name of the resource to be accessed on the server is set. The "getMainURL" method returns the following URL: "http://79.116.198.156:8080/ChatNewsRestWebService/rest". The class which does the connections, the "WebServiceConnection" extends the "AsyncTask" class from Android, so that the connection is not done on the same thread with the user interface. Like this, the user interface is not blocked, the task which has to send a request to the server and wait for the response runs asynchronous and the whole result is published on the UI thread. The reason why I start the asynchronous task as seen above, with the "execute" method, although in the "WebServiceConnection" there is no method called "execute", is that this method is specific for the starting of an asynchronous task in Android. The asynchronous task in Android consists of four steps. The first step is the "onPreExecute" method, which normally is used to set up the task by showing a progress bar in the user interface. The second step is the "doInBackground method" which is invoked on the back thread immediately after the finishing of the "onPreExecute" method, and is used to perform the background actions which may take a long time. In my case, the "doInBackground" method takes the URL sent from the taped button, in this case, and, according to the type of method parses the url and send it to the server as a POST or as a GET task. In the example above, there is a post task, where a "HttpPost" object is created with the received URL and then, from it is parsed into an entity with the help of the "setEntity" method called on the "HttpPost" object instance. Then the "HttpClient" object is invoked to execute the request using the default context:

```
case POST:
    HttpPost httpPost = new HttpPost(url);

    httpPost.setEntity(new UrlEncodedFormEntity(params));

    httpResponse = httpClient.execute(httpPost);
    break;
```

Figure 5.3: client POST method

When a GET method is called by the user interface, the parameter of the WebServiceConnection has to be GET and no NameValuePairs have to be set, just the URL for the resource has to be known. In the asynchronous task, there will be created a “HttpGet” object with the specfied URL, and calling the “execute” method on the “HttpClient” with the “HttpGet” object as the parameter, will do all the work needed.

```
case GET:
    HttpGet httpGet = new HttpGet(url);
    httpResponse = httpClient.execute(httpGet);
    break;
```

Figure 5.3: client GET method

Now it is up to the server to do the correct job: the REST web services receives the URL with the mapped pattern in the “web.xml file”: “http://79.116.198.156:8080/ChatNewsRestWebService/rest/login” and like this, the server knows that it has to access the “login” resource. How does the server do that? JAX-RS has a set of annotations which help make this very easy. The “@Path” annotation has the value of a relative URI which indicates where the Java class will be hosted or a method. The “@GET” annotation is a request method designator which corresponds to the HTTP GET method, and it will process the HTTP GET requests. The “@POST” method is the request method designator, which corresponds to the HTTP POST method, and it handles it requests. The “@Consumes” annotation is used to specify the MIME media type which can be consumed by the resource annotated, which was sent by the client. The “@Produces” annotation specifies the MIME media type a resource can produce and send back to the client. This means, that in order for the server to know the resource to use, it has to be annotated properly. For example, on the next page, in the snip of my login resource, the class is annotated with the “@Path(“/login”)” annotation, so that the server knows, that with the received URL, it has to access this resource.

```
@Path("/login")
public class AppuserLoginRessource {

@POST
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
@Produces(MediaType.APPLICATION_JSON)
public Answer loginAppuser(MultivaluedMap<String, String> appuserParams) {

      String username = appuserParams.getFirst("username");
      String password = appuserParams.getFirst("password");
```

Figure 5.4: log in method on server

As there can be observed, the login method is annotated so that it knows that it has to “consume” an “URLENCODED” message from the client, which contains the “MultivaluedMap” with the parameters: username and password which are parsed into simple strings. After sending the information to the database and verifying if the information is valid, the server “produces” the answer and sends it to the asynchronous task which is waiting at the client. The MIME type which is sent to the client is a JSON. JSON is presented at the end of this chapter.

Finally, step two of the asynchronous task at the client is done, it receives the answer from the server and moves on to step three: the “onProgressUpdate” method, which can display on the user interface any form of progress while in the background, the task is still working. And step four, the last one, is invoked by the UI thread after the background work is done and the result of the background computation is passed to this step as a parameter. So at the client, I receive a JSON object which now needs to be correctly parsed so that it can be further processed.

And this is how the connection is done and everything works together without any kind problems.

