WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

FIELD: INFORMATYKA
SPECIALIZATION: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH

ENGINEERING THESIS

Mobile application providing offline access to data published in a student services system

Aplikacja mobilna zapewniająca dostęp offline do danych publikowanych w systemie obsługi studentów

AUTHOR:
Justyna Skalska

SUPERVISOR:
Dr inż. Tomasz Kubik, W4/K9

GRADE:

WROCŁAW, 2020

*To my **Mother**, thank you for always believing in me and keeping my spirits up.*

*To my partner, **Aleksandra**, thank you for being with me when I needed it and putting up with my moods. Especially these last few months.*

*To **Małgorzata** and **Grzegorz**, thank you for constantly making me laugh. It really helped.*

*To my supervisor, **Dr inż. Tomasz Kubik**, thank you for your time, support, and expertise during my studies and this project. I appreciate the last minute help and understanding.*

Contents

List of Figures

List of listings

Abbreviations

ACID (*Atomicity, Consistency, Isolation, Durability*)
AJAX (*Asynchronous JavaScript and XML*)
AOT (*Ahead-Of-Time*)
API (*Application Programming Interface*)
APK (*Android Application Package*)
BLoC (*Business Logic Component*)
DAO (*Data Access Object*)
DSL (*Domain Specific Language*)
FPS (*Frames Per Second*)
IDE (*Integrated Development Environment*)
Java EE (*Java Enterprise Edition*)
JIT (*Just-In-Time*)
JSON (*JavaScript Object Notation*)
JVM (*Java Virtual Machine*)
MVC (*Model-View-Controller*)
POJO (*Plain Old Java Object*)
SDK (*Software Development Kit*)
STX (*Streaming Transformations for XML*)
UI (*User Interface*)
URI (*Uniform Resource Identifier*)
WAP (*Wireless Application Protocol*)
WORA (*Write Once, Run Anywhere*)
XML (*eXtensible Markup Language*)
XSLT (*eXtensible Stylesheet Language Transformations*)
YAML (*YAML Ain't Markup Language*)

Chapter 1

Introduction

We are living in the digital era where a mobile phone is everybody's best friend. The Internet is all around us, and it almost feels like a commodity. People consider it a fundamental human right, like water. They are used to being online every day from the moment they open their eyes until the moment they shut them again at night. Everyone is obsessed with immediate access to information. That is why mobile applications are so significant these days. They provide a quick way of finding the data that we are interested in.

Not so long ago, mobile phones allowed only for calls and messages. After some time, simple applications started to appear. Most of them were mobile games that people could download on their computers and transfer to their devices. They could also download them using, then very slow, Internet connection and WAP protocol. Currently, there are stores where users can search for apps by name, type, or functionalities. It is very comfortable because customers can find everything that they want in one place. It is also convenient for developers because they get access to a bigger audience when publishing their program on such a platform.

On the other hand, we could use our computers for all tasks. The problem is that they are massive and heavy in comparison to phones. People carry the latter with them everywhere. They allow us to connect to the Internet, use built-in GPS, gyroscope, Bluetooth, cameras, and more. To use computers, also laptops, we have to sit down, turn them on, sometimes log in, only then we can order some things or browse the Internet. That is why there is a need for mobile applications. They allow us to pay phone bills, order clothes, play games, and chat with friends, et cetera. It does not matter if we are traveling in a metro or lying on a couch in a living room.

There are some places where the Internet connection is extremely slow, unstable, or it does not exist. When this happens, mobile applications should be ready to handle the situation. They download data when there is a connection and then store it locally. Thanks to this, users do not have to have constant access to the World Wide Web. Sometimes a service from which our app is getting all the data is down. Saving information on a device allows users to use the app even though the server is inaccessible.

Some universities have their mobile applications allowing students to browse news feed from their faculty and university, see calendars, grades, and other information. They let users send and check emails from other students and lecturers. Some educational institutions do not have mobile apps. They use systems that can be accessed only via web browsers. Wrocław University of Science and Technology also has one. It is not optimized for mobile devices, and some functionalities are difficult to access from a phone. The service also gets unavailable from time to time, returning an error message. Usually, when many people try to access it at the same time, for example, during an examination session. That has been an inspiration for the project described in the thesis. It is designed to improve the mobile experience and allow users to access the data when the system is offline.

Two similar solutions exist in Poland. Both of them are represented in Figure 1.1. One was explicitly created for the University of Warsaw (Fig. 1.1a) [27]. We draw inspiration from it to include some functionalities like news feed and mailbox in our project. Unlike the first app, the second one (Fig. 1.1b) can show data for more than one university [25]. Users have to choose what university they attend, and then they can see their calendar, grades, and some other data from the chosen university system. This is what we wanted to achieve while designing our application. Most of our attention went to creating a simple and extensible API. The program represented in the document was designed to provide better user experience than the second solution described here, and it is supposed to be easily expendable by YAML files, new widgets, and views.

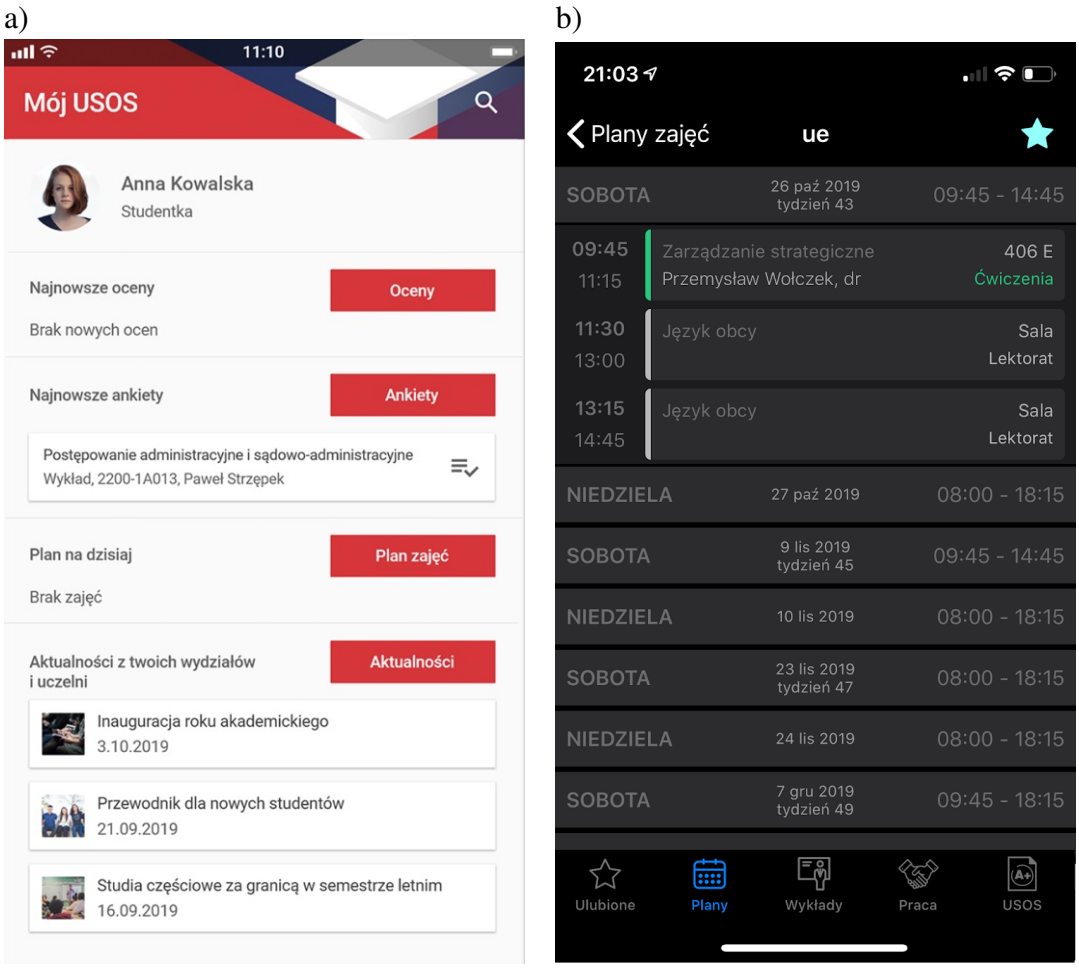

Figure 1.1: Interfaces of two mobile applications supporting student service used in Poland: a) Mobilny USOS UW, b) Kiedy wykład

1.1. Objectives and Scope of Thesis

The main objective of this thesis is to create a mobile application for both iOS and Android, which will allow users to log in with credentials for their educational institution system, and get access to data published in a student service system. It will store some of the information in a local SQLite database to let users access them offline. Thanks to it, students will be able to see the details when the student services system is down, or they have limited access to the Internet.

The mobile application will allow logged-in students to access:

- university or faculty news feed;
- profile details;
- calendar with class details;
- list of grades for every semester;
- messages and their details;
- list of payments.

Besides the mobile application, the project requires a server that will act as a service transforming data between two schema-incompatible systems. The app will send a data request to the server using JSON format. This request will be processed there, mapped to a schema of a selected student services system. After it has been mapped, it will be sent to the system requesting data. The last step is to transform the received information and return it to the mobile application in an expected format. The mapping files for universities will be created using the YAML language. This structure will allow the system to be easily expanded with new universities.

1.2. Thesis Structure

The current chapter acts as an introduction to the topic, provides examples of similar applications created in the past, and explains what the objectives and scope of the thesis are. The second chapter talks about the overall description of the project. It includes a use case diagram with functional as well as non-functional requirements and lists all technologies used during the development of the application. The third part features system architecture along with the database and user interface design. It includes wireframes depicting every screen available in the mobile application. The fourth chapter covers the project implementation. It goes into detail about the database configuration as well as the mobile app and server code. The fifth chapter explains how the applications were tested and provides code snippets. The seventh chapter presents the appearance of the created system and describes the installation and implementation of the program in a production environment. The last chapter summarizes the effects of work and outlines possibilities for the future development of the project.

Chapter 2

Project Assumptions

2.1. Overall Description

The main goal of the project is to create a mobile application that can be connected to an API of any university. It will be done using the Flutter framework and the Dart language. The app will be ready to run on both iOS and Android devices. Every student is required to have an existing account in their university system and log in to the mobile app. After they log in, students will gain access to these pages:

- home (user details and news);
- calendar with a schedule;
- grades;
- messages;
- message details;
- payments.

None of application's functions can be used by an unauthenticated user. Users cannot make changes to any data shown on the listed pages. It is obtained from their university services system and stored on their local devices as read-only information.

When a student logs in, a token received from the student service system will be saved to secure storage on the mobile device. It will be removed if the user explicitly logs out of the application. The token will be used to obtain data from the university system. It will also allow the system to store information about users and will not require them to log in each time they open the application. To log in, users will have to complete a form by selecting their university from the drop-down list and providing login and password to the account in their university system.

The homepage will show user details and news from the users' faculty and university. The next page will display a calendar that users can click on to select a date. After picking the day, students will receive a list of planned lectures with specifics like start and end time, lecturer, classroom, et cetera. The grades and payments pages will provide a list of items with some type-specific details. The former will also allow students to calculate the overall average grade of all grades relative to the ECTS weighting of the courses they have completed. The last one is the messages page. It will present all messages received by users in their university system. After they select one of the e-mails, they will be redirected to a more detailed view.

2.2. Functional Requirements

All functional requirements are listed below. They define the functions of the system as a specification of behavior between outputs and inputs. The operations available to the user are collected in the following use case diagram (Fig. 2.1).

- The software system should be integrated with university APIs.
- The system will translate JSON from the mobile application request into JSON compatible with the university API.
- The software automatically validates customers against the university API.
- The system will limit access to authorized users.
- Users should be able to browse calendars and calendar events.
- Users should be able to view the grades list.
- The system should allow users to calculate their average grade.
- Users should be able to browse messages.
- Users should be able to see message details.
- Users should be able to browse the payments list.
- Users should be able to see the news feed.
- Users should be able to view user details.
- Every user has to have an account in one of the available university systems.

2.3. Non-functional Requirements

The non-functional requirements are listed below. They specify criteria that can be used to evaluate system performance rather than specific behaviors.

- The system will be composed of 3 layers: mobile database, mobile application, and transformation server.
- The mobile SQLite database will store user data, grades, calendar events, payments, and messages.
- The mobile application will support Polish and English.
- The transformation server will be created using the Java language and Spring Framework.
- The server will use Spring Boot to accelerate the preparation and configuration.
- The transformer will utilize Spring Could Config to provide a dynamic and simple way to update the university API configurations. The server will not have to be restarted to apply new settings because they will refresh automatically.
- The configuration will be served as YAML files.
- All requests will utilize the JSON format.
- The transformation specification will use JSON format and will be handled by the Jolt library.
- The authentication will be managed by the server and the university's external API.
- The mobile application will be made using the Dart language and Flutter framework. It will allow generating two applications for both iOS and Android, using one codebase.
- MockServer will be used to mock the external API to which the transformation server will send requests. It will ease the development and speed up testing.
- Docker Compose will be used to define and run multi-container Docker applications, in this case, the transformation server and its configuration.
- Version control will be supported by Git and GitHub platforms.
- All tasks and issues will be managed with GitKraken Glo Board.
- All development will be done using Android Studio IDE.

2.4. Use Cases

A use case diagram (Fig. 2.1) represents the main functionalities available to users. All of them require users to be authenticated. The diagram is simplified and does not contain connections between the "Authenticate/Authorize" use case and all the others.

