

The Collection of Practical Assignments for Students Software Projects

Lubomír Benko
José Daniel González-Domínguez
Zenón José Hernández-Figueroa
Tomáš Jakúbek
Jozef Kapusta
Jaromír Landa
Juan Carlos Rodríguez-del-Pino
Ján Skalka
Peter Švec
Pavel Turčínek

www.fitped.eu

2021

The Collection of Practical Assignments for Students Software Projects

Published on

November 2021

Authors

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

Tomáš Jakúbek | Mendel University in Brno, Czech Republic

Jozef Kapusta | Pedagogical University of Cracow, Poland

Jaromír Landa | Mendel University in Brno, Czech Republic

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Peter Švec | Teacher.sk, Slovakia

Pavel Turčínek | Mendel University in Brno, Czech Republic

Reviewers

Anna Stolińska | Pedagogical University of Cracow, Poland

Dušan Junas | Teacher.sk, Slovakia

Cyril Klimeš | Mendel University in Brno, Czech Republic

Piet Kommers | Helix5, Netherland

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Michal Švec | Teacher.sk, Slovakia

Graphics

Marcela Skalková | Teacher.sk, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union



The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



Licence (licence type: Attribution-Non-commercial-No Derivative Works) and may be used by third parties as long as licensing conditions are observed. Any materials published under the terms of a CC Licence are clearly identified as such.

All trademarks and brand names mentioned in this publication and all trademarks and brand names mentioned that may be the intellectual property of third parties are unconditionally subject to the provisions contained within the relevant law governing trademarks and other related signs. The mere mention of a trademark or brand name does not imply that such a trademark or brand name is not protected by the rights of third parties.

© 2021 Constantine the Philosopher University in Nitra

ISBN 978-80-558-1796-5

Table of Contents

INTRODUCTION	10
WEB DEVELOPMENT.....	11
WEB GAME	12
Available Solutions	12
REQUIREMENTS.....	12
Functional Requirements	12
Non-functional Requirements	12
APPLICATION DESIGN	12
Technology and Architecture Selection.....	12
USER INTERFACES	12
SOLUTION	13
Prepare the environment.....	13
Create motion	16
Shot	18
Possibilities of other game function.....	18
AUTHENTICATION AND MENU BASED ON USER'S ROLES FOR A WEB APP IN PHP	19
Available Solutions	19
REQUIREMENTS.....	19
Functional Requirements	19
Non-functional Requirements	20
Use Case Diagram.....	20
APPLICATION DESIGN	20
Technology and Architecture Selection.....	20
Data Model.....	20
User Interface	21
SOLUTION	23
Introduction.....	23
Creating and populating the database	23
Database class	25
Security class	26
User class.....	26
View class	28
Navigation class.....	29
Home page	30
Login page	30
Logout page.....	31
CSS	31
Source Code.....	33
CREATE, RETRIEVE, UPDATE, DELETE, AND LIST USERS FOR A WEB APP IN PHP.....	34
Available Solutions	34
REQUIREMENTS.....	34

Functional Requirements	35
Non-functional Requirements	35
Use Cases Diagram	35
APPLICATION DESIGN	36
Technology and Architecture Selection	36
Data Model	36
User Interface	37
SOLUTION	38
Introduction	38
Specific database queries and server-side data validation	38
List of users	42
Add a new user	43
View user profile	45
Edit user profile	46
Delete user account	47
Change user password	47
Form validation in browser. The example of the add user from	48
Delete user account with Ajax	50
Source Code	52
CHAT IN VUEJS	53
Available Solutions	53
REQUIREMENTS	53
Functional Requirements	53
Non-functional Requirements	54
Use Case Diagram	54
APPLICATION DESIGN	54
Technology and Architecture Selection	54
User Interfaces	55
SOLUTION	56
Introduction	56
Clone repository	56
Create Vue.js project using Vue CLI	57
Project directory description	57
Install Vuetify	57
Installation of the necessary dependencies	58
Vue.js documentation	58
Start development mode	58
Remove Hello world template	58
Vue single file component	59
The Vue instance	59
SPA development in general	59
Creating AppBar component	60
Data and props in general	61
Props and directives	62
Adding router	65
Login and register forms	66
Adding routes	68
Adding links	69
HTTP client setup	70
HTTP requests	71

Token manager class	73
Login and logout flow	74
Room components	77
Secured routes	82
Message components	83
SIMPLE FORMS.....	89
JQUERY ANIMATION	90
HAMMER HITTING TURTLE	91
MOBILE APPLICATIONS.....	92
TO-DO APPLICATION FOR ANDROID IN JAVA.....	93
Available Solutions	93
REQUIREMENTS.....	93
Functional Requirements	93
Non-functional Requirements	93
APPLICATION DESIGN	93
Technology and Architecture Selection.....	93
Data Model.....	94
User Interfaces	94
SOLUTION	94
Application database.....	94
SharedPreferences	97
SplashScreenActivity	97
ToDoListActivity.....	99
AddEditTaskActivity.....	100
TaskDetailActivity	102
Settings Activity.....	103
Working with images.....	104
GOOGLE MAP APPLICATION TEMPLATE	106
Available Solutions	106
REQUIREMENTS.....	106
Functional Requirements	106
Non-functional Requirements	106
APPLICATION DESIGN	106
Technology and Architecture Selection.....	106
Data Model.....	106
User Interfaces	107
SOLUTION	107
LocationManager	107
MapManager.....	108
MarkerManager	109
MapFragment.....	111

TO-DO APPLICATION WITH MAPS	115
Available Solutions	115
REQUIREMENTS.....	115
Functional Requirements	115
Non-functional Requirements	115
APPLICATION DESIGN	115
Technology and Architecture Selection.....	115
Data Model.....	116
User Interfaces	116
SOLUTION	116
Design definition	116
Fragment	117
Navigation chart	117
Creation of the database.....	119
Database class	120
Implementation of the list of tasks	122
Class to display the list	123
List of elements	124
Adapter.....	125
LayoutManager	126
Adding and modifying the task.....	127
Custom Views for the adding of tasks	130
Layout.....	130
Class.....	131
BaseClasses.....	132
Use in code	134
Dependency injection.....	134
Options menu	136
WORKING WITH MAP	137
Getting the key from the console.....	138
Map in fragment.....	139
 MAP BOX APPLICATION TEMPLATE	 141
Available Solutions	141
REQUIREMENTS.....	141
Functional Requirements	141
Non-functional Requirements	141
APPLICATION DESIGN	141
Technology and Architecture Selection.....	141
SOLUTION	142
Connection	142
Activation	146
POSITION ON THE MAP.....	147
MARKERS.....	149
Area boundary.....	152
DIRECTIONS.....	154
CONCLUSION	157

COMPASS.....	158
Available Solutions	158
REQUIREMENTS.....	158
Functional Requirements	158
Non-functional Requirements	158
SENSORS	158
Communication with the environment	159
Sensor categorization	159
List of sensors in the device	159
ACCELEROMETER - ACQUISITION OF DATA FROM SENSORS.....	160
Coordinate system	161
Principle of accelerometer operation	161
Capturing values.....	162
Registration and unregistration	164
Reading data from the sensor	165
Data interpretation	166
GEOMAGNETIC SENSOR.....	167
Compass	167
LIGHT SENSOR.....	172
SIMPLE GAME WITH ACCELEROMETER.....	173
STEP DETECTOR.....	174
DATA PROCESSING	176
CUSTOMER-PRODUCTS DATA MODEL DEVELOPMENT	177
Available Solutions	177
REQUIREMENTS.....	178
DATA MODELLING	179
ENTITY IDENTIFICATION	179
RELATIONSHIP IDENTIFICATION	180
ATTRIBUTES.....	187
KNOWLEDGE DISCOVERY FROM LOG FILE	191
REQUIREMENTS.....	191
Methodology	191
SOLUTION	191
Data cleaning.....	191
User/session identification.....	194
Data transformation.....	197
Data analysis.....	200
KNOWLEDGE DISCOVERY USING SEQUENCE PATTERN MINING.....	203

KNOWLEDGE DISCOVERY USING CLUSTER ANALYSIS	204
SOURCE CODES	205
BIBLIOGRAPHY	206

Introduction

A publication that you hold in your hands, or rather, it may be said that you are reading it from your displays, provides a cross-section view on tasks and projects that:

- illustrates the principles of selected technologies that are currently used in the development of various types of applications,
- describes the methodologies, procedures, and creation of solutions for some complex problems solved at the level of development environments or frameworks.

The publication is one of the results of the project Work-Based Learning in Future IT Professionals Education (ERASMUS+ Programme 2018, KA2, project number: 2018-1-SK01-KA203-046382). It aims to provide enough skills and information to develop a specific type of application as quickly as possible.

Tasks, respectively, the projects presented in the publication are divided into three categories, which copy the skills needed for the application development in the current and probably in the near future market.

The first category of applications consists of applications designed for the web, with assignments going from a simple web application through backend PHP applications focused on working with a database to using the Vue front-end framework to create a complex application.

The second category covers the development of mobile applications, which consists of application types focused on working with databases, mapping data, and mobile device sensors.

Data-oriented projects complement the publication. One of the projects provides a comprehensive procedure for designing a large database and the other analytical processing of data from logs, which reflect the user's activity in the web application environment.

Each assignment contains specifications so that the reader gets the same basis as a developer in a technology company. For some tasks, the specification is very detailed, in others very brief and leaves the understanding and processing of the assignment to the developer.

The published projects were selected to prepare students for the system of lifelong learning, which in the field of IT usually begins, even with excellent university training, immediately after graduating from university.

The collection thus represents a way to bring university education closer to work-based learning or, conversely, to apply a work-based learning strategy in university education. Although "collections of assignments" are not very preferred in recent times, mainly due to the rapid advancement of technologies, project assignments are formulated to sufficiently separate from the technologies used, and analogous assignments can be solved within various technologies.

It is appropriate to apply a project approach to solving these problems - the solution of each problem is a separate project set in the real world through proper technology and based on searching for information in the web environment.

In addition to the description of the problem and the procedure for solving it, complex source codes are also available. However, we did not use any storage or versioning system that we require students and developers to use, but to provide their availability, we decided to publish them directly on the page of this publication.

WEB DEVELOPMENT

Web Game

The presented example shows a process of game development with JavaScript.

The web application introduces the basic principles for animation and game development with JavaScript and jQuery. It shows the traditional shooter game. The gamer is a hunter. The objects are flying ducks. The game is controlled by a mouse.

Recommended Number of Developers

One.

Available Solutions

There are a lot of similar games. To name some:

- Moorhuhn (<https://www.crazygames.com/game/moorhuhn-shooter>)
- Moorhuhn Shooter (<https://keygames.com/moorhuhn-shooter-game>)

Requirements

The application will be created in JavaScript and jQuery. We recommend using some HTML or JS editor like Sublime Text, Notepad++, Visual Studio Code, etc. The development requires a basic knowledge of Object-oriented programming and JavaScript.

Functional Requirements

- a) The user can move with the pointer on the game environment
- b) Flying objects (ducks) will fly in the game environment. The movement will be from right to left and back from left to right.
- c) The user can shoot down the flying object (duck) by the right button on the mouse.
- d) The user can see the current state of the count of the rifle cartridges.

Non-functional Requirements

- a) web browser Firefox or Chrome
- b) JavaScript programming language

Application Design

Technology and Architecture Selection

The application is written using the JavaScript programming language.

User Interfaces

The interface and design of selected students' solutions (Bartłomiej Pietras, Damian Serwatka, Magdalena Mucha, Tymoteusz Bojarski).



Fig. 1 Background examples

Solution

We have to solve the following steps for the development of the web game:

1. Prepare the environment
2. Set the default setting for every duck
3. Create motion
4. Solve shot down of duck
6. Counter of shot-downs

Prepare the environment

All code of game we will create into one HTML file. It is a basic website with an HTML5 structure.

```
<!DOCTYPE html>
<html>
<head>

</head>
<body>
```

```
</body>
</html>
```

We don't prefer to download and host jQuery myself. We can include it from a CDN (Content Delivery Network). We add the link to CDN into <head> element.

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
```

The part <body> of our game is very simple. It contains counter of shot-downs (text information in <h1> element).

The most important part of our game is the game environment. We create it into one div element. The id of this element is "space".

We have to create flying objects (ducks). There will be only four objects of ducks together at one time. If the user shoots down the duck, we show the effect of the shootdown, and we create a new duck in the previous flying object (and change the coordinates of a new flying object).

We will use 4 <div> elements for ducks with id bird1, bird2, bird3 and bird4.