In the next, and last, subchapter I will present what JSON is and how I used it in my client application.

5.2. JSON: JavaScript Object Notation

JSON is a lightweight data-interchange format which is very easy for humans to read, write and understand, but also for the machines to parse and generate. As it name says, JSON is based on a subset of the JavaScript Programming Language. JSON is built on two structures: a collection of name/value pairs which is called object or and ordered list of values which is called array. The JSON format is often used for serializing and transmitting structured data over a network connection, and it is mostly used to send data between the server and a client application.

Java supports the JSON format. In my project I needed to use the "JSONObject" class and the "JSONArray" class. The "JSONObject" class represents an unordered collection of name/value pairs. Its writing form is between curly braces, with colons between the names and values, and commas between values and names. The values supported in a "JSONObject" are: Boolean, JSONArray, JSONObject, Number and String, or the JSONObject.NULL object. The "JSONArray" class is an ordered sequence of values, which writing form is a string wrapped in square braces with commas between the values. The "JSONArray" class supports values like: Boolean, JSONArray, JSONObject, Number and String, or the JSONObject.NULL object.

On the next is a snip of how an answer from the server is parsed at the client. So the server sends a JSON to the client, and the client processes it meaningfully. The client receives the answer in the "onPostExecute" method of the asynchronous task which calls a method on the UI thread. In the example I will explain every possibility that can occur when receiving the message that a user can login or not. It can happen that the user does not exist, or he introduces a wrong password, or he introduces everything correct and can log in to the application.

```
public void handleResponse(String response) {

    try {
        JSONObject jso;
        jso = new JSONObject(response);

        String answer = jso.getString("answer");

        if (answer.equalsIgnoreCase("User " + username.getText().toString()
                + " doesn't exist !")) {
            Toast.makeText(getApplicationContext(), answer,
                    Toast.LENGTH_LONG).show();
        }

        if (answer.equalsIgnoreCase("Incorrect password !")) {
            if (nrAttempts == 1) {
                Toast.makeText(getApplicationContext(), answer,
                        Toast.LENGTH_LONG).show();
                attempts.setVisibility(View.INVISIBLE);
                security.setVisibility(View.VISIBLE);
                password.setEnabled(false);
                username.setEnabled(false);
            } else {
                Toast.makeText(getApplicationContext(), answer,
                        Toast.LENGTH_LONG).show();
                attempts.setVisibility(View.VISIBLE);
                nrAttempts--;
                if (nrAttempts == 2) {
                    attempts.setText("Number of attempts left: "
                            + nrAttempts);
                    attempts.setTextColor(Color.rgb(255, 165, 0));
                } else if (nrAttempts == 1) {
                    attempts.setText("Number of attempts left: "
                            + nrAttempts);
                    attempts.setTextColor(Color.RED);

                }
            }
        } else if (answer.equalsIgnoreCase("Success !")) {

            LoginActivity.usernameS = username.getText().toString();
            Toast.makeText(getApplicationContext(), answer,
                    Toast.LENGTH_LONG).show();
            Intent intent = new Intent();
            intent.setClass(this, FragmentChangeActivity.class);
            startActivity(intent);

            startService(new Intent(this, ServiceA.class));
```

Figure 5.3: JSON parsing

First, a new JSONObject is created from the "response" string which contains the answer from the server. The "response" string is actually a JSON text string. For example, this is how the JSON string looks like, for the array with all users from the database: (it is only a snip)


```
{"allusers":
[{"avatar":"iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAYAAAB5fY51AAAACXBIWXMAAAsTAAALEwEA

                    ...
Z2mekk6bBpguVDBSE3Ai+TeFs0I+1bW/w8blxk/SyWuMgAAAABJRU5ErkJggg==","email":"amicu"
,"firstname":"amicu","id":"22","lastname":"amicu","password":"c40e13ff3df80f2d0e
1aabd82907a4d0","phone":"0985757474","securityphrase":"c40e13ff3df80f2d0e1aabd82
907a4d0","username":"amicu"}]}
```


Figure 5.4: JSON string

So this is the kind of message the client receives. In the figure 5.3, there can be observed that after creating a new JSONObject from the received string, I parse the answer into a string so that I can use it easier. If the username does not exist, it means that the received string looks as follows: {"answer":"User username does not exist!"}. If it is this case, then, a Toast is shown on the UI, which informs the user that the data introduced is not correct, because the user he introduced does not exist. Otherwise, if the answer is "Incorrect password!", again the user is informed with a toast that he should rewrite the password correctly. And in the last case, if everything is correct and the answer is "Success!" from the server, than the corresponding activity is started and the user is logged in.