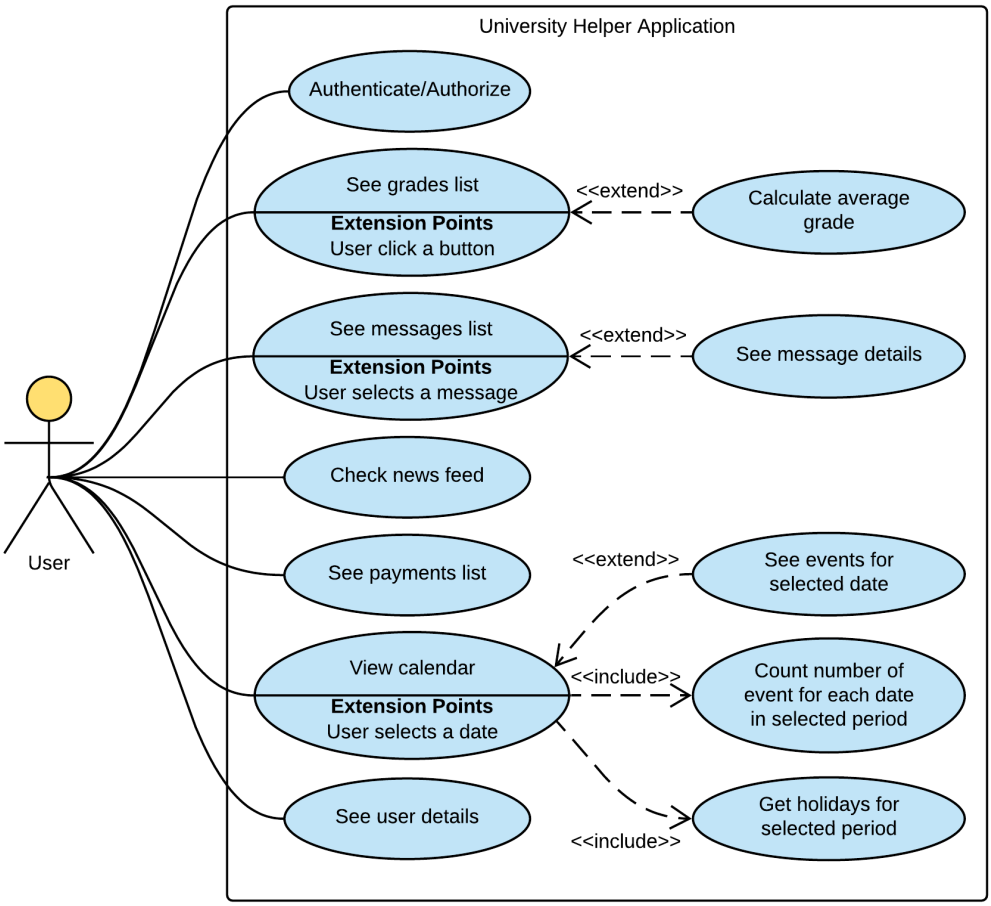

Figure 2.1: Use case diagram

Authenticate/Authorize

Primary Actor: User
Scope: Mobile application
Level: Fish level
Brief: The user logs in to the mobile application.
Trigger: The user open the mobile application.
Preconditions: The user has an account in one of the available university systems.
Basic flow:

1. The system's login screen has a drop-down with available universities.
2. The user chooses her/his university and enter credentials.
3. The user submits the form.
4. The mobile application sends the credentials to the server.
5. The server validates the credentials and sends a response back to the mobile application.
6. The system redirects the user to the homepage.

Postconditions: The user is authenticated and can access all screens of the mobile application.

See grades list

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see her/his grades.

Trigger: The user selects "Grades" screen from the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the "Grades" page.
2. The mobile application sends a request to the server.
3. The server fetches data from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the view with the list of grades.

Postconditions: The user can see the list of her/his grades.
Extensions:

a. Calculate average grade:
    1. The user clicks on the "Calculate average" button.
    2. The system calculates an average grade for all semesters and displays it to the user.

See messages list

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see her/his messages.
Trigger: The user selects "Messages" screen from the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the "Messages" page.
2. The mobile application sends a request to the server.
3. The server fetches data from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the view with the list of messages.

Postconditions: The user can see the list of her/his messages.
Extensions:

a. See message details:
    1. The user selects a message from the list.
    2. The system redirects the user to a page showing message details.

See payments list

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see her/his payments.
Trigger: The user selects "Payments" screen from the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the "Payments" page.
2. The mobile application sends a request to the server.
3. The server fetches data from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the view with the list of payments.

Postconditions: The user can see the list of her/his payments.

Check news feed

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see news from her/his faculty and university.
Trigger: The user goes to the homepage of the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the homepage.
2. The mobile application sends a request to the server.
3. The server fetches news from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the news feed section.

Postconditions: The user can see the list of news in the news feed section of the homepage.

View calendar

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see her/his calendar.
Trigger: The user selects "Calendar" screen from the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the "Calendar" page.
2. The mobile application sends a request to the server.
3. The server fetches data from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the calendar with the list of events.

Postconditions: The user can see her/his calendar. **Extensions:**

a. See events for selected date:
    1. The user selects a date from the calendar.
    2. The system shows a list of events for the selected date.

Get holidays for selected period

Primary Actor: User
Scope: Mobile application
Level: Fish level
Brief: Get holidays from an external API for a selected period.
Trigger: The user selects "Calendar" screen from the mobile application or chooses another period.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The mobile application sends a request to the external API.
2. The external API sends a response with a list of holidays for the selected period.
3. The mobile application loads the data and refreshes the calendar with the list of holidays.

Postconditions: Holidays for the selected period are loaded into the system and shown on the calendar page.

Count number of event for each date in selected period

Primary Actor: User
Scope: Mobile application
Level: Fish level
Brief: Count events for every date in the selected period.
Trigger: The user selects "Calendar" screen from the mobile application or chooses another period.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.
- The calendar events are stored in the system.

Basic flow:

1. The mobile application loads events for each date in the selected period.
2. The mobile application counts all events and refreshes the calendar with the number of events per day.

Postconditions: The number of events for each day in the selected period is shown on the calendar page.

See user details

Primary Actor: User
Scope: Mobile application
Level: Sea level
Brief: The user wants to see profile details.
Trigger: The user goes to the homepage of the mobile application.
Preconditions:

- The user has an account in one of the available university systems.
- The user is authenticated.

Basic flow:

1. The system redirects the user to the homepage.
2. The mobile application sends a request to the server.

3. The server fetches user data from the university system and sends a response back to the mobile application.
4. The mobile application loads the data and refreshes the profile info section.

Postconditions: The user can see the profile details section on the homepage.

Chapter 3

Project Design

3.1. System Architecture

The system consists of two main components, as shown in Figure 3.1. The first is the Flutter application, which is installed on the user's smartphone. On the first run, it will create a local SQLite database with the required tables. Whenever a user logs in or navigates to a page, it will send a request for data to the second component, the server. It is composed of configuration and transformation modules. The former is responsible for providing the YAML configuration file in the transformation application. Because both use Spring Cloud Config, the transformation application does not need to be restarted when the config changes. Each time it is updated, the configuration server will pick up the changes and serve the updated content. Now beans from the client that were using the configuration must be refreshed. A simple way to do this is to send a GET request to the "/ refresh" endpoint provided by the Spring Boot Actuator.

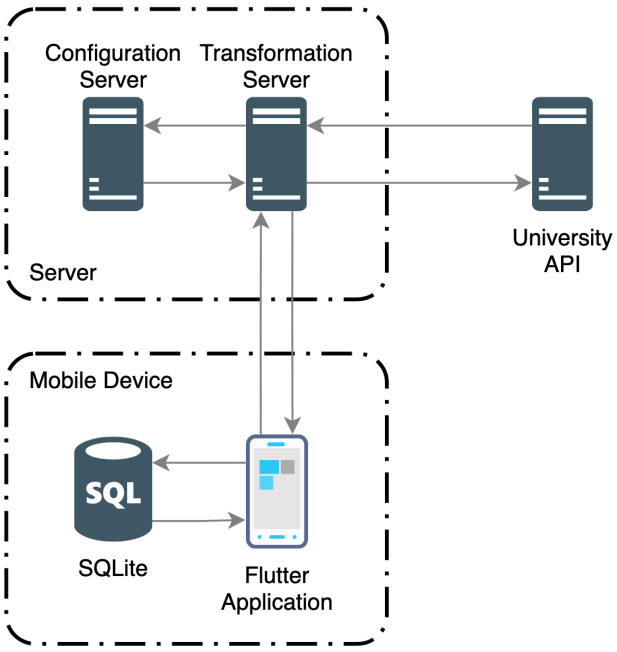

Figure 3.1: System architecture

When the transformation server receives a request, for example from Listing 3.1, it reads the university and searches for a configuration for this particular institution.

Listing 3.1: Sample calendar request

```
{
    "username": "pwr123456",
    "university": "pwr",
    "startDate": "2020-01-01T00:00:00.000",
    "endDate": "2020-02-01T00:00:00.000",
}
```

A sample YAML configuration for calendars is shown in Listing 3.2. It contains an object with the name "pwr" and a configuration for this university. Every config entry has to contain a name, endpoint address, a specification for Jolt transformation for request and response.

Listing 3.2: Sample YAML configuration for calendars

```
calendars:
    - name: 'pwr'
      address: 'http://localhost:1080/calendars'
      request: '[{"operation": "shift","spec": {"username": "user","studentNumber":
          ↪ "indeks","startDate": "date.start","endDate": "date.end"}}]'
      response: '[{"operation": "shift","spec": {"events": {"*": {"eventName": "
          ↪ events.[&1].name","room": "events.[&1].classroom","type": "events.[&1].
          ↪ eventType","date": {"start": "events.[&2].startDateTime","end": "events
          ↪ .[&2].endDateTime"},"*": "events.[&1].&"}}}}]'
```

Every transformation operation requires a specification. Below is an example of conversion from one JSON schema (List. 3.3) to another (List. 3.4) using a JSON spec file (List. 3.5).

Listing 3.3: JSON transformation input

```
{
    "events": [{
       "eventName": "Internetowe bazy danych",
       "room": "C-16, s. L2.6",
       "lecturer": "Dr inż. Jan Kowalski",
       "type": "2",
       "date": {
          "start": "2020-01-17T09:15:00Z",
          "end": "2020-01-17T11:00:00Z"
       }
    }]
}
```

Listing 3.4: JSON transformation output

```
{
    "events": [{
       "name": "Internetowe bazy danych",
       "classroom": "C-16, s. L2.6",
       "lecturer": "Dr inż. Jan Kowalski",
       "eventType": "2",
       "startDateTime": "2020-01-17T09:15:00Z",
       "endDateTime": "2020-01-17T11:00:00Z"
    }]
}
```

Operation type stated in the spec copies data from the input to the output tree without changing. Then we go inside the `events` array and get each entry with "*". The next step is to map `eventName` to `name` inside the output `events` array. Later we map `room` to `classroom` and `type` to `eventType`. Further, we go inside the `date` object and get and `end` and map it to `startDateTime` and `endDateTime` inside the output `events` array. The last action is to get the remaining, not mapped keys, and copy them into the `events` array.

Listing 3.5: Transformation specification

```
[{
    "operation": "shift",
    "spec": {
       "events": {
          "*": {
             "eventName": "events.[&1].name",
             "room": "events.[&1].classroom",
             "type": "events.[&1].eventType",
             "date": {
                "start": "events.[&2].startDateTime",
                "end": "events.[&2].endDateTime"
             },
             "*": "events.[&1].&"
          }
       }
    }
}]
```

3.2. Database Design

The mobile database is very straightforward. There are five separate tables as seen in Figure 3.2. All of them are created during the first run of the application. Data received from the server is stored locally in the SQLite database. Every time the application sends a successful request, information in a chosen table is updated with new data. If there is no connection to the Internet, the app takes already stored details from the database instead of sending a request. Thanks to it, users can have constant access to data.

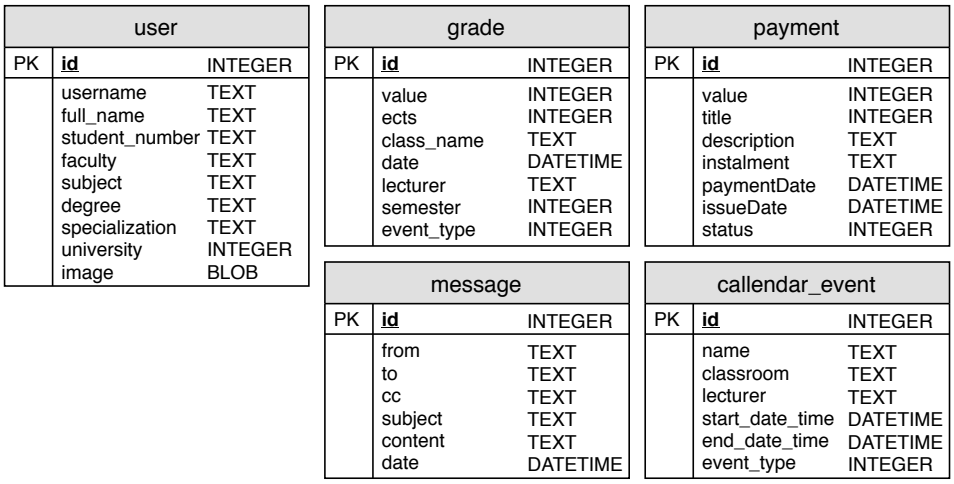

Figure 3.2: ERD diagram for the mobile database

3.3. User Interface Design

All mobile applications should be easy to use and intuitive. In our app, we used Material Design [12] created by Google. It is also a part of the Flutter toolkit. To make the development quicker and easier, we created wireframes for all the main views of the application using a tool called Sketch [24]. It is simple to use and has a lot of available libraries.

Every mobile application requires a logo. To design a temporary one (Fig. 3.3), we used a website created by FreeLogoDesign [11]. It allows users to create logos by selecting templates and then customizing them using the built-in HTML5 wizard. It is free to use for low-resolution images, which was sufficient for our purpose.

Figure 3.3: Temporary logo

There are seven main screens available with a navigation flow as shown in Figure 3.4. The first screen is a login screen. After users enter the correct login and password, they are redirected to the homepage. There they can see their profile info and news for their university and faculty. On the bottom, there is a navigation bar from which they can get access to an additional four pages. The first of them is the calendar page where they can see their schedule. The next one is the grades page where they can view all their grades. After it, there is the messages page where users can access their e-mails and go to the messages details page. The last one is the finances page. It allows users to see all of their payments.

