Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Maksud Ganapijev

Extending a multiple-user android application for historical monuments

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Maksud Ganapijev

Extending a multiple-user android application for historical monuments

Bachelorthesisbased on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr. rer. nat. Henning Dierks
Second Examiner : Prof. Dr.Ing. Marc Hensel

Day of delivery 23. Oktober 2018

Maksud Ganapijev

Title of the Bachelorthesis

Extending a multiple-user android application for historical monuments

Keywords

Java, Android, Linux, Spring Boot, Tomcat, MySQL, Apache, representational state transfer, REST, model view controller, mvc, web application, software architecture

Abstract

The focus of the following thesis is to extend a multiple user application built for an Android platform. This application would give the users an opportunity to teach and learn the history of ancient Rome.

All the instruments that were used in order to develop the application are free and open-source: Java, Android, Linux, Spring Boot, Tomcat, MySQL, Apache. The description of each tool listed above can be found in the following chapters of the thesis.

As a result, it is expected to have an interactive Android application, that will be capable of maintaining users database, uploading and storing information about historical places and what is the most important provide a graphical user interface to perform all of these operations.

Maksud Ganapijev

Titel der Arbeit

Erweiterung einer mehrbenutzer Android-Anwendung für die historischen Denkmäler

Stichworte

Java, Android, Linux, Spring Boot, Tomcat, MySQL, Apache, representational state transfer, REST, model view controller, mvc, web application, software architecture

Kurzzusammenfassung

Ziel der folgenden Abschlussarbeit ist es, eine Multi-User Anwendung auf einer Android-Plattform zu erweitern. Diese Anwendung gibt den Benutzern die Möglichkeit, die Geschichte des Römischen Reiches zu lehren und zu erlernen.

Alle bei der Entwicklung der Anwendung verwendeten Hilfsmittel sind kostenlos und open-source verfügbar: Java, Android, Linux, Spring Boot, Tomcat, MySQL, Apache. Eine detaillierte Beschreibung dieser besagten Hilfsmittel erfolgt in den nachfolgenden Kapiteln dieser Arbeit.

Das erwartete Ergebnis wird eine interaktive Web-Anwendung sein, welche die Pflege einer Nutzerdatenbank und das Hochladen und Speichern von Informationen über historische Stätten ermöglicht und darüber hinaus, was am wichtigsten ist, eine graphische Benutzeroberfläche zur Verfügung stellt, um alle diese Prozesse durchzuführen.

Acknowledgment

First of all, I would like to thank Prof. Dr. rer. nat. Henning Dierks for mentoring me throughout the whole project. Without his organizing tips and realistic evaluation of the project status, I would not have been able to achieve as much as I did in the end.

I would also like to thank all of my university professors for giving me the knowledge, that has become a foundation for my future plans and helped me to stand firmly on my own two feet in my life.

Last but not the least, I would like to thank my father Vitaly Ganapiev and my mother Roma Ganapijeva for supporting me spiritually throughout the studying process and my life in general.

Contents

List of Figures

1. Introduction

With the coming of the software giants like Apple, Amazon, and Google, smart-phone applications have become an inseparable part of humans everyday life. Today smart-phone applications reside in every possible sphere: business, organization, self-development, education and more. This study will focus on the usage of smart-phone applications in an educational sector.

The project **Civitas** is a system developed for the purposes of teaching and learning a history of the ancient Rome. The system combines in itself an Android application and a server application with its database, that stores information about discovered historical places of the Roman Empire, or so-called **artefacts**.

The Android application serves as a **Graphical User Interface** (GUI) and includes the map with locations of the artefacts, menus, and logic to access the server. The server provides an interface to access a database, which contains a text, image and audio information about the artefacts. It is assumed that clients of the **Civitas** application would be university professors or university students since it can be used for both teaching and learning purposes.

1.1. The purpose of the investigation

Civitas project was started by a previous team of developers and as a result, a working system was delivered, however, it lacked some features that would make an experience of the application more pleasing for the user. Missing features as they were listed in the technical description of the thesis:

1. For the user registered in the **Civitas** system, there should be a possibility to recover a password in case the user forgot it.
2. **Civitas** system should have a functionality to create users with different user permissions, which would allow registered users to modify their own content and restrict them from modifying someone else's. In addition, there should be a super user, who would have the rights to modify all the content, for moderation purposes.
3. The existing filtering system is none of the convenience of developers nor users because it only allows users to filter the data by a predefined list of values. Which leads one to a logical finding to introduce a search field. The best part about having a search function is that its logic can be changed, without the user even noticing it.
4. Every registered user has an opportunity to create artefacts. Some of the information uploaded by the users could be wrong, so the users should have a chance to inform the each other about inaccurate artefact data. An artefact rating system was requested for this reason.
5. An existing interface allows allow users to create the artefacts, but does not allow to modify them. **Civitas** application user interface should be modified in a way, that every user would have a functionality to modify the artefacts as well.

1.2. Problem statement

One could ask, why not to write the application from scratch and bother with the given code? A simple explanation is because the former Civitas team did a large amount of work on the project. And it was not possible to replicate this work in the given time.

More can be said, an existing project code base will always be an advantage to a skilled developer and an obstacle to an incompetent one. Every good developer should have an ability to interpret the provided code and integrate new features into an existing system.

After analyzing the technical description (Section 1.1), one would assume that the task would only involve working on an Android application (Section 2.3) of the **Civitas** system, but soon

enough deeper analysis of the project has shown, that it would impossible to achieve desired results without re-working the server-side (Section 2.4) of the project, which of course would reduce the time available for the main task.

1.3. The background

As it was mentioned in section 1.1, there already was an amount of progress achieved by the previous team of developers. At the start of the bachelor project **Civitas** system contained the following functioning modules:

1. Android application
    a) An interface to register and log-in a user
    b) An interface to upload and fetch text information about an artefacts
    c) A local storage of an image and audio information of the artefacts
2. Server application
    a) A server application is written in Java with pre-programmed controller and persistence classes, with an implemented logic to allow the Android application actions mentioned above
    b) A configured database
    c) A comma separated value file with text information about monuments, as well as a directory with images and audio files
    d) Credentials to access a remote server with running back-end application

1.4. Thesis and general approach

A study contains four main sections structured in the following way: an analysis of the problem, a design of an architecture, a final realization of the designed architecture and ultimately a testing of the final realization.

1.5. The criteria for success

The criteria of success of this study can be extracted from the project task and it would be a system that would allow users to login into **Civitas** system, as well as upload, fetch and search through the text, image, and audio information about the artefacts. However, the criteria of success are more than just a technical task and its completion.

The project is a chance for a developer to try his organization and programming skills in a development of a complex real-world Android application, make mistakes and understand how not to make them again, find multiple solutions for a single problem, develop a mindset to overcome appearing obstacles and finally estimate a complexity and accordingly arrange the resources. All of the aspects mentioned above can be considered criteria for success of this study.

2. Analysis

A picture is worth a thousand words, that is why **Unified Modeling Language** (UML) diagrams are so widely practised in a software engineering. The UML diagrams create a common ground for every level of users, a language that can be understandable for business users and software engineers.

The reason that UML diagrams play an important role in software design is that diagrams help to organize software development process, by diving it into smaller stages. In order to achieve success, it is an absolute must for every project to be visualized "on a paper" first.

2.1. Design tools, theoretical background

This section contains a detailed analysis of the problems with diagrams and descriptions, and will develop a theoretical base for a convenient reading of the Design chapter 3 and the Realization chapter 4.

The project is almost entirely written in Java programming language, so the focus of the thesis will be mainly on Java components. However, it is worth mentioning that **eXtensible Markup Language** (XML) and **Structured Query Language** (SQL) were utilized too. XML was used for creating the Android application views and SQL for the communication with the database.

The reason that Java is so popular sits behind Java application properties. Applications written in Java are completely agnostic of an underlying operating system (OS), thanks to the **Java Run-time Environment** (JRE). Run-time environment is a somewhat nebulous term, which collectively refers to the **Virtual Machine** (VM) in which your code runs. **Java Virtual Machine** (JVM) runs the code and stores standard libraries that the code may expect to find, no matter where it is installed.

2.1.1. Android OS run-time environment

Civitas Android application can be run on any device, which is running Android OS. The Android OS has a Linux-like core and uses a virtual machine to run applications. Since most of the Android applications are still written in Java, an Android application running process is very similar to Java's with its JVM. It is not a secret, that many developers claim JVM to be memory and power consuming, for this reason, Google developed an optimized solution exclusively for an Android OS and called it **Dalvik Virtual Machine** (DVM).

In detail, the difference between JVM and DVM is the following. The Java source and library code go through the standard Java complication process and ends up as a Java bytecode, but instead of running it directly on JVM, it is being further minimized and compiled once again into a Dex bytecode, then finally translated into machine code and run in DVM (Figure 2.1)

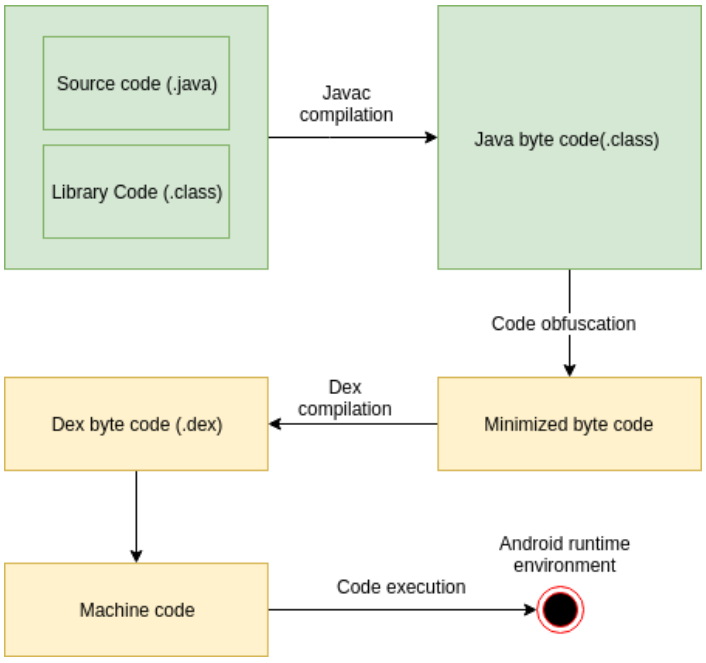


Figure 2.1.: Android code processing diagram

2.1.2. Android Studio, Android API

Due to sophisticated source-to-library code dependencies and complex build process, the Android programming is not a kind of programming one can easily do in a random code editor. Luckily enough, Google provided developers with an **Integrated Development Environment** (IDE) with components, that take care of all of the complications. The IDE is called Android Studio.

The Android Studio would be useless without its main Android **Application Programming Interface** (API) component. It contains the essential building blocks of an Android application:

1. An **Activity** - is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for a developer in which a developer can place a **User Interface** (UI). While activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows, or nested inside of another activity.
2. A **View** occupies a rectangular area on the screen and is responsible for drawing and event handling. A View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.).
3. A **Fragment** represents a part of a user interface in a **FragmentActivity**. One can combine multiple Fragments in a single Activity to build a multi-pane UI and reuse a Fragment in multiple Activities. A fragment can be thought of, as a section of an activity, which has its own function stack, receives its own input events, and which one can add or remove while the activity is running.
4. **Representational State Transfer** (REST) service libraries contain the functionality required for creating web services.

An Android application is an environment where a software and hardware components are constantly exchanging information and statuses on the system state. Most of these API calls are executed in parallel to the application code and have prescribed function (callback) to run right after this API call has finished executing. This is necessary in order to prevent a block of an execution of the Activity and Fragment code, which responsible for the view rendering, so in case of the smallest delay user would immediately see this lag in an animation of the application.

A mean of programming described above is called **asynchronous**. The DVM instead of waiting for the program to reach the return statement just jumps to an execution of a further code. There is no need to wait for a return statement because a callback function will take care of the return argument (Figure 2.2).

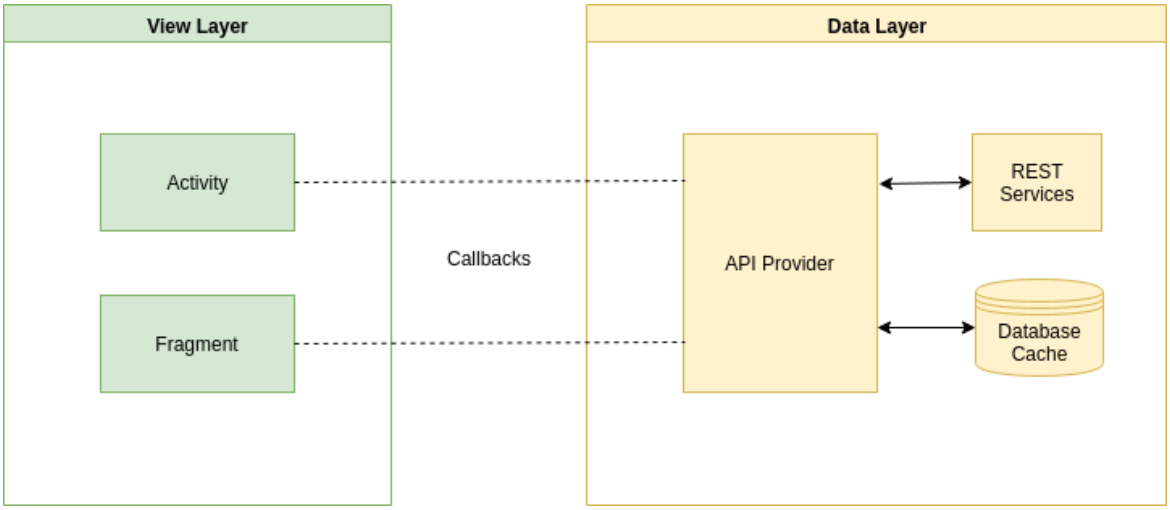


Figure 2.2.: View and Data layer relation in Android application

In case of a web-application return argument for a callback is often a response from a server. This leads us to the server (or so-called back-end) introduction section (Sec. 2.1.3).