Only the client application is aware of the user which is currently logged in on the device it is running, the server, as mentioned, does not keep any state.

Now that every technical aspect has been presented, explained and exemplified, the next chapter presents the application with its functionality and many screenshots to make it easier to understand.

Chapter 6

The application: Stay informed

As I presented in the chapters above, the goal of my application was to connect many edge technologies and make the work together optimal. The client uses the following main technologies: Java 7, Android 4.0, RSS, connected with the Glassfish server which uses: Java 7, REST Web Service, JPA, MySQL, EJB. I put this technologies together to build an application for communication and keeping up to date with anything that happens around the world or with friends.

The application runs connected to a server, that is why it needs to have access to the network, so that in can access the server kept in one place from any wireless or 3G network around the world.

Below I will start presenting the functionality and the use case of my application along with screenshots.

6.1 Use case

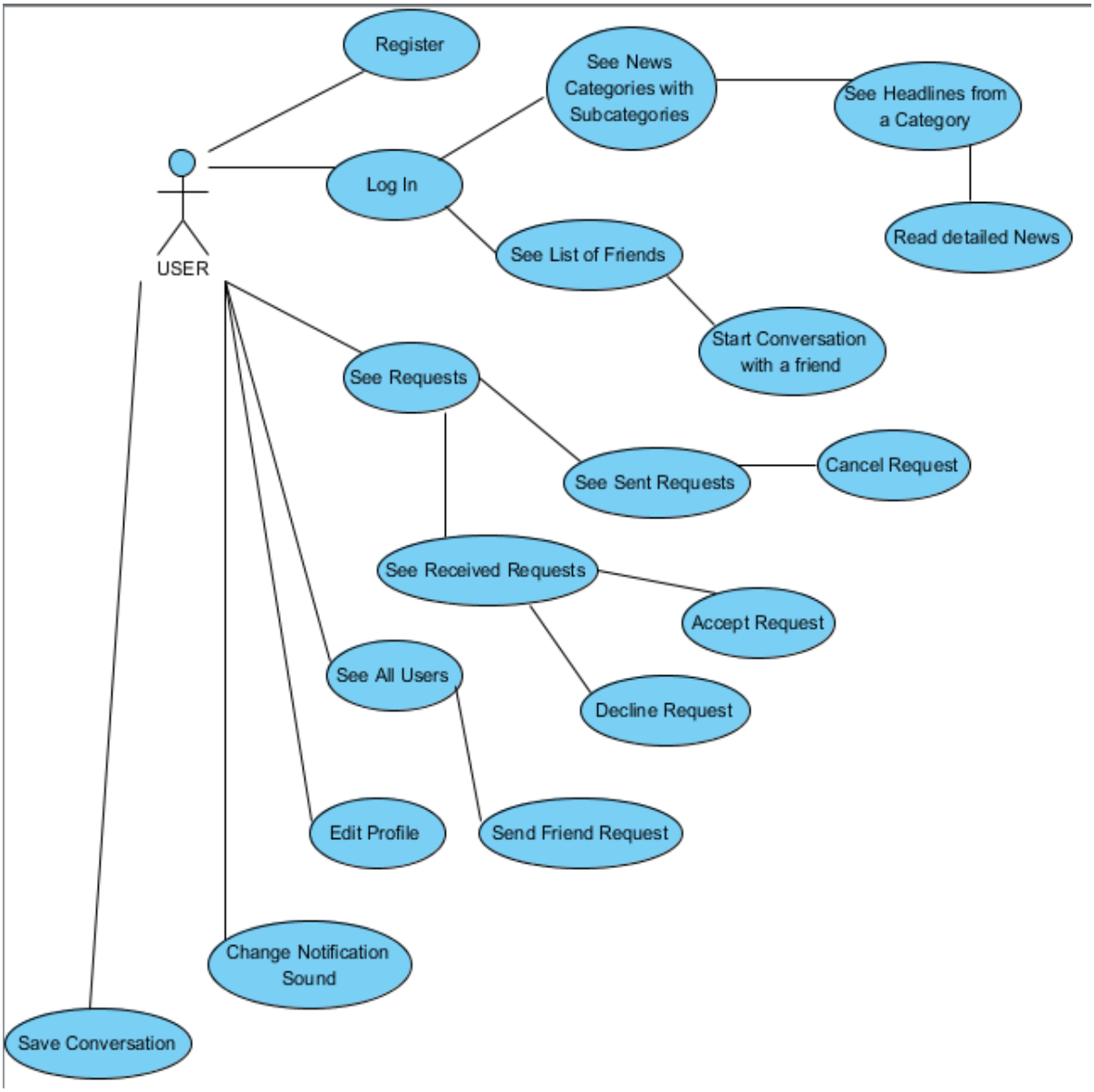


Figure 6.1 Use case

As we can observe in the figure above, at first, when the application starts, the user can log in or create an account. After creating a new account with success the user can immediately log in, because when the new account is created, it is immediately sent through the network to the server and into the database.

When logging in, the user can see the news categories with every subcategory. By taping a subcategory, he can see the headlines and then by taping a headline he can read the complete news. From the detailed news, the user has the possibility to get to the headlines back or to return directly to the screen with all the categories and subcategories.