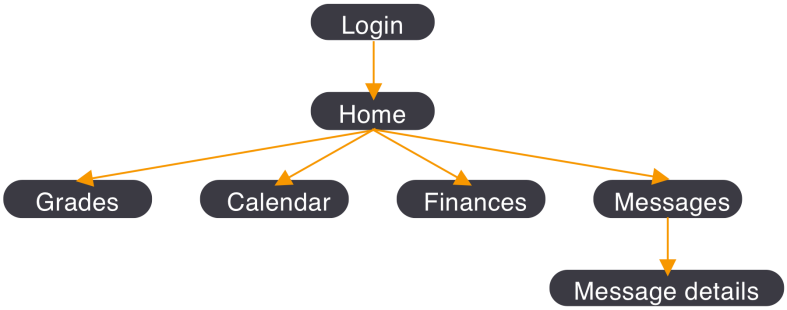

Figure 3.4: Mobile UX flow

The first screen that is presented to users is the login screen (Fig. 3.5a). There is a big logo of the application in the middle of the view. There are also two input boxes on the bottom of the screen with a button to submit them. If the data provided is incorrect, users are presented with a red error snack bar.

The home screen (Fig. 3.5b), is the main page of the application. It contains profile info with the logged-in student's data. There is also a news feed section where users can access news from their university or faculty, depending on the configuration. In the top right corner of the screen, there are two icons. The first one allows users to log out of the application. By clicking on the next one, users can access the settings.

The calendar screen (Fig. 3.6a) contains a calendar with the list of dates. If a date is a holiday, there is an indicator shown in the right top corner of the container. An indicator at the bottom of the container informs users that they have some lectures during the selected date. When users click on a date, a list of classes is shown under the calendar. Every row contains the start and the end time of the event, its name, university teacher, classroom, and a type of the event.

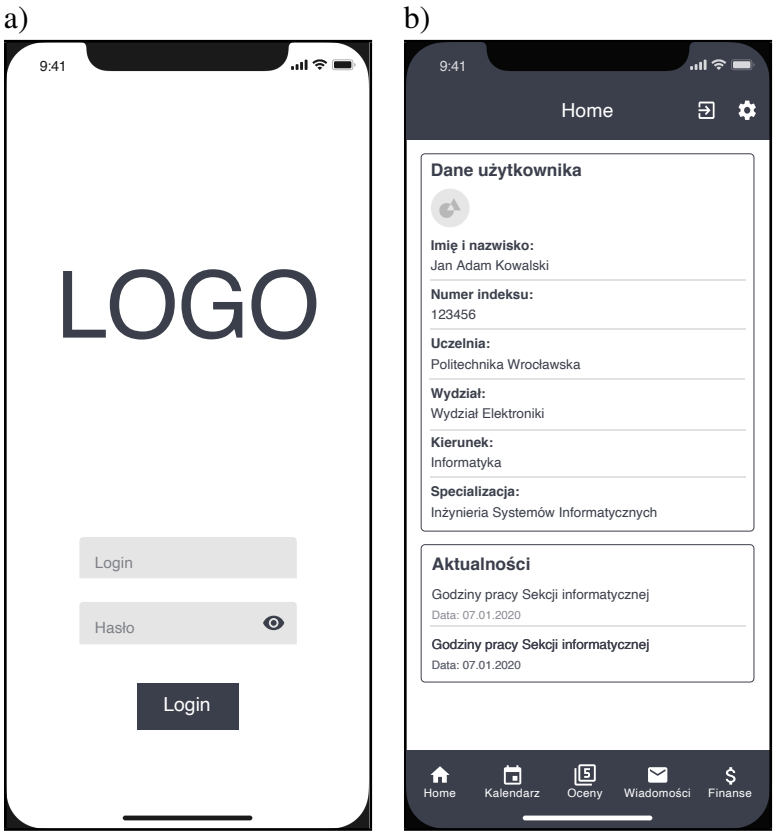

Figure 3.5: Wireframes: a) login screen, b) home screen

The next screen (Fig. 3.6b) is the finances page. It is composed of a list of payments with their details. Every entry includes an amount, status, name, and the date of the payment's issue. Some of them also contain an installment number.

The last screen (Fig. 3.6c) consists of a grades list. Each entry contains a grade, number of ECTS credits, course name, type and university teacher, and issuing date. The icon in the top right corner allows users to calculate their average grade for a semester or the whole studies.

The two last screens shown in Figure 3.7 present a list of messages and their details. The first page is composed of the email sender, topic, date received, and partial contents. When customers click on an email, they are redirected to the details page where the full contents of the email are shown along with Cc'd recipients. After users have read the email, they can get back to the emails list by clicking an arrow on the top of the screen. In the top right corner of the screen, there is an icon that is shown only when the user university's API allows for sending messages.

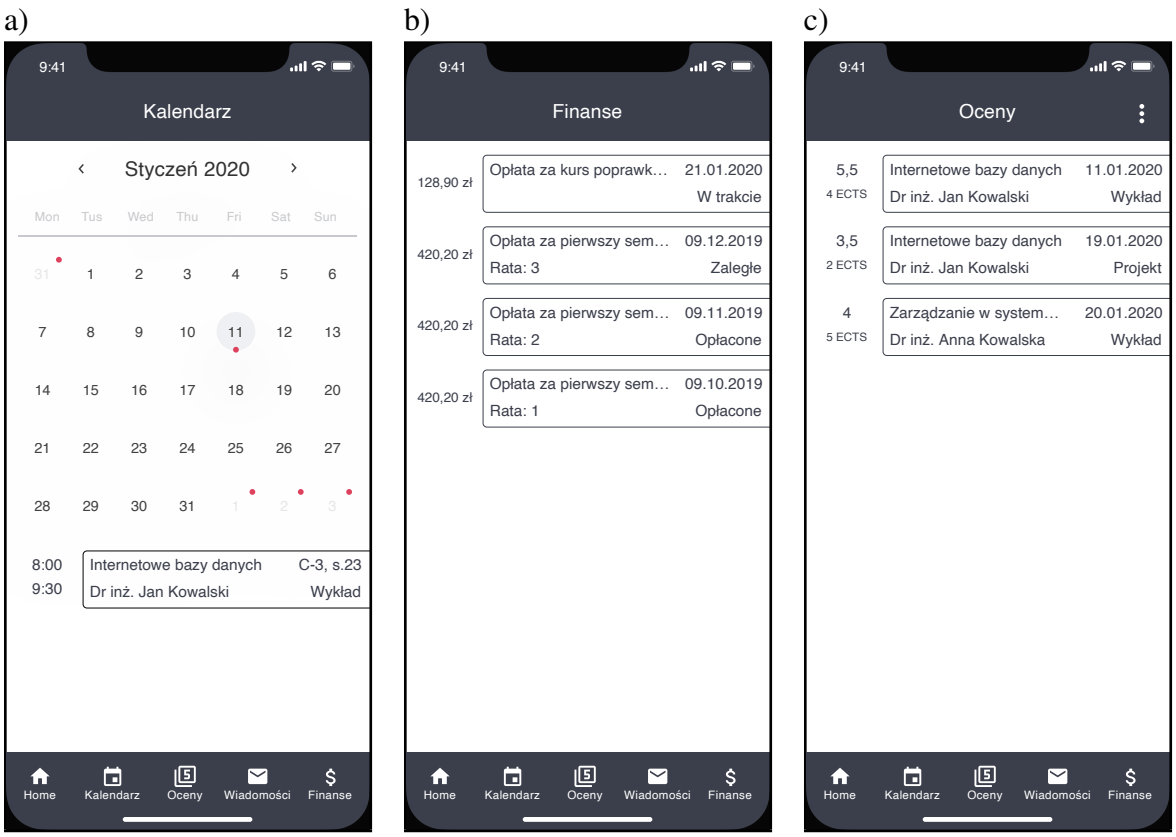

Figure 3.6: Wireframes: a) calendar screen, b) finances screen, c) grades screen

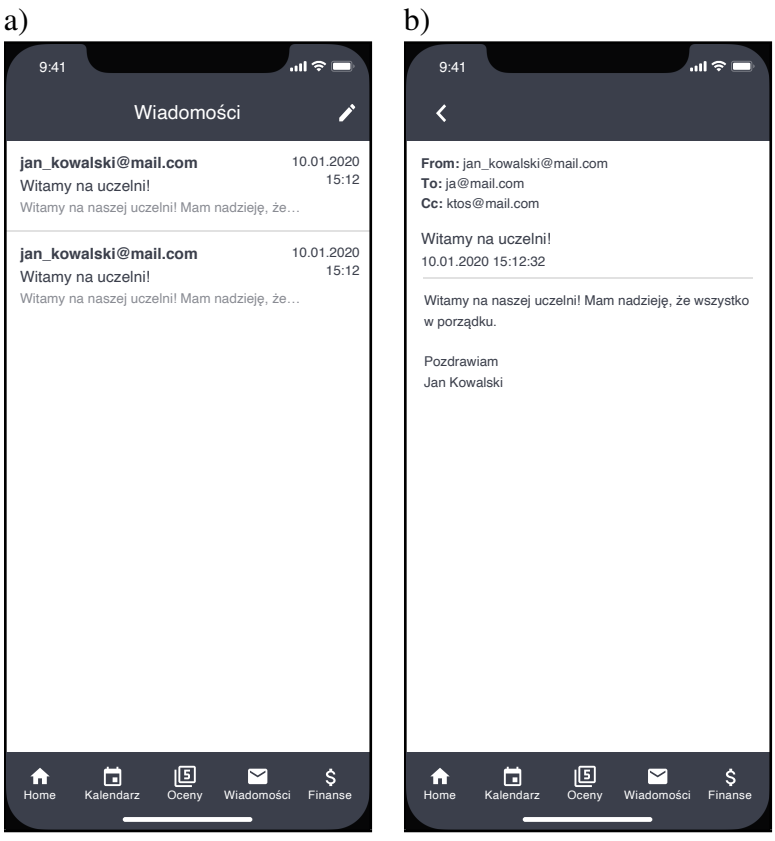

Figure 3.7: Wireframes: a) messages screen, b) message details screen

Chapter 4

Project Implementation

4.1. Design Patterns

The main architectural pattern used during the development of the mobile application is called BloC (Business Logic Component) [16]. It is a pattern created by Google, especially for Flutter framework. To manage the flow of data within an app, it uses Reactive Programming [33], which in simple terms, is programming with asynchronous data streams. Flutter user interfaces are composed of smaller parts called widgets. When a state of a widget changes, it is redrawn automatically on the screen. If there are lots of widgets used, their state can change multiple times, and they will be redrawn, sometimes unnecessary. That can lead to performance issues. This is where the BLOC comes into play. It helps to separate logic from the user interface while maintaining the Flutter reactive model of redrawing widgets.

A BLoC has two main components: Sinks and Streams, both of which are provided by a StreamController. A Stream is a source of asynchronous data events, which provides a way to receive a sequence of events. These events can be user inputs, hover events, variables, click events, network requests, and others. A Sink is a place where data or events are added and can be read by a Stream. Everything in the application should be represented as a stream of events. Some widgets submit events to the BLoC, which acts as a middleman, and other widgets will receive and react to processed data, as shown in Figure 4.1.

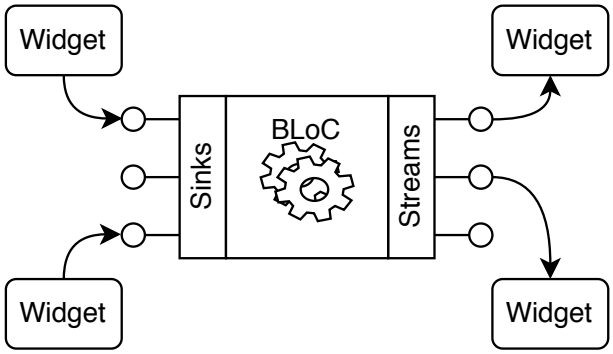

Figure 4.1: Visual representation of the BLoC pattern (based on [7])

Using the BLoC pattern, we can manage the state of an application clearly, and it helps in writing concise and clean code. The user interface is separated from the business logic, which makes it easier to change the user interface without modifying the business logic.

4.2. Project Management

GitKraken Glo

All tasks created while developing the project were managed with GitKraken Glo Board [5]. It is a Kanban board for task and issue tracking, which is fully integrated with GitKraken. It allows adding labels, descriptions, comments, assignees, and many others to created cards. As seen in Figure 4.2, four statuses of tasks turned out to be enough for this system (backlog, on hold, doing, and done). Usage of the mentioned Kanban board simplified the development because it was easy to see what was already done and what still needs to be implemented.

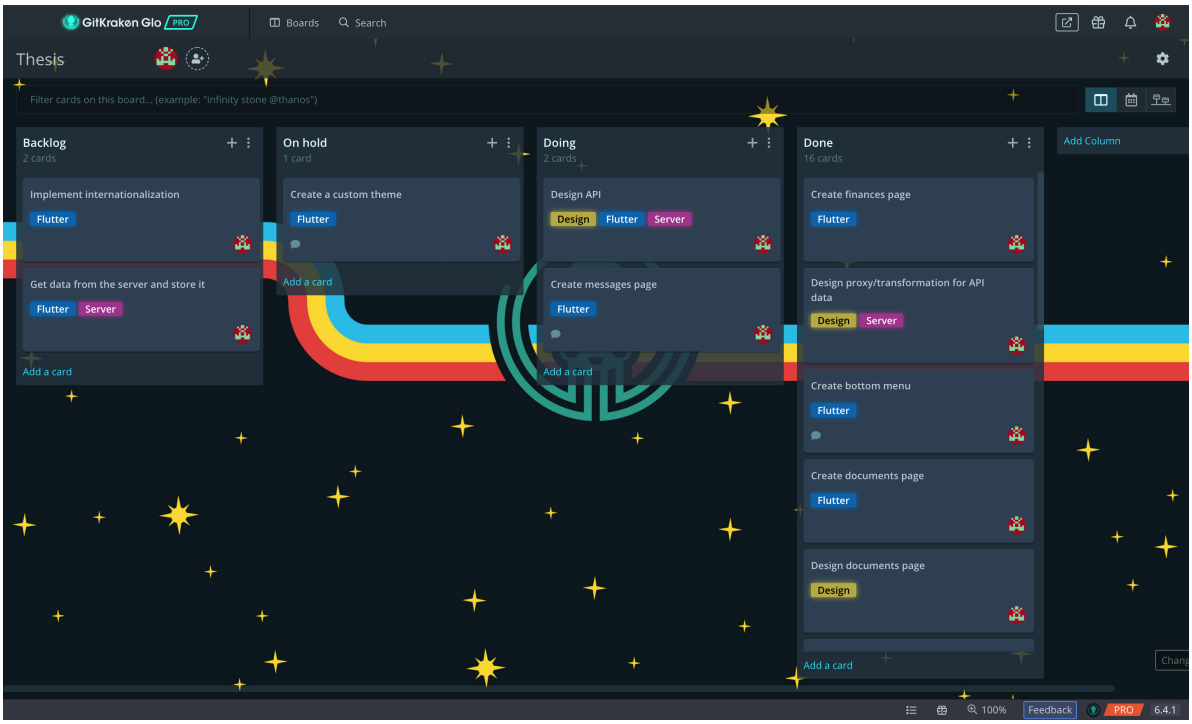

Figure 4.2: An example of GitKraken Glo Board

GitFlow

GitFlow was used to ease the development of tasks. It is a branching model for Git, created by Vincent Driessen [10]. It makes parallel development very easy because it isolates new changes from finished work. The new development is done in feature branches. They are only merged back into the main code when the developer thinks that the code is ready for release. If a person is asked to switch from one task to another, all they need to do is commit changes and create a new feature branch for the new task. When the task is done, they check out the original feature branch and can continue where they left off. For the purpose of the project, the GitFlow was slightly altered. A new type of branch was introduced, a bugfix branch. All fixes were committed only to this type of branch. Figure 4.3 represents a part of the commit history for the mobile application, which uses GitFlow.