```
<h1>Counter of shot-downs: <span id="counter"></span></h1>
<div id="space"></div>
<div id="pointer"></div>
<p></p>
<div id="bird1"></div>
<div id="bird2"></div>

<div id="bird3"></div>
<div id="bird4"></div>
```

We have to create a good look design for elements. We can use CSS to define it.

For a better graphical design, we can replace **background-color** with **background-image** and load some pictures of the environment and flying ducks (animated gif).

Do not forget that CSS style we have to input into the <head> element.

We can see that for all elements we set the style for the cursor. The cursor CSS property sets the type of mouse cursor, if any, to show when the mouse pointer is over an element. For our game, we can switch off the cursor and set this style to none.

```
<style>
    #space{
        width:800px;
        height:600px;
        margin:10px 10px 10px 10px;
        background-color:yellow;
        border:1px solid black;
        /*float:right;*/
        cursor:none;
    }

    #pointer{
        width:10px;
        height:10px;
        background-color:red;
        border:1px solid black;
        position:absolute;
        cursor:none;
```

```

        border-radius:50%;
    }

    #bird1,#bird2,#bird3,#bird4{
        width:30px;
        height:30px;
        position:absolute;
        cursor:none;
        background-color:brown;
    }
</style>

```

Additional source code we will create in jQuery. For this reason, all next code we will add into the "main" jQuery method **`$(document).ready()`**

The first function in **`$(document).ready()`** is even listener **`mousemove`**. For every mouse move over the space, we set new coordinates for the pointer (rifle sight). We set it in a simple way by **`.css`** method.

```

<script>
$(document).ready(function() {

    $("#space").mousemove(function(event) {
        $("#pointer").css("top",event.pageY+"px");
        $("#pointer").css("left",event.pageX+"px");
    });

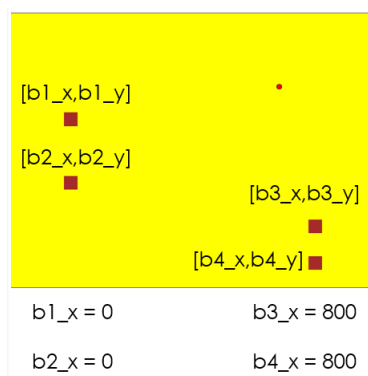
});
</script>

```

Set the default setting for every duck

There are many basic settings for ducks. All settings we add at the beginning of the method **`$(document).ready()`**.

At the beginning of our game, we set that all ducks (birds) live. The second important settings are coordinates for ducks (birds). X-coordinates for the first two ducks are near the left border of the environment (**`b1_x = 0; b2_x = 0;`**), and for the last two ducks are near the left border (**`b3_x = 800; b4_x = 800;`**).



We will change only X-coordinates during the game.

We set random Y-coordinates for ducks. We can place ducks with Y-coordinates with the **`.css()`** method.

```

var counter = 0;
var b1_x = 0;
var b2_x = 0;
var b3_x = 800;
var b4_x = 800;

/* set all bird to live*/
var b1_live = true;
var b2_live = true;
var b3_live = true;
var b4_live = true;

var b1_y,b2_y,b3_y,b4_y;
/* generate random position (top) for 4 birds*/
b1_y = Math.floor(Math.random()*600);
b2_y = Math.floor(Math.random()*600);
b3_y = Math.floor(Math.random()*600);
b4_y = Math.floor(Math.random()*600);

$("#bird1").css("top",b1_y);
$("#bird2").css("top",b2_y);
$("#bird3").css("top",b3_y);
$("#bird4").css("top",b4_y);

```

Create motion

There are a few interesting methods for animation in JavaScript. For the motion in our game, we use **setInterval()** method. This method calls a function or evaluates an expression at specified intervals (in milliseconds).

The setInterval() method will continue calling the function until **clearInterval()** is called, or the window is closed.

A simple example of the motion of a bird (div element with square design) is in the following source code. We use method **setInterval()**. This method calls its own function **game()** every 20 milliseconds. We calculate X-coordinate every 20 milliseconds in this function and we set a new value (new position) by **.css** method.

```

<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
<style>
    #bird1{
        width:30px;
        height:30px;
        position:absolute;
        cursor:nono;
        background-color:brown;
    }
</style>
<script>
    $(document).ready(function(){
        var b1_x = 0;

        /*Function setInterval() set system to call function game()
        every 20 millisecond*/
        requestId = setInterval(game,20);

        /*every 20 milliseconds will be this function call*/

```



```

        function game() {
            /*effect - flying*/
            b1_x += 3;
            $("#bird1").css("left", b1_x + "px");
        }
    });
</script>
</head>
<body>
    <div id="bird1"></div>
</body>
</html>

```

The motion of ducks in the game is similar to in the previous example. We set method **game()** with **setInterval()**. The method **game()** is the main method for our game. We have to check if bird1 live. If yes, we can recalculate X-coordinate for motion from left to right. If the bird1 do not live, we have to create a fall down effect. We can create it by recalculating Y-coordinate.

Next two conditions (**if(b1_x > 800)** and condition **if(b1_y > 600)**) are for checking the collision with border. The first condition is if the bird1 is at the right border. We just only set X-coordinate to 0 and set a new position on the left border.

If **bird1** is at the bottom of the game environment, it means that **bird1** does not live, and the fall down effect is done. We have to "create" a new bird (duck) into an <div> element with **id="bird1"** and set a new coordinates for a new duck. It is important to set bird1 as a live flying object back.

```

requestId = setInterval(game, 20);

/*every 20 milliseconds will be this function call*/
function game() {

    /*effect - flying*/
    if(b1_live) {
        b1_x += 3;
    } else {
        b1_y += 5;
        $("#bird1").css("top", b1_y + "px");
    }

    $("#bird1").css("left", b1_x + "px");

    /*if some bird is out of space (environment) for game - left/right*/
    if(b1_x > 800) {
        b1_x = 0;
        $("#bird1").css("top", Math.floor(Math.random() * 600));
    }

    if(b1_y > 600) {
        b1_live = true;
        //for Y coordinate
        b1_y = Math.floor(Math.random() * 600);
        $("#bird1").css("top", b1_y);

        //for X coordinate
        b1_x = 0;
        $("#bird1").css("left", b1_x);
    }
}

```

We would like to show examples clearly and simply. For this reason, we create only motion for the bird1.

Do not forget to create the motion for bird2, bird3 and bird4.

Shot

The main part of our game we solved into **game()** method. There is a solution for the fall down effect in this method. It is necessary to check if the duck (bird) is living permanently for effects and coordinates changing.

We can solve the shot down of duck very simply. We create an event listener for the event click on the div element for a duck. Essential for use in this method is the set that duck does not live.

```
$("#bird1").click(function() {
    b1_live = false;
    counter += 1;
    refresh_counter();
});
$("#bird2").click(function() {
    b2_live = false;
    counter += 1;
    refresh_counter();
});
$("#bird3").click(function() {
    b3_live = false;
    counter += 1;
    refresh_counter();
});
$("#bird4").click(function() {
    b4_live = false;
    counter += 1;
    refresh_counter();
});
```

Counter of shot-downs

The last step in our game is the actualization of the counter of shot-downs. We create this counter in a span element with **id="counter"**.

```
function refresh_counter() {
    $("#counter").text(counter);
}
```

Possibilities of other game function

The main aim of our example was to show the essential processes for creating a simple game with jQuery and JavaScript. The main emphasis was on simplicity and clarity. In the following list, we show the possibilities of other game functions and improvements:

- More ducks - with similar steps, we can generate more ducks (not only 4)
- Random motion of every bird (not only from left to right and right to left)
- Change bird's size - effect for flying in, flying out of position
- Change the size of weapon - change cursor size (!!!other rules for the shot, shot-down)

- Level selector (faster mode, more birds),
- Show the level of a hunter (or count of lives)

Authentication and Menu Based on User's Roles for a Web App in PHP

This task is in the area of web applications. More specifically, it is a PHP¹ development of an authentication system and a role-based menu system for a web application.

The goal is to build an authentication system and a menu for a general-purpose web application. The development will not release a fully functional application and will not cover any unresolved problem, but rather will be used to demonstrate and practice the basic concepts of the PHP web programming language and their application to the development of session-based authentication systems role-based menus. The task includes PHP components, hypertext markup language (HTML) pages, forms, sessions, database access, and web application aesthetics design. A basic authentication system and a menu system for a web application are good examples covering all these features.

The students will develop an authentication system with the login and logout actions and a basic menu system aware of the authenticated user's role. The goal is that students use from scratch some of the technologies involved in the development of web applications. The application components to develop will be generic and ready to be adapted for a specific complete application. The development of components commonly used in many web applications increases the possibility of applying the acquired knowledge in the future.

Recommended Number of Developers

The number of developers and the time to fulfil the task requirements vary based on the developer's knowledge of the technology and skills. It is assumed that the student has some knowledge or experience with PHP, HTML, Cascade Style Sheets (CSS) and Data Bases (DB).

The task is suitable for 1 or 2 junior developers with an estimated 10-12 person-hours duration.

Available Solutions

There are a great number of solutions to this problem on the internet. The proposed task here highlights the reusability and security.

Requirements

Any integrated development environment (IDE) with support for PHP web development can resolve the task. If a proper IDE is not available and the task will be developed in the context of a teaching environment using Moodle, the VPL plugin can be used. A modern web browser is also needed for development.

Functional Requirements

Main functional requirements:

- a) A non-authenticate user can log in with credentials.

¹ <https://www.php.net>

- b) The application shows the menu for the user's role. A non-authenticate user has a special role.
- c) An authenticated user can log out.

Non-functional Requirements

- a) PHP 7.2 or higher.

Use Case Diagram

The diagram of use cases is: There are four actors and three specific uses cases. Other use cases are open as proof of concept.

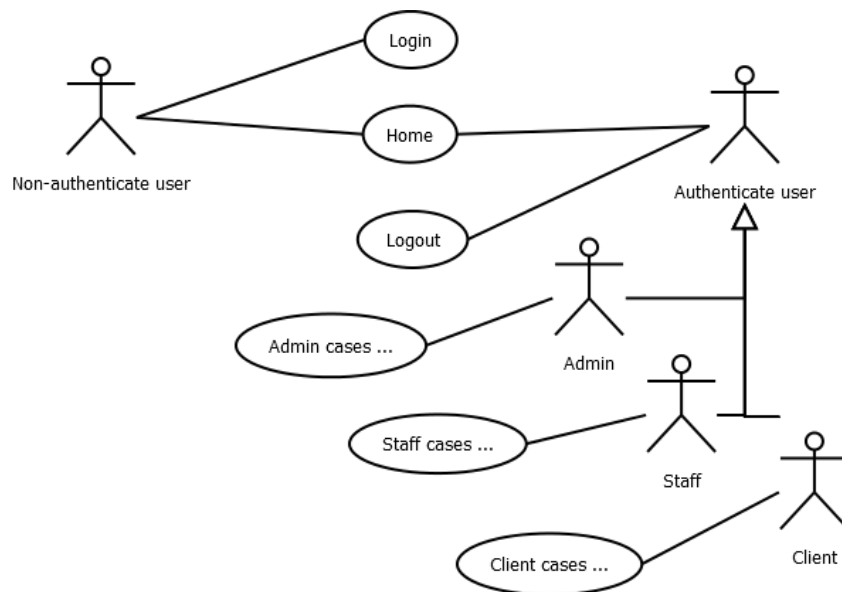


Fig. 2 Use cases

Application Design

Technology and Architecture Selection

The architecture of the application is simple – it uses a different HTML page for each action generated on the server-side. The idea is to show students the basic web development technologies. The use of frameworks or other modern technologies can accelerate the development, hiding what is happening in the server or even browser. With this approach, the browser shows HTML and sends forms data and URL requests to the server. The server runs the application responding to the browser requests, generating HTML pages, and accessing the data storage.

Data Model

The component uses the database table named users showed below. The fields used are common in tables storing users' data for the intended purpose.

- **id.** Record identification auto-incremented.
- **account.** Account name.
- **password.** Hashed password to access the account
- **name.** User real name.

- **role.** The role is an integer with the following interpretation:
 - 1 => Admin
 - 2 => Staff
 - 3 => Client

These roles are enough for a proof of concept.

- **email.** User email.
- **address.** User full address.
- **mobile.** User mobile number.

users	
id	INTEGER
account	NVARCHAR(20)
password	NVARCHAR(32)
name	NVARCHAR(200)
role	INTEGER
email	NVARCHAR(200)
address	NVARCHAR(200)
mobile	NVARCHAR(200)

• Fig. 3 Users DB table

User Interface

The user interface is simple; every page has four zones: header, navigation, content, and footer. The header shows the name of the application, the page name, and the logo image on the right. The navigation contents the menu with a horizontal line of options: the home page, the user's role menu options, and the login or logout menu option. The content shows the page information. The footer shows the contact and copyright staff.



Fig. 4 Home page for non-authenticated users

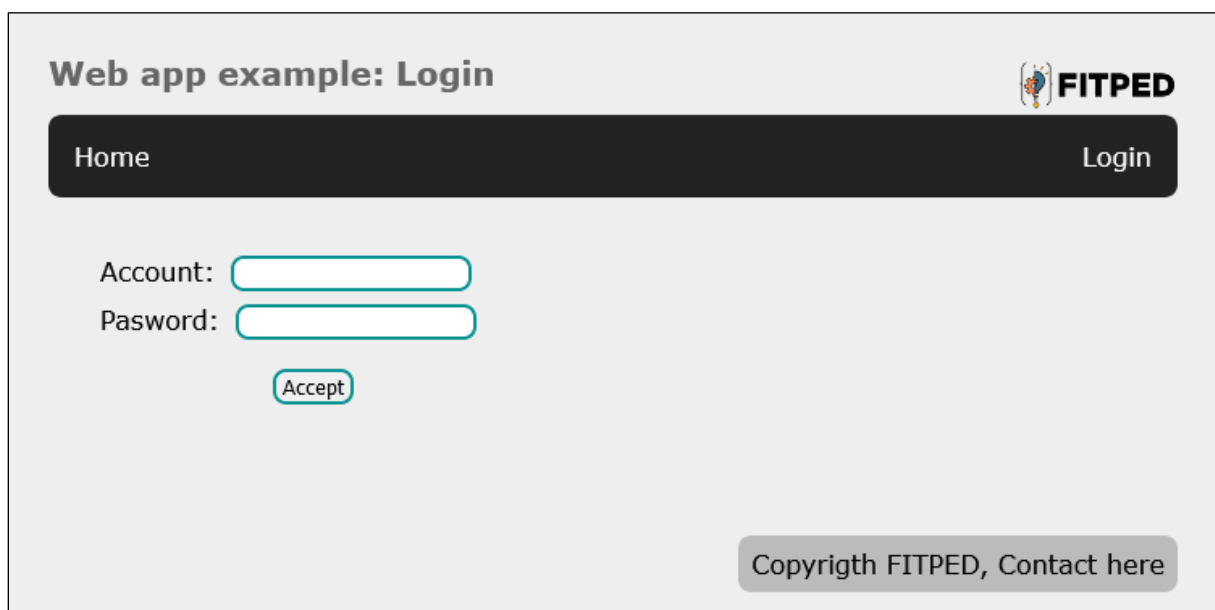


Fig. 5 Login page



Fig. 6 Home page for users with the administration role

Solution

Introduction

The next sections describe the development of the web application components, including creating the database, PHP classes, PHP generated web pages and CSS. Following the idea that the content of the tutorial must be didactic and serve as a proof of concept, not a development with a result prepared to be used "as is", its design has prioritized the simplicity of the solution over the full functionality. Nevertheless, the safety of the generated components has been significantly taken into account. The description will detail the development of each featured source file justifying its need, describing how to use it and highlighting the important code. Alternatives to the chosen approach will also be indicated.

The first of next sections establishes the database management system (DBMS) to be used and describes how to create a test DB; the following sections show the classes that represent and supports the development components, some files that generated web pages using the classes, and finally, the last section will show the CSS.

Creating and populating the database

The creation of the database and the addition of test data are necessary to start testing and using the components to be created. PHP has multiple frameworks to access databases. For versatility and simplicity, the [PHP Data Objects](https://www.php.net/manual/en/book.pdo.php)² (PDO) extension has been chosen. [SQLite](https://www.sqlite.org/index.html)³ will be used as BDMS, but just by changing a parameter of the connection command, another BDMS could be used.

As indicated above, only the "users" table from the database is required for these components. This table has some common fields, but two of them worth be highlighted:

- **role:** this field stores the user's role as an integer, could have been designed with a table of roles and permissions for each role.

² PDO <https://www.php.net/manual/en/book.pdo.php>

³ SQLite <https://www.sqlite.org/index.html>

- **password:** To avoid passwords compromised in case of unauthorized access to the table, they are not stored in plain text but cyphered using MD5 hashing.