2.1.3. Server software stack

The fact that **Civitas** will have an information about thousands of historical artefact forces one to consider, where to store all that information. A great solution for this problem would be a device, which is always online and ready to serve this information to the **Civitas** application users (web server). It sounds like a simple solution, however, there is more to it. A web server consists out of a physical server (device connected to an internet), server operating system and software used to manage HTTP communication.

An HTTP Server is necessary for the server software to be able to perform web communication. **Apache HTTP Server** is a software part of a web server system that distributes local content or services (e.g servlets) to users over the internet.

Once the information has been sent from a client device to a server, the server has to know how to process the received information. This functionality is described inside server applications or so-called servlets. In short, a servlet is a software component that extends the capabilities of a server.

As it was mentioned above Java can be an instrument to develop a tool under any platform, if this platform has JRE installed, so Java was used to program servlets as well, specifically Spring Boot framework. This framework contains all the libraries required for servlet programming and database communication.

Java servlets, just like any other Java program need require a run-time environment. **Apache Tomcat Server** is used as a run-time environment for servlets, it implements required servlet web socket functionality and simplifies servlet deployment and debug process.

Finally to store the artefact information **Civitas** uses **MySQL** relational database. MySQL database understands SQL and stores data in form of the tables. A useful fact, SQL tables can be easily exported to a comma-separated values file and then converted back to an SQL table.

Once it was decided upon which software components will be used to implement the project (Sec. 2.2), the next logical step was to develop the project architecture.

A server architecture diagram (Figure 2.3) shows how all of the components mentioned above work together.

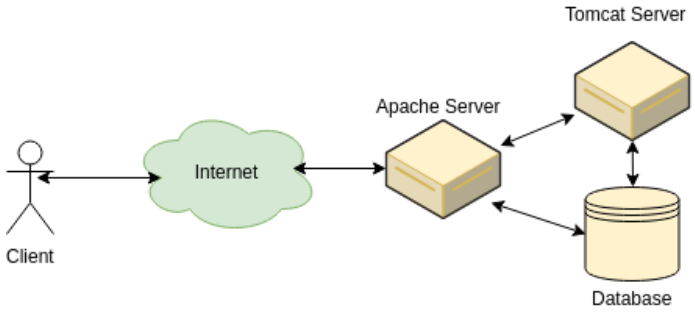


Figure 2.3.: Server architecture

2.2. Project architecture

The base application provided by the previous developer team was developed in the REST style.

It is worth mentioning, that another good option for the given task would be **Simple Object Access Protocol** (SOAP). Both architecture styles could be used to achieve the same outcome and both provide great scalability. However, the main reason REST architecture is chosen over SOAP is due to its simplicity. Further discussion of the advantages and disadvantages of both is out of the scope of this thesis and will not be developed.

Additional resources in form of a server are available, therefore Android API will be only utilized as a front-end layer or a **Graphical User Interface** (GUI). In addition to a GUI, Android API contains the main request dispatch pipeline (Android Volley library).

A brief introduction into a project architecture can be seen in Figure 2.4.

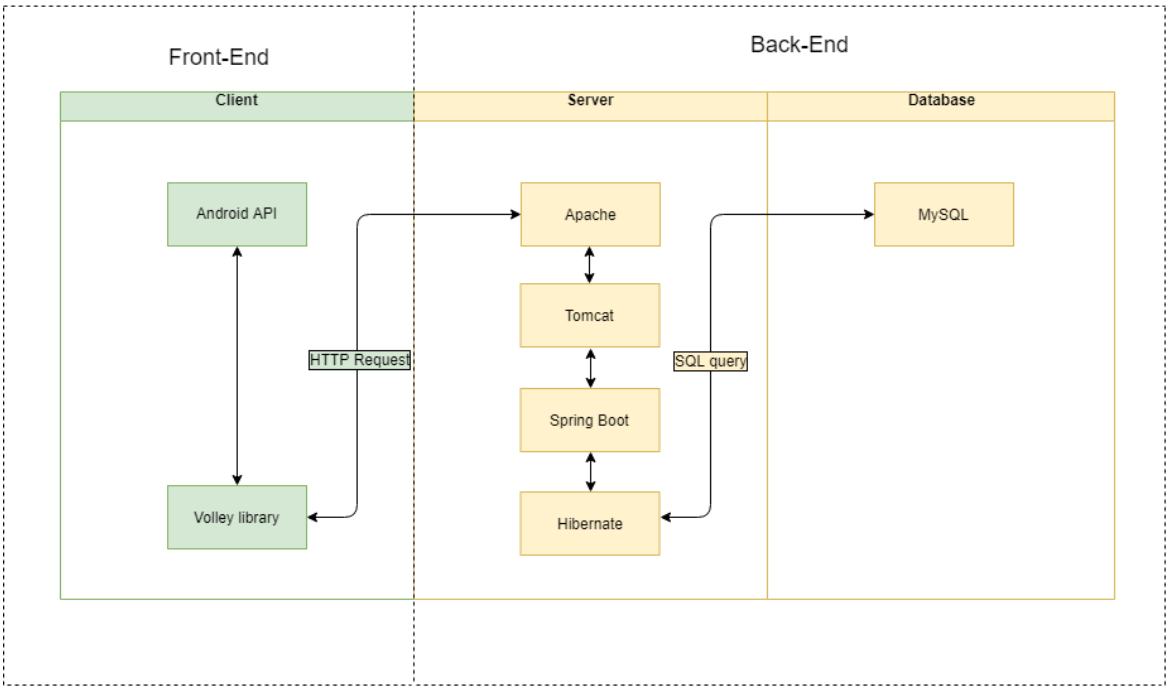


Figure 2.4.: Application architecture

With a help of an Android Volley library **Civitas** communicates with a back-end in a request-response manner. At the back-end, the Apache server delivers a request to the Tomcat, which is running a Spring Boot application. A server application performs required operations with the data passed in a request body and sends back a response.

2.3. Front-end

The front-end, or the client-side of **Civitas** system is fully implemented inside an Android application. An Android API offers a great variety of tools for software development, especially for the UI creation.

The technical description of a project had the following requirements from the client-side:

1. The user should be able to create an account and login into Civitas system
2. The users must be able to upload pictures and additional information
3. The application must support three different user account levels: regular users, trusted users and administrators
4. The content uploaded by users must be automatically shown in the application
5. The app must include an interactive map, that shows the current location of the user as well as various artefacts on the map, that has already been added to the application
6. If the user clicks on a marker on the map, a pop-up must appear showing the user additional information about the artefact

The Android application was built by utilizing fragment-activity concept. This concept allows to build a block of application and re-use it later in other places; See diagram 2.5.

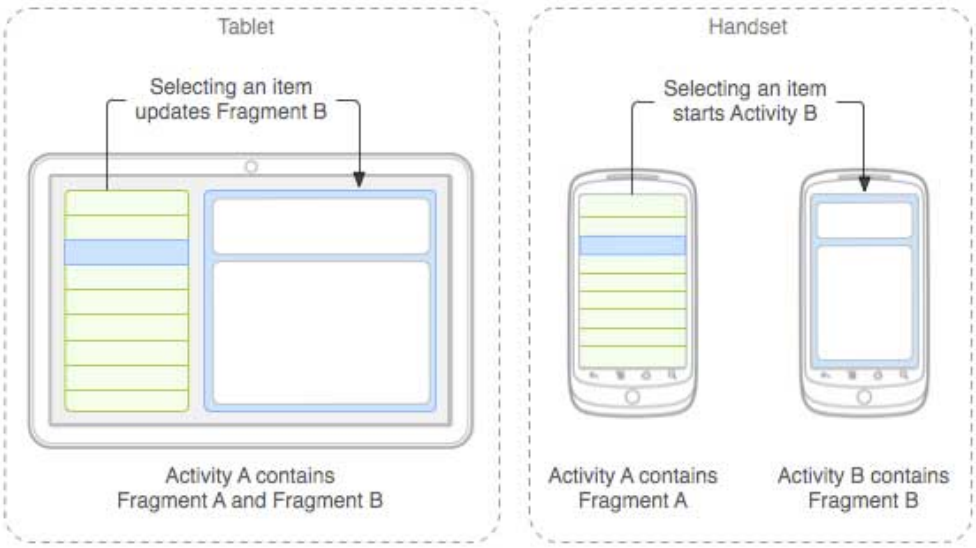


Figure 2.5.: Android fragment concept

After an analysis of a provided **Civitas** Android application the following diagram, describing application functionality was built (see Figure 2.6). A user with an Android application sends various types of requests, i.e to fetch, upload, or delete data. The data is then formed into a request (GET, POST, DELETE) with its body and submitted to a server.

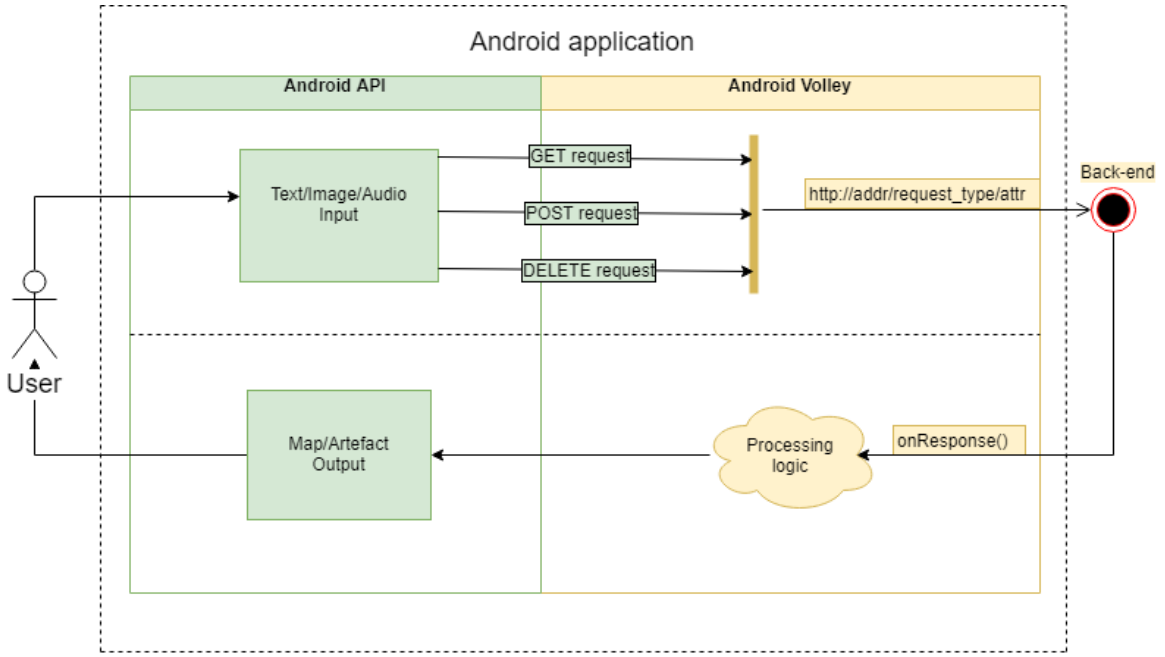


Figure 2.6.: Client-side general functionality diagram

To understand the communication between the Android application and a server, it is crucial to know the structure of the request (see Figure 2.7).

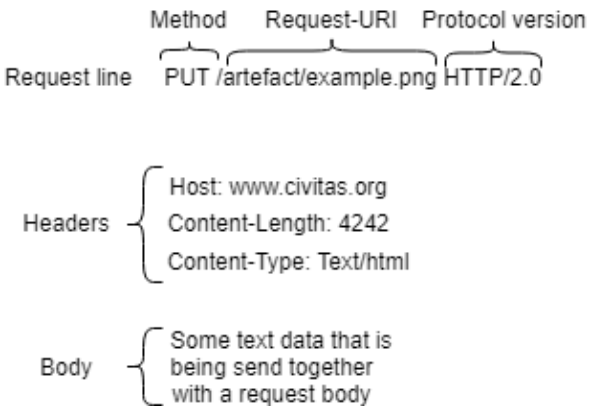


Figure 2.7.: Request structure

A request method and Uniform Resource Identifier (URI) in the request line will later determine the response of the server. A request head and body contain the data sent by the user and an information about a request, such as data type and content length. The server then answers with a response, which is very similar in the structure to a request.

2.4. Back-end

The project description implied a working back-end, therefore, during the first phases of the project development, the back-end was treated as a properly functioning black box system. However, after a deeper code debug and test runs (section 5) functionality limitations were found, such as:

1. Instead of uploading image and audio files to a server, the files were stored locally. As a consequence one user could not fetch an image or audio information of another user.
2. Restricted access to the file system of the server. As a consequence **Access Forbidden** error on every attempt to store files on the server.
3. A filter list implementation would lead to a data inconsistency right after the first filter list update because filter values were hard-coded. A description of this problem is better developed in section 3.1.6.

After several unsuccessful attempts to fix issues from the front-end side, a decision was made to access the server and debug the code. Luckily, server access credentials were provided by the previous developers. However, right after the first connection attempt, it became clear that access to the server was restricted by the host administrators.

Finally, it was decided to replicate the back-end in the lab environment, so it would be possible to use the debugging in its full potential. Further description of the back-end migration process can be found in the realization section (section 4.2.1).

Since all the analysis steps are accomplished, it is possible to proceed to the design and implementation part (Section 3).

3. Design

This section describes an implementation process of the main **Civitas** project components, an implementation of the project is mainly divided into two parts: front-end and back-end.