4.3. Tools and Technologies

Flutter

Flutter is an open-source toolkit created by Google [14]. It can be used to develop natively compiled applications for mobiles (iOS, Android), desktops (Mac, Linux, Windows, Google

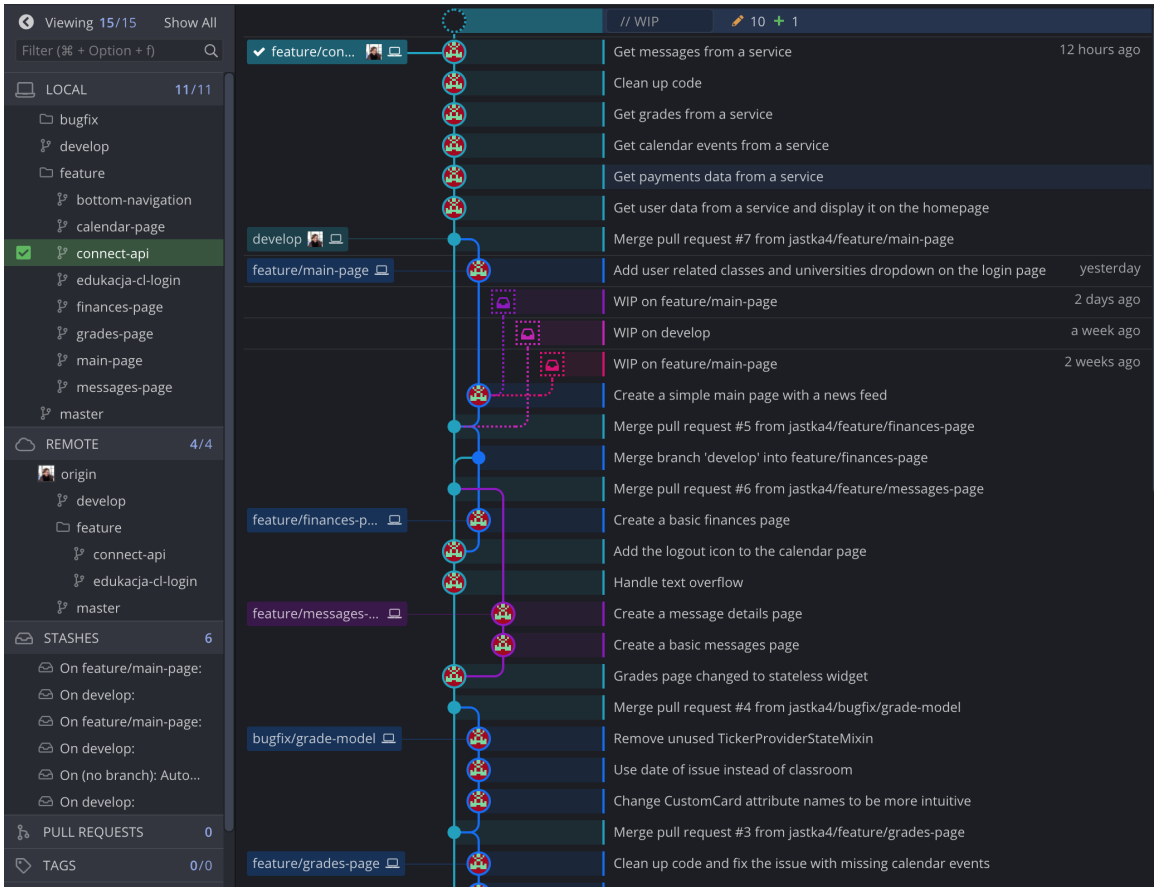

Figure 4.3: Part of the commit history for the mobile application

Fuchsia), and the web from a single codebase. Flutter's engine utilizes Google's Skia graphics library for low-level rendering. It is written primarily in C++. It also leverages platform-specific SDKs, for example, iOS and Android.

Flutter applications are created using the Dart language. The toolkit runs in the Dart virtual machine that features a just-in-time execution engine. It can be used while debugging an app to do a "hot reload", which modifies source files and then injects them into a running application. Flutter added a stateful hot reload on top of it, wherein most cases, changes to source code can be reflected immediately in the working app without requiring a restart or any loss of state.

Release versions of Flutter apps are compiled with ahead-of-time (AOT) compilation on both iOS and Android, making Flutter's high performance on mobile devices possible [31].

Creating UIs in Flutter involves using composition to assemble widgets from other widgets. According to the docs, "A widget is an immutable description of part of a user interface" [14]. They define what their view should look like, given their current state and configuration. The widget rebuilds its description when it's state changes because the framework diffs against the previous representation to determine the minimal changes needed in the underlying render tree to transition from one state to the next.

Flutter aims to provide 120 frames per second (FPS) performance on devices capable of 120Hz updates and 60 FPS otherwise.

Complex widgets can be built from other smaller ones. An app is actually the largest widget of all of them, often called "MyApp". Text component is a widget, but so is its TextStyle, than defines things like color, size, font-weight, and family. Some widgets represent things, others, like TextStyle, represent characteristics. There are also ones that do something, like StreamBuilder and FutureBuilder.

An alternative option to using widgets is to use the Foundation library's methods directly. They can be used to draw text, shapes, and imagery directly on the canvas. One of the frameworks that utilized this is the open-source Flame game engine.

Dart

Dart is a client-optimized programming language developed by Google [13]. It is multiplatform and can be used to build desktop, mobile, backend, and web applications.

It is object-oriented, class defined, garbage-collected language. It uses a C-style syntax and can be transcompiled into JavaScript. It supports abstract classes, interfaces, mixins, static typing, reified generics, and a sound type system.

Dart code can be compiled ahead-of-time into machine code. Applications build with Flutter, a mobile app SDK built with Dart, are deployed to app stores as AOT compiled Dart code.

Dart version 2.6 was accompanied by a new extension dart2native. It allows composing a Dart program into self-contained executables on the macOS, Linux, and Windows desktop platforms. Earlier, this feature only exposed capability on iOS and Android mobile devices via Flutter [28].

SQLite

SQLite is a relational database management system contained in a C-language library [26]. It is embedded into the end program, unlike many other database management systems. It is built into most computers, all mobile phones and is bundled inside lots of applications that people use every day. It is a popular selection as an embedded database software for client/local storage in application software such as web browsers. It is one of the most used database engines, as it is used today by several widespread browsers, embedded systems, such as mobile phones, operating systems, among others. SQLite has bindings to many programming languages.

SQLite is ACID-compliant and follows PostgreSQL syntax while implementing most of the SQL standard. It uses weakly and dynamically typed SQL syntax that does not guarantee the domain integrity. A string can be inserted into a column defined as an integer. SQLite will try to convert data between formats when appropriate, for example, the string "1234" into an integer. However, it does not guarantee such conversion and will store the data as-is if such conversion is not possible [29].

Android Studio

Android Studio is an integrated development environment created by Google and built on JetBrains' IntelliJ IDEA software [30]. It was specially designed for Android development. It can be downloaded on macOS, Windows, and Linux based operating systems. Previous versions of Android Studio were based on Eclipse IDE.

Flutter apps can be built using any text editor combined with Flutter command-line tools. It is best to use the Flutter plugin with Android Studio. The plugin provides users with, for example, syntax highlighting, code completion, widget editing assist, run and debug support.

Java

Java is a general-purpose, object-oriented, class-based programming language [17]. It lets developers write once, run anywhere, so it can run on all platforms that support it without the need for recompilation. It has been designed to have as few implementation dependencies as possible. Java applications are usually compiled into bytecode that can be run on any JVM, regardless of the architecture of the computer. The syntax of Java is similar to C++ and C, but it has fewer low-level facilities than either of them. According to GitHub, Java is currently one of the most popular programming languages used, particularly for client-server web applications [32].

Spring Framework

Spring Framework is an open-source Java platform and a container for inversion of control [22]. Its core features can be used in developing any Java applications. There are also extensions for creating web applications on top of the Java Enterprise Edition (Java EE) platform. The framework targets to make Java EE development more comfortable to use and promotes good programming practices by enabling a POJO-based programming model. Even though the framework does not impose any specific programming model, it has become popular in the Java community as an add-on or even a replacement for the Enterprise JavaBeans model.

Spring Boot is an open-source micro-framework maintained by Pivotal company [20]. It is pre-configured by the Spring team with the best configuration possible and use of the Spring platform and third-party libraries so that developers can quickly get started without wasting time on preparation.

Spring Cloud Config provides client and server-side support for externalized configuration in distributed systems [21]. The Config Server is a central spot to manage external properties for apps across all environments. The configuration can be used with any application running in any language.

The Spring Web MVC platform provides MVC (Model-View-Controller) architecture and components that can be used to create loosely connected and flexible web applications [19]. The MVC pattern separates various aspects of the application (business logic, user interface logic, and input logic), providing a loose relationship between these.

- Model - a POJO that encapsulates the application data;
- View - responsible for rendering the model data;
- Controller - responsible for processing user requests, building the appropriate model, and passing it to the view for rendering.

Jolt

Jolt (JsOn Language for Transform) is a transformation library written in Java [2]. It allows developers to convert one JSON structure to another using a schema created in JSON. The tool provides a set of transformation types:

- shift - copies data from input to the output tree;
- default - applies default values to the tree;
- remove - removes data from the tree;
- sort - sorts map keys alphabetically;
- cardinality - adjusts the cardinality of input data.

Each of the types has its DSL, which is called a specification, that defines the new structure for outgoing JSON data.

A basic approach for converting JSON to JSON in Java is to use XSLT or STX. The conversion sequence would look like this:

JSON -> XML -> XSLT/STX -> XML -> JSON

With Jolt, the conversion sequence is simplified and looks like this:

JSON -> Specification JSON -> JSON

The out-of-the-box Jolt should be able to do most of the structural transformation. Any complex transformation logic which cannot be expressed in standard terms can be plugged in via Java extension class with Jolt.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange text format that is easy for a machine to parse and generate, but also for humans to read and write [1]. It is entirely language-independent and uses array data types and attribute-value pairs to transmit data objects. It is a widespread data format and serves as a replacement for XML in AJAX systems.

YAML

AML Ain't Markup Language (YAML) is a data serialization language that has a human-readable format [18]. It is commonly used for configuration files and in applications where data is being transmitted or stored. It targets many of the same communication applications as XML but has minimal syntax. It uses both Python-style indentations to indicate nesting, and a more compact format that uses "[]" for lists and "{}" for maps making YAML 1.2 a superset of JSON.

MockServer

MockServer is an open-source platform used to mock systems via HTTP or HTTPS [6]. It was designed to simplify integration testing by mocking systems such as web services or websites. It also helps in developing code against a service that is not complete or is unstable.

Docker

Docker is a platform for developing, shipping, and running applications [8]. It enables to separate applications from the infrastructure so that they can be delivered quickly. It also provides an ability to package and run applications in a loosely isolated environment called a container. Many different containers can be run simultaneously on a given host.

Docker team also made Docker Compose. It is a tool for defining and launching Docker applications using multiple containers [9]. It utilizes YAML files to configure the application's services. Then they can be created and started with just a single command.

4.4. Database Configuration

The database is elementary and does not contain any relations. That is why the model-first approach was used during the implementation. All tables are created on a smartphone during the first run of the mobile application. The `DBProvider` class presented in Listing 4.1 is a singleton and creates a new database only if none exists on the device. If the `universityTable.db` exists, it is opened and used throughout the application.

Listing 4.1: The database provider class

```
1  class DBProvider {
2    DBProvider._();
3
4    static final DBProvider db = DBProvider._();
5    static Database _database;
6
7    Future<Database> get database async {
8      if (_database != null) return _database;
9
10     _database = await initDB();
11     return _database;
12   }
13
14   initDB() async {
```

```
15      Directory documentsDirectory = await getApplicationDocumentsDirectory();
16      String path = join(documentsDirectory.path, 'universityHelper.db');
17      return await openDatabase(path, version: 1, onOpen: (db) {},
18          onCreate: (Database db, int version) async {
19        await db.execute(CREATE_USER_TABLE);
20        await db.execute(CREATE_CALENDAR_EVENT_TABLE);
21        await db.execute(CREATE_GRADE_TABLE);
22        await db.execute(CREATE_MESSAGE_TABLE);
23        await db.execute(CREATE_PAYMENT_TABLE);
24      });
25    }
26  }
```

The `initDB` method creates new tables with the previously created scripts. An example of a script creating a user table is shown in Listing 4.2.