```

/** Create Tables */
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    account NVARCHAR(20) NOT NULL,
    password NVARCHAR(32) NOT NULL,
    name NVARCHAR(200) DEFAULT '',
    role INTEGER DEFAULT 2,
    email NVARCHAR(200) DEFAULT '',
    address NVARCHAR(200) DEFAULT '',
    mobile NVARCHAR(200) DEFAULT ''
);
/*
Roles:
1 => admin
2 => staff
3 => client

```

Code 1 Creates users table - data_base.sql

Notice the use of hash function md5 of '1' and '2' as the password. For a real app, a better hash function with salt is recommended, see '[Safe Password Hashing](https://www.php.net/manual/en/faq.passwords.php)'⁴ section on the PHP home page <https://www.php.net/manual/en/faq.passwords.php> for more details.

```

/** Populate Tables */
INSERT INTO users (account, password, name, role) /* password=1 */
VALUES ('admin', 'c4ca4238a0b923820dcc509a6f75849b',
    'Thepo Werful', 1);
INSERT INTO users (account, password, name, role, email) /* password=1 */
VALUES ('staff1', 'c4ca4238a0b923820dcc509a6f75849b',
    'Wor Kerall', 2, 'wor.kerall@shop.com');
INSERT INTO users (account, password, name, role, email, address, mobile) /*
password = 1 */
VALUES ('client1', 'c4ca4238a0b923820dcc509a6f75849b',
    'Findsop Portunities', 3, 'findsop.portunities@gmail.com',
    'New York City, USA', '621111111');
INSERT INTO users (account, password, name, role, email, address, mobile) /*
password = 2 */
VALUES ('client2', 'c81e728d9d4c2f636f067f89cc14862c',
    'Iwant Itall', 3, 'iwant.itall@gmail.com',
    'Paris, France', '621123456');

```

Code 2 Populates users table - data_base.sql

The VPL users can create an activity and add the "data_base.sql" file to the "execution files", and the "pre_vpl_run.sh" script showed below. The script will create and populate the database before each run. The "data_base.sql" file must be marked as "Files to keep when running".

```

#!/bin/bash
sqlite3 data_base.bin < data_base.sql

```

Code 3 pre_vpl_run.sh script

⁴ PHP Safe Password Hashing <https://www.php.net/manual/en/faq.passwords.php>

Database class

The DB class concentrates on the actions to manage and access the database. The class, not build to create objects, is a utility class with all its methods static. The methods are as follow:

- **get_connection()**. This method returns a connection to the app DB with a singleton design pattern. For each run, it always returns the same object avoiding multiple unneeded connections to the DB. To connect to other DBMS, change here the \$dsn variable.

```
<?php
class DB {
    private static $connection=null;
    public static function get_connection() {
        if(self::$connection === null){
            $dsn = 'sqlite:' . __DIR__ . '/../data_base.bin';
            self::$connection = new PDO($dsn);
            self::$connection->exec('PRAGMA foreign_keys = ON;');
            self::$connection->exec('PRAGMA encoding="UTF-8";');
            self::$connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        }
        return self::$connection;
    }
}
```

Code 4 DB::get_connection() - db.php

- **execute_sql(\$sql, \$parms=null)**. To simplify the access to the DB using PDO this method encapsulates the run of a SQL statement and returns a PDOStatement object to access the data or false if fails. **Important:** for security reasons the query and the data are separated to avoid SQL code injection. **Never create a SQL statement concatenating data and code.**

```
public static function execute_sql($sql, $parms=null) {
    $db = self::get_connection();
    $ints = $db->prepare ( $sql );
    if ($ints->execute($parms)) {
        return $ints;
    }
    return false;
}
```

Code 5 DB::execute_sql() - db.php

- **get_records(\$sql, \$parms=null)**. This method goes further by returning an array with the registers obtained by the SQL statement. Each register is an array where each pair of record field/value is an array key/value. This method is commonly used for SQL "select" statements that return a small number of registers.

```
public static function get_records($sql, $parms=null) {
    $ints = self::execute_sql($sql, $parms);
    if ($ints == false) {
        return [];
    }
    $ints->setFetchMode(PDO::FETCH_NAMED);
    return $ints->fetchAll();
}
```

Code 6 DB::get_records() - db.php

- **user_exists(\$account, \$pass)**. This method returns the registered user data that matches the account and password provided or returns false if not found. Notice the use of the md5 hash

function on the password to search in the table and the use of '?' instead of the plain data in the SQL statement avoiding the SQL code injection threat.

```
public static function user_exists($account, $pass) {
    $param = [$account, md5($pass)];
    $query = 'SELECT * FROM users WHERE account=? and password=?';
    $res = self::get_records($query, $param);
    if (count($res) != 1) {
        return false;
    } else {
        return $res[0];
    }
}
```

Code 7 DB::user_exists() - db.php

Security class

The roles of a user can be 'admin', 'staff', 'client', or 'non-authenticated'. The Security class provides the method **allowed_roles(\$roles)** that return true if the current user's role is included in the parameter \$roles. The parameter admits one or several roles separated by '|'. Also admits the roles groups 'all' and 'authenticated'.

This method is **key for app security**. Each action start must be limited using this method to the user's roles with the right to do it. A common newbie fail is the idea that not showing a link in the menu is enough to avoid the use of a function by other users.

```
<?php
include_once 'user.php';

class Security{
    public static function allowed_roles($roles) {
        $role = User::get_user_role();
        foreach (explode('|', $roles) as $allowed) {
            $allowed = strtolower(trim($allowed));
            if ($role == $allowed ) {
                return true;
            }
            if ($allowed == 'all') {
                return true;
            }
            if ($allowed == 'authenticated'
                && $role != 'non-authenticated') {
                return true;
            }
        }
        return false;
    }
}
```

Code 8 Security::allowed_roles() - security.php

User class

The user utility class groups the methods related to the current user and authentication. This class provides getting the current user, the current user role, user login, and logout.

The user authentication is done using the `$_SESSION` super global PHP variable of type array. This variable, unique for each browser, is saved and restored in each PHP execution, which means that the values saved in this variable remain from run to run.

The User class provides the following methods:

- **start_session()**. This method allows calling `session_start()` with no error if already called. The call to `session_start()` is required if using `$_SESSION` variable. To ensure the existence of the `$_SESSION` variable, this method can be called before accessing it.

```
<?php
include_once 'db.php';
class User{
    public static function start_session() {
        if (session_status() === PHP_SESSION_NONE) {
            session_start();
        }
    }
}
```

Code 9 User::start_session() - user.php

- **get_logged_user()**. This method returns the current user's data as an array, or false if no user is logged. The user's data is get from the `$_SESSION` variable.

```
public static function get_logged_user() {
    self::start_session();
    if (!isset($_SESSION['user'])) return false;
    return $_SESSION['user'];
}
```

Code 10 User::get_logged_user() - user.php

- **get_user_role()**. This method returns the authenticated user's role as a string or 'non-authenticated' if not authenticated.

```
private static $roles = array(
    1 => 'admin',
    2 => 'staff',
    3 => 'client'
);
public static function get_user_role() {
    $user = self::get_logged_user();
    if ($user == false || !isset(self::$roles[$user['role']])) {
        return 'non-authenticated';
    } else {
        return self::$roles[$user['role']];
    }
}
```

Code 11 User::get_user_role() - user.php

- **login(\$account, \$pass)**. If the account and password belong to a registered user, the method saves the user as authenticated and returns true else return false. The `$_SESSION['user']` array element saves the user's data so indicating that there is an authenticated user.
- **logout()**. This removes the element array `$_SESSION['user']` so indicating that there is no authenticated user.

```
public static function login($account, $pass) {
    self::start_session();
    $res = DB::user_exists($account, $pass);
}
```

```

    if ($res != false) {
        $_SESSION['user'] = $res;
        return true;
    }
    return false;
}

public static function logout() {
    self::start_session();
    unset($_SESSION['user']);
}
}

```

Code 12 User::login() and User::logout() - user.php

View class

The View class methods allow the construction of the HTML pages of the application that the browser shows. There is a method that generates the start of the page, another method that generates the end, and also a method that shows a message on the page. The use of this class is important to easily generate uniform pages in all parts of the application.

The View class provides the following methods:

- **text2html(\$text)**. This method takes a text string and returns it but replacing all codes interpretable as HTML in the text with HTML entities. The resulting string has only text interpretation. To avoid cross-scripting attacks, the use of this function is required when showing text on any app page.

```

<?php
include_once 'navigation.php';
include_once 'security.php';
class View{
    const APP_NAME = 'Web app example';
    public static function text2html($text) {
        return htmlentities($text, ENT_QUOTES, 'UTF-8');
    }
}

```

Code 13 View::text2html() - view.php

- **start_page(\$title, \$roles)**. This method shows the initial part of an app HTML page, including the navigation menu. The title is shown in the header of the page. The method before showing the page checks if the current user's role matches the required. If the check fails show a page with a message and stop the script. Including the \$role parameter in this method forces the programmer to indicate the roles that can do the action.