On the screen with the news categories, by sliding on the device screen from the left margin, the list of friends and their locations appears. By taping a friend, a conversation screen is opened and the two friends can chat by sending text messages to each other. From here the user can slide the screen again on the left and chose another friend to talk to, or he can chose to see the news categories again by taping the “news” icon.

From the screen with the news categories or the chat screen, the user can also tap on the “requests” icon so that he is sent to a new screen where he can see what friend requests he sent, and can cancel them, and also what friend requests he received and can either accept them or decline them. From the same screen the user has the possibility to tap on the “all users” icon and he will be sent to the screen with all the users from where he can tap on a user and see the user details and he can chose to send him a friend request.

Also the user has the possibility to edit his profile at any time or he can chose to change his sound for the notification. If he is in the chat screen, the user can save his conversation.

If the user wants to log out he can simply tap the back button of the device two times and he will be sent to the log in screen. If the user does not want to log out, so that he keeps on receiving messages from friend, but he wants to do something else with his device, he simply has to minimize the

application by taping the “minimize” icon. Like this, the user can do whatever he wants on his device and still receive messages from his friends. Every time a new message comes in, the user receives a notification. By taping the notification, the user is sent to the screen with the chat and he can continue to send messages with his friend.

6.2 Functionality

At first, when starting the application, the login screen will be shown, where the user can either introduce his username and password or he can register to create a new account. Below is a screenshot of how it looks:

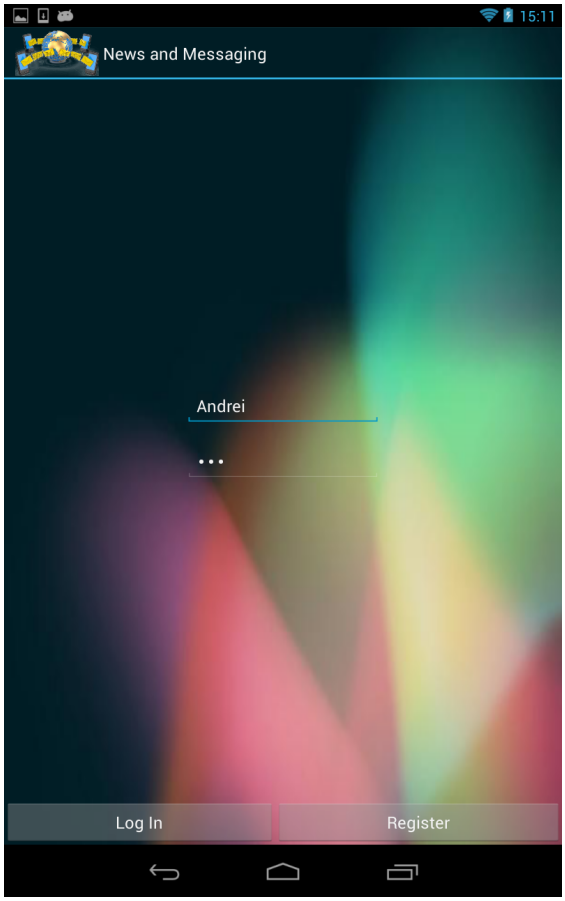


Figure 6.2: Start Page

By clicking the "Register" button the user is sent to the register page where he can create a new account:

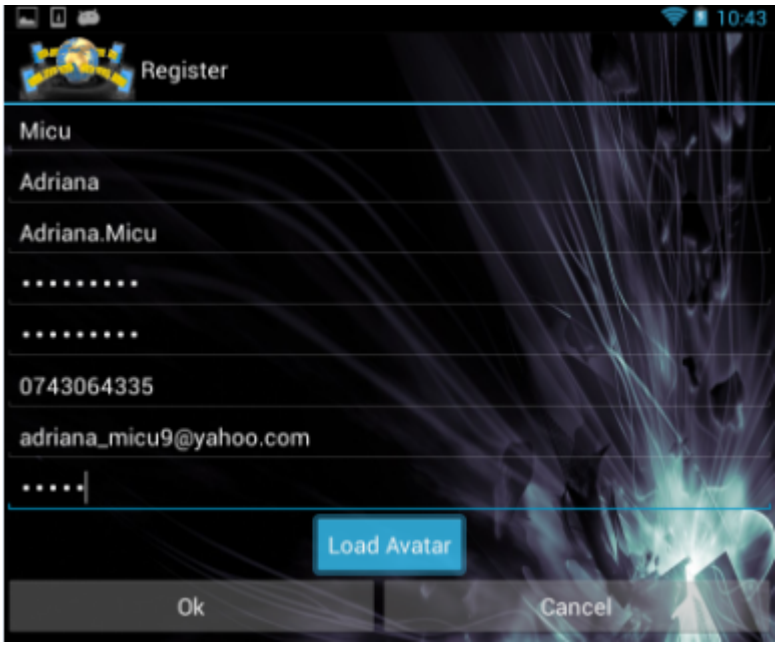


Figure 6.3: Register

By taping the "Load Avatar" button, the user can select a picture from his device Gallery and it will be added as his profile picture into the database. The parsing of the image is done as follows on the server and on the client, Actually, the client sends to the server a long byte-array which is parsed and set into the database as a "Longblob" Object. So that the parsing is done correct, and the server puts a valid file into the database, which can be parsed back on the client side I used the following classes:

```
String st = jso.optJSONArray("appuserLocation")
        .optJSONObject(i).get("avatar").toString();

byte[] blob = Base64.decode(st.getBytes(), Base64.DEFAULT);
Bitmap bmp = BitmapFactory.decodeByteArray(blob, 0, blob.length);
```

Figure 6.4: Client parsing

```
String avatar = appuserParams.
        getFirst("avatar");

edit.setAvatar(Base64.decodeBase64(avatar));
```

Figure 6.5: Server parsing

After the user selects an avatar from the device gallery, it will be shown above the “Load Avatar” button, as it can be seen on the next page:

Figure 6.4 Image loaded from Gallery

The user can then already log in with his new account. After logging in, the first thing seen is a screen with categories of news and their subcategories.