3.1. Front-end

As it was described in a front-end theory section (Sec. 2.3), a client-side is entirely implemented within the Android application. One of the most important blocks of an Android application is a GUI. When building a GUI it is common to discuss use-cases with a client and then develop use-case diagrams.

Use-case diagrams are usually referred to as behaviour diagrams, that describe a set of actions (use-cases), that users can perform with the system.

3.1.1. User use-cases

Use-case diagram 3.1 demonstrates, that all the required functionality can be integrated in the following use-cases:

1. Register/Login screen allows a user to create an account and login into the **Civitas** system
2. Map View screen allows a user to view the map and observe all the artefact in the selected location
3. Create Artefact allows a user to create his own artefact and store it in the **Civitas** system
4. View Artefact allows a user to view information on the artefact and rate another's artefacts
5. Filter/Search allows a user to get rid of inessential data
6. Rate artefacts allows a user to rate the artefacts created by other users

In order to construct the user use-case diagram (Diagram 3.1), client requirements listed above were taken in account. After user use-case diagram was constructed the GUI was built.

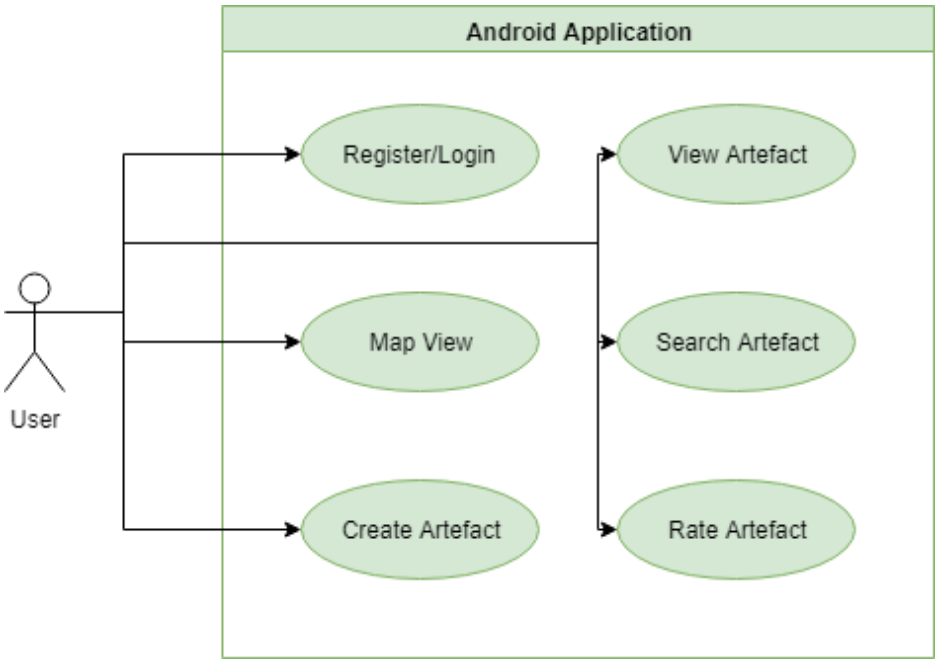


Figure 3.1.: User use-case diagram

This section continues with an accurate description of the design process of each of the features displayed in the Diagram 3.1.

3.1.2. Register, Login, Password recovery

An authentication system is a mandatory component for **Civitas** project. It should include user register function, login function and password recovery logic.

Previously implemented authentication system utilized classical user name, user password authentication approach. Complications in this approach appear with an attempt to implement a password recovery and user verification logic. There is no simple solution to implement these features without setting up a mail server.

Fortunately, Google introduced an authentication API, that takes care of both problems listed above. In addition, Google authentication API simplifies login and registration process for the user. An example of the Google Sign-In process is displayed in the Google Sign-in image set 3.2.

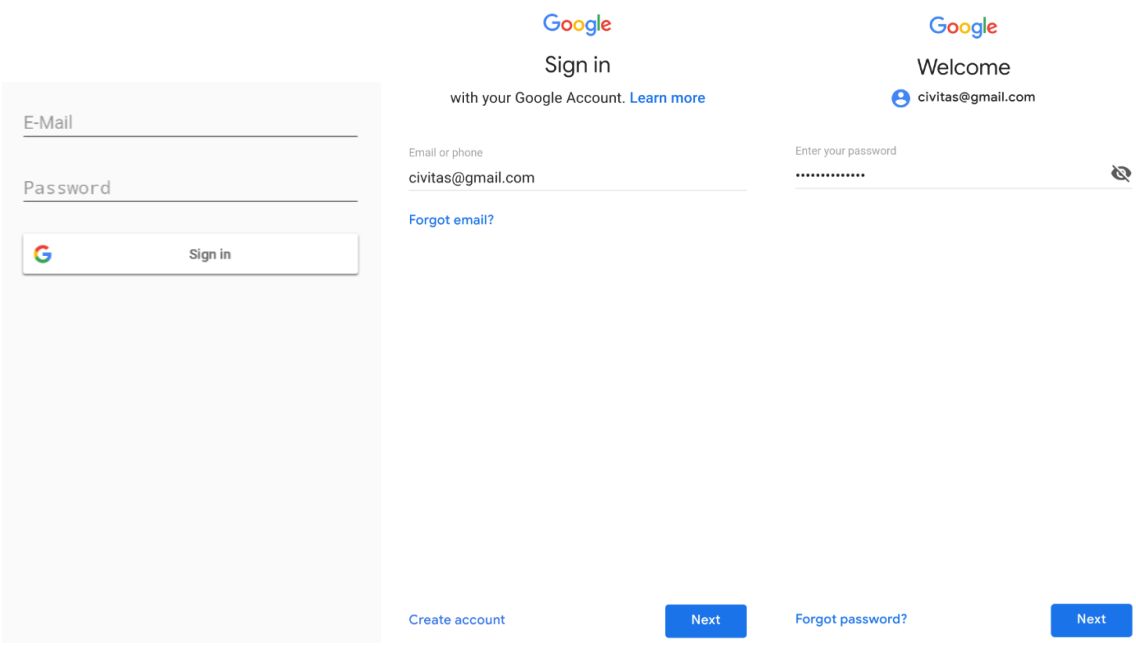


Figure 3.2.: Google Sign-in process

Before implementing the Google Sign-In, a developer must ensure the following is true:

1. Encryption and application credentials were obtained from the keystore. Critical information: during the development phase, the application is signed with encryption keys, that are stored in the debug.keystore file. Sign-in will fail if the wrong file is used.
2. The application should be registered in Firebase Console, with the files obtained from the step before. A developer will be provided with "google-services" JSON extension file.
3. JSON configuration file has to be placed in the "app" project folder 3.3 and application re-built.

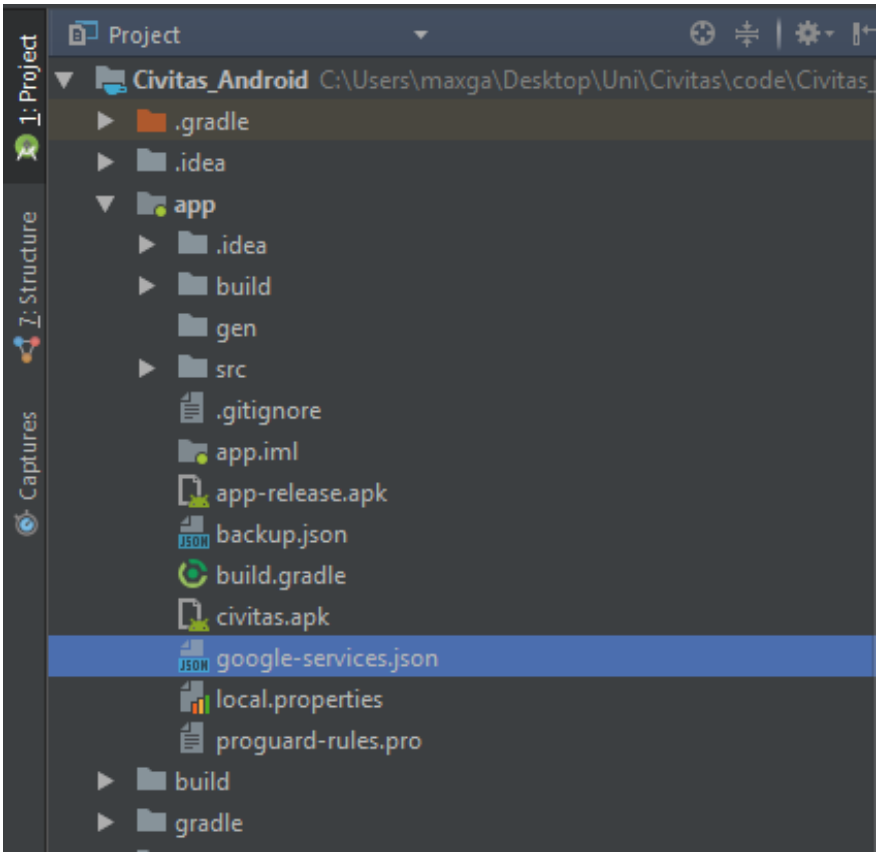


Figure 3.3.: Android project file structure, where to insert a google-services file

As authorization timing diagram shows (Fig. 3.4), an authentication API provided by Google works on the principle of token exchange. When a user sends a request for a sign in through the GUI, a listener uses Google Services library to request a token from the Google servers, later on, the same token can be used to perform other operations related to Google Web API, which means a token has to be requested only once.

An authentication service provided by Google would only allow developers to fetch user information, but each project requires a developer to store some internal information about users. In case of **Civitas** it is an additional parameter defining a user status in the **Civitas** system. In order to store user status information, once Google authentication API responds with a successful sign in result (Fig. 3.4), a request with a user first and last name, e-mail and a user status (user or admin) is sent to the back-end for storage. If the user already exists in the system, a server just responds with a status code 200 (successful response code) and a user object.

By default, on the first authentication, a user is given normal user access rights, which allow a user to view all the artefacts, as well a create and modify his own artefacts.

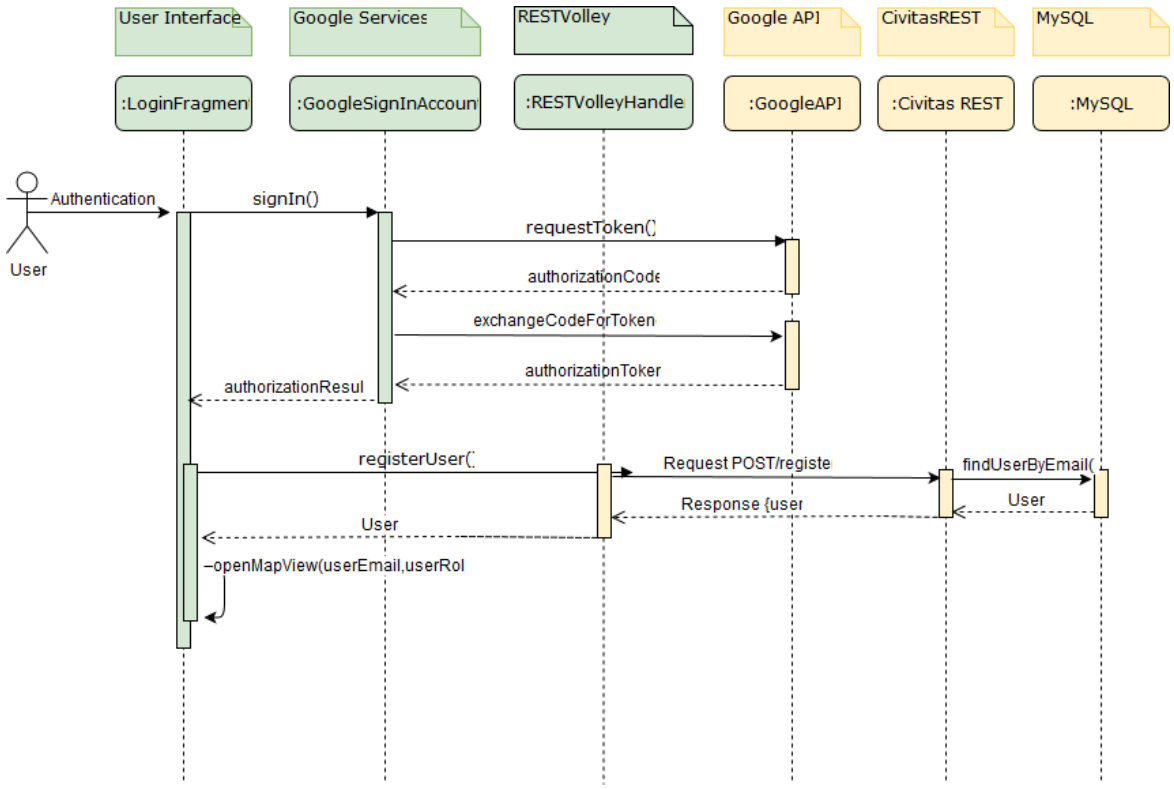


Figure 3.4.: Authorization with Google API

3.1.3. Viewing a map

An **Activity** class implements parallel code execution functionality, so **MapActivity** is also used to fetch additional resources from the server. Once the **MapFragment** reaches ready state, the **MapActivity** fetches artefacts and categories, by sending a request to the server. A server answers with a response, containing the lists of the artefacts and filter values (See timing diagram 3.5).