Listing 4.2: SQL script to create a user table

```
1  static const CREATE_USER_TABLE = 'CREATE TABLE user ('
2        'id INTEGER PRIMARY KEY,'
3        'username TEXT,'
4        'full_name TEXT,'
5        'student_number TEXT,'
6        'faculty TEXT,'
7        'subject TEXT,'
8        'degree TEXT,'
9        'specialization TEXT,'
10       'university INTEGER,'
11       'image BLOB'
12       ');';
```

4.5. Mobile Application Implementation

The mobile application is the largest part of the project. It uses Flutter framework and the BLoC pattern, which imposes a particular folder structure (Fig. 4.4):

- `.dart_tools` – used by pub and other tools.
- `.idea` – contains IntelliJ's project specific settings files.
- `android` – stores Android native project that is used when building the app for Android devices.
- `build` – holds the compiled code of the Flutter application.
- `images` – contains all image assets used by the app. It could also be placed inside the `lib` folder.
- `ios` – only available when working on macOS. Stores XCode iOS native project that is used when building the app for iOS devices.
- `lib` – holds Dart files containing Flutter application code.
- `blocs` – composed of different BLoC components used throughout the app. Each of them has a dedicated folder for storing the main BLoC logic, event, and state classes.
- `common` – contains common files (providers, helpers, enums).
- `dao` – a place with all DAOs.
- `models` – keeps all models.
- `repositories` – a folder containing repositories that are used by components to fetch data from the API or database, depending on the availability. Inside each of the files is a logic determining if users have an active connection to the Internet. They also store downloaded data in a database.
- `services` – holds services responsible for fetching data from the API.

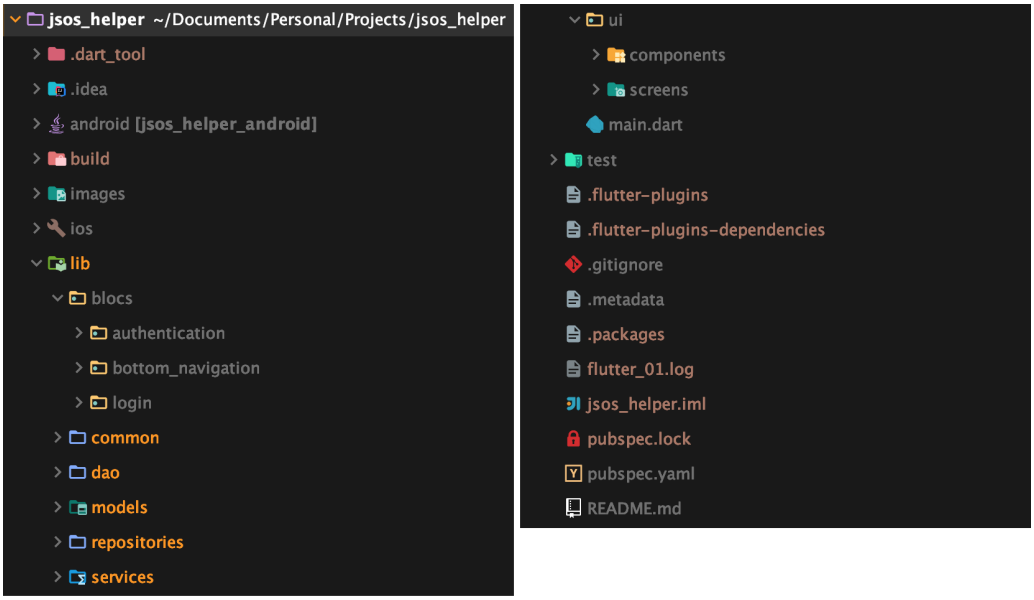

Figure 4.4: Folder structure of the Flutter application

- `ui` – stores all UI elements that are divided into reusable `components` and `screens` holding views.
- `tests` – contains all tests written for the application.

In addition to these structures, there is a `.gitignore` file used for the program versioning. `pubspec.yaml` is a place where developers can provide all the required dependencies and project configuration.

BLoC

The first type of elements are the Business Logic Components (see Section 4.1). They manage the flow of data within the application using asynchronous data streams. In the context of the program, a BLoC takes a stream of events as an input and transforms them into a stream of states as output (Fig. 4.5).

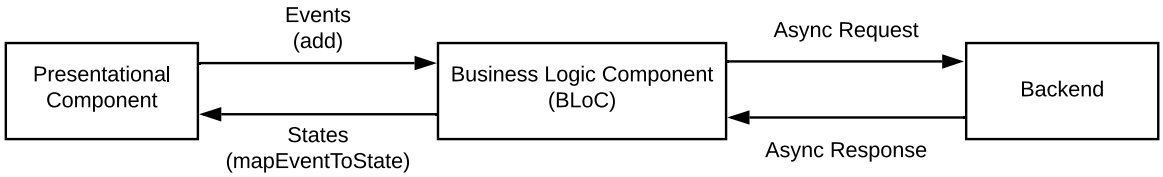

Figure 4.5: High-level BLoC diagram (based on [3])

Events are the input of a BLoC. They are often UI events, such as pressing a button. Events are added and then converted to states. The Listing 4.3 shows an example of the event implementation. It is a simple object that stores `username`, `password`, and `university`. It has a simple constructor and a `toString` method. `Equatable` overrides == and `hashCode` so objects can be easily compared.

Listing 4.3: Login event

```
1  abstract class LoginEvent extends Equatable {
2    LoginEvent([List props = const []]) : super(props);
3  }
4
```

```
5  class LoginButtonPressed extends LoginEvent {
6    final String username;
7    final String password;
8    final University university;
9
10   LoginButtonPressed({
11     @required this.username,
12     @required this.password,
13     @required this.university,
14   }) : super([username, password, university]);
15
16   @override
17   String toString() =>
18       'LoginButtonPressed { username: $username, password: $password, university:
             ↪ $university }';
19 }
```

States are the output of a BLoC. All presentation components can listen to the state stream and redraw their parts based on the given state. Listing 4.4 represents a simple class used for login, with multiple states available. Each of them can have different attributes and implementation; for example, `LoginFailure` differs from the rest.

Listing 4.4: Login state

```
1  abstract class LoginState extends Equatable {
2    LoginState([List props = const []]) : super(props);
3  }
4
5  class LoginInitial extends LoginState {
6    @override
7    String toString() => 'LoginInitial';
8  }
9
10 class LoginLoading extends LoginState {
11   @override
12   String toString() => 'LoginLoading';
13 }
14
15 class LoginFailure extends LoginState {
16   final String error;
17
18   LoginFailure({@required this.error}) : super([error]);
19
20   @override
21   String toString() => 'LoginFailure { error: $error }';
22 }
```

The last but not least is the actual logic of the BLoC (List. 4.5). It uses previously defined state and event classes, and it has two attributes (`userRepository` and `authenticationBloc`). The initial state of the BLoC is set to `LoginInitial`. Besides the constructor and a simple getter, there is a `mapEventToState` method. It converts the incoming events into states which are consumed by the presentation layer. If the event is `LoginButtonPressed`, the state is changed into `LoginLoading`. Next, it tries to authenticate the user and adds a `LoggedIn` event for the `authenticationBloc`. The last part is to change the state into the initial state. If any error occurs, the state is changed to `LoginFailure`, and the details of the error are passed there.

Listing 4.5: Login BLoC

```
1  class LoginBloc extends Bloc<LoginEvent, LoginState> {
2    final UserRepository userRepository;
3    final AuthenticationBloc authenticationBloc;
```

```
4
5   LoginBloc({
6     @required this.userRepository,
7     @required this.authenticationBloc,
8   }) : assert(userRepository != null),
9        assert(authenticationBloc != null);
10
11  LoginState get initialState => LoginInitial();
12
13  @override
14  Stream<LoginState> mapEventToState(LoginEvent event) async* {
15    if (event is LoginButtonPressed) {
16      yield LoginLoading();
17
18      try {
19        final token = await userRepository.authenticate(
20          username: event.username,
21          password: event.password,
22          university: event.university,
23        );
24
25        authenticationBloc.add(LoggedIn(
26            token: token,
27            username: event.username,
28            university: event.university));
29        yield LoginInitial();
30      } catch (error) {
31        yield LoginFailure(error: error.toString());
32      }
33    }
34  }
35 }
```

DAO

Another type of components used in the system are DAOs. It is a pattern that provides an abstract interface for communication between the application and the database. DAO methods are very similar, and an example is shown in Listing 4.6. The first step is to establish a connection to SQLite. After it has been successful, we query the database, map the results to `Grade` object, and return the list.

Listing 4.6: Getter method defined in the grades DAO

```
1 Future<List<Grade>> getGradesBySemester(int semester) async {
2   final db = await DBProvider.db.database;
3   var res =
4       await db.query('grade', where: 'semester = ?', whereArgs: [semester]);
5   List<Grade> list =
6       res.isNotEmpty ? res.map((c) => Grade.fromMap(c)).toList() : [];
7   return list;
8 }
```

Model

The model is a simple data structure with lots of properties. It has a constructor, `toMap`, and `toString` methods. Besides them, there are factory constructors capable of creating objects from JSON (List. 4.7) or map structure.

Listing 4.7: Payment factory constructor

```
factory Payment.fromMap(Map<String, dynamic> json) => new Payment(
      id: json["id"],
      value: json["value"],
      title: json["title"],
      description: json["description"],
      instalment: json["instalment"],
      paymentDate: DateTime.parse(json["paymentDate"]),
      issueDate: DateTime.parse(json["issueDate"]),
      status: PaymentStatus.values[json["status"]],
    );
```

Repository

The next type is the repository, which is responsible for fetching data from the database or API, depending on the availability. The `getCalendarEvents` method (List. 4.8) obtains customer's data such as username and university, which are required by the API. Next, we verify if the mobile device has an active Internet connection. If not, we get calendar events from the local database. If the user has access to the web, we try to fetch events from the service and update the local database.

Listing 4.8: Getter method defined in the calendar repository

```
Future<List<CalendarEvent>> getCalendarEvents(
    DateTime first, DateTime last) async {
  String _username = await storageRepository.getUsername();
  University _university = await storageRepository.getUniversity();
  if (await ConnectionStatusSingleton.getInstance().checkConnection()) {
    List<CalendarEvent> events = await _calendarService.fetchCalendarEvents(
        _username, _university, first, last);
    _calendarDao.updateCalendarEvents(first, last, events);
    return events;
  } else {
    return _calendarDao.getCalendarEvents(first, last);
  }
}
```

Service

The service is a component responsible for fetching data from an external API. `fetchCalendarEvents` method (List. 4.9) retrieves calendar events for selected dates and a student. It sends a POST request with JSON body and waits for a response. After it successfully obtained the data, it maps the response to the `CalendarEvent` and returns it. If the `statusCode` is different from 200, it throws an exception.

Listing 4.9: Calendar service

```
class CalendarService {
  static const URL = GlobalConstants.BASE_API_URL + '/calendar';

  Future<List<CalendarEvent>> fetchCalendarEvents(String username,
      University university, DateTime start, DateTime end) async {
    final response = await http.post(URL,
        headers: {
          'Content-Type': 'application/json',
        },
        body: jsonEncode(
          {
            'username': username,
```

```
13            'university': university.abbreviation,
14            'startDate': start.toIso8601String(),
15            'endDate': end.toIso8601String(),
16          },
17        ));
18    if (response.statusCode == 200) {
19      return _parseCalendarEvents(response.body);
20    } else {
21      throw Exception('Unable to fetch calendar events from the REST API');
22    }
23  }
24
25  List<CalendarEvent> _parseCalendarEvents(String responseBody) {
26    final parsed = json.decode(responseBody)["events"];
27    return parsed.isNotEmpty
28        ? parsed
29            .map<CalendarEvent>((json) => CalendarEvent.fromJson(json))
30            .toList()
31        : [];
32  }
33 }
```

UI

The last type of components in the application are UI widgets. They allow building views by simply stacking widgets on top of each other. Listing 4.10 shows a `build` method of the home screen. It draws a `Scaffold` with an app bar build by a private method `_buildAppBar`. Under it, there is a `Container` with a scrollable view, that has two rows. The first one is the profile info, and the second is the news feed. It also uses `_authenticationBloc` to manage user data.

Listing 4.10: Build method of the home screen

```
1  @override
2  Widget build(BuildContext context) {
3    final AuthenticationBloc _authenticationBloc =
4        BlocProvider.of<AuthenticationBloc>(context);
5
6    return Scaffold(
7      appBar: _buildAppBar(_authenticationBloc),
8      body: Container(
9        margin: EdgeInsets.only(top: 16.0),
10       padding: const EdgeInsets.symmetric(horizontal: 8.0, vertical: 4.0),
11       child: SingleChildScrollView(
12         child: Column(
13           children: <Widget>[
14             Row(children: <Widget>[
15               Expanded(child: _buildProfileInfo(_authenticationBloc))
16             ]),
17             Row(children: <Widget>[Expanded(child: _buildNewsFeed(context))]),
18           ],
19         ),
20       ),
21     ),
22   );
23 }
```

4.6. Server Implementation

The server is composed of two server modules (Fig. 4.6): configuration (`*-server-config`) and transformation (`*-server-client`). Both were created with Spring Framework (Spring MVC, Spring Boot, and Spring Cloud Config). The main project has its own `pom.xml` with a simple parent config. There are two separate folders for each of the modules. There is a global `.gitignore` file for versioning of the program. `.idea` and `.mvn` contain project-specific settings for IntelliJ IDEA and Maven.

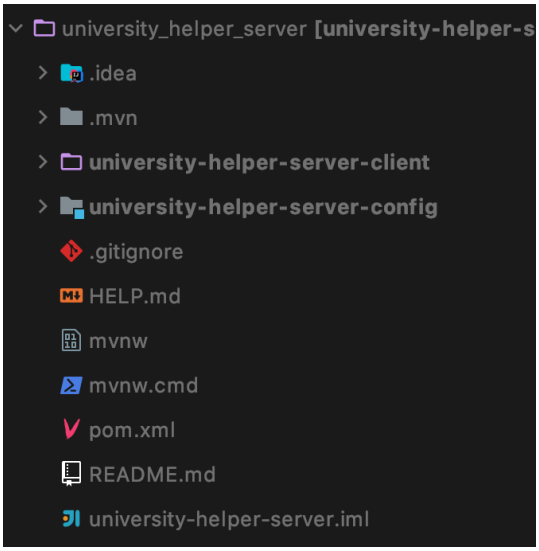

Figure 4.6: Folder structure of the transformation server

Figure 4.7 represents a folder structure for the first of the modules. It includes:

- `java` – contains a package with the main class that acts as an entry point for the entire program.
- `resources` – keeps a properties file, which is a configuration for the Spring Cloud Config.
- `test` – contains all tests written for the module.