```

public static function start_page($title, $roles){
    if (! Security::allowed_roles($roles)) {
        $message = 'You are not granted to do this action';
        View::start_page('Problem', 'all');
        View::message($message);
        View::end_page();
        die();
    }
    $title = self::text2html($title);
    $html_head =
        "<!DOCTYPE html>"
        "<html>"
        "<head>"

```

```

        <meta charset=\"utf-8\">
        <link rel=\"stylesheet\" type=\"text/css\" href=\"style.css\">
        <title>{$title}</title>
    </head>
    ";
    $logo_src = '../image/logo.png';
    $html_body_start =
    "
        <body>
        <header>
            <img src='{$logo_src}'><h1>" .
                self::APP_NAME . " : {$title}</h1>
        </header>
    ";
    echo $html_head;
    echo $html_body_start;
    Navigation::show_navigation();
}

```

Code 14 View::start_page() - view.php

- **message(\$text)**. Shows a message on the page.
- **end_page()**. Generates the end of the page showing a standard footer.

```

public static function message($text) {
    $text = View::text2html($text);
    $html = "<div>{$text}</div>\n";
    echo $html;
}

public static function end_page() {
    $html =
        "
            <footer>Copyrighth FITPED, Contact here</footer>
        </body>
    </html>";
    echo $html;
}
}

```

Navigation class

The navigation class allows generating the navigation menu by calling its `show_navigation()` method. The menu with a private attribute contains the menu for each role. For each role and array with key/value, with the key being the text of the menu option and the value the link of the option. The menu options are added from left to right. If the array key is equal to `'--right--'`, the next menu options align right.

The `show_navigation()` method gets the current user's menu based on its role and generate the menu using ``, `` and `<a>` HTML tags. The menu options align right are get by adding the class `'menu-right'` to the corresponding `` tags.

```

<?php
include_once 'user.php';
class Navigation{
    private static $menu = array (
        'non-authenticated' => array(
            'Home' => 'index.php', '--right--' => '',
            'Login' => 'login.php'),
    );
}

```

```

        'admin' => array(
            'Home' => 'index.php', 'Users' => 'users.php',
            'Reports' => 'reports.php', 'Products' => 'products.php',
            '--right--' => '', 'Logout' => 'logout.php'),
        'staff' => array(
            'Home' => 'index.php', 'Products' => 'products.php',
            'Shipments' => 'shipments.php', '--right--' => '',
            'Logout' => 'logout.php'),
        'client' => array(
            'Home' => 'index.php', 'Products' => 'products.php',
            'Orders' => 'orders.php', 'Profile' => 'profile.php',
            '--right--' => '', 'Logout' => 'logout.php'),
    );
    public static function show_navigation() {
        $role = User::get_user_role();
        $role_menu = self::$menu[$role];
        $clase = '';
        $html = '<nav><ul>';
        foreach ( $role_menu as $menu_item => $link ) {
            if ($menu_item == '--right--') {
                $clase = "class='menu-right'";
                continue;
            }
            $menu_item = View::text2html($menu_item);
            $html .= "<li $clase><a href='$link'>$menu_item</a>";
        }
        $html .= '</ul></nav>';
        echo $html;
    }
}

```

Code 15 Navigation class - navigation.php

Home page

The classes created make it easy to create pages with a common look. Generating the 'Home page' requires a few lines of code.

```

<?php
include_once 'class/view.php';
View::start_page('Home page', 'all');
?>
    <h1>This is a Home page example</h1>
    <?php
View::end_page();

```

Login page

The login page starts by checking if the own form data has been sent by the browser to check the account and password. If the checks succeed, the check is done, and a page with a message is shown with the result. If no form is sent, the login form is generated and showed by the browser. Notice the request of the HTTP POST method in the form. Newbies may be confused with the behaviour of this page because, really, there are two pages inside this code. The code can generate the login form or, if form data is sent, do the login check and generate a page with the result. Notice that for each HTTP protocol request (link click or form submit), PHP runs the corresponding code and generates a response without further interaction with the browser.

```

<?php
include_once 'class/view.php';
include_once 'class/user.php';
include_once 'class/security.php';
if (isset($_POST['account']) && isset($_POST['password']))
    && Security::allowed_roles('non-authenticated') ) {
    $logged_in = User::login($_POST['account'], $_POST['password']);
    if ($logged_in) {
        $title = "Logged in";
        $text = "You are logged in user " . user::get_logged_user()['name'];
    } else {
        $title = "Login failed";
        $text = "Error: account or password mismatch";
    }
    View::start_page($title, 'all');
    View::message($text);
    View::end_page();
} else {
    View::start_page('Login', 'non-authenticated');
    ?>
    <form method="POST">
        <label>Account: <input type="text" name="account"></label> <br>
        <label>Pasword: <input type="password" name="password"></label> <br>
        <input type="submit" value="Accept">
    </form>
    <?php
    View::end_page();
}

```

Code 16 login.php

Logout page

The logout page is easier than the login page, and the main action is to call `User::logout()`. The rest of the code checks the user's role and showing the page result. Really this page and login page can be run with no role control due to the lack of negative effects.

```

<?php
include_once 'class/view.php';
include_once 'class/user.php';
include_once 'class/security.php';
if (Security::allowed_roles('authenticated')) {
    User::logout();
}
View::start_page('Log out', 'non-authenticated');
View::message('You are logged out');
View::end_page();

```

Code 17 logout.php

CSS

CSS is an important part of all web design. In this case, all pages generated by the call to `View::start_page()` will use a common CSS by including an external file `style.css` that is shown below.

The `style.css` file has different parts:

- **Navigation:** The navigation menu is get by changing the tags `` `` `<a>` inside that `<nav>`. The main property changes are for `<u>` are "list-style-type: none" and "overflow: hidden", for `` "float: left", and for `<a>` "display: block".

```

nav ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
  overflow: hidden;
  background-color: #222;
  border-radius: 8px;
  clear: both;
}

nav li {
  float: left;
  text-align: center;
}

.menu-right {
  float: right;
}

nav li a {
  color: white;
  display: block;
  padding: 16px 16px;
  text-decoration: none;
  border-radius: 8px;
}

nav li a:hover {
  background-color: #555;
  color: yellow;
}

```

- **Body, header, and h1:** Notice that the image in the header (the logo) floats to the right.

```

body {
  font-family: Verdana, Helvetica, Arial, sans-serif;
  background-color: #EEE;
  margin: 25px;
}

header img {
  float: right;
  width: 100px;
}

h1{
  color: #666;
  font-size: 1.25em;
}

```

- **Form.**

```

form {
  margin: 2em;
}

```



```
input {
  margin: 0.3em;
  border: 2px solid #079292;
  border-radius: 8px;
}

input[type=submit] {
  margin-left: 8em;
  margin-top: 1em;
}
```

- **Message class and footer.** The footer is shown fixed at the bottom right of the window by applying the properties "display: block", "position: absolute", "bottom: 0" and "right: 25px".

```
.message {
  padding: 16px 16px;
  margin: 1em auto 1em auto;
  background-color: #BBB;
  border-radius: 8px;
  text-align: center;
}