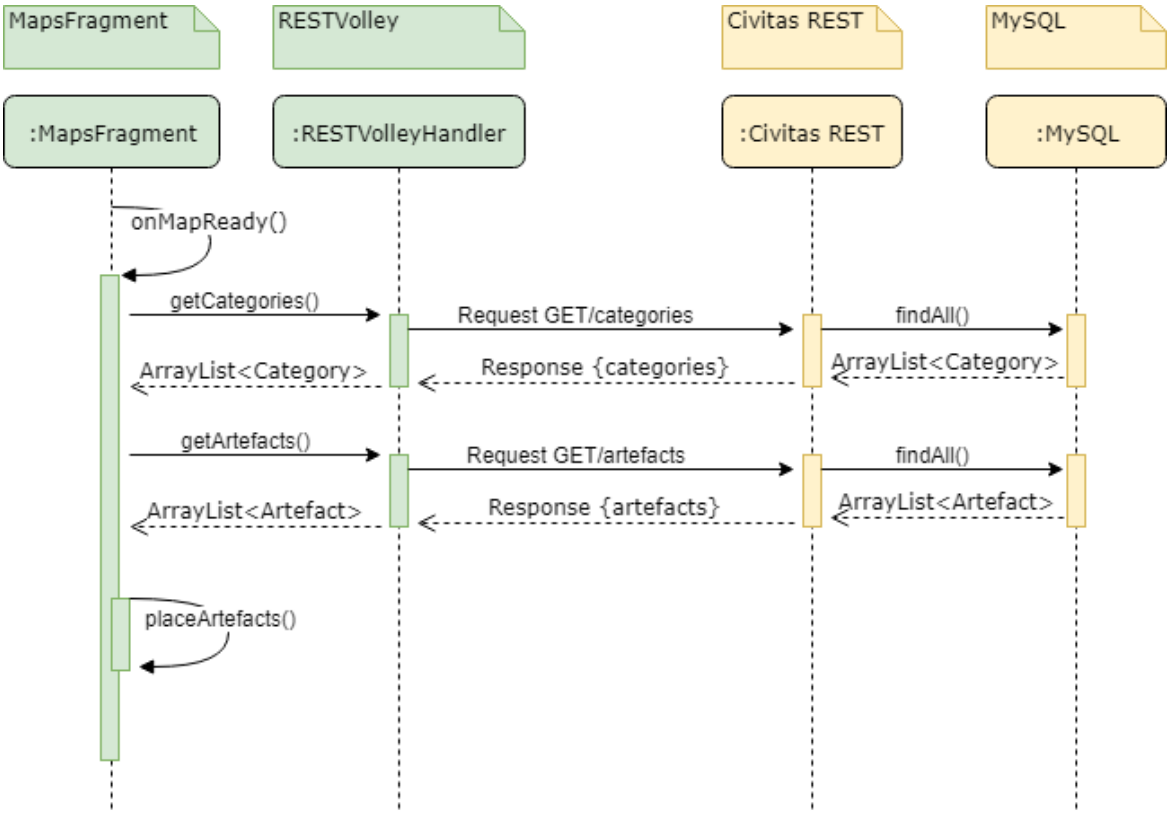


Figure 3.5.: MapFragment timing diagram

The idea behind storing filter values on the server is clarified in section 3.1.6.

3.1.4. Creating an artefact

An artefact creation view is one of the most complex in **Civitas** Android application. **AddArtefactActivity** class includes fragments and views: **AddArtefactGeneralV**, **MapView**, **AddArtefactPicturesCardView** and **AddArtefactImageDetailActivity**. It is easier to understand a structure of the **AddArtefactActivity** by looking at the class diagram 3.6.

A short analysis of the task of each module:

1. **AddArtefactGeneralView** is used as an input for the general text information about the artefact and its category selection.
2. **MapView** is a re-used **MapFragment** from the application main screen, it allows the user to choose the right location of the artefact on the map.
3. **AddArtefactImageDetailActivity** is a separate activity window, which involves image views, audio handler and an access to a camera. A storage logic, such as URL generation is stored in this class too.
4. **AddArtefactPictureCardView** displays all the artefact image items in a form of card list.

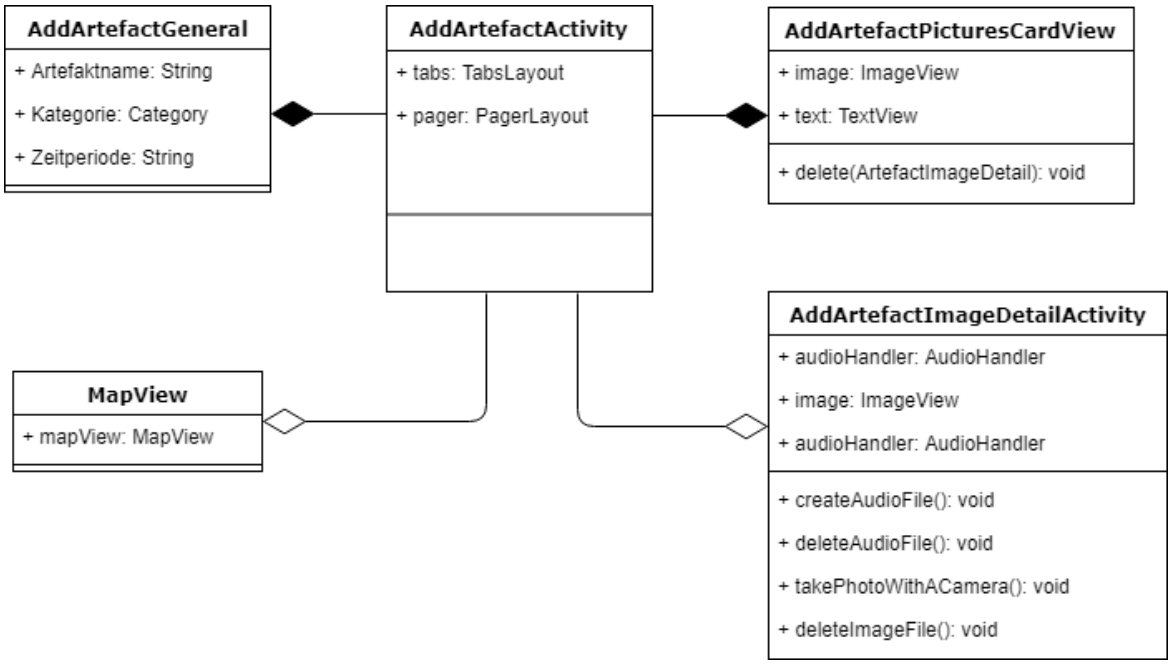


Figure 3.6.: Add artefact feature class relation diagram

For a better understanding of the process of how an artefact is created and then uploaded to a server, it is required to understand what does an abstract term **Artefact** mean.

An **Artefact** term includes in itself the the following programmatic parts (classes):

- An **Artefact** class holds the general information about the artefact
- An **ArtefactItem** class contains a broader information about the artefact, as well as the links pointing to an artifact image and audio files
- Optional image and audio files related to an artefact

A class diagram (Diagram 3.7) shows the contents of the **Artefact** and **ArtefactItem** classes and their relation.

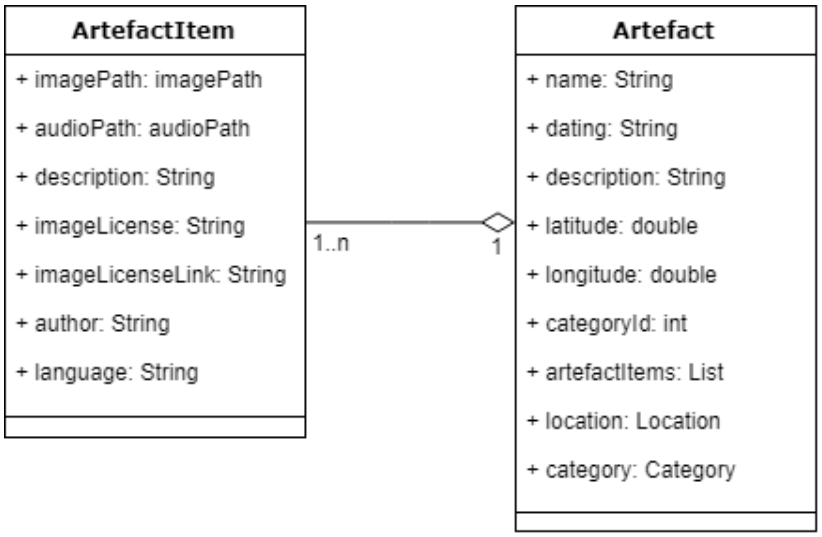


Figure 3.7.: ArtefactItem and Artefact class relation diagram

In order to create an artefact a user has to tap the **AddArtefact** button, which will display an artefact editor view. If all the necessary data was filled in correctly **Finish** button will trigger an upload process.

An upload process is divided into three steps. An upload process is displayed in a timing diagram 3.8. In a first step, the general information that is stored inside an **Artefact** object is uploaded. If an upload process was successful the server responds with the same **Artefact** object, which is later added to the main map view (section 4.1.2). As a second step an information stored inside an **ArtefactItem** is uploaded to a server and lastly, an image and audio data is uploaded.

The step two and three could be merged into a one since they belong to the same **upload-ArtefactItems()** method. However, in order for the data to be uploaded to a server, each of the steps requires a separate HTTP request.

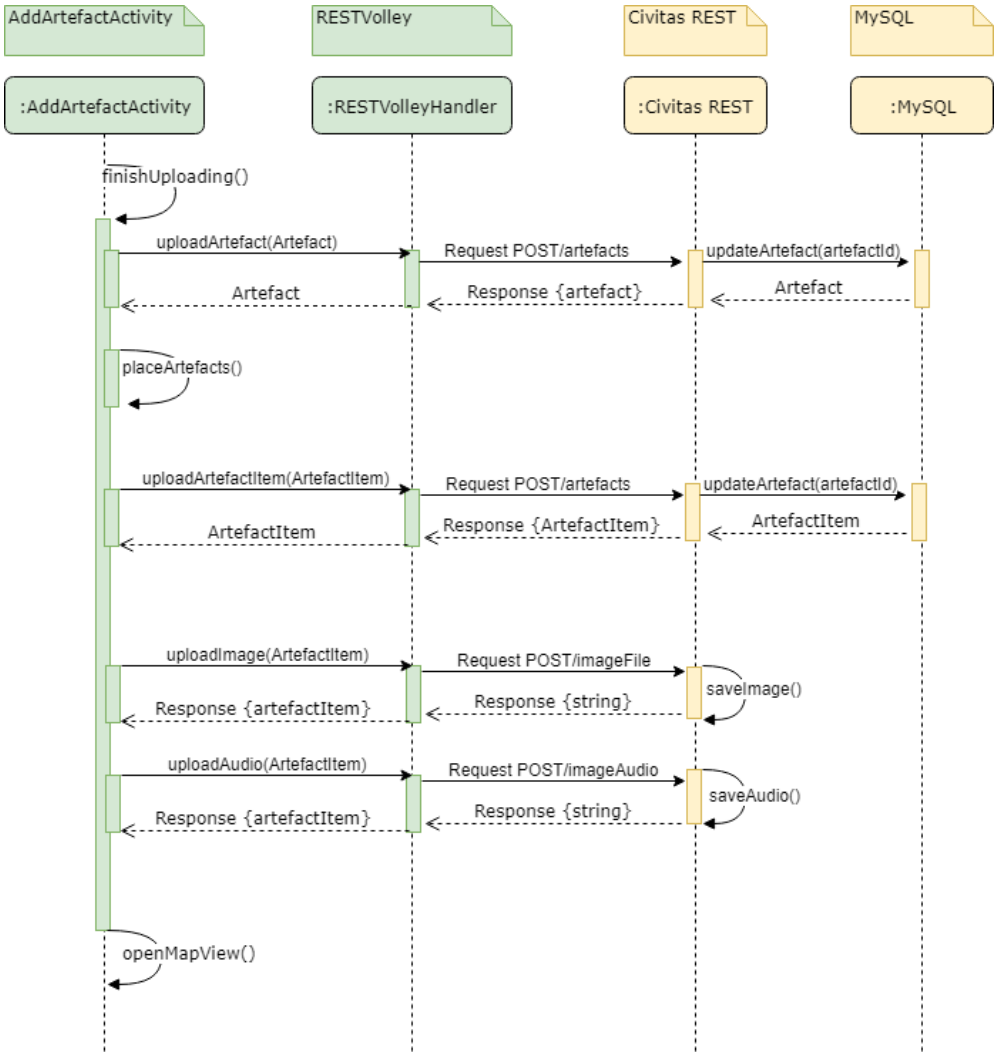


Figure 3.8.: Add artefact timing diagram

It is important to note, when uploading the data from the client device to the server, the parameters of a request body must match the parameters list (argument list) described within the controller on the server side, otherwise a server would send a response with an error status 500 (**Internal Server Error**). This prevents users from uploading random, or harmful data onto a server.

More about a servlet controller and controllers argument list can be found in the server architecture section 3.2.1.

3.1.5. Viewing and editing an artefact

Once a user has been successfully logged in to the **Civitas** system, a user is presented with a map containing icons of the artefacts in the selected location (Figure 4.2).

Each of the icons is a button that is attached to a **viewArtefact** listener. Once a user clicks on one of the artefact icons located on the map, a listener launches a **ViewArtefactActivity** and the window with artefact data is presented to the user (Fig. 4.4).

The process described above seems to be logical and error-less, but there is a case when some of the information (e.g. an image or an audio file) is missing. In this case, this information has to be implicitly fetched from the server. A flowchart diagram (Diagram 3.9) displays the application logic in case there is a missing data in the **ArtefactDetailActivity**.