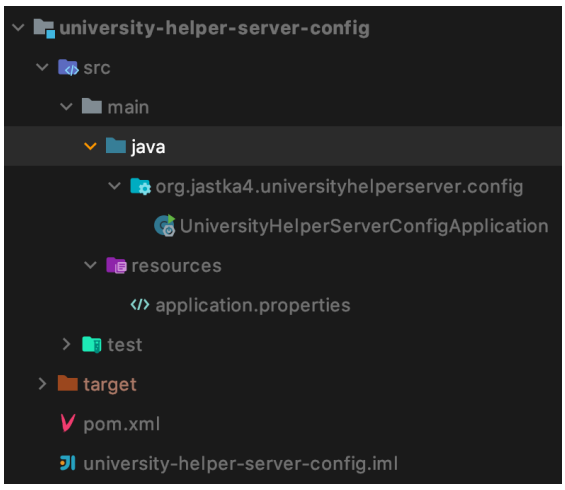

Figure 4.7: Folder structure of the configuration module

The `UniversityHelperServerConfigApplication` file (see Listing 4.11) is the main class of the configuration module. `@EnableConfigServer` annotation was used to enable Spring Cloud Config for the program. It allows to share the specified configuration between many microservices and applications.

Listing 4.11: Server configuration main class

```
1 @SpringBootApplication
2 @EnableConfigServer
3 public class UniversityHelperServerConfigApplication {
4
5         public static void main(String[] args) {
6                 SpringApplication.run(UniversityHelperServerConfigApplication.class,
                      ↪ args);
7         }
8 }
```

The properties for the Spring Cloud Config (see Listing 4.12) have the following meanings:

- `server.port` – sets the server on which the module will run.
- `spring.cloud.config.server.git.uri` – specifies the URI of the remote repository containing the property files that will be served by the configuration service.
- `spring.cloud.config.server.git.clone-on-start` – determines whether the configuration from the repository should be cloned when starting the module.

Listing 4.12: Spring Cloud Config properties

```
1 server.port=8888
2 spring.cloud.config.server.git.uri=https://github.com/jastka4/university-helper-
      ↪ config
3 spring.cloud.config.server.git.clone-on-start=true
```

Figure 4.8 represents a folder structure for the second of the modules. It includes:

- `common` – a folder containing enums available in the system, a transformation POJO used to store configuration from the YAML file, and university constants required for transformation. There is also a new exception type created specifically for this system.
- `config` – contains transformation configuration class. It is a simple POJO that stores a list of every type, for example, payments, finances, et cetera.
- `controllers` – a place where all controllers are placed. In this case there is only the main one, which handles every incoming request.
- `services` – holds services for handling requests and transformations. It contains only interfaces, and the `impl` folder, which stores implementations of the said interfaces.
- `resources` – keeps a properties file, which is a configuration read by Spring Cloud Config.
- `test` – contains all tests written for the module.

In addition to these structures, there is a `UniversityHelperServerApplication` file that acts as an entry point for the entire program. The last but not least is `pom.xml`, which contains information, configuration, build directory, source directory, test source directory, plugins, dependencies, et cetera used by Maven to build the module.

Method shown in Listing 4.13 is responsible for handling all calendar requests. It reads the incoming JSON and sends it to `sendRequestToUniversityApi` method with a parameter that indicates that the request was of calendar type. If exception happens during the execution 400 status is returned with a JSON response containing the error message.

Listing 4.13: Method that handles all calendar requests

```
1 @PostMapping(value = "/calendar")
2 public ResponseEntity<String> getCalendarEvents(@RequestBody final String request)
      ↪ {
3     try {
```

```
4         return ResponseEntity.ok().body(sendRequestToUniversityApi(new JSONObject(
              ↪ request), RequestType.CALENDARS));
5     } catch (UnsupportedUniversityException e) {
6         return ResponseEntity.badRequest().body(e.getMessage());
7     }
8 }
```

The `sendRequestToUniversityApi` method shown in Listing 4.14 is responsible for sending a request to the chosen university's API and returning the response. The first step is to transform the input JSON to the format expected by the API. To do it, we need to get the university from the request and store it for future use. The next action is to send the transformed request and get the response. The last step is to transform the answer to the format readable by the mobile application. If the initial request contains a university not supported by the configuration or the config is missing, the `UnsupportedUniversityException` is thrown.

Listing 4.14: Method that handles requests for university APIs

```
1  private String sendRequestToUniversityApi(final JSONObject request, final
       ↪ RequestType requestType) throws UnsupportedUniversityException {
2      final String university = transformationService.getUniversity(request);
3      final Optional<Transformation> transformation = transformationService.
           ↪ getTransformationForUniversity(university, requestType);
4
5      if (transformation.isPresent()) {
6          final String transformedRequest = transformationService.transformJson(
               ↪ request, transformation.get(), LoadType.REQUEST, university);
7          final JSONObject response = requestService.sendPostRequest(transformation.
               ↪ get().getAddress(), transformedRequest);
8          return transformationService.transformJson(response, transformation.get(),
               ↪ LoadType.RESPONSE, university);
9      }
10
11     throw new UnsupportedUniversityException("Missing configuration for " +
           ↪ university);
12 }
```

The primary method used for transformation is shown in Listing 4.15, and it utilizes the functions of the Jolt library. First, the request must be converted from a string to map of strings required by the library. Then we check if JSON is a request or response from the API and we get the matching transformation. Two last steps are to create the `Chainr` object from the

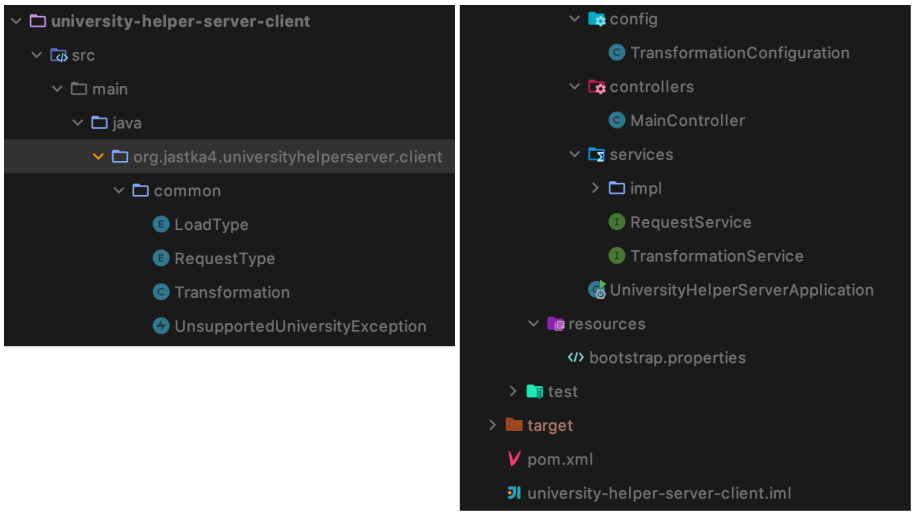

Figure 4.8: Folder structure of the transformation module

transformation obtained in the previous step and to finally transform the JSON input and return it as a string.

Listing 4.15: Method transforming JSON

```
1  public String transformJson(final JSONObject jsonRequest, final Transformation
       ↪ transformation, final LoadType loadType, final String university) {
2      final Map<String, String> inputJSON = (Map<String, String>) JsonUtils.
           ↪ jsonToObject(jsonRequest.toString());
3
4      final List<Object> chainrSpecJSON;
5      if (LoadType.REQUEST.equals(loadType)) {
6          chainrSpecJSON = JsonUtils.jsonToList(transformation.getRequest());
7      } else {
8          chainrSpecJSON =
9                    JsonUtils.jsonToList(transformation.getResponse());
10     }
11     final Chainr chainr = Chainr.fromSpec(chainrSpecJSON);
12
13     final Object transformedOutput = chainr.transform(inputJSON);
14     return JsonUtils.toJsonString(transformedOutput);
15 }
```

Chapter 5

Project Tests

5.1. Mobile Application Tests

5.1.1. Unit Tests

A unit test tests a single class, method, or function [15]. The purpose of the unit test is to check the correctness of a logical unit under various conditions. External dependencies of the tested unit are usually mocked out. Unit tests generally do not render to screen, read from, or write to disk, or receive user actions from outside of the process running the test.

An example of such a test is shown in Listing 5.1. First, we have to mock the `UserRepository` and `AuthenticationBloc` to eliminate any distortions. In the `setUp` method, we instantiate a new instance of `AuthenticationBloc` to ensure that each test runs under the same conditions and does not affect subsequent tests. The `tearDown` method runs after each test and destroys all unused objects. Next is a group of tests that imitates a user pressing the login button. First, we set the expected response, which consists of three states. In the end, we expect that the `LoginBloc` will yield `LoginInitial`, followed by a `LoginLoading` and lastly a `LoginInitial` in response to the `LoginButtonPressed` event.

Listing 5.1: Flutter login BLoC test

```
1  class MockUserRepository extends Mock implements UserRepository {}
2
3  class MockAuthenticationBloc extends Mock implements AuthenticationBloc {}
4
5  void main() {
6    LoginBloc loginBloc;
7    MockUserRepository userRepository;
8    MockAuthenticationBloc authenticationBloc;
9
10   setUp(() {
11     userRepository = MockUserRepository();
12     authenticationBloc = MockAuthenticationBloc();
13     loginBloc = LoginBloc(
14       userRepository: userRepository,
15       authenticationBloc: authenticationBloc,
16     );
17   });
18
19   tearDown(() {
20     loginBloc?.close();
21     authenticationBloc?.close();
22   });
23
```

```
24    group('LoginButtonPressed', () {
25      test('emits token on success', () {
26        final expectedResponse = [
27          LoginInitial(),
28          LoginLoading(),
29          LoginInitial(),
30        ];
31
32        when(userRepository.authenticate(
33          username: 'valid.username',
34          password: 'valid.password',
35          university: University.pwr,
36        )).thenAnswer((_) => Future.value('token'));
37
38        expectLater(
39          loginBloc,
40          emitsInOrder(expectedResponse),
41        ).then((_) {
42          verify(authenticationBloc.add(LoggedIn(
43            token: 'token',
44            university: University.pwr,
45            username: 'valid.username',
46          ))).called(1);
47        });
48
49        loginBloc.add(LoginButtonPressed(
50          username: 'valid.username',
51          password: 'valid.password',
52          university: University.pwr,
53        ));
54      });
55    });
56  }
```

5.1.2. Widget Tests

A widget test (sometimes called the component test) tests a single widget [15]. The goal of it is to check that the widget's UI looks and interacts as expected. Testing a widget involves many classes and requires a test environment that provides the appropriate widget life cycle context.

For instance, widgets that are being tested should be able to receive and respond to user events and actions, perform layout, and instantiate child widgets. This kind of test is more comprehensive than a unit test. Though, like unit tests, a widget tests' environment is replaced with an implementation much more straightforward than a fully developed UI system.

Two examples of widget tests are shown in Listing 5.2. First, we need to create a `CustomCard` widget that will be tested in the next steps. The component is responsible for rendering items in grades and payments lists (see Figure 3.6). The `buildTestableWidget` method wraps the widget with the `MaterialApp` and `MediaQuery` so that it can be built and tested correctly.

The next part checks if the "CustomCard widget has aside, right and left widgets". First, we have to build `CustomCard` inside the test environment by using the `pumpWidget` method provided by WidgetTester. It builds and renders the provided widget.

The following steps use the `find` method to search through the widget tree for the aside, right, and left widget. Then, as a result, we expect to locate one `asideWidget`, two `rightWidget`s, and one `leftWidget` component. The test ends successfully if all expected values match what has been found by using `Finder`.

The second test also uses the `CustomCard` type. It verifies that the "CustomCard widget has a color". First, we have to build the component, and then we can create a predicate used to filter widgets. We want to find only components that are `Containers` and have a decoration attribute equal to the specified `BoxDecoration`. If such components are found, the test passes.

Listing 5.2: Flutter widget tests

```
void main() {
  final CustomCard customCard = CustomCard(
    asideWidgets: <Widget>[Text('asideWidget')],
    rightWidgets: <Widget>[Text('rightWidget'), Text('rightWidget')],
    leftWidgets: <Widget>[Text('leftWidget')],
    color: Colors.red,
  );

  Widget buildTestableWidget(Widget widget) {
    return MediaQuery(data: MediaQueryData(), child: MaterialApp(home: widget));
  }

  testWidgets('CustomCard widget has aside, right and left widgets',
      (WidgetTester tester) async {
    await tester.pumpWidget(buildTestableWidget(customCard));

    final asideFinder = find.text('asideWidget');
    final rightFinder = find.text('rightWidget');
    final leftFinder = find.text('leftWidget');

    expect(asideFinder, findsOneWidget);
    expect(rightFinder, findsNWidgets(2));
    expect(leftFinder, findsOneWidget);
  });

  testWidgets('CustomCard widget has a color', (WidgetTester tester) async {
    await tester.pumpWidget(buildTestableWidget(customCard));

    WidgetPredicate widgetColorPredicate = (Widget widget) =>
        widget is Container &&
        widget.decoration ==
            BoxDecoration(
                gradient: new LinearGradient(
                    stops: [0.02, 0.02],
                    colors: [customCard.color, Colors.white]),
                borderRadius: new BorderRadius.all(const Radius.circular(6.0)));

    expect(find.byWidgetPredicate(widgetColorPredicate), findsOneWidget);
  });
}
```

5.2. Server Application Tests

The transformation server required some manual testing when creating the configuration YAML files. All of it was done from the Postman application [23]. A sample POST request to fetch calendar data shown in Listing 5.3 was used to test the transformation configuration.