footer {
  display: block;
  position: absolute;
  bottom: 0;
  right: 25px;
  padding: 8px 8px;
  margin: 1em auto 1em auto;
  background-color: #BBB;
  border-radius: 8px;
  text-align: right;
}
```

Code 18 style.css

Source Code

The solution source code is available for download from the FITPED server. The source code organization must follow the structure shown in Fig. 7. The PHP scripts that generate pages are in the top directory, the PHP classes files are in the class subdirectory, and the images are in the image subdirectory.



Fig. 7 Source code structure

Create, Retrieve, Update, Delete, and List Users for a Web App in PHP

This task is in the area of web applications. More specifically, it is a PHP⁵ development of users accounts management system for a web application. This management system provides listing, adding, editing, deleting, and changing passwords of users' accounts by users with the admin role.

The goal is to build a users' accounts management system for a general-purpose web application. The development will not release a fully functional application and will not cover other related problems as the web page template, the user's authentication, and the menu system⁶. This tutorial will be used for describing and practice the basic concepts of Create, Retrieve, Update, and Delete (CRUD) registers in a database (DB) using the PHP and JavaScript programming languages and their application to the development of a users' accounts management system for a web application. The task includes PHP components, hypertext markup language (HTML) pages, forms, database access, forms checks, and asynchronous requests to the server (AJAX) in JavaScript. A users' accounts system for a web application is a good example covering all these features.

The students will develop the users' accounts system with the features of listing, adding, editing, deleting, and changing passwords of users' accounts aware of the authenticated user's role. The development will consider the existence of an already developed programmatic web page template, authentication system, and menu system. It has the goal that students get skills by developing CRUD in common data records and using some of the technologies involved in the development of web applications. On the other hand, the development of components commonly used in many web applications increases the possibility of applying the knowledge acquired in the future.

The application components to develop will be generic and ready to be adapted for a specific full application, allowing CRUD of different types of data records.

Recommended Number of Developers

The number of developers and the time to fulfil the task requirements vary based on the developer's knowledge of the technology and skills. It is assumed that the student has some knowledge or experience with HTML, databases, PHP, and JavaScript in web development.

The task is suitable for 1 or 2 junior developers with an estimated duration of about 12-16 man-hours.

Available Solutions

There are a great number of solutions to this type of problem on the internet. The proposed task here highlights readability, reusability, user-friendly, and security.

Requirements

Any integrated development environment (IDE) with support for PHP and JavaScript web development can be used to resolve the task. If a proper IDE is not available and the task will be

⁵ <https://www.php.net>

⁶ For more details, see the corresponding tutorial in this book.

developed in the context of a teaching environment using Moodle, the VPL plugin can be used. A modern web browser is also needed for development.

Functional Requirements

Main functional requirements:

- a) List of users.
- b) Add a new user.
- c) View a user profile.
- d) Edit a user profile.
- e) Delete a user account.
- f) Change a user password.

Non-functional Requirements

- a) PHP 7.2 or higher.
- b) Modern browser.

Use Cases Diagram

The diagram of use cases is the common one for this type of application component. There is an actor named Admin with six specific use cases. The other use cases were developed at another tutorial in this book.

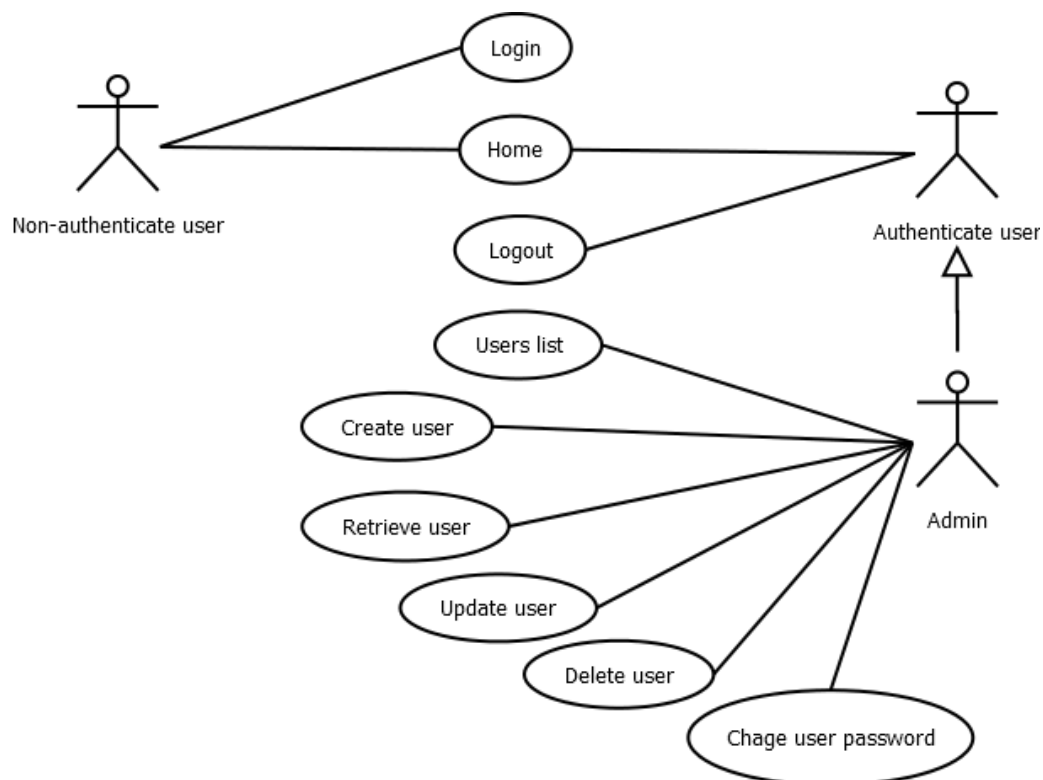


Fig. 8 Use cases

Application Design

Technology and Architecture Selection

The architecture of the application is mainly simple with a different server-generated page for each use case. Some pages include JavaScript help in the form validation, and one use case uses AJAX. The idea is to show students the basic web development used for a CRUD component. The use of frameworks or other modern technologies can accelerate the development, hiding what is happening on the server or browser side. With this approach, the browser shows HTML, sends forms data and URL requests to the server, and in several cases, the JavaScript running in the browser can help the user interface or interact with the server to do actions. The server runs the application by generating HTML pages for the browser requests, answering AJAX requests, and accessing the data storage.

Data Model

The component uses the database table named *users* showed below. The fields used are common in tables storing users' data for the intended purpose.

- **id.** Record identification auto-incremented.
- **account.** Account name.
- **password.** Hashed password to access the account
- **name.** User real name.
- **role.** The role is an integer with the following interpretation:
 - 1 => Admin
 - 2 => Staff
 - 3 => Client

These roles are enough for a proof of concept.

- **email.** User email.
- **address.** User full address.
- **mobile.** User mobile number.

users	
id	INTEGER
account	NVARCHAR(20)
password	NVARCHAR(32)
name	NVARCHAR(200)
role	INTEGER
email	NVARCHAR(200)
address	NVARCHAR(200)
mobile	NVARCHAR(200)

Fig. 9 Users DB table

User Interface

The page template is simple; every page has four zones: header, navigation, content, and footer. The header shows the name of the application, the page name, and the logo image on the right. The navigation contains the menu with a horizontal line of options: the home page, the user's role menu options, and the login or logout menu option. The content shows the page information. The footer shows the contact and copyright hints.



Fig. 10 Page template

The user iterations for reaching the different Admin use cases are centred on the "List of users" page that allows adding new users, view, edit and delete a selected user. The "View user" page also allows edit, delete, or change the password of a selected user.

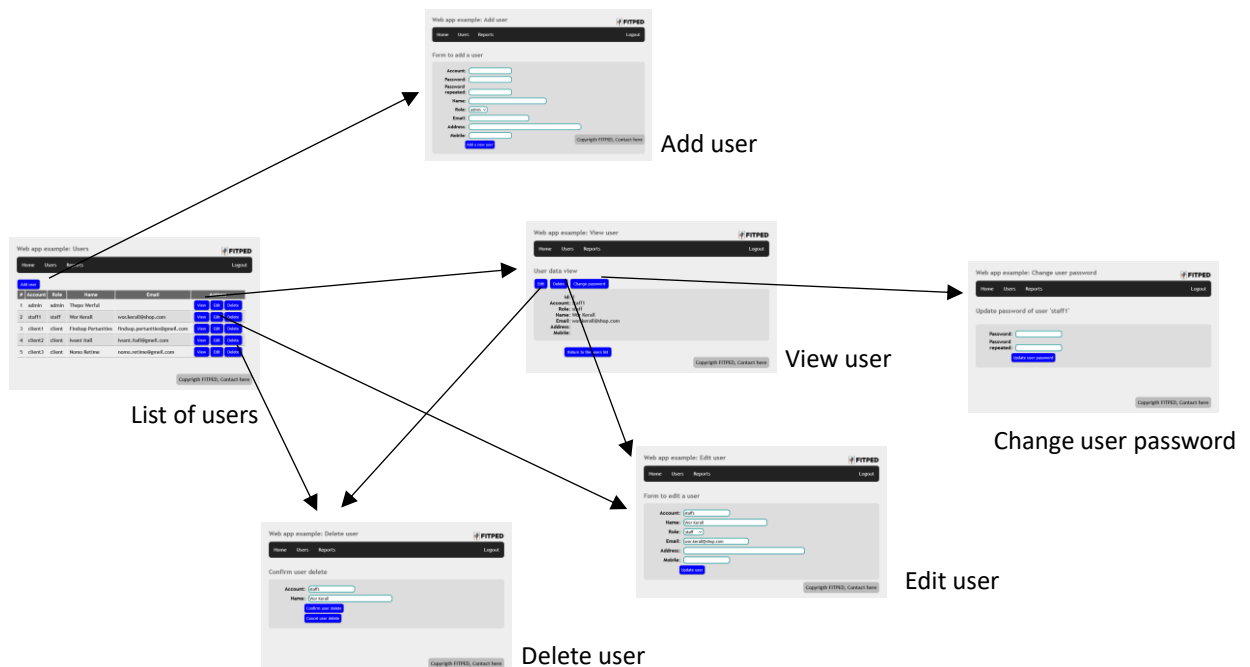


Fig. 11 Login page

Solution

Introduction

The next sections describe the development of the web application components, including the PHP-generated web pages, specific database queries, data validation, and AJAX requests. Following the idea that the content of the tutorial must be didactic and serve as a proof of concept, not a development with a result prepared to be used "as is", its design has prioritized the simplicity of the solution over the full functionality. Nevertheless, the safety of the generated components has been significantly taken into account. The description will detail the development of each featured source file justifying its need, describing how to use it and highlighting the relevant code. Alternatives to the chosen approach will also be indicated.

The first of the next sections establishes specific database access and data validation in PHP as a base for implementing related use cases. The following sections show the developments of use cases without JavaScript use, and finally, the last section will show the use of JavaScript to improve the user interface.

Specific database queries and server-side data validation

Each use case specific to the admin's role needs database queries that, to reuse and easy use, is better to encapsulate in methods with a proper name and a clear interface. The use cases need to get all users, save a new user, get a user by id, update a user register, delete a user register by id, and update a user password. Also, adding and updating a user needs to validate the size and format of the related fields. The resulted methods are located in the **User class**, but locating them in other classes is also valid.

The proposed User class methods are listed following:

- **User::get_all_users()**. This method returns all users registered in the system as an array. Each element in the array contains an array with each field/value of the user record as a key/value pair. Notice that this method is not appropriated for systems with a large number of users. In this case, the method may need filters and an output size limit.

```
public static function get_all_users() {
    return DB::get_records('SELECT * FROM users');
}
```

Code 19 User::get_all_users() - user.php

- **User::get_user(\$id)**. This method returns the user with the record id specified or false if not found. This method is used whenever one user's information is need based on their id.

```
public static function get_user($id) {
    $res = DB::get_records('SELECT * FROM users WHERE id = ?', [$id]);
    if (count($res) == 1) {
        return $res[0];
    } else {
        return false;
    }
}
```

Code 20 User::get_user(\$id) - user.php

- **User::check_user_data(\$data)**. This method checks if the data for adding or updating a user record is valid. The method returns true if the data is valid or a string with the problem found

if not. This type of method is needed to guarantee the format and limits of the data to be saved or updated. Some fields of data must fulfil some format for security reasons as the minimum size of the account name or the format of the password. Other fields must not surpass the maximum field size limit due to the generations of DB errors in some DBMS. **For security reasons, this type of method is always needed even when the limits are set on HTML input tags or JavaScript code is available to validate the format in the browser.** Notice that these barriers are effective for helping the common users to introduce the data in the correct format but are very easy to overcome for an attacker. When adding a new user, also method `User::check_user_password()` must be called.

```
public static function check_user_data($data){
    if (strlen($data['account']) < 6 || strlen($data['account']) > 20) {
        return "Account name too short or long, min 6 chars and max 20.";
    }
    if (strlen($data['name']) < 4 || strlen($data['name']) > 32) {
        return "User name too short or long, min 4 chars and max 32";
    }
    if (! in_array($data['role'], ['1', '2', '3'])) {
        return "Invalid role value.";
    }
    $regemail = '/^[^@ ]{3,}@[^@ \.]{3,}[^@ ]*\.[^@ \.]{2,}[^@ ]*$/' ;
    if ($data['email'] != '' &&
        preg_match($regemail, $data['email']) != 1 ) {
        return "Invalid user email value.";
    }
    if (strlen($data['email']) > 200 ) {
        return "User email too large, maximum 200 chars.";
    }
    if (strlen($data['address']) > 200 ) {
        return "User address too large, maximum 200 chars.";
    }
    if (strlen($data['mobile']) > 200 ) {
        return "User mobile too large, maximum 200 chars.";
    }
    return true;
}
```

Code 21 User::check_user_data(\$data) - user.php

- **User::check_user_password(\$data).** This method checks for the equals of the 'password' and the 'repeated password' and the minimum password size. The method returns true if the password is valid or a string with the problem found if not.

```
public static function check_user_password($data){
    if ($data['password'] != $data['passwordrep']) {
        return "Passwords mismatch.";
    }
    if (strlen($data['password']) < 8 ) {
        return "Password too short, minimum 8 chars.";
    }
    return true;
}
```

- **User::add_user(\$data).** This method tries to add a new user to the 'users' table. The method returns value "true" if adding succeeds; else returns a string containing a description of the problem found. To get a correct run, the caller must provide all the needed fields. The method checks and processes the data and execute the database query:

- Cleans the data by removing (trimming) unneeded spaces at the start and end of the fields.
- Calls to `check_user_data()` method.
- Updates 'password' field to save as its md5 hash result.
- Runs and checks the database query. The creation of a unique index with only the 'account' field grants that two users cannot have the same account name.

```
CREATE UNIQUE INDEX [indexaccount] on [users] ([account]);
```

```
public static function add_user($data) {
    if ( ($check = self::check_user_password($data)) !== true) {
        return $check;
    }
    $fields = ['account', 'password', 'name',
               'role', 'email', 'address', 'mobile'];
    $cleandata = [];
    foreach ($fields as $field) {
        $cleandata[$field] = trim($data[$field]);
    }
    if ( ($check = self::check_user_data($cleandata)) !== true) {
        return $check;
    }
    $cleandata['password'] = md5($cleandata['password']);
    $sql = "INSERT INTO users (" . implode(", ", $fields) . ") ";
    $sql .= "VALUES (: " . implode(", :", $fields) . ")";
    try {
        $res = DB::execute_sql($sql, $cleandata);
    } catch(Exception $e) {
        return "Account already exist";
    }
    if ($res->rowCount() == 1) {
        return true;
    }
    return "Unkown error";
}
```

Code 22 User::add_user(\$data) - user.php

- **User::update_user(\$data).** This method is similar to `add_user()` but without updating the password field. The user's password update is done in a specific method `User::update_user_password($data)`.

```
public static function update_user($data){
    $fields = ['account', 'name', 'role', 'email', 'address', 'mobile'];
    $cleandata = [];
    foreach ($fields as $field) {
        $cleandata[$field] = trim($data[$field]);
    }
    if ( ($check = self::check_user_data($cleandata)) !== true) {
        return $check;
    }
    $sqlset = '';
    foreach ($fields as $field) {
        if (strlen($sqlset) > 0) $sqlset .= ', ';
        $sqlset .= "$field = :$field";
    }
    $sql = "UPDATE users SET $sqlset WHERE id = :id";
    $cleandata['id'] = $data['id'];
    try {
        $res = DB::execute_sql($sql, $cleandata);
    }
```



```

    } catch(Exception $e) {
        return "Account already exist";
    }
    if ($res->rowCount() == 1) {
        return true;
    }
    return "User not found";
}

```

Code 23 User::update_user(\$data) - user.php

- **User::update_user_password(\$data).** This method updates a user's account password. The data parameter is an array containing the 'id', 'password', and 'passwordrep' keys with the corresponding values. If the method updates the password, then returns true, else it returns a string containing a description of the problem found.

```

public static function update_user_password($data){
    if ( ($check = self::check_user_password($data)) !== true) {
        return $check;
    }
    $cleandata = array ( 'id' => $data['id'],
                        'password' => md5($data['password']));
    $sql = "UPDATE users SET password = :password WHERE id = :id";
    try {
        $res = DB::execute_sql($sql, $cleandata);
    } catch(Exception $e) {
        return "Internal error";
    }
    if ($res->rowCount() == 1) {
        return true;
    }
    return "User not found";
}

```

- **User::delete_user(\$id).** This method deletes the user record with the indicated id. If deletes the user's account, then returns true else, returns a string containing a description of the problem found.