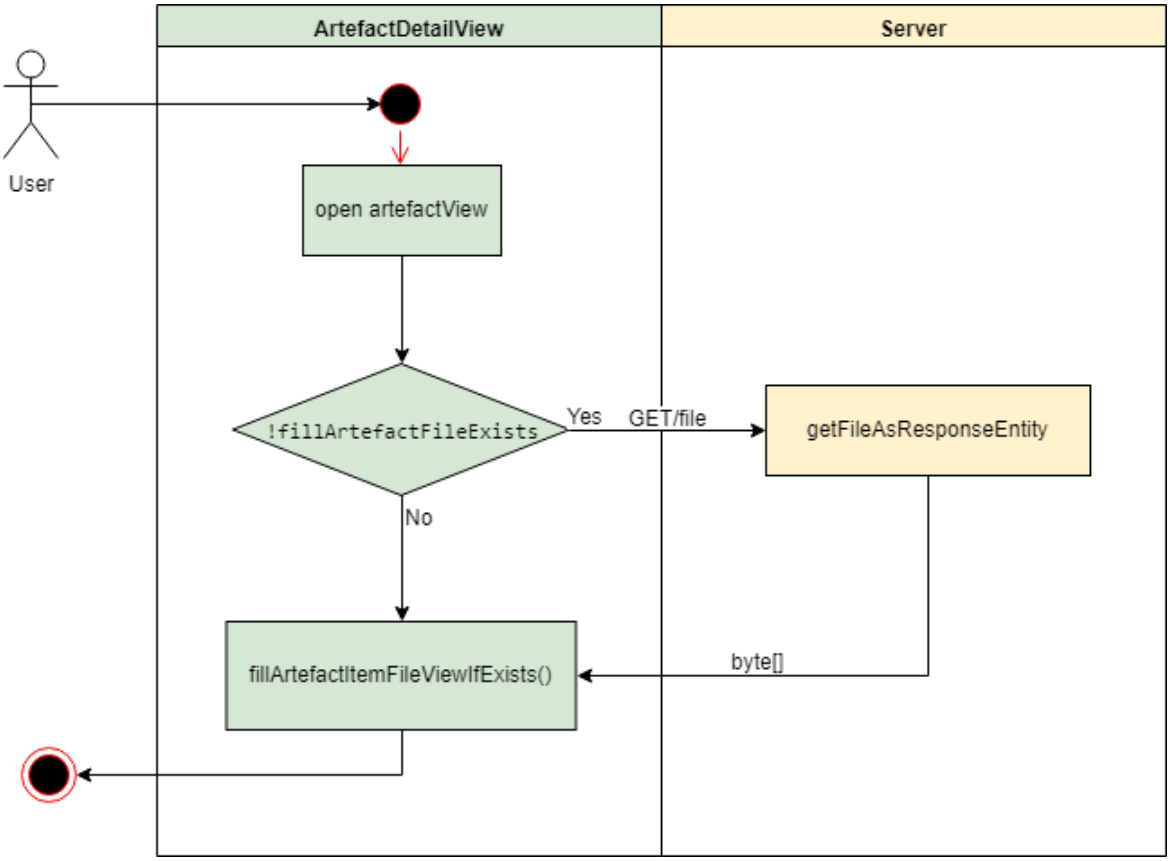


Figure 3.9.: Artefact view flowchart

In order to implement an edit artefact functionality it was possible to re-use the **AddArtefactActivity** from the section **Create Artefact** 3.1.4. An edit button launches **AddArtefactActivity** and passes required parameters to the **Activity**.

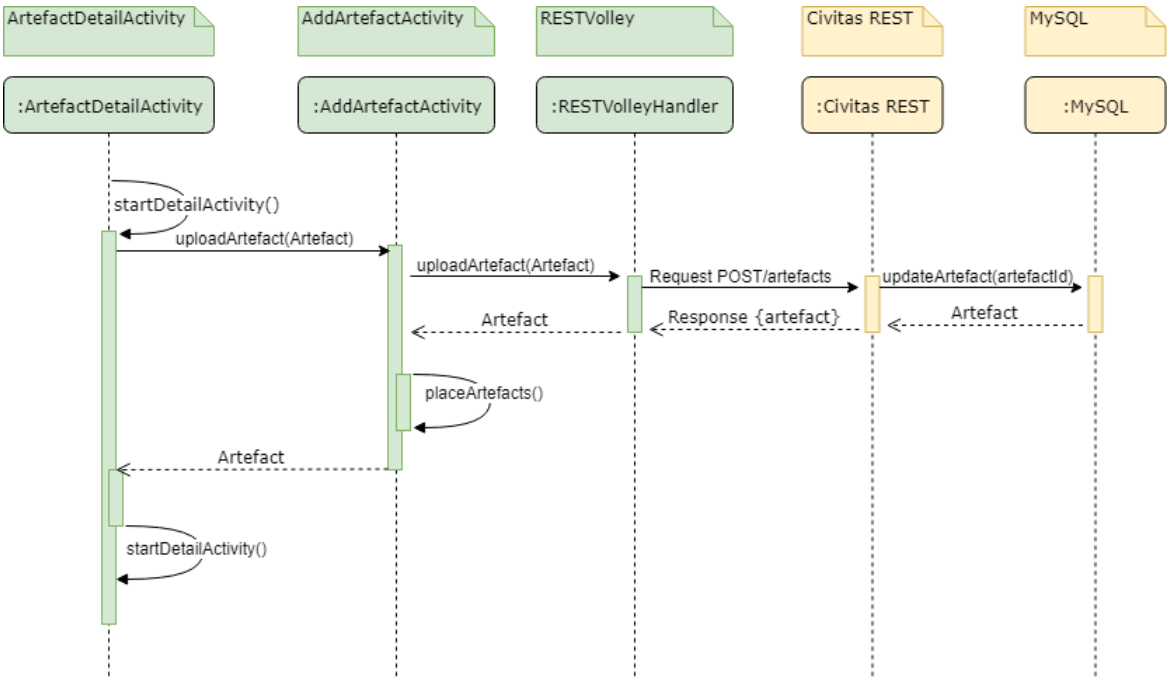


Figure 3.10.: Editing artefact timing diagram

3.1.6. Search function and filters

Filters and a search function simplify an access to the information by sorting out irrelevant data. Existing filters allow the sorting by time period and artefact category type.

There are two potential problems in the previous filter implementation:

- Filter choice is very limited for the user and only allows sorting by hard-coded artefact category types and year when the artefact existed
- Values for the artefact categories and time period are hard-coded in the Android application as a **String** type list. In future, this could potentially lead to a problem, once developer will try updating the existing filter list.

During the development phase of filters, two possible implementations were found. The first approach is to send a GET all artefacts request to fetch all the artefact information on the initialization (like it is done in section Map View) 3.5) and then filter out the artefacts. The second approach is to use a GET artefacts request with an additional filter parameter, which will filter artefacts directly in a database and return filtered results.

There are advantages to the both of approaches. The first approach requires only one heavy query to the database, data is sorted locally later. The second approach would require many small queries each time a filter value is changed and the artefacts that were received can be directly displayed on the map, without further post processing. It was decided to go with the approach.

An example of how a **Category** filter works is displayed in the diagram 3.11. A **categoryId** parameter is appended to the end of a GET request, so the server would only return artefacts, cointaining matching categoryId.

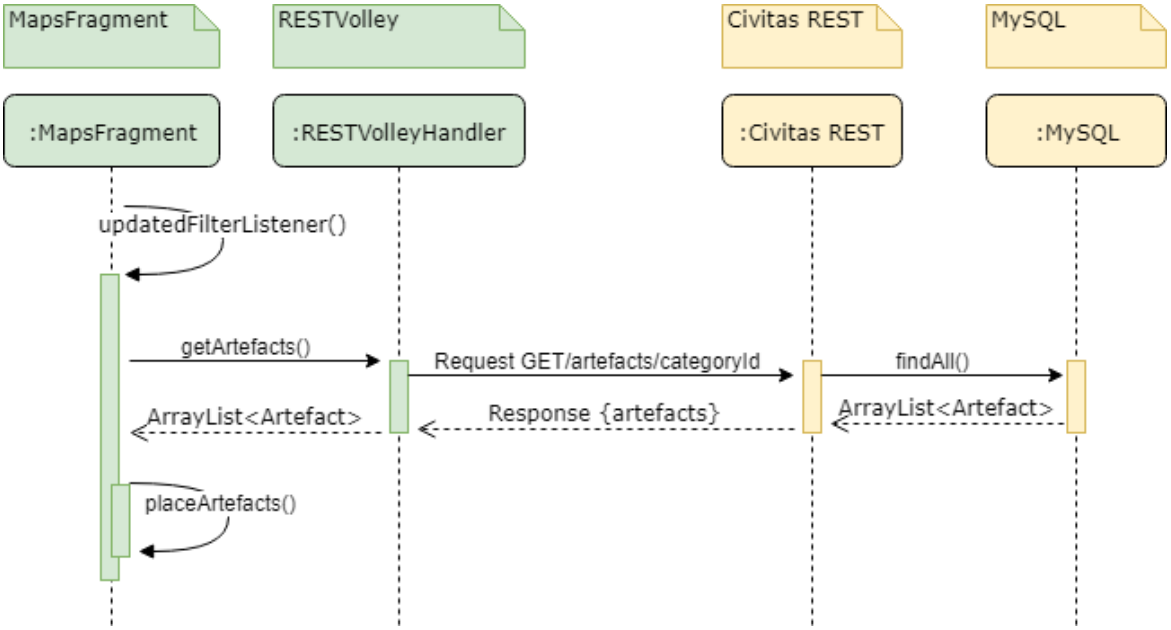


Figure 3.11.: Fetching filtered artefacts timing diagram

3.1.7. File transfer

A process of uploading artefacts has already been discussed previously, however, an important step is still missing. The project specification requires image and audio files to be transferred through Web and stored on a server.

A general concept for file a transfer exists, so an image and audio files can be treated in the same way. In order to transfer a file, a front-end converts it into a **bitmap** array. To be able to utilize same REST API in order to send a file through the web, a **bitmap** array is converted into a string and added to a request body as a parameter as it is displayed in a timing diagram below (Fig. 3.12).

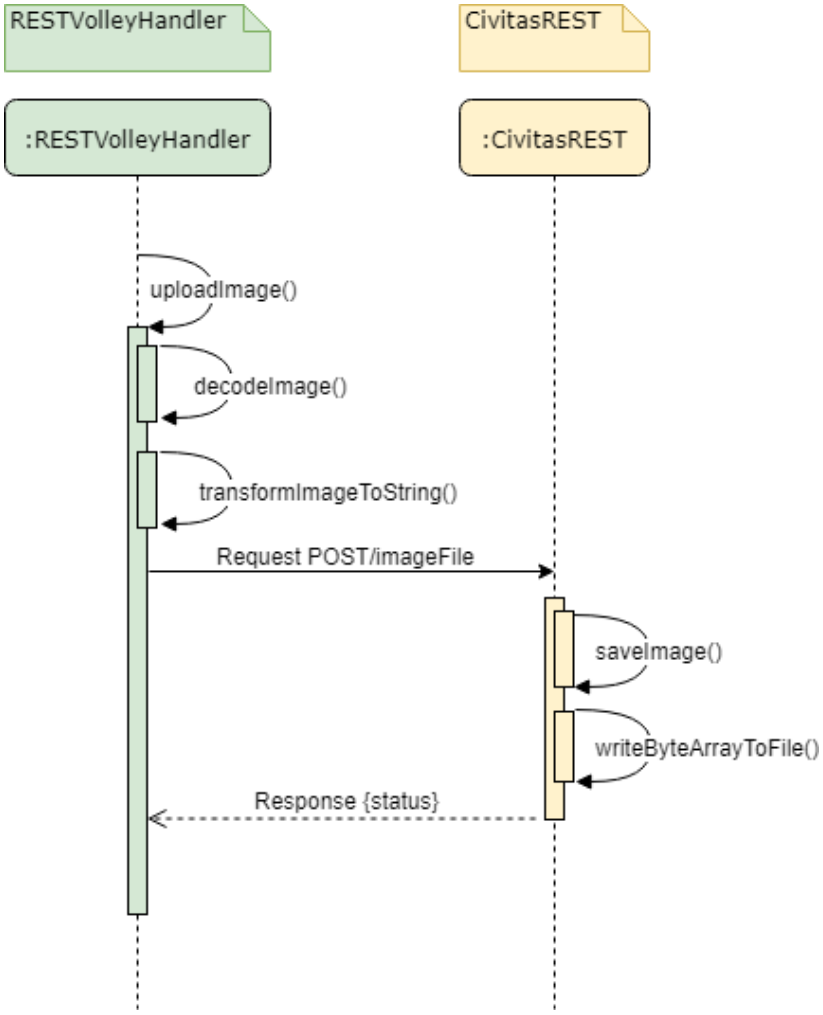


Figure 3.12.: Transferring files through web

3.2. Server-side

In parallel to the front-end, some of the back-end components had to be modified. This section continues with a description of how exactly these back-end components were modified.

3.2.1. Server stack architecture

A server application was built in an architecture style, that closely reminds Model-View-Controller architecture (Fig. 3.13) with exception, that Civitas Boot Spring application does not render the view itself, but responds to the client with the requested data, which is later being processed by android application view (see Fig. 2.6), so the Android application plays a View role in Model-View-Controller architecture.