Listing 5.3: Sample content of the POST request to fetch calendar data

```
{
        "university": "pwr",
```

```
3         "username": "pwr230115",
4         "studentNumber": "123456",
5         "startDate": "2020-01-01",
6         "endDate": "2020-01-31"
7  }
```

Apart from the manual testing, the transformation was validated by unit tests. One of them is demonstrated below (List. 5.4) and checks if the function responsible for translating all incoming requests and responses works as expected. It is a Spring Boot test, as indicated by the `@SpringBootTest` annotation. At the top of the `TransformationServiceTest` class are the static strings used to prepare the tests:

- `REQUEST_SPEC` - Jolt specification for the request;
- `RESPONSE_SPEC` - Jolt specification for the response;
- `ADDRESS` - the API endpoint;
- `UNIVERSITY` - the chosen university;
- `CALENDAR_JSON_INPUT` - sample calendar request;
- `CALENDAR_JSON_OUTPUT` - expected calendar response.

`@BeforeAll` annotation placed above the `initAll` indicates that the method executes only once, before any test function. Inside it, a new transformation is initialized using predefined static strings.

The last method, called `transformCalendarJson`, tests the `transformJson` method from the `transformationService` class, which was autowired at the beginning. The `output` variable stores the result of the JSON transformation, which is compared, int the end, with the expected output. If they are equal, the test passes, if they are not the same or there was an exception while creating a new `JSONObject`, the test fails.

Listing 5.4: Tests for the transformation service requests

```
1  @SpringBootTest
2  class TransformationServiceTest {
3      @Autowired
4      TransformationService transformationService;
5
6      private final static String REQUEST_SPEC = "[{\"operation\": \"shift\",\"spec\":
           ↪  {\"username\": \"user\",\"studentNumber\": \"indeks\",\"startDate\": \"
           ↪ date.start\",\"endDate\": \"date.end\"}}]";
7      private final static String RESPONSE_SPEC = "[{\"operation\": \"shift\",\"spec
           ↪ \": {\"events\": {\"*\": {\"eventName\": \"events.[&1].name\",\"room\":
           ↪ \"events.[&1].classroom\",\"type\": \"events.[&1].eventType\",\"date\":
           ↪ {\"start\": \"events.[&2].startDateTime\",\"end\": \"events.[&2].
           ↪ endDateTime\"},\"*\": \"events.[&1].&\"}}}}]";
8      private final static String ADDRESS = "http://localhost:1080/calendar";
9      private final static String UNIVERSITY = "pwr";
10
11     private final static String CALENDAR_JSON_INPUT = "{\"university\": \"pwr\",\"
           ↪ username\": \"pwr123456\",\"studentNumber\": \"123456\",\"startDate\":
           ↪ \"2020-01-01\",\"endDate\": \"2020-01-31\"}";
12     private final static String CALENDAR_JSON_OUTPUT = "{\"user\":\"pwr123456\",\"
           ↪ indeks\":\"123456\",\"date\":{\"start\":\"2020-01-01\",\"end
           ↪ \":\"2020-01-31\"}}";
13
14     private static Transformation transformation;
15
16     @BeforeAll
17     static void initAll() {
18         transformation = new Transformation(UNIVERSITY, ADDRESS, REQUEST_SPEC,
               ↪ RESPONSE_SPEC);
```

```
19      }
20
21      @DisplayName("Test JSON transformation")
22      @Test
23      void transformCalendarJson() throws JSONException {
24          final String output = transformationService.transformJson(new JSONObject(
                ↪ CALENDAR_JSON_INPUT), transformation, LoadType.REQUEST, UNIVERSITY);
25          assertEquals(CALENDAR_JSON_OUTPUT, output);
26      }
27  }
```

Chapter 6

Project Presentation

The created application runs on both iOS and Android devices. It was installed only on one real device, iPhone XS, and it worked as expected. The iOS device simulation included in Xcode [4] is very accurate, and it well illustrates the final look of the app. All screenshots shown in this chapter were created using the iOS simulator. The main reason for that is that all testing data has been prepared on a local machine, including data in the SQLite database and mocked university API. The transformation server and its configuration have not been deployed to any external server, so they both have to be run locally.

Two images shown in Figure 6.1 represent the login screen in two states. The first one is the basic page with a university drop-down menu and two input fields for user login and password. The next image depicts the previous screen with the expanded university drop-down.

The screen seen by users after they log in is the homepage (Fig. 6.2a). At the top, there are two icons (logout and settings). When users click on the first one, they get logged out of the app. The second one should open a settings tab where users could change some properties. It has not

Figure 6.1: Screenshots of the mobile application: a) login page, b) login page with the university drop-down menu

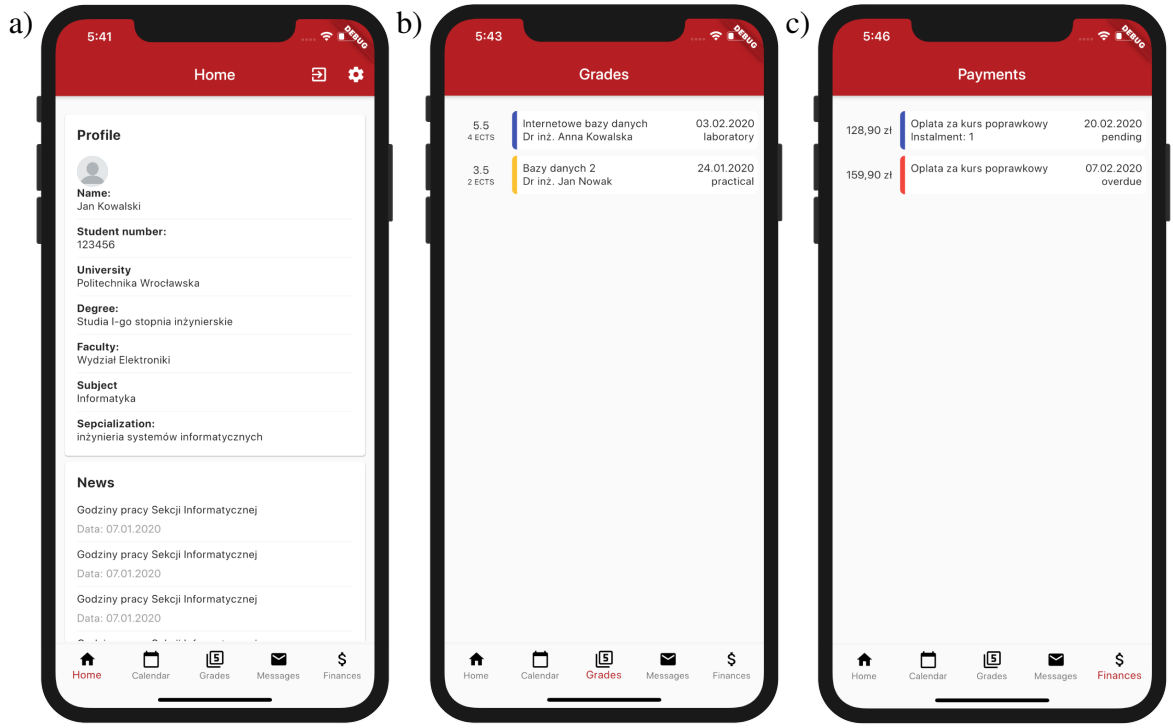

Figure 6.2: Screenshots of the mobile application: a) homepage, b) grades page, c) payments page

been implemented yet. Also, the default application language is set to English. As of now, the language change functionality has not been implemented.

In the middle, there is a "Profile" section where users can see their details like name, student number, university, degree, faculty, subject, and specialization. This segment also contains a user profile photo retrieved from the system. The next area consists of news from users' faculty and university websites. All screens, except the login page, have bottom navigation. It allows users to change tabs and quickly see which page they are currently on.

The page shown in Figure 6.2b lists all user grades with their details. On the left side, there is a section with the grade value and number of ECTS credits associated with the class. On the right, starting from the top, there is the class name, date of issue, lecturer, and type of class.

The page presented in Figure 6.2c shows a list of payments related to the current user. Each entry has a value (in Polish złotys), name, date of payment, and status. The number of the installment is an optional property and does not have to occur in every element.

One of the most important screens in the application is the calendar page (Fig. 6.3). It shows all classes that the user have to attend. The first image (Fig. 6.3a) does not present any events planned for the selected date. All red numbers indicate that it is a weekend or a holiday. The house at the top right corner means that the university is not working and the user can stay at home instead of going to class.

The image in Figure 6.3b shows a different view of the calendar. It can be swapped using the button in the top right corner of the page. There are three available appearances:

- one month;
- 2 weeks;
- a week.

The indicator in the lower-left corner of a day tells users how many events they have planned. When users select a date, a list of classes appears with details such as the beginning and the end of the class, its name, lecturer, classroom, and type. The viewed month can be changed by swapping the screen left or right or by clicking on one of the arrows at the top of the page.

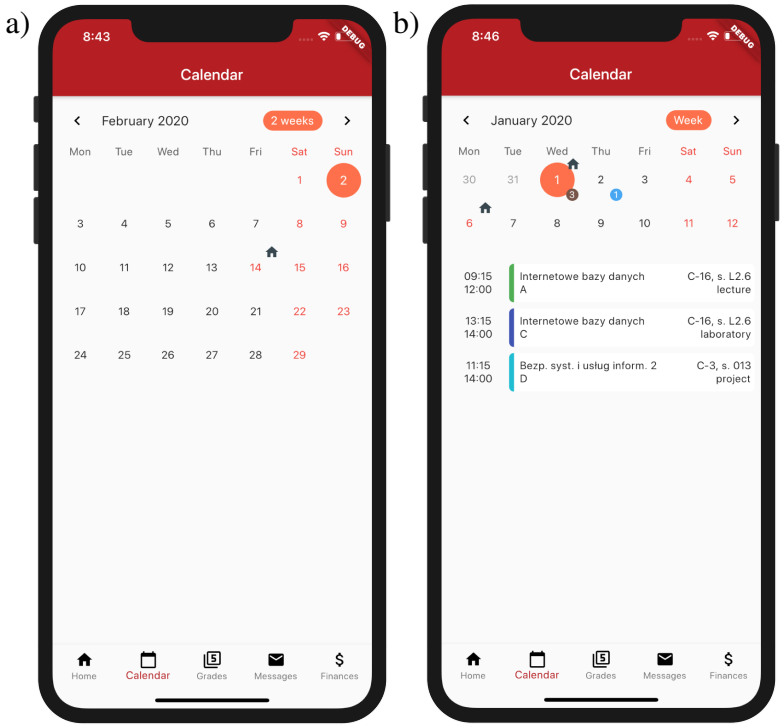

Figure 6.3: Screenshots of the mobile application: a) calendar page with no classes, b) calendar page with available classes

The messages page (Fig. 6.4a) lists all messages received by users in their university system. On the left-hand side, there is a section displaying specifics like sender, topic, and partial content. On the right, there is a date of receipt. After users select one of the listed messages, they are redirected to a more detailed page (Fig. 6.4b) where they can view, in addition to previously seen details, the entire content of the message, and CC recipients.

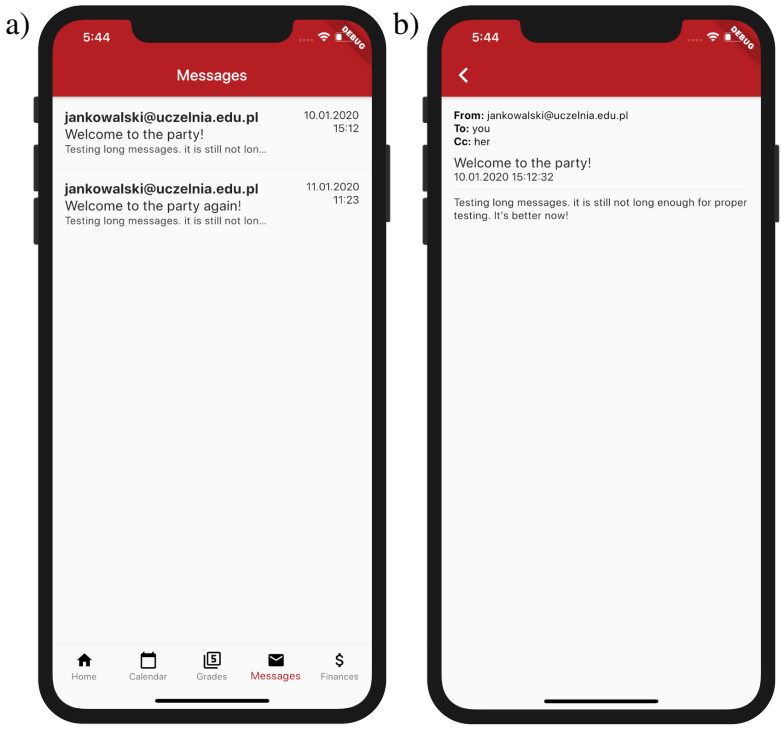

Figure 6.4: Screenshots of the mobile application: a) messages page, b) message details page

Chapter 7

Summary

7.1. Degree of Achievement

The main objective of the thesis was to design and develop a system that students could use to access data in their student services system. It was supposed to be comfortable to use and quick to access. The vital thing was that it should work on both mobile operating systems, iOS, and Android. It ought to have a local SQLite database where all received data could be stored and accessed offline if a need arises. The transformation server configuration had to be easily editable and expendable and should not require any restarts. It was supposed to connect two schema-incompatible systems and allow them to communicate with each other. The server should act as a middleman between the mobile application and external APIs. All of the previously stated objectives have been achieved, and the system is almost ready for publication in one of the digital distribution services.

Not all functional and non-functional requirements have been implemented. For example, users cannot change the language of the application since the only available language, as of now, is English. Additionally, students cannot calculate their average grade. Other requirements have been met.

7.2. Further Development

As mentioned earlier, the core of the application has been completed, and after minor changes, it can be published in the App Store or Google Play. Before all this takes place, some universities will have to agree to share the API of their student services system. If this happens, all transformations would have to be defined and stored in the system. Luckily, the architecture of the server provides an effortless way to implement the required changes. The main service is, in a sense, a microservice which allows for easy extensions if the service design is in any way incompatible with the given API. The application requires a considerable amount of testing before it can be published anywhere. It would need a trial run on a smartphone to see how long it takes to communicate with the external API through the transformation server.