```

public static function delete_user($id){
    $sql = "DELETE FROM users WHERE id = :id";
    $parms = array('id' => $id);
    try {
        $res = DB::execute_sql($sql, $parms);
    } catch(Exception $e) {
        return "Internal error";
    }
    if ($res->rowCount() == 1) {
        return true;
    }
    return "User not found";
}

```

Code 24 delete_user(\$id) - user.php

List of users

This page shows the list of users as a data table. To generate the main content of the page (See Fig. 12), the program does the following steps:

- Show a link to "Add user."
- Get the data calling o User::get_all_users().
- Shows the table header
- For each user
 - Shows user's data as a table row. Notice the use of the View::text2html() function.
 - Shows actions on the user as links to corresponding pages. Notice that use query string in URL to pass the user's id to the page action.
- Shows the end of the table.

Web app example: Users


Home Users Reports Logout

Add user

#	Account	Role	Name	Email	Actions
1	admin	admin	Thepo Werful		View Edit Delete
2	staff1	staff	Wor Kerall	wor.kerall@shop.com	View Edit Delete
3	client1	client	Findsop Portunities	findsop.portunities@gmail.com	View Edit Delete
4	client2	client	Iwant Itall	iwant.itall@gmail.com	View Edit Delete
5	client3	client	Nomo Retime	nomo.retime@gmail.com	View Edit Delete

Copyright FITPED, Contact here

Fig. 12 Example of "List of users" page

```
<?php
include_once 'class/view.php';
View::start_page('Users', 'admin');
echo "<a href='adduser.php' class='button'>Add user</a>";
$users = User::get_all_users();
if (count($users) > 0) {
    $roles = User::get_roles();
    echo "<table><tbody>\n";
    echo "<tr><th>#</th><th>Account</th><th>Role</th><th>Name</th>";
    echo "<th>Email</th><th>Actions</th></tr>\n";
    $sec = 0;
    foreach ($users as $user) {
        $sec++;
        $id = $user['id'];
        echo "<tr>";
        echo "<td>$sec</td>";
        echo "<td>" . View::text2html($user['account']) . "</td>";
        echo "<td>" . $roles[$user['role']] . "</td>";
        echo "<td>" . View::text2html($user['name']) . "</td>";
        echo "<td>" . View::text2html($user['email']) . "</td>";
        echo "<td>";
        echo "<a href='viewuser.php?id=$id' class='button'>View</a> ";
    }
}
```

```

        echo "<a href='edituser.php?id=$id' class='button'>Edit</a> ";
        echo "<a href='deleteuser.php?id=$id' class='button'>Delete</a>";
        echo "</td>";
        echo "</tr>\n";
    }
    echo "</tbody></table>\n";
}
View::end_page();

```

Code 25 userslist.php

Add a new user

The 'adduser.php' program, like all other forms builder programs in this tutorial, does two different things: if it is detected that the form was sent, process it and shows the result; if not, generate the page with the form. Newbies may be confused with the behaviour because the program does two different processes and generates two different pages but not at the same time. The code in 'adduser.php' can generate the form to add a new user or, if form data is sent, add a new user and show the result as a page.

System behaviour:

- When a user clicks on a link to 'adduser.php' then the browser sends the URL to the server. The server runs 'adduser.php' the first time generating the form page. The browser shows the form page to the user (See Fig. 13).
- When the user fills the form and clicks to send the form, the browser sends the same URL and form data to the server. The server runs 'adduser.php' again and processes the sent data generating a page with a message. The browser shows the generated page (See Fig. 14). The browser sends the form data to the same URL because the form action attribute is empty. The browser sends the form data to the URL indicated in the form action attribute, but empty indicates the current URL.

Notice that for each HTTP protocol request (link click or form submit) PHP runs the corresponding code and generates a response without further interaction with the browser.

```

<?php
include_once 'class/view.php';
include_once 'class/user.php';
View::start_page('Add user', 'admin');
// Checks if form data received
$fields = ['account', 'password', 'passwordrep', 'name',
           'role', 'email', 'address', 'mobile'];
$formdata = true;
foreach ($fields as $field) {
    if (!isset($_POST[$field])) {
        $formdata = false;
        break;
    }
}
if ($formdata) {
    $res = User::add_user($_POST);
    if ($res === true) {
        View::message("User '{$_POST['account']}' created");
    } else {
        View::message("Error creating user: $res");
    }
    echo "<br><span class='field-name'></span>";
}


```

```

    echo "<a href='userslist.php' class='button'>Users list</a>";
} else {
    $roles = User::get_roles();
    $starttags = "<label><span class='field-name'>";
    $midtags = "</span>: <input type='";
    $endtags = "></label><br>\n";
    echo "<h1>Form to add a user</h1>";
    echo "<form method='POST'>\n";
    echo "${starttags}Account${midtags}'text' name='account'${endtags}";
    echo "${starttags>Password${midtags}'password' name='password'${endtags}";
    echo "${starttags>Password repeated${midtags}'password'";
    echo " name='passwordrep' ${endtags}";
    echo "${starttags}Name${midtags}'text' name='name' size='40' ${endtags}";
    echo "${starttags}Role</span>: <select name='role'>\n";
    foreach ($roles as $key => $name) {
        echo "<option value='$key'>$name</option>\n";
    }
    echo "</select${endtags}";
    echo "${starttags}Email${midtags}'text' name='email' size='30'${endtags}";
    echo "${starttags}Address${midtags}'text' name='address'";
    echo " size='60' ${endtags}";
    echo "${starttags}Mobile${midtags}'text' name='mobile' ${endtags}";
    echo "<span class='field-name'></span>";
    echo "<button type='submit'>Add a new user</button><br>\n";
    echo "</form>\n";
}
View::end_page();

```

Code 26 adduser.php

Web app example: Add user


Home Users Reports Logout

Form to add a user

Account:

Password:

Password repeated:

Name:

Role:

Email:

Address:

Mobile:

Add a new user

Copyright FITPED, Contact here

Fig. 13 Example of "Add user" form page



Fig. 14 Example of "Add user" result page

View user profile

The "View user" shows a page containing one user's data (See Fig. 15). The user to show comes with the request as the query string. PHP converts the URL query string to the `$_GET` superglobal variable that the programmer uses to get the user's id. The page also shows links to "edit", "delete", and "change password" actions on the user account. These links carry the user's id in the query string. Notice the use of the function `View::text2html` to avoid problems with especial HTML codes and cross-scripting threats.

```
<?php
include_once 'class/view.php';
include_once 'class/user.php';
View::start_page('View user', 'admin');
if (isset($_GET['id'])) {
    $id = $_GET['id'];
    $user = User::get_user($id);
    if ($user !== false) {
        $roles = User::get_roles();
        $userid = View::text2html($id);
        $account = View::text2html($user['account']);
        $role = $roles[$user['role']];
        $name = View::text2html($user['name']);
        $email = View::text2html($user['email']);
        $address = View::text2html($user['address']);
        $mobile = View::text2html($user['mobile']);
        echo "<h1>User data view</h1>";
        echo "<a href='edituser.php?id=$id' class='button'>Edit</a>\n";
        echo "<a href='deleteuser.php?id=$id' class='button'>Delete</a>\n";
        echo "<a href='changepassword.php?id=$id' class='button'>";
        echo "Change password</a><br>\n";
        echo "<div class='user-view'>\n";
        echo "<span class='field-name'>id</span>: $userid<br>\n";
        echo "<span class='field-name'>Account</span>: $account<br>\n";
        echo "<span class='field-name'>Role</span>: $role<br>\n";
        echo "<span class='field-name'>Name</span>: $name<br>\n";
        echo "<span class='field-name'>Email</span>: $email<br>\n";
        echo "<span class='field-name'>Address</span>: $address<br>\n";
        echo "<span class='field-name'>Mobile</span>: $mobile<br>\n";
        echo "</div>\n";
    }
}
```

```

echo "<br><span class='field-name'></span>";
echo "<a href='userslist.php' class='button'>Users list</a>";
View::end_page();

```

Code 27 viewuser.php

Web app example: View user

Home Users Reports Logout

User data view

Edit Delete Change password

id: 6
 Account: fitpedtest
 Role: client
 Name: Name of example user
 Email: fitpedtest@fitped.eu
 Address: The World
 Mobile: 1111111111

Users list

Copyright FITPED, Contact here

Fig. 15 Example of "View user" page

Edit user profile

The "Edit user" case is similar to the "Add user" case, but the password is not updated here. The form contains the saved user's data values to avoid the user introduces again know data (See Fig. 16). See the use of View::text2html function again to show the previous data in the form by populating the value attribute of the input tags. The password is not updated here because the system does not have the original password because only the hash of the password is saved. See file 'edituser.php' in the source code for more details.

Web app example: Edit user

Home Users Reports Logout

Form to edit a user

Account: client2
 Name: Iwant Itall
 Role: client
 Email: iwant.itall@gmail.com
 Address: Paris, France
 Mobile: 621123456

Update user

Copyright FITPED, Contact here

Fig. 16 Example of "Edit user" page

Delete user account

The "Delete user" buttons go to a delete confirm page (See Fig. 17). The page shows the user's account and name. The page also contains a form with the user identification hidden to use if the user confirms the deletion. Notice that the links generate HTTP get requests but the forms that change data in the system must use the HTTP post method. The delete process gets the user's id from the `$_POST` superglobal PHP variable. See file 'deleteuser.php' in the source code for more details.

Web app example: Delete user

Home Users Reports Logout

Confirm user delete

Account: client2
Role: client
Name: Iwant Itall

Confirm user delete
Cancel user delete

Copyrighth FITPED, Contact here

Fig. 17 Example of "Delete user" page

Change user password

The "Change user password" button only appears on the view user page and, when clicked, shows a form to change the password (See Fig. 18). This form asks to repeat the new password but does not ask to enter the previous password again, as seen on many websites nowadays. Notice that this form is for the admin role that does not need to know the password to change it, indeed requesting to enter the old password might be a big problem. See file 'changepassword.php' in the source code for more details.

Web app example: Change user password

Home Users Reports Logout

Update password of user 'client2'

Password:
Password repeated:

Update user password

Copyrighth FITPED, Contact here

Fig. 18 Example of "Change user password" page

Form validation in browser. The example of the add user from.

The in-browser form validation allows checking the characteristic of the data entered by the user before sending it to the server. The advantage of doing it in-browser is that the user can correct the problems found without reentering the data. The validation commonly shows the problem description near the input field (See

Fig. 19 19). The validation must be written in JavaScript and does not eliminate the need for data validation in the server because it is easy for a hacker to avoid the browser version.

Web app example: Add user

Home Users Reports Logout

Form to add a user

Account: Account too short. Minimum 6 chars

Password: Passwords mismatch

Password repeated:

Name: Name too short. Minimum 4 chars

Role:

Email: Email bad format

Address:

Mobile:

Add a new user

Copyright FITPED, Contact here

Fig. 19 Example of add user form showing validation warnings

The JavaScript code in this application is in the file "scripts.js" and must be load with the page. To load the code in all the application Web pages the method View::end_page() includes the HTML code `<script src='scripts.js'></script>`. The load at the end of the pages allows the code to access the page at load-time.

The form validation must be done before data submit and the event 'submit' that the form triggers before sending the data is perfect for this purpose. This event, in conjunction with the functions preventDefault() and submit(), allows controlling if send or not the data to the server. The preventDefault() method of an event avoids programmatically that the default action gets done. For example, it avoids sending the data to the server when the user clicks on the submit button or avoids following a link when the user clicks on it. The method submit() of a form in JavaScript submit the data to the server programmatically.

The validation has some auxiliary functions (See Code 28):

- **getWarningTag(name).** Returns the object associated with the tag that shows the validation error message for the input with the name indicated.
- **getValue(name).** Returns the value in the indicated input.

- **check(condition, w_tag, warning).** This function sets the message (warning) of validation error in the w_tag if the condition is true.

```
function getWarningTag(name) {
    return document.querySelector('#warning_' + name);
}
function getValue(name) {
    var tag = document.querySelector("input[name='" + name + "']");
    return tag.value;
}
function check(condition, w_tag, warning) {
    if (condition) {
        w_tag.innerHTML = warning;
        return true;
    }
    return false;
}
```

Code 28 Auxiliar functions for form validation - script.js

- **validateAddUser(e)** is the event handler for the 'submit' event of the form tag. The function removes the default submit action and checks the data for the specified limits and formats, and shows a warning for each problem found. Finally, if no problem is found for limits and formats, the function does the last check, searching if the account name already exists. If the data pass all checks, the form is submitted.

```
function validateAddUser(e) {
    e.preventDefault();
    var w_account = getWarningTag('account');
    var w_name = getWarningTag('name');
    var w_password = getWarningTag('password');
    // Cleans previous warning
    w_account.innerHTML = '';
    w_name.innerHTML = '';
    w_password.innerHTML = '';
    // Read input values
    var account = getValue('account');
    var name = getValue('name');
    var password = getValue('password');
    var passwordrep = getValue('passwordrep');
    no_send = false;
    no_send = check(account.length < 6, w_account,
        'Account too short. Minimum 6 chars') || no_send;
    no_send = check(account.length > 20, w_account,
        'Account too slarge. Maximum 20 chars') || no_send;
    no_send = check(name.length < 4, w_name,
        'Name too short. Minimum 4 chars') || no_send;
    no_send = check(name.length > 32, w_name,
        'Name too slarge. Maximum 32 chars') || no_send;
    no_send = check(password != passwordrep, w_password,
        'Passwords mismatch') || no_send;
    no_send = check(password.length < 8, w_password,
        'Passwords too short. Minimum 8 chars') || no_send;
    if ( ! no_send ) {
        var form = document.querySelector('#add-user-form');
        submitIfAccountNotExists(account, form);
    }
}
```

Code 29 validateAddUser() - scripts.js

- **submitIfAccountNotExists(account, form)** is the function that call `validateAddUser(e)` to submit the form data if account selected not exists. If the account is already in use, the function shows a warning near the account input field. To check if the account exists, the function consults the server making an AJAX request. In this case, the function sends the account name to check in the URL query string to the `ajaxaccountexists.php` server script and gets the answer as JSON data.