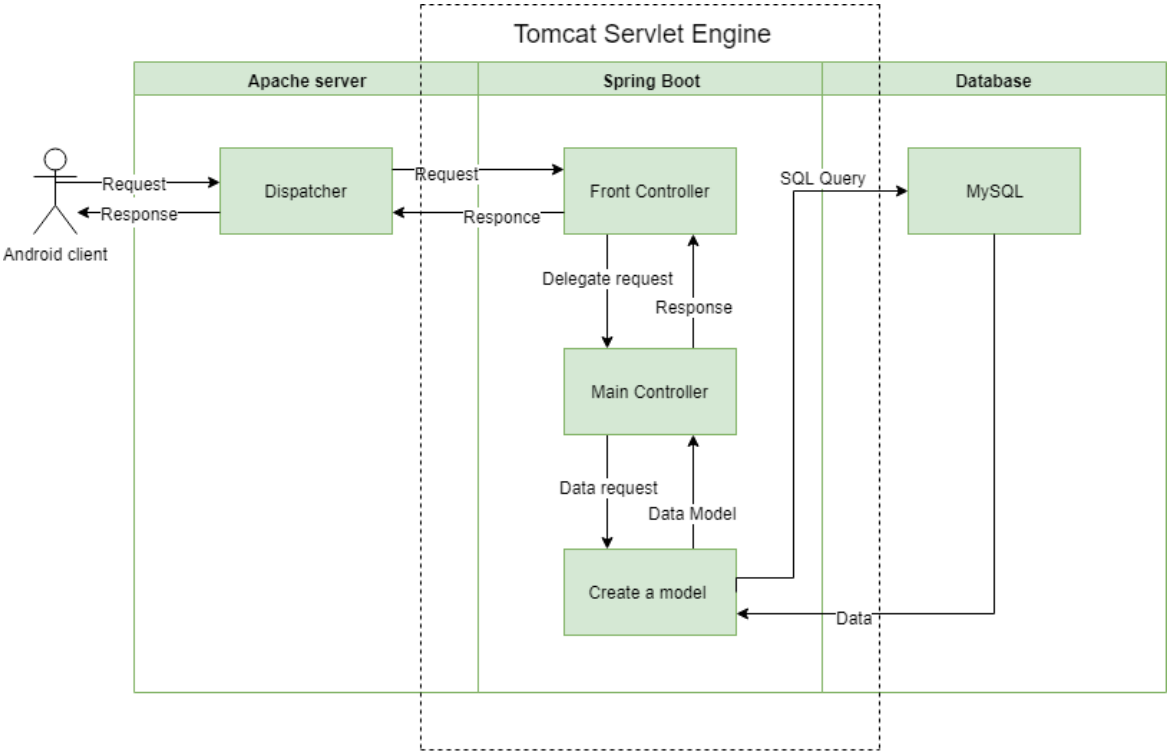


Figure 3.13.: MVC architecture

To begin with, once an Android client sends any request to a server, Apache server looks for a system service that is listening on the provided network port and dispatches the request to it (Apache Tomcat service, since the only service, that this project is utilizing). After, the

service the Tomcat receives a request and passes on to a front controller. The front controller contains routes linked to main controllers with main logic and link to a database.

Since a general functionality of the back-end is clear, it is now possible to move on to the more specific topics related to the back-end design.

3.2.2. User rights and artefact ownership

At the moment it might not be clear what the user rights and the artefact ownership concepts are.

A user rights concept is an abstract term which sets the constraints on the user actions. **Civitas** has the following levels of user access rights:

- **Guest** level. Since a Google Authentication service makes it so easy to log in, it was decided to drop it, because a **Guest** access level would be redundant.
- **User** level. Includes guest's functionality, plus allows users to create, rate, edit and delete it's own artefacts.
- **Admin** level. Includes user's functionality, plus a right to edit and delete any artefact.

To develop a user rights concept, in the **User** entity the field **userType** was introduced (see entity relation diagram 3.14).

An artefact ownership is another abstract term for a concept which helps to determine the belonging of an **Artefact** to a **User**. The relation of a **User** to an **Artefact** is dependent on an **ArtefactItem** entity, which stores **userId** and **artefactId**. This way an A **User** entity can easily have access to an **Artefact** fields, and vice-versa.

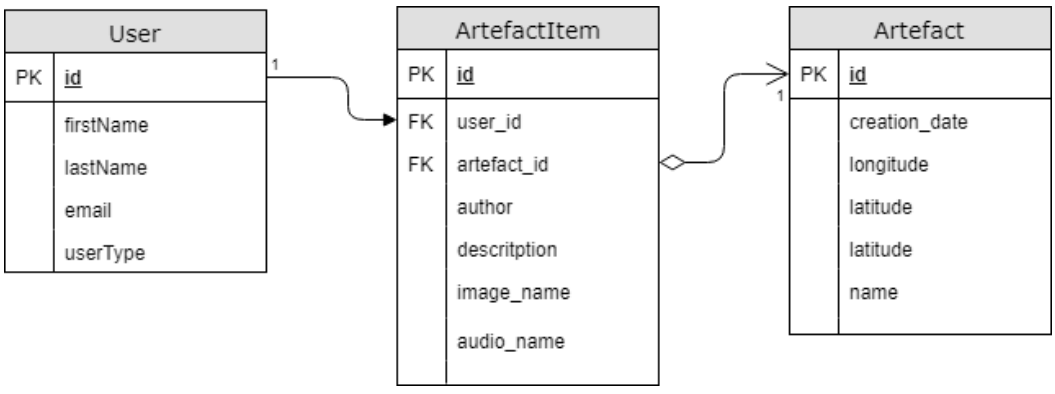


Figure 3.14.: User to ArtefactItem to Artefact entity relation diagram

It was mentioned above that every registered user by default is given normal user rights and currently there is no interface (admin control panel) for the user rights management. This means that at the moment access rights elevation is only possible on a database level.

The admin control panel is a good idea for the future work for the **Civitas** project.

3.2.3. Creating an artefact

It should be clear by now, how do the front-end and back-end interact in between in order to create and store an artefact. However, one important aspect is missing, it is a description of the entity relations.

If one wants to build a feature, which would alter any of the existing data in the database, it is a must to know which table rows are utilized by other tables.

What is meant by that? If one will have a look at the entity relation diagram (Diag. 3.15), one would notice, that in example an **ArtefactItem** entity re-uses an **Artefact** id field, so what is supposed to happen, when a database administrator decides to delete a field in the **Artefacts** table without thinking in advance about entity relations? In the best case a database will just throw an error, informing a developer about a hanging relation, but in the worst case the database would just look up all the hanging relations and delete related data together with the **Artefact** row.

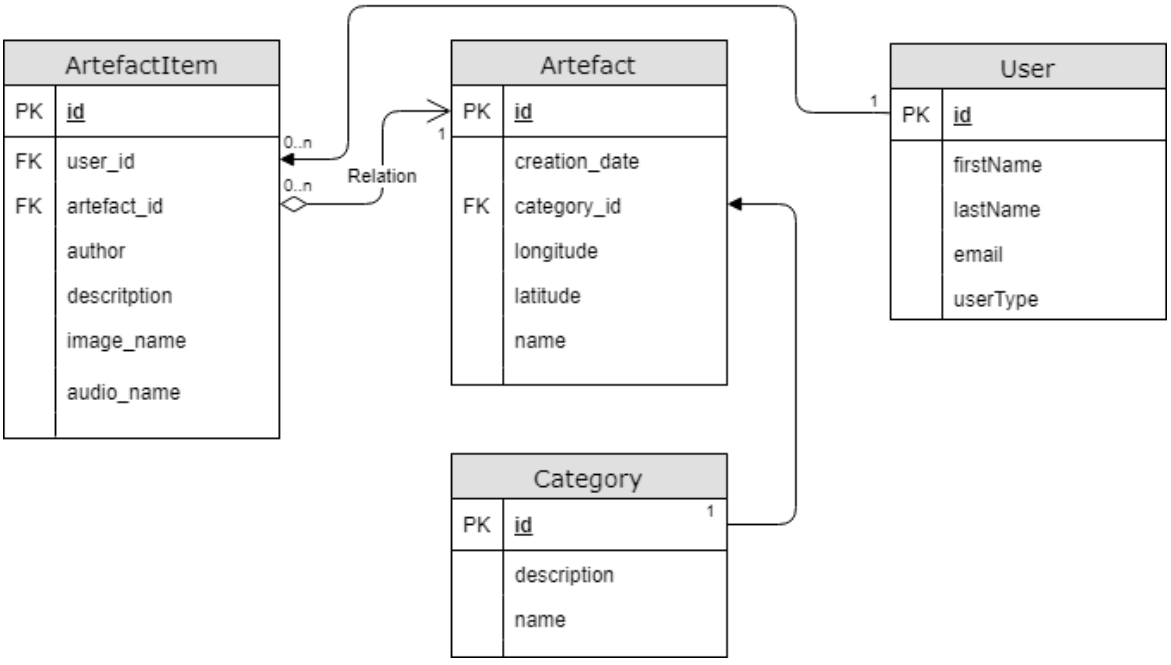


Figure 3.15.: Artefact, ArtefactItem, User and Catergory entity relation diagram

Nowadays, in order to prevent the data loss, most of the databases have protection mechanisms against the problem described above. However, if one wants to master the back-end, knowing such things like the entity relations would increase a confidence of the developer and speed up the development process.

3.2.4. Artefact rating system

An application is planned to be used in the educational purposes, so it is important that every user gets the right information about a historical place. Considering that everyone can register and contribute into the artefact database, it is possible to utilize the same human resource to detect artefacts with a false information.

It was decided to use 5-star artefact rating concept. Every registered user is able to set one rating for each artefact. Once the user has voted the average artefact score will be displayed, instead of the voting scale. To make sure that every user votes only once, the artefact entity was updated with an extra field which keeps the record of users who has already voted; See rating system flowchart 3.16

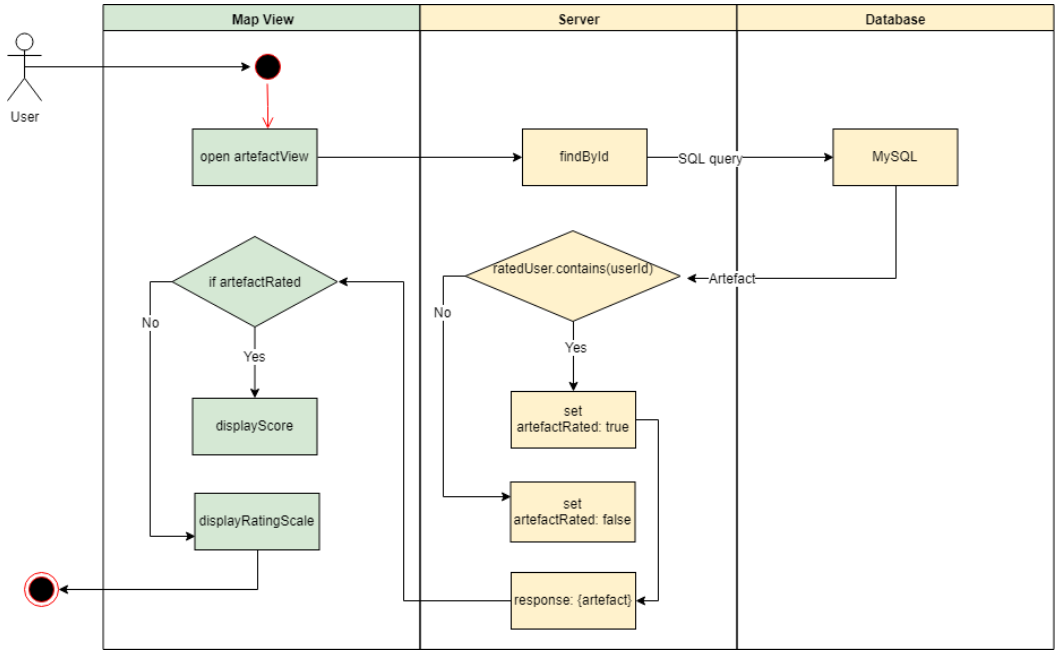


Figure 3.16.: Rating system flowchart

A total rating result is then averaged and displayed as a 5-star rating bar. Designed logic is scalable and can be extended with other features, in example review section.

4. Realization

This section combines all the previous work and presents a final realization of the implemented features.

4.1. Client-side

4.1.1. Google Sign-in

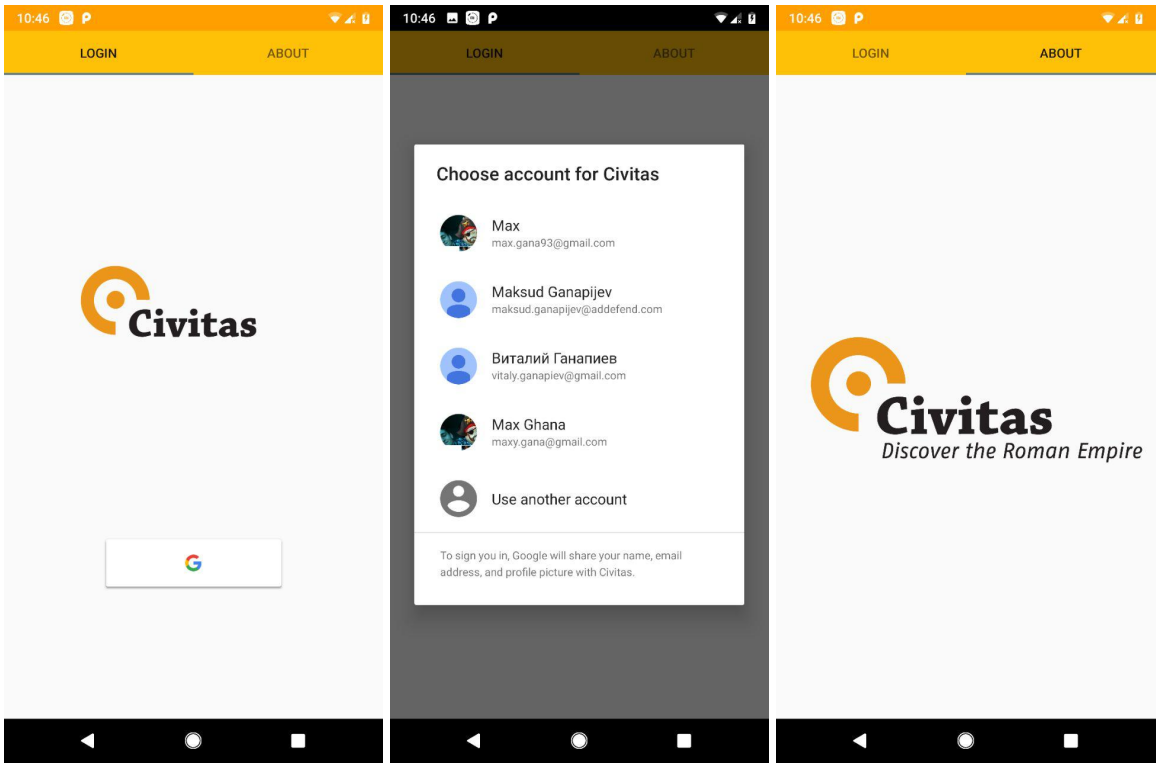
The first screen, that is presented to the user is a login screen. There was an attempt to simplify a login process for the user and implementation process for the developer. A great advantage of utilizing Google Authentication is that a user can log in into the Civitas system, with just two clicks, without providing any text data and without verification.