7.3. Conclusion

Creating mobile applications using Flutter is very quick and effortless. It allows programmers to design beautiful user interfaces using Material Design. The documentation of the framework and the Dart language is extensive, simple to access, search, and read. Flutter gets more popular every day, which translates into a rapidly growing package base. The best thing about the

framework is that it can create two (iOS and Android) native applications from one codebase. Each of them can be modified separately if any need arises. It was a real pleasure to use it for this project.

The transformation service required lots of designing and thinking. The best part of creating the whole system was to come up with the best available architecture for the designed tasks. Coding was just a small portion of the actual work. That shows the importance of the design phase throughout the entire application development cycle.

The project has been very educative and can be easily extended in the future. Let us hope that the development will continue so that it can be eventually released to the public. It can eventually help many students find the right class or access their grades.

References

[1] Introducing JSON. https://www.json.org/json-en.html [accessed on 1 February 2020].

[2] Jolt GitHub Repository, Sept. 2019. https://github.com/bazaarvoice/jolt [accessed on 21 January 2020].

[3] F. Angelov. Flutter Login Tutorial with "flutter_bloc", Oct. 2018. https://medium.com/flutter-community/flutter-login-tutorial-with-flutter-bloc-ea606ef701ad [accessed on 3 February 2020].

[4] Apple Inc. Xcode, 2020. https://developer.apple.com/xcode/ [accessed on 2 February 2020].

[5] Axosoft, LLC. GitKraken Glo Boards, 2020. https://www.gitkraken.com/glo [accessed on 2 February 2020].

[6] J. D. Bloom. MockServer, Jan. 2020. http://www.mock-server.com/ [accessed on 1 February 2020].

[7] D. Boelens. The BLoC Pattern: From Bad to Acceptable Flutter Code, Aug. 2018. https://www.didierboelens.com/2018/08/reactive-programming---streams---bloc/ [accessed on 2 February 2020].

[8] Docker Inc. Docker Documentation, 2019. https://docs.docker.com/ [accessed on 1 February 2020].

[9] Docker Inc. Overview of Docker Compose, 2019. https://docs.docker.com/compose/ [accessed on 1 February 2020].

[10] V. Driessen. A successful Git branching model, 2010. https://nvie.com/posts/a-successful-git-branching-model/ [accessed on 2 February 2020].

[11] FreeLogoDesign.org. FreeLogoDesign. https://www.freelogodesign.org/ [accessed on 3 February 2020].

[12] Google. Material Design, 2020. https://material.io/design/ [accessed on 2 February 2020].

[13] Google Dart. Strona języka Dart, Jan. 2019. https://www.dartlang.org [accessed on 30 January 2020.

[14] Google Flutter. Flutter Documentation, Jan. 2020. https://flutter.dev/docs [accessed on 14 January 2020].

[15] Google Flutter. Testing Flutter apps, 2020. https://flutter.dev/docs/testing [accessed on 3 February 2020].

[16] A. Oghenero. Understanding Flutter Bloc Pattern, Sept. 2019. https://blog.soshace.com/understanding-flutter-bloc-pattern/ [accessed on 30 January 2020].

[17] Oracle. JDK 13 Documentation, Jan. 2020. https://en.wikipedia.org/wiki/Java_(programming_language) [accessed on 1 February 2020].

[18] Oren Ben-Kiki, Clark Evans, Ingy döt Net. YAML Ain't Markup Language (YAML™) Version 1.2, Jan. 2009. https://yaml.org/spec/1.2/spec.html [accessed on 30 January 2020.

[19] Pivotal Software, Inc. Spring - MVC Framework, Jan. 2020. https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html [accessed on 2 February 2020].

[20] Pivotal Software, Inc. Spring Boot, Jan. 2020. https://spring.io/projects/spring-boot [accessed on 1 February 2020].

[21] Pivotal Software, Inc. Spring Cloud Config, Jan. 2020. https://spring.io/projects/spring-cloud-config [accessed on 1 February 2020].

[22] Pivotal Software, Inc. Spring Framework, Jan. 2020. https://spring.io/projects/spring-framework [accessed on 1 February 2020].

[23] Postman, Inc. Postman homepage. https://www.getpostman.com/ [accessed on 3 February 2020].

[24] Sketch B.V. Sketch - The digital design toolkit, 2020. https://www.sketch.com/ [accessed on 2 February 2020].

[25] SophScope Sp. z o.o. Kiedy wykład. http://kiedywyklad.pl/ [accessed on 2 February 2020].

[26] SQLite Consortium. SQLite Documentation, Jan. 2019. https://www.sqlite.org/docs.html [accessed on 30 January 2020.

[27] Uniwersytet Warszawski. Aplikacja Mobilny USOS już dostępna, Oct. 2018. https://www.uw.edu.pl/aplikacja-mobilny-usos-juz-dostepna/ [accessed on 2 February 2020].

[28] Wikipedia contributors. Dart (programming language), Dec. 2019. https://en.wikipedia.org/wiki/Dart_(programming_language) [accessed on 14 January 2020].

[29] Wikipedia contributors. SQLite, Dec. 2019. https://en.wikipedia.org/wiki/SQLite [accessed on 14 January 2020].

[30] Wikipedia contributors. Android Studio, Jan. 2020. https://en.wikipedia.org/wiki/Android_Studio [accessed on 14 January 2020].

[31] Wikipedia contributors. Flutter (software), Jan. 2020. https://en.wikipedia.org/wiki/Flutter_(software) [accessed on 14 January 2020].

[32] Wikipedia contributors. Java (programming language), Jan. 2020. https://docs.oracle.com/en/java/javase/13/ [accessed on 1 February 2020].

[33] Wikipedia contributors. Reactive programming, Jan. 2020. https://en.wikipedia.org/wiki/Reactive_programming [accessed on 2 February 2020].

Appendix A

Description of the attached CD/DVD

The attached disk contains code and builds of the created applications (Fig A.1). The `mobile` directory contains `code` and `release` folders. Inside the first one is the source code of the Flutter mobile application with license and readme files. The second one holds Android application packages (APK) for three different architectures (`app-arm64-v8a-release.apk`, `app-armeabi-v7a-release.apk`, `app-x86_64-release.apk`), as well as a build dedicated to iOS devices (`jsos_helper.app`). All of them can be run on a smartphone or a computer using emulation software.

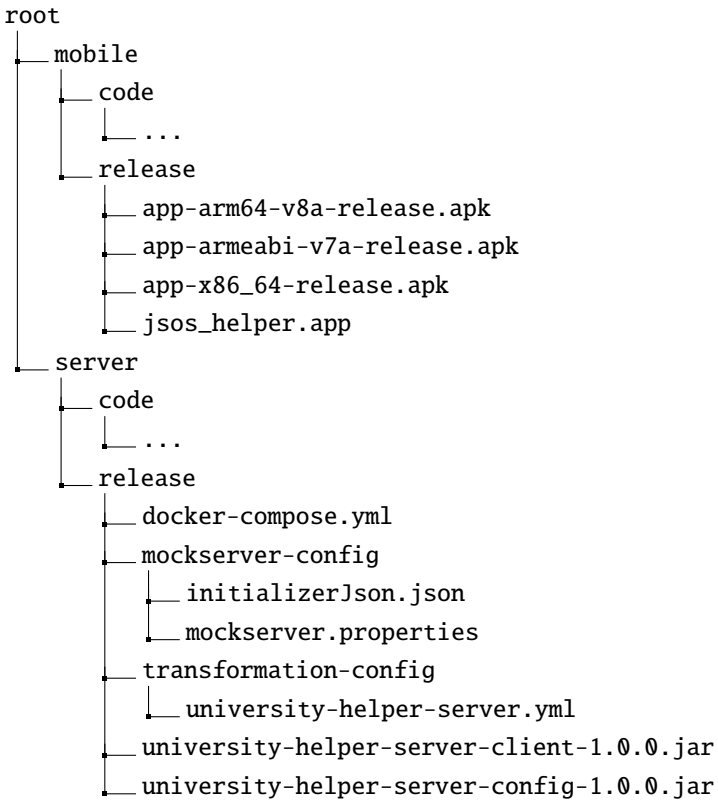

Figure A.1: Disk directory structure

The server subtree contains the `code` folder with the source code of the translation and configuration modules. Inside the parent project, there are license and readme files. In the `release` folder, there is the `docker-compose.yml` file for run-

ning the entire server and the mocking service at once. The `mockserver-config` holds MockService startup configuration. The `transformation-config` stores Spring Cloud Config properties used to translate JSON schemas. Last but not least, there are two jars (`university-helper-server-client-1.0.0.jar`, `university-helper-server-config-1.0.0.jar`), one for each server module.

Appendix B

System Deployment

The mobile application can be installed from the attached APK files for Android devices. To do so, we need to copy a package for the chosen architecture onto a smartphone. Then we only have to click on the copied application and let it install on the device. Installing the iOS package is more complicated, so it is much better to run it in an XCode simulator instead of a mobile machine. We simply have to start the virtual device and drag and drop our package onto its screen. The Android application can also be run on an Android emulator.

In the case of this project, it is best to run it on a computer instead of a mobile device. We use MockServer to mimic the behavior of a real student services system API. The mock server is not embedded into the mobile application, and because of this, logging in and browsing screens is not possible for applications installed on smartphones.

The best way to test this proof of concept is to run the mobile application using an Android or iOS emulator and start the backend server using the steps described below.

Docker and Docker Compose were used to simplify the startup of the server application and the API mocking service. Both tools are required for the next steps, so they must be installed on the system. To start the whole backend, we need to run the following commands from the disk root directory:

```
cd server/release
docker-compose up
```

You can also run and stop Docker in the detached mode:

```
docker-compose up -d
docker-compose down
```

Appendix C

User's Guide

This documentation guides users through one of the use cases (see Section 2.4). It leads step by step through authentication, messages page and message details page.

Figure C.1a represents the login page. To be able to use the application, users need to login to the system with credentials for their student services system. To do this, they need to complete the required fields and actions:

1. drop-down with a list of available universities;
2. the username used to log in to the student services system;
3. the password used to log in to the student services system;
4. click on the "Login" button.

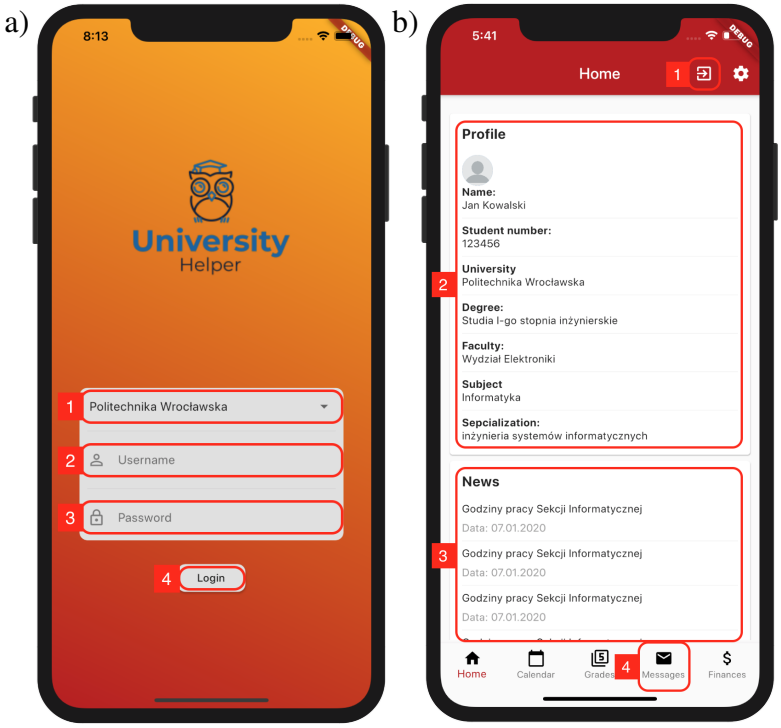

Figure C.1: User's guide: a) login page, b) homepage

After the successful login users are redirected to the homepage (Fig. C.1b), where they can see sections:

1. logout button;
2. user profile information;

3. news feed;
4. link to the messages page.

When users click on the link, they are redirected to the messages page (Fig. C.2a). Where there are two segments:

1. message entries;
2. navigation with a link to the homepage.

Just after users select one of the messages from the list, they are moved to another screen shown in Figure C.2b. The message details page is composed of:

1. back button - used to get back to the messages page;
2. message details;
3. message content.

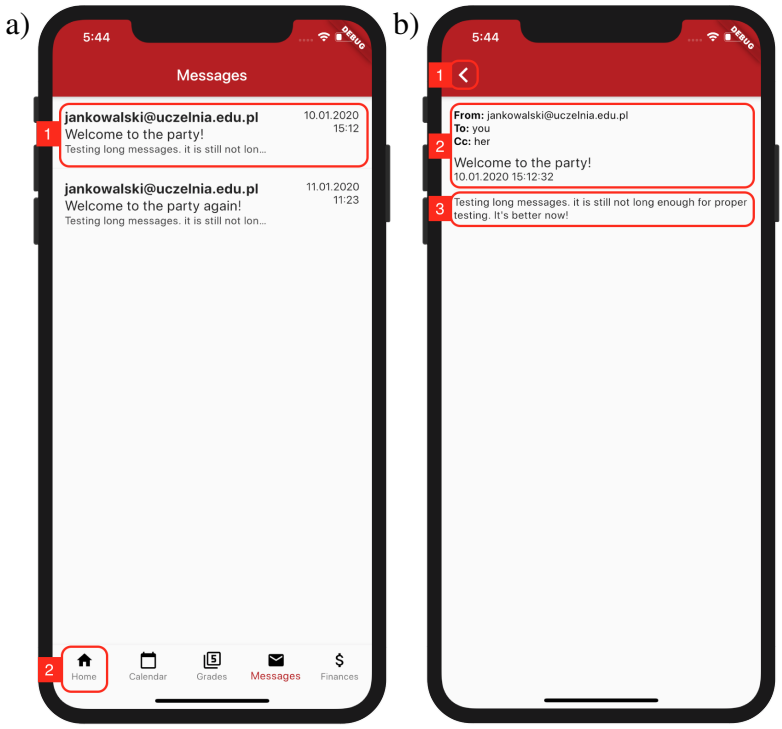

Figure C.2: User's guide: a) messages page, b) message details page