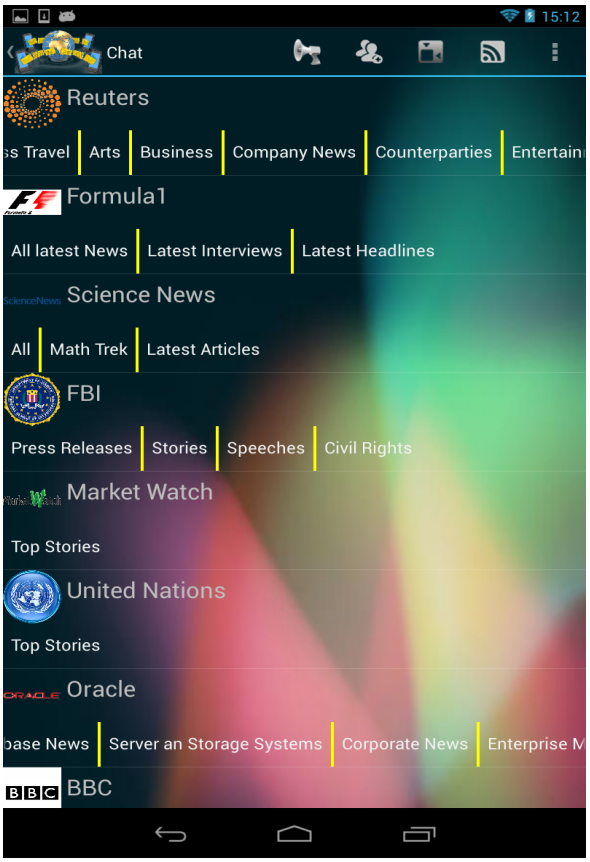


Figure 6.5: News

From the figure 6.5 screen, the user can either tap on a subcategory to read its headlines or he can slide the screen on the left side to view his list of friends and their locations. By taping a subcategory, the user can read the headlines and by taping a headline, the full detailed news shows up, and the user can read it completely. Returning to the main screen, the user slides the screen on the left and sees the following:

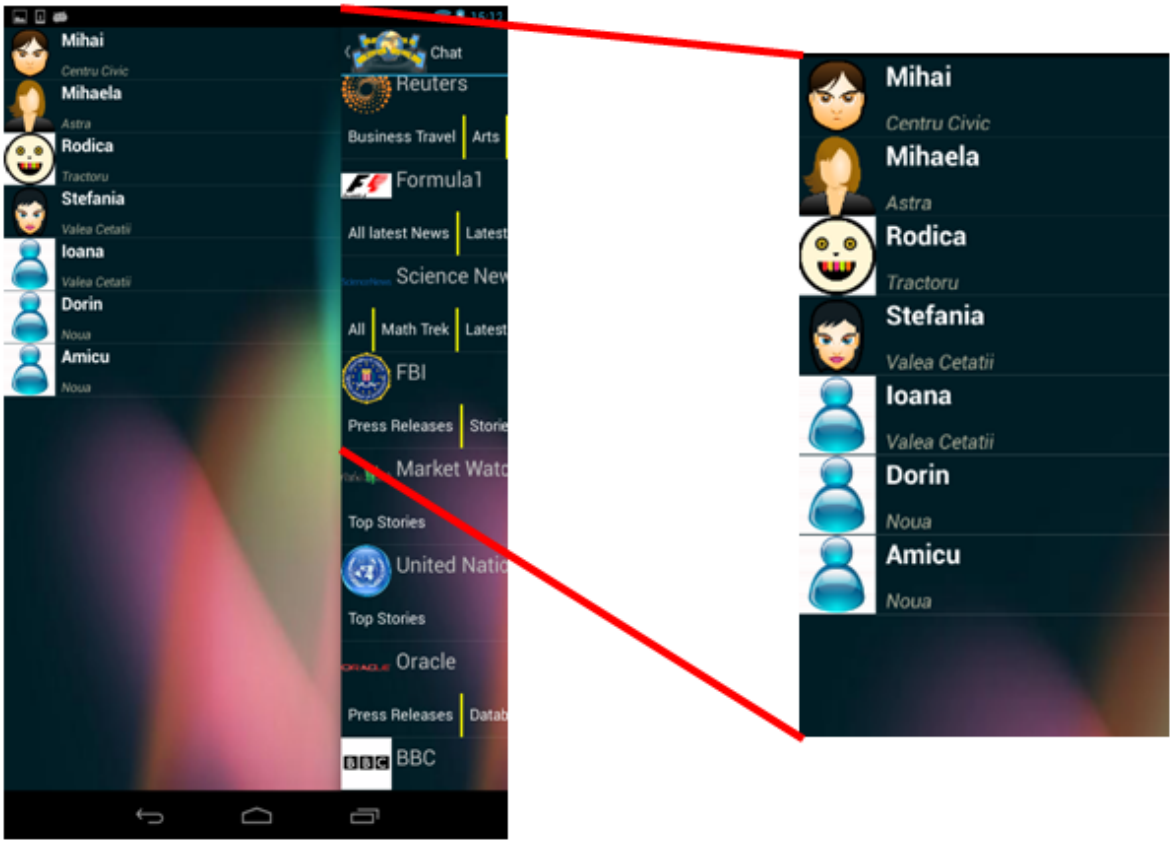


Figure 6.5: Friend List

Figure 6.6: Usernames and Locations

The locations are set as every user logs in using the GPS from the device. By taping a user the conversation can begin. When the user taps a friend to talk to him, the service which verifies if messages are sent between those two, starts. The two users can chat by sending text messages to each other. They introduce text in the text box, and then, taping the “Send” button, the server receives the message, the service on the friends device finds it by sending the request to the

server, and like this the users chat in real time. This is how the chat screen looks like:

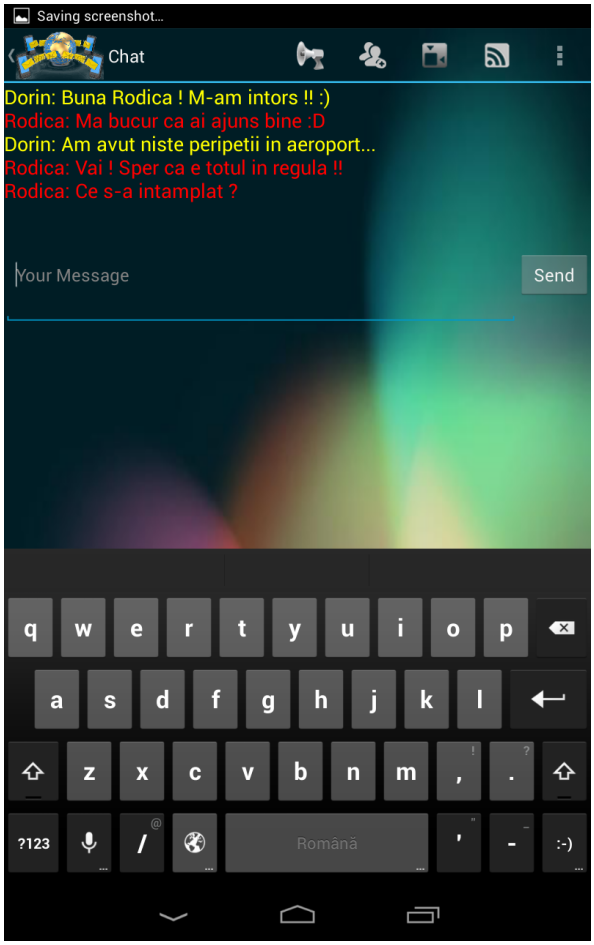


Figure 6.7: Chatting

As it can be observed, the application has some icons in the action bar, which perform different tasks. The first icon, from left to right, if taped, redirects the user to the screen on which he sees two tabs: send and received requests. The second icon, redirects the user to the screen where he can see all the users and by taping on a user in the list, he can choose to send friend requests to the user he wants to. The next icon minimizes the application and starts the notification service, so that the user is able to receive messages no matter what he does with his device. The last icon is used to redirect the user to

the screen with the news. Then follow the menu items which do not have an icon and which will never be shown in the action bar. These three items are presented below:

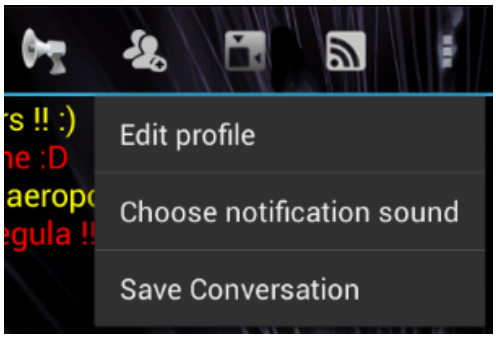


Figure 6.8: Menu items

The first item: "Edit profile" sends the user to a new screen where he can edit his profile details. The next item: "Choose notification sound", triggers a pop up where the user can listen to every sound an choose the one he would like more to hear when a notification shows up. Next, the "Save Conversation" item appears in the menu, only if the user is on the conversation screen, so that he can save the conversation as a ".doc" file to his device.

For example, the screenshot below shows the screen where a user sees the requests he received. By taping the green "tick" sign, the user is added to his friends list, and by taping the "X" sign, he declines the friend request and the user will not be added to his friends list.