A final realization of the user authentication screen is presented in the image set 4.1. The look of the login screen, as well as the authentication process may slightly differ depending on an Android version.

At the moment an **About Screen** looks quite basic, but already provides enough information on what is the application about. Of course, it can anytime be easily modified to meet clients requirements.

Google Sign-In works with the Google Authentication server. A user signs in with Google credentials, which are encrypted and transmitted to a server, then credentials are checked for validity and authentication or failure response is sent back.

If an authentication process was successful the user will be able to observe a screen with a map view and the artefacts in the surrounding location, realization of which is described in the following section (Section 4.1.2).



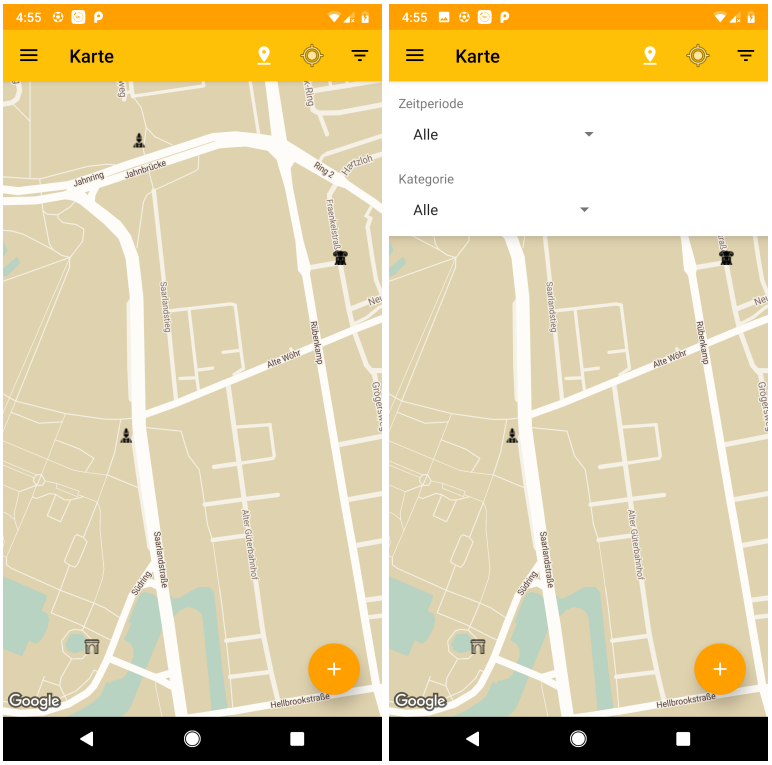
(a) Login screen (b) Account selection (c) About page

Figure 4.1.: Sign in screen in Civitas application

4.1.2. Viewing a map

The idea of the **MapView** and **MapActivity** is to display all the artefacts in the selected location.

Visually **MapView** contains a map with the artefacts, context menu with filters and a floating **AddArtefact** button, an image set 4.2 illustrates the **MapView**. The **MapActivity** is also running several background tasks in order to load additional data.



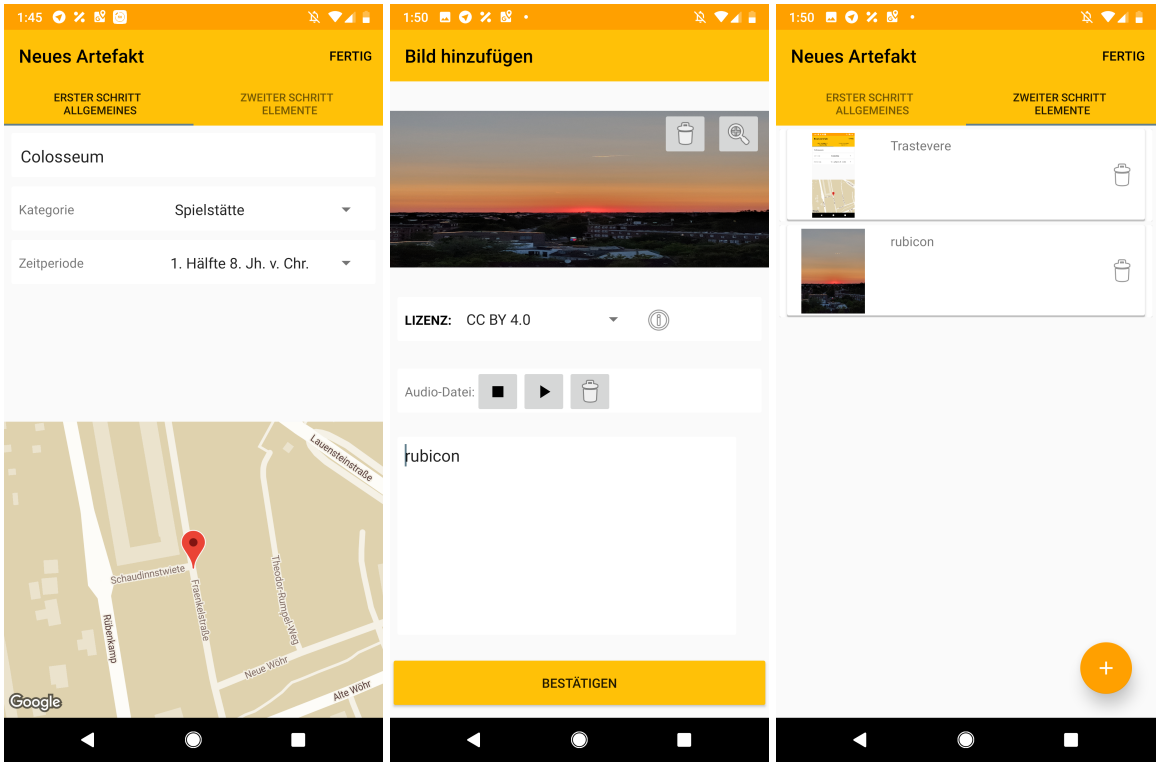
(a) Map with artefacts (b) Filter menu

Figure 4.2.: Map view in an Android application

A tap on the **AddArtefact** button (a plus in a yellow circle), brings up an artefact creation screen to the user, realization of which follows in the next section.

4.1.3. Creating an artefact

One will notice this from the first glance, the UI for creating artefacts is a complex activity. It combines various Android API components and custom created Activities, such as **MapView**, **ImageView**, **TextViews**, **CardView**, **MediaController**. A design of the **AddArtefact** feature can be seen on the image set 4.3.



(a) Login (b) Authentication (c) Authentication

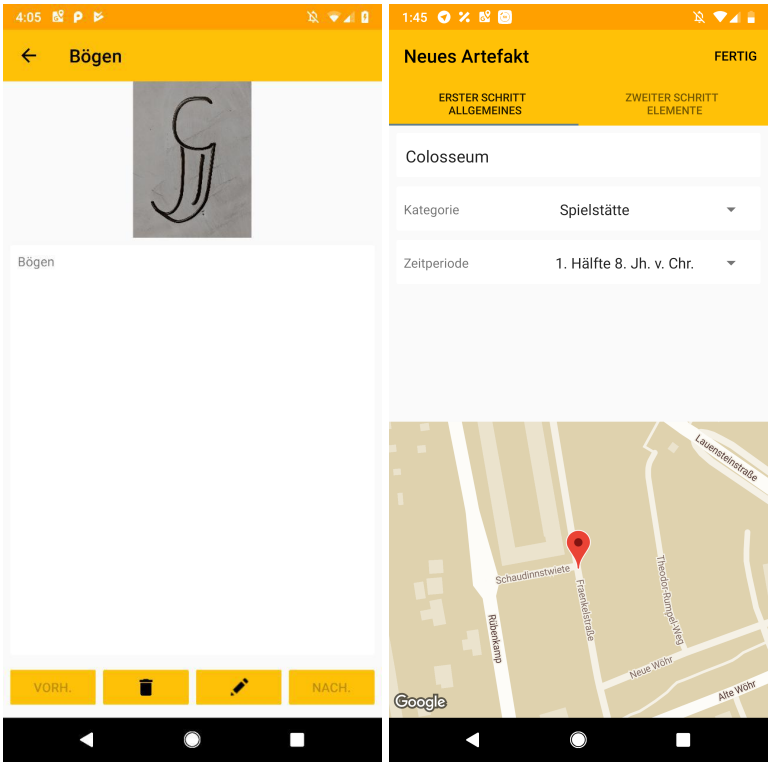
Figure 4.3.: Add artefact view

A user can input a various information about the artefact, classification information, images and an audio note.

After creating an artefact, a user should have an opportunity to view it, a UI realization is in the next section.

4.1.4. Viewing and editing an artefact

As it was described in a Map View section (Sec. 4.1.2), a user can view information on the artefact by clicking on any artefact icon on the **Map View**. An icon click starts the **ArtefactViewActivity**, where a user can view information about the artefact (Fig. 4.4).



(a) View artefact (b) Edit artefact

Figure 4.4.: Viewing an artefact

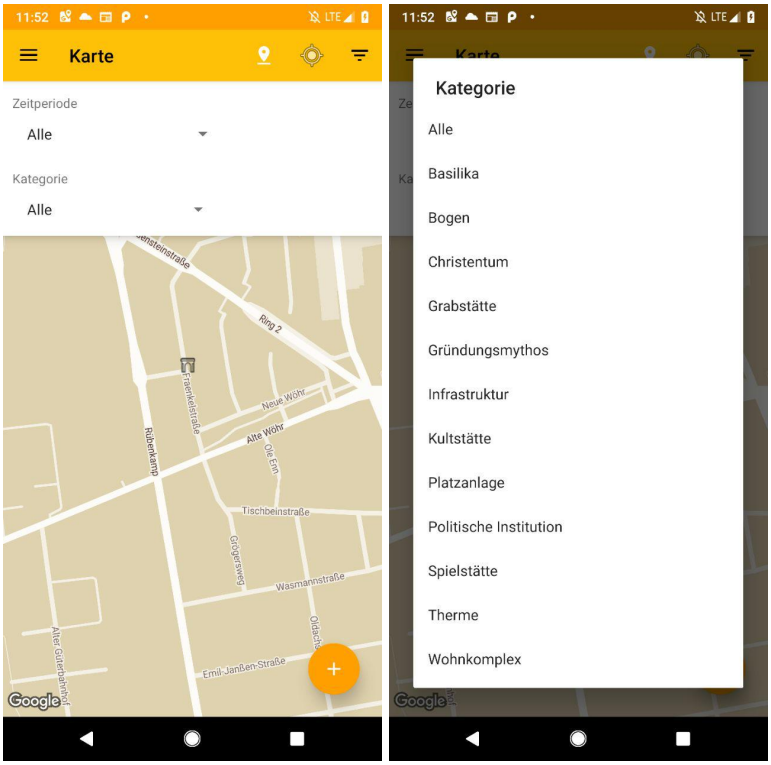
It is interesting to note, that in order to develop an artefact editing feature, it was possible to re-use create artefact component.

In order to restrict other users from editing not their own content, the edit button only turns active if an id of a logged in user matches with the user ID recorded in the artefact entity, or a user **userType** is an admin.

4.1.5. Search function and filters

As it was mentioned above, the filter implementation was re-worked. The difference is that filter values are now stored on the server and the Android application fetches filter values during the launch.

This implementation gives the developer a chance to modify filter values, without taking care of the synchronization of the filter values between the server and the Android application.



(a) (b)

Figure 4.5.: Artefact filter realization

The Android application has other minor improvements, however, they are too small to have a separate section, but are just enough to be mentioned. A login page was reworked, and has an about part now, as well as delete artefact function was added to the artefact view.

Here the section with the realization of all the front-end modifications ends, and it brings this study to the last section of this chapter: the back-end realization.

4.2. Back-end

The first step into this study was reading the project description, and the second step was to open the provided Android application and try it out.

The first Android application use-case that was run, was an attempt to create an artefact and view it on the same device, the second use-case test that was run is displayed in the use-case diagram 4.6, and only then one could notice that if the **User 1** has uploaded an artefact with an image and audio information, the **User 2** could not see this image or audio information on his device. This test has started the whole back-end debug story, which later grew into a back-end migration.