```
function submitIfAccountNotExists(account, form) {
    var request= new XMLHttpRequest();
    request.onreadystatechange = function() {
        if(this.readyState == XMLHttpRequest.DONE && this.status == 200) {
            var res= JSON.parse(this.responseText);
            if(res.exist === false) {
                form.submit();
            } else {
                getWarningTag('account').innerHTML = 'Account name already exists';
            }
        }
    };
    queryString = '?account=' + encodeURIComponent(account);
    request.open("get", "ajax/ajaxaccountexists.php" + queryString);
    request.send();
}
```

Code 30 submitIfAccountNotExists(account, form) – scripts.js

The `ajaxaccountexists.php` server script is the server-side of the AJAX request (See Code 31). This script gets the account name from the `$_GET` superglobal variable, consults the database, and answers the request as an object sent in JSON format. Notice that this PHP script shows plain text (JSON object) and not an HTML page. The script also checks for the user's role to ensure access limitations.

```
<?php
include_once '../class/security.php';
include_once '../class/db.php';
if (Security::allowed_roles('admin') && isset($_GET['account'])) {
    $sql = "SELECT id FROM users WHERE account = ?";
    $parms = [ $_GET['account'] ];
    $result = DB::get_records($sql, $parms);
    $answer = new stdClass();
    $answer->exist = count($result) == 1;
    echo json_encode($answer);
}
```

Code 31 ajaxaccountexists.php

Delete user account with Ajax

The AJAX requests can have many uses. Another example here is deleting a user account at the "List of users" page without changing of page. The "delete" links in the "List of users" page are changed to launch a request to the server by JavaScript instead of changing the page (See Fig. 20).

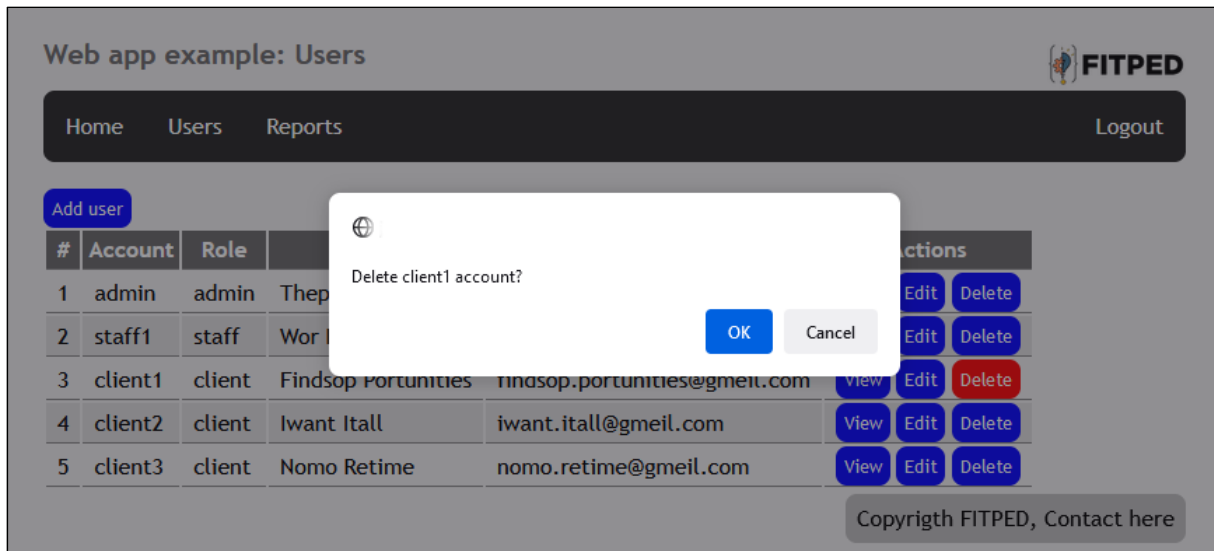


Fig. 20 Example of deleting a user's account with AJAX

- **deleteAjax(e)** is the JavaScript function launched when the user clicks on any of the "delete" links in the page "List of users". This function uses the event parameter to know the link where the user clicks (e.target). The HTML page also contains information to help the function to do its job. The delete link tags contain data with the user's id to delete. The account name is in an HTML tag with the id attribute named 'account + id', and the 'tr' tag of the user in the table is also identified by a 'row + id' (See Code 32). The function asks for delete confirmation, and, if accepted, the function sends the user's id as a JSON object in the HTTP post method payload. The answer is also received as JSON data. If the server deletes the user, then the function removes the row with the user's information from the table.

```
$id = $user['id'];
echo "<tr id='row$id'>";
echo "<td>$sec</td>";
echo "<td id='account$id'>" . View::text2html($user['account']) . "</td>";
echo "<td>" . $roles[$user['role']] . "</td>";
echo "<td>" . View::text2html($user['name']) . "</td>";
echo "<td>" . View::text2html($user['email']) . "</td>";
echo "<td>";
echo "<a href='viewuser.php?id=$id' class='button'>View</a> ";
echo "<a href='edituser.php?id=$id' class='button'>Edit</a> ";
echo "<a data-userid='$id' class='button delete-ajax'>Delete</a>";
echo "</td>";
echo "</tr>\n";
```

Code 32 Partial userslist.php file

```
function deleteAjax(e) {
    e.preventDefault();
    var id = e.target.dataset.userid;
    var account = document.querySelector('#account' + id).innerHTML;
    if (!confirm("Delete " + account + " account?")) {
        return;
    }
    var request = new XMLHttpRequest();
    request.onreadystatechange = function() {
        if(this.readyState == XMLHttpRequest.DONE && this.status == 200) {
            var res = JSON.parse(this.responseText);
            if(res.deleted === true) {
                var tr = document.querySelector('#row' + id);
            }
        }
    };
    request.open('POST', 'delete-user.php', true);
    request.setRequestHeader('Content-Type', 'application/json');
    request.send(JSON.stringify({id: id}));
}
```

```

        tr.parent.remove(tr);
    }
}
};
request.open("post", "ajax/ajaxdeleteuser.php");
request.send(JSON.stringify({'id': id}));
}

```

Code 33 deleteAjax(e) - scripts.js

The **ajaxdelete.php** server script (See Code 34) is the server-side of the delete request and is similar to **ajaxaccountexists.php**. This script gets the user's id to delete from a JSON object in the request payload, delete the user account, and answers the request with an object sent in JSON format.

```

<?php
include_once '../class/security.php';
include_once '../class/user.php';
if (Security::allowed_roles('admin')) {
    $jsondata = file_get_contents("php://input"); // Read payload
    $data = json_decode($jsondata);
    $answer = new stdClass();
    $answer->deleted = User::delete_user($data->id);
    echo json_encode($answer);
}

```

Code 34 ajaxdeleteuser.php

The event handlers need to be set for specific tags and event names. An anonymous function at the end of "scripts.js" sets the handlers by searching for the specific tags and calling to **addEventListener** tag method.