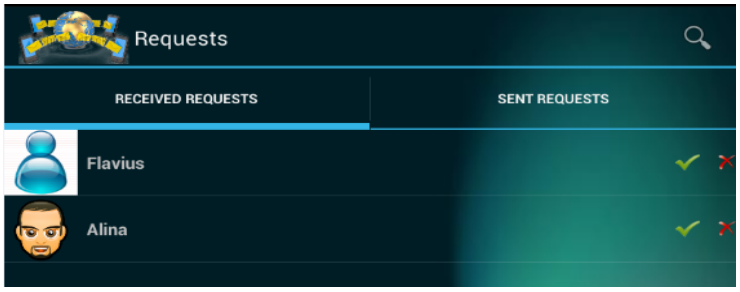


Figure 6.9: Received Requests

When the application is minimized and the user which is logged in receives a message, a notification will be shown on the user's device status bar:

Figure 6.10: Notification in status bar

Above, the first icon represents the icon of the "News and Messaging" application. If the user wants to see who and what wrote him, this is how it looks like:

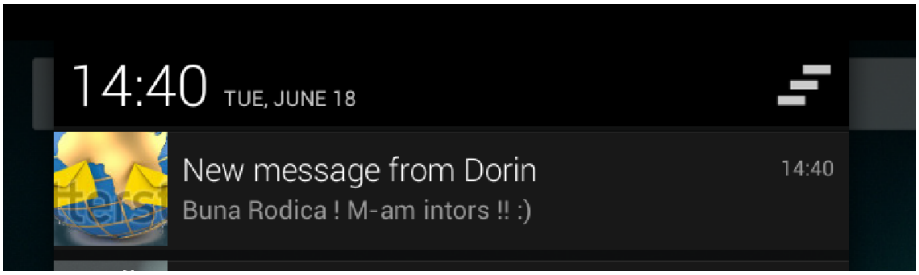
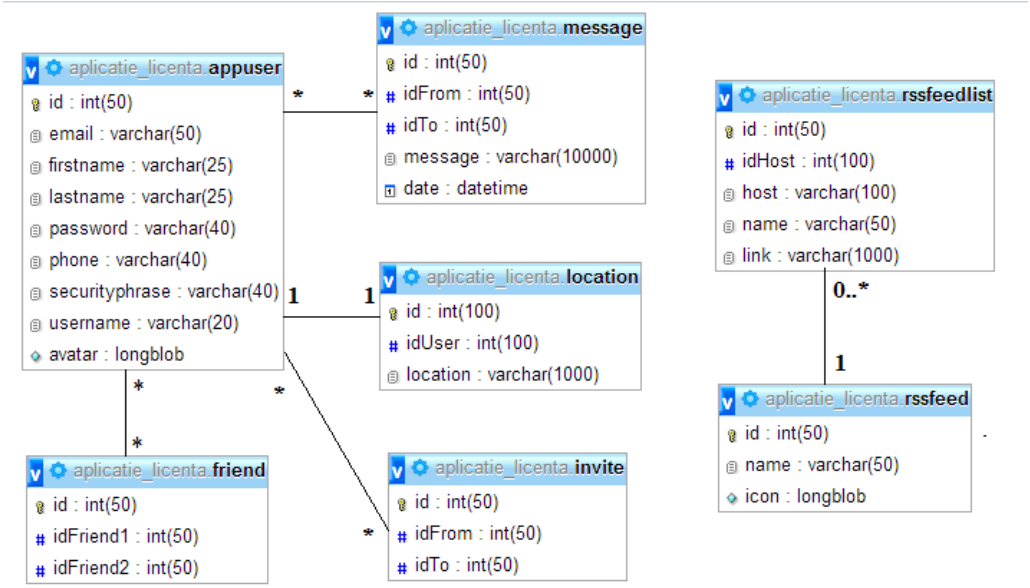


Figure 6.11: Notification

6.3 The Database

As I mentioned in the chapters before, I used a MySQL database for my project. Below is the structure of the tables in the database:



In the picture from the previous page are the mapped objects which resulted from the entities I created in the server-side application.

And these are the most important aspects of the functionality of my application. This kind of combination between the news and the chat was not implemented before. I chose to do this kind of application, because I wanted to implement everything that the expression “to stay informed” means.

Also the news part can easily be extended, by simply adding new RSS links with new feeds.