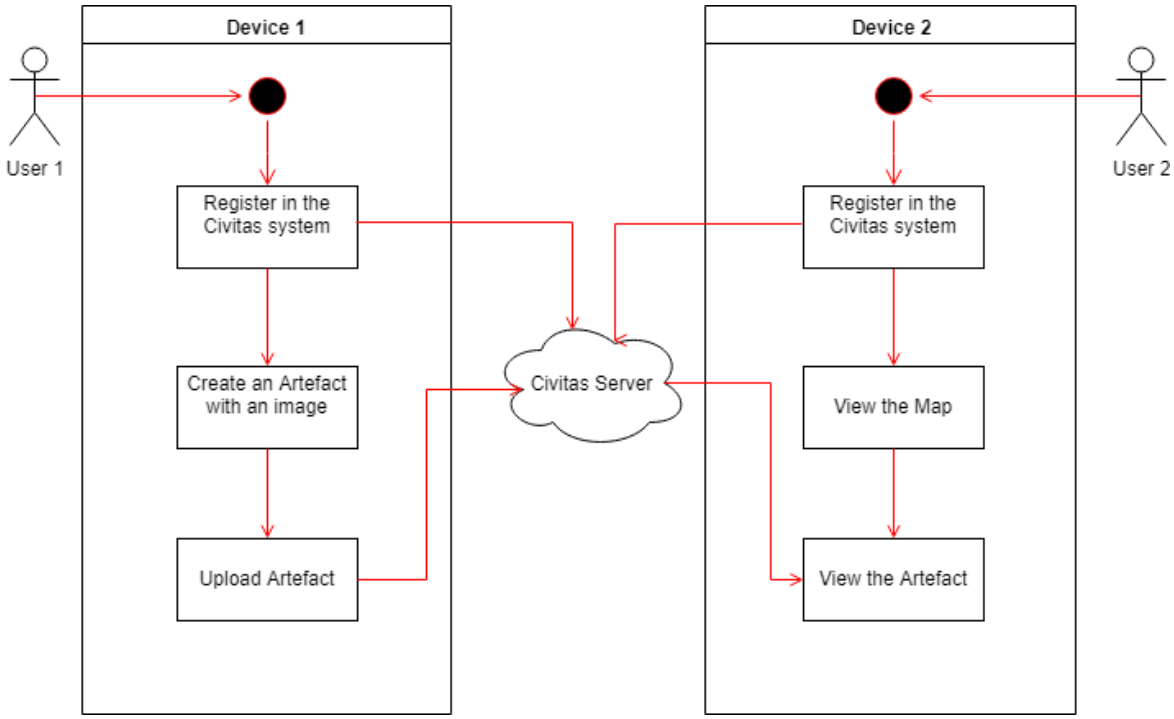


Figure 4.6.: Upload artefact use-case

All the information that the developer was provided after a test, was a success response with no warning or error message. It was then decided to try to access a server with the **SSH** (Secure Shell) and **FTP** (File Transfer Protocol) protocols with the credentials provided with the project files. Both attempts were unsuccessful, and it seemed like the server application host was not interested in hosting the **Civitas** application anymore, and the only possible solution in this situation was to replicate the server architecture on the private server, without seeing the original one.

After a documentation analysis it was decided upon which software stack will be utilized for the back-end of **Civitas** (Section 3.2.1).

4.2.1. Back-end deployment

During this study a great amount of work has been dedicated to the research of how the server software works, and how to write the **Boot Spring** servlets and how to configure them, so they can later be accessed remotely by the user. It was a progress for the project development generally, but for this study, it has come with a cost of time, and some of the features listed in the requirement could not be implemented.

Nevertheless, the project benefited from the migration onto a private server, due to the following reasons:

- The developer has full control over the server machine, as a result, possibility to install additional software packages if needed.
- A developer has a possibility to write a web interface to access a server file system and store any kind of additional data.
- Allows a developer to perform fast deployments, without introducing third parties, as well as allows to and access all logs, as a result, speeds up the debugging process.

As a result, a virtual private server with the pre-installed Debian distribution was configured to run the **Civitas** back-end application. The server is currently being paid from the developer's budget, so it makes sense that the back-end will migrate at least once again.

For the future work on this project, a sheet with deployment instructions was written. This sheet was added to an appendix CD.

5. Tests

Surely enough every developer at some point of his career, asked oneself why write tests at all? The list can be endless, but here are some major reasons why a developer should write tests:

- Improve and maintain the product quality
- Increase a confidence when shipping a product
- Perform routine checks, that humans can't perform fast
- Helps to write shorter, but better understandable code

5.1. Unit tests

In software engineering, unit testing is a testing method used in order to test individual blocks of a source code or modules of a computer program. It makes sense to run unit tests first because they are fast to run and easy to write.

5.1.1. Testing back-end with Postman mock service

Throughout the development process, delays on the front end or back end can hold up developer from completing his work efficiently. Postman allows a developer to work on the endpoints (controllers) without relying on a front-end UI in order to simulate API requests. Postman mock service provides an option to test back-end endpoints without involving front-end.

A set of tools available for endpoint testing is displayed in the "Postman mocking service interface" figure (Fig. 5.1).

In this example a mock test for **POST/artefact** endpoint is displayed. A developer can select a request type: GET, POST, DELETE, UPDATE; add the parameters into a body of a request and send it to the back-end. The UI block under parameter list will then display response status and response body contents if any data was attached to it.

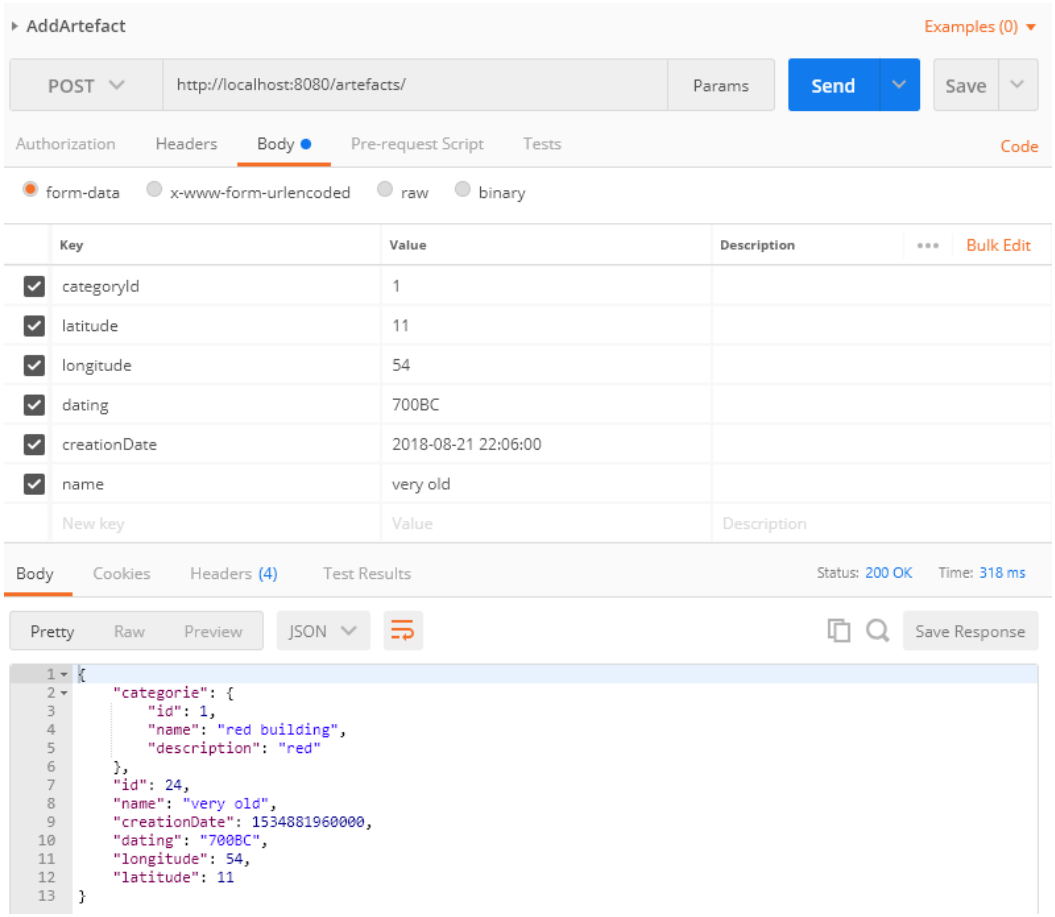


Figure 5.1.: Postman mocking service interface

6. Conclusion

Great number of the projects are started every day, but end up in stagnation, and even more ideas that never moved into actual conception for a various reasons. Not everything is meant to be finished, but many people have a boatload of projects and to-dos that have been put on the shelf, and software development projects is no exception.

In this study, a software developer had a task to make a progress on an existing multiple-user Android application for historical monuments, by implementing the features described in the technical task of the project, as well features not listed in the technical task, but in developer's opinion would improve the usability of the Android application.

6.1. Deliverable

As a result, an existing **Civitas** system has the new features listed further in the text.

Now has an Android application with a comfortable one-click login feature. The users of the application have an interface to exchange the image and audio information as well. The users and the administrators of the **Civitas** system also can edit and delete artefacts from the **Civitas** database.

Furthermore, now **Civitas** back-end exists independently from the previous host, which gives the back-end developers an unlimited room for the project improvement. An instruction sheet has been added to the appendix section, for anyone who would want to deploy **Civitas** on a new machine.

Due to the problems with an access to a server that occurred at the beginning of the project, a time constraint did not allow to implement a search feature and an artefact rating feature.

6.2. Future work

There is still a lot of work that can be done in order to improve the **Civitas** project.

- A refactoring of an Android code. At the current stage, it would be nice to completely test the **UI** and fix all the navigation bugs.
- A security improvement. First of all, the token system should be introduced for the users who are signed in. Second, at the moment all the data is being transmitted with the unprotected GET and POST requests, so encryption should be added for the data protection. And lastly, a captcha method should be introduced, when adding an artefact, to prevent a potential problem with the bots in the future.
- A search field has to be introduced for the easier access to the data.
- An artefact rating and comment section should be added to the artefact view, which will make the application more interactive for the users.
- A page with a list of the artefacts owned by the user should be introduced for easy access to the artefacts. Currently, a user has o look for his artefacts on the map.
- And finally an admin control panel.

There are more features that one can think of in order to improve **Civitas**, but the presented list is enough to make up a task of another bachelor project.

It seems like there is always the room for an improvement, not just in the software engineering, but everywhere. One has to do a constant analysis of mistakes in order to understand at what point and what went wrong, so in the future this knowledge could be utilized in order not to make these mistakes again and achieve better results.

Bibliography

[Aleksa Vukotic 2011] ALEKSA VUKOTIC, James G.: *Apache Tomcat 7*. 2011. – URL https://books.google.de/books?id=kNVe8lzTec0C&hl=de

[aleksys 2017] ALEKSYS: *How to write in Spring in 2017*. 2017. – URL https://habr.com/post/333756/

[AloneCoder 2017] ALONECODER: *REST is a new SOAP*. 2017. – URL https://habr.com/company/mailru/blog/345184/

[Bert Bates 2009] BERT BATES, Kathy S.: *Head First Java, 2nd Edition*. 2009. – URL http://shop.oreilly.com/product/9780596009205.do

[Brian Totty 2009] BRIAN TOTTY, Marjorie Sayer Anshu Aggarwal Sailu R.: *HTTP: The Definitive Guide*. 2009. – URL http://shop.oreilly.com/product/9781565925090.do

[DigitalOcean 2014] DIGITALOCEAN: *How To Install and Configure Apache Tomcat on a Debian Server*. 2014. – URL https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-apache-tomcat-on-a-debian-server

[Feiktisov 2011] FEIKTISOV, Artiom: *REST vs SOAP. Feel the difference*. 2011. – URL https://habr.com/post/131343/

[fspare 2014] FSPARE: *Java 8, Spring, Hibernate, SSP*. 2014. – URL https://habr.com/post/222077/

[Gariks 2013] GARIKS: *Simple example on using Volley library*. 2013. – URL https://habr.com/post/188860/

[Google 2016] GOOGLE: *Add Google Sign-In to Your Android App*. 2016. – URL https://developers.google.com/identity/sign-in/android/

[Google 2018a] GOOGLE: *Android fragment concept (Figure 2.5)*. 2018. – URL https://developer.android.com/guide/components/fragments

[Google 2018b] GOOGLE: *Fundamentals of Testing*. 2018. – URL https://developer.android.com/training/testing/fundamentals

[Google 2018c] GOOGLE: *Send a simple request*. 2018. – URL https://developer.android.com/training/volley/simple

[Griffiths und Griffiths 2017] GRIFFITHS, David ; GRIFFITHS, Dawn: *Head First Android Development, 2nd Edition*. 2017. – URL https://www.oreilly.com/library/view/head-first-android/9781491974049/

[Percival 2017] PERCIVAL, Rob: *The Complete Android N Developer Course*. 2017. – URL https://www.udemy.com/complete-android-n-developer-course/learn/v4/overview

[Phillip Webb 2018] PHILLIP WEBB, Josh Long Stephane Nicoll Rob Winch Andy Wilkinson Marcel Overdijk Christian Dupuis Sebastien Deleuze Michael Simons Vedran Pavic Jay Bryant Madhura B.: *Spring Boot Reference Guide*. 2018. – URL https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[PostdotTechnologies 2018] POSTDOTTECHNOLOGIES, Inc.: *Postman documentation*. 2018. – URL https://www.getpostman.com/docs/v6/

[Raphael Hertzog 2015] RAPHAEL HERTZOG, Roland M.: *The Debian Administrator's Handbook*. 2015. – URL https://debian-handbook.info/browse/stable/

[Reto Meier 2018] RETO MEIER, Ian L.: *Professional Android*. 2018. – URL https://www.amazon.com/Professional-Android-Reto-Meier/dp/1118949528

[Richardson 2013] RICHARDSON, Leonard: *RESTful Web APIs: Services for a Changing World*. 2013. – URL http://shop.oreilly.com/product/0636920028468.do

[ShareLaTeX 2018] SHARELATEX: *ShareLaTeX guides*. 2018. – URL https://www.sharelatex.com/learn

[Sharma 2018] SHARMA, Lakshay: *Postman Tutorial*. 2018. – URL http://toolsqa.com/postman-tutorial/

[SiteGround.com 2018] SITEGROUND.COM: *phpMyAdmin Tutorials*. 2018. – URL https://www.siteground.com/tutorials/phpmyadmin/

[TheApacheSoftwareFoundation 2017] THEAPACHESOFTWAREFOUNDATION: *Apache Tomcat 8 documentation*. 2017. – URL http://tomcat.apache.org/tomcat-8.0-doc/

A. Appendix

This Bachelor Thesis contains an appendix of program listings and program code on a DVD disk. This Appendix is deposited with Prof. Dr. rer. nat. Henning Dierks.

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, October 23, 2018

City, Date

sign