```

(function() {
    var form = document.querySelector('#add-user-form');
    if (form) {
        form.addEventListener('submit', validateAddUser);
    }
    var ajaxs = document.querySelectorAll('.delete-ajax');
    ajaxs.forEach(function(aTag) {
        aTag.addEventListener('click', deleteAjax);
    });
})();

```

Code 35 The anonymous function that sets the events handlers - scripts.js

Source Code

The solutions source code is available for download from the FITPED server. The tutorial has two versions of the source code: one of PHP and the other of PHP with JavaScript. The source code organization follows the structure shown in Fig. 21. The PHP scripts that generate pages are in the top directory, the PHP classes files are in the class subdirectory, the AJAX server-side code scripts are in the ajax subdirectory, and the images are in the image subdirectory.

Fig. 21 Source code structure



Chat in VueJS

This task is suitable for the area of web applications. More specifically, the development of the frontend using JavaScript framework Vue.js⁷.

Create frontend part for web application - internet chat. The application will not cover any unresolved problem but rather will be used for demonstration and practice the basic concepts of the Vue.js framework and the development of frontend applications in general. This includes component design and communication, routing, token authorization and management, operations over local storage and HTTP requests. The chat application is a good example covering all these features.

The main task of the student is the implementation of the client part (or user interface – UI) according to the following assignments using the Vue.js framework. The application will communicate with the server via the REST API interface, which is already prepared and described in the `/server` folder (see Solution section below).

The application will allow user registration, login, room creation and chatting. A more detailed description of the function is also given below.

Recommended Number of Developers

It is assumed that the student has some knowledge or experience with programming (ideally with JavaScript or other scripting languages). This document will not cover the general basics of programming or JavaScript syntax.

The project is suitable for one junior developer. The estimated duration is about 14-18 hours.

Available Solutions

Basically, there are many simple applications used for communication in chat rooms. There are countless such applications available, and it makes no sense to name them here.

Requirements

Any reasonable **text editor** can be used to develop the application, but I highly recommend Visual Code⁸, which is very popular (and free). It will be necessary to have **NodeJS**⁹ runtime environment installed locally on the PC. After installation, it should be possible to list the `node` and `npm` versions in the terminal (commands: `node -version` and `npm --version`). A modern web **browser** (Google Chrome, Firefox or similar) and favourite **terminal** will also be very useful in the development.

Functional Requirements

Main functional requirements:

- a) The user can create a new account.
- b) The user can log in with credentials.

⁷ <https://vuejs.org/>

⁸ <https://code.visualstudio.com/>

⁹ <https://nodejs.org/en/>

- c) The browser remembers the logged-in user and automatically logs him in after reloading the application.
- d) The user can see a list of all chat rooms.
- e) The user can filter rooms by attribute (title or description).
- f) The user is able to create a new room.
- g) The user can enter the room and view messages which are regularly updated.
- h) The user can see a regularly updated list of room users.
- i) The user can send short messages visible to all people in the room.
- j) The user can leave the room and visit another one.

Non-functional Requirements

- a) Vue.js 2.x. – JavaScript or TypeScript (if someone prefers).

Use Case Diagram

The use case diagram is not necessary for such a simple application. There is only one actor in the application. He communicates directly with the system. The list of individual use cases is given above.

Application Design

Technology and Architecture Selection

The architecture of the application is designed in a modern way as a single-page application. SPA is a web application with one HTML page, which is initialized once, and only the necessary content is changed during further interactions. Almost the entire application runs directly in the client browser, while the server is mostly used for authentication and as a source or storage of data.

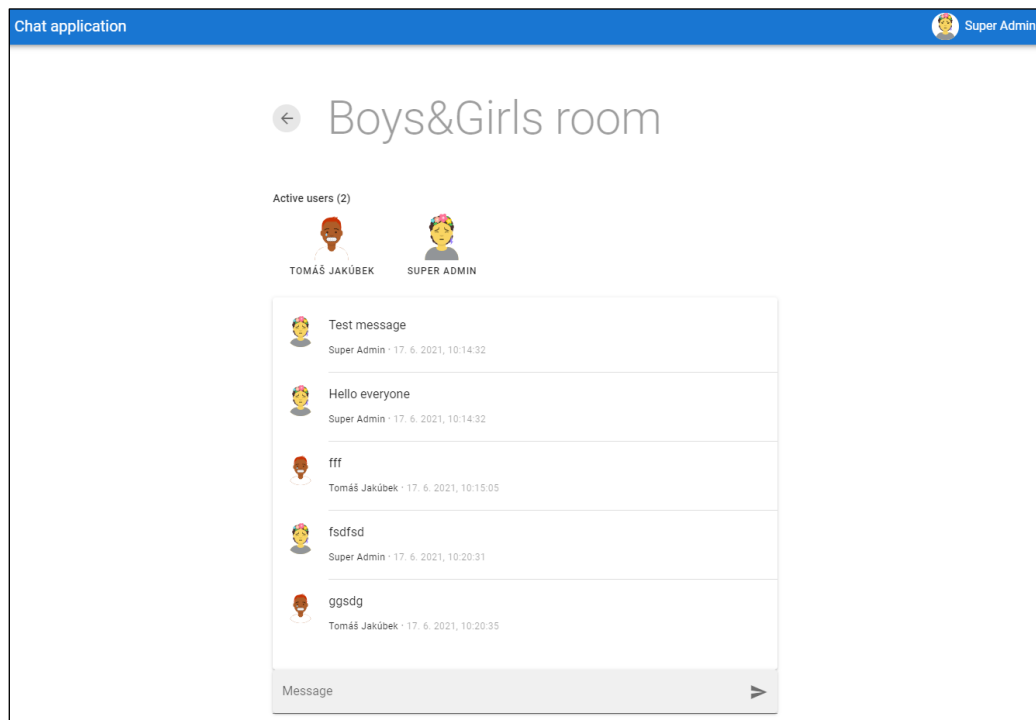
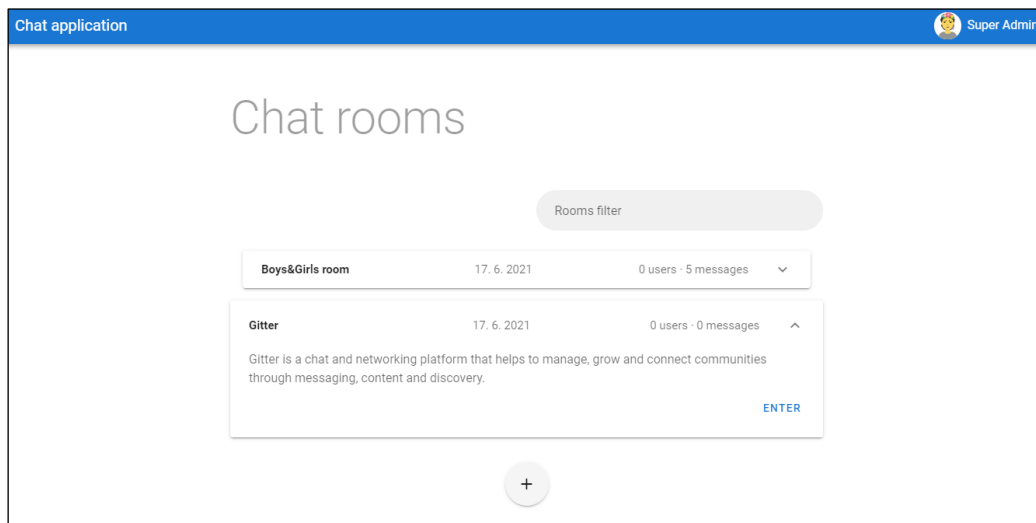
Compared to traditional server-oriented applications, the main difference is the way data is received and sent after the initial HTTP request. Single-page applications use AJAX (Asynchronous JavaScript And XML) to transfer data between the server and the client, which are usually in JSON format. This means that the moment the data reaches the client, the client partially re-renders the HTML page without having to refresh the entire page. That is why the entire HTML code is not fetched every time, and this is reflected in the speed of SPA applications.

Vue.js was chosen as the SPA framework (there are alternatives like React¹⁰ or Angular¹¹). Vue.js is a community-managed framework with a very fast learning curve and constantly growing attention of new developers. It has great documentation, many libraries, and tools, so it is a very suitable choice for students or beginners in general.

¹⁰ <https://reactjs.org/>

¹¹ <https://angular.io/>

User Interfaces



The image displays two distinct user interface forms side-by-side. The left form, titled 'Create new room', includes a sub-header 'You will be the admin of the new room.' and two text input fields labeled 'Room title' and 'Room description'. At the bottom, it features two buttons: 'CANCEL' in red and 'CREATE' in grey. The right form, titled 'Register', contains several fields: an 'Email' field with the value 'some@' and a red error message 'E-mail must be valid' below it; a 'Password' field with the value 'aaa' and a toggle icon; 'First name' (value 'anonym') and 'Last name' (value 'Foo') fields; and a gender selection section with radio buttons for 'Male', 'Female', and 'Other' (which is selected). A 'REGISTER' button is located at the bottom right of this form.

Fig. 22 Interface definition

Solution

Introduction

The following text describes a step-by-step tutorial focused on creating a frontend web application in the Vue.js framework. The basic concepts of this framework, as well as the main features of the application, will be covered. However, it is practically impossible to cover everything in detail. Therefore, in the text, you will find links to the official documentation of individual tools or mentioned features. These links are very useful if you don't understand something or want to know more. Also, some repeating parts of the code are intentionally omitted to shorten and maintain the overall readability of the text. But don't worry! The complete solution can be found in the attached GIT repository, where you can always take a look if something is not clear and also for inspiration. As this is just a simple demo application, some styling stuff may not be fine-tuned in detail and covered in the following text. The purpose of the text is to introduce the Vue.js framework, and pixel-perfect styling of the application is a secondary task in this case. Therefore, I leave the door open to your own initiative in this direction. I wish you good luck.

Clone repository

In the beginning, it is necessary to clone the GIT repository (link in the next section). The cloned project contains:

- `server` folder with the necessary information in the `readme.md` file. It is important to follow the instructions to install and run the server locally. After starting the server, it is also possible to view the Swagger¹² documentation. This documentation clearly displays the server API, and the programmer can see what endpoints are available in the API, what individual data endpoints expect or return in the response. Again, a description of how to view this documentation can be found in the `readme.md` file.

In short, it is a simple NodeJS server that provides a REST API between the created frontend application and the "database". For simplicity, the server does not connect to any real database. But instead, it stores all data in memory and forgets them when turned down. Such a setting does not require any other technical requirements necessary for your development (database engine, docker, permissions setup etc.) and shares the same

¹² <https://swagger.io/>

requirements as the frontend application. Also, the implementation of the server is not the subject of this task.

- `client_solved` folder with a complete implementation of the frontend, which the participant should obtain by elaborating the following sections.

Create Vue.js project using Vue CLI

Vue CLI¹³ is a full system for rapid Vue.js development, providing tools for interactive project scaffolding and many other features. To install Vue CLI globally on a PC, please run:

```
npm install -g @vue/cli
```

Please, make sure then that `vue --version` returns the installed version. Enter the root directory of cloned repo and run `vue create client` to create a new Vue.js project called `client`. Then select the default preset (Vue 2, babel, eslint) and press enter. After a few seconds (sometimes a few minutes), a new folder called `client` is initialized with the default (*Hello World!*) Vue.js project. And that's it! You just set up your (probably first) Vue.js project using Vue CLI.

Project directory description

The folder that was created by Vue CLI contains the following files and subfolders:

- `/node_modules` – the folder where all installed dependencies live,
- `/public` – favicon and one and only HTML page in the whole SPA project - `index.html`,
- `/src` – source code divided into components,
- `babel.config.js` – webpack¹⁴ configuration,
- `package.json` & `package-lock.json` – project description with a list of dependencies, useful dev scripts (`server`, `build`, `lint`) and eslint¹⁵ configuration,
- `README.md` – description of useful scripts.

Install Vuetify

Vuetify¹⁶ is a Vue UI library with beautifully handcrafted material components. It is an open-source project for building user interfaces for web and mobile applications. Building a friendly application interface with a great user experience is a skill that requires practice and knowledge. While Vuetify won't make you a skilled UX practitioner overnight, it will help provide a solid start to those who are new in this area.¹⁷

Within this assignment, we will use Vuetify as the UI library. Alternatively, if you don't like Vuetify, there are a lot of¹⁸ other UI libraries for Vue.js. If you are a CSS master, you can also continue without a UI library and write your own styles.

¹³ <https://cli.vuejs.org/>

¹⁴ <https://webpack.js.org/>

¹⁵ <https://eslint.org/>

¹⁶ <https://vuetifyjs.com/en/>

¹⁷ <https://www.sitepoint.com/get-started-vuetify/>

¹⁸ <https://athemes.com/collections/vue-ui-component-libraries/> or <https://www.codeinwp.com/blog/vue-ui-component-libraries/>

To instal Vuetify, we will use the Vue CLI plugin (in the root directory):

```
vue add vuetify
```

Then select `default` recommended preset and wait for installation. Great, you just loaded the preconfigured Vuetify into the project.

Installation of the necessary dependencies

When developing the application, we will need the following libraries:

- `vue-router`¹⁹ – router for Vue.js,
- `axios`²⁰ – promise based HTTP client.

To install them, we use a npm package manager²¹ and its command `npm install` (make sure you run the command in the `/client` folder – where `package.json` is located):

```
npm install vue-router axios
```

After the installation is complete, these dependencies must appear in `package.json` in the `dependencies` section.

Vue.js documentation

Vue.js has excellent documentation at <https://vuejs.org/v2/guide/>. If you come across any problem or additional question in the process, you will probably find information here. You can also try to use other dev pages like <https://stackoverflow.com/>, etc.

Start development mode

To start the application in development mode, run the following command:

```
npm run serve
```

If successful, we should see the *Hello world* application when we open <http://localhost:8080> in the browser. If necessary, we can change the default port (8080) in

`package.json>scripts>serve` script using the optional `--port` parameter (e.g. `vue-cli-service serve --port 1234`). Development mode includes hot-reloading, so every time you save a file, the code is automatically recompiled in the background, and the application is reloaded.

Remove Hello world template

Since we don't need the default *Hello world* template, it's a good idea to remove it. To do so, we must:

- delete `/src/components/HelloWorld.vue` file (there should be an error)

¹⁹ <https://router.vuejs.org/>

²⁰ <https://github.com/axios/axios>

²¹ <https://www.npmjs.com/>

- `/src/App.vue` requires `HelloWorld` so we need to delete three references there – 1 in `template` and 2 in `script` section (`import` statement and `components` property in Vue instance)

Now the application should start working again, and we should see the Vuetify logo in the default header at the top.

Vue single file component

In the previous step, you first encountered the Vue component. The Vue component is usually defined in one file (with `.vue` extension) called single file component²² with three main sections - `template`, `script`, and `style`:

- `template` – a template that is recompiled into HTML code. This section is mandatory.
- `script` – this section defines a Vue instance (component) that is exported so that it can be used elsewhere. This section is optional.
- `style` – CSS styles are defined in this section. These styles are applied globally by default. This behaviour can be undermined by the defined `scoped` attribute, which will scope the styles to this component. It is also possible to use any CSS preprocessor (it is necessary to install it first²³) and use a more convenient CSS syntax. This section is also optional in `.vue` files.

The Vue instance

Vue object lives in `script` section in a single file (`.vue`). When you create a Vue instance, you pass in an options object. It is a regular vanilla JS object with several reserved keys (e.g., `data`, `methods`, `computed`, `components`, etc.) and behaviour. We will see later what *options* are being talked about.

SPA development in general

When developing frontend applications, the individual code is divided into separate components. These components communicate with each other (they move data from parent to child or vice versa), and it is advisable to keep them reasonably large so that each component solves one specific behaviour or just displays a small part of the HTML page (so it is not good to implement the whole app in one component). Medium-sized applications contain dozens of smaller components that are reusable in many places and configured via the input interface (as will be mentioned later). During the application life cycle, components are created and destroyed (or hidden). However, in each SPA frontend framework, there is a "main" component (ancestor of all others), which is always present and loaded first.

In this case, it is the `App.vue` component. The `App` component is the first to load when the application starts. See the `main.js` file in which the Vue application (root Vue instance) is created (`new Vue ({...})`) and rendered into an HTML DOM element with the `#app` id.

²² <https://vuejs.org/v2/guide/single-file-components.html>

²³ <https://cli.vuejs.org/guide/css.html>

Creating AppBar component

The App component defines the top bar by default (see `template`). It is more optimal to move this bar (with all relevant stuff) to our separate component `AppBar`. This component will display a top bar (navbar) with information about the logged user and buttons for login, logout, and register.

To do so, it is necessary to create a new file in the `/src/components` folder and name it `AppBar.vue`. We will add the following code in the new component:

```
// AppBar.vue

<template>
  <v-app-bar app dense color="primary" dark>
    <v-toolbar-title>Chat application</v-toolbar-title>
    <v-spacer />
    <div>
      <v-btn outlined>
        Logout
      </v-btn>
    </div>
    <div>
      <v-btn outlined>
        Login
      </v-btn>
      <v-btn outlined>
        Registration
      </v-btn>
    </div>
  </v-app-bar>
</template>

<script>
export default {};
</script>
```

We have just defined the template of our first `AppBar` component using Vuetify built-in UI components (elements/components starting with the prefix "v-" come from Vuetify). For a list of all Vuetify components, see <https://vuetifyjs.com/en/components/>. As you can see, third-party components, common HTML elements or our own components are mixed quite normally.

We do not use this component anywhere yet. We want to use it in the `App` component. Therefore, we go back to the `App` component, remove everything between `<v-app-bar>...</v-app-bar>` (including), and import the newly created component instead. To do so, we will use ES6 import at the top of the `script` section. Subsequently, we must register this component to the local scope of the `App` component (components can also be registered globally²⁴, but this is not best practice). Therefore, we modify the options of the `Vue` instance (JS object mentioned earlier) and add `components` part to this object. The final version of the `App` component is:

```
// App.vue

<template>
  <v-app>
    <app-bar/>
```

²⁴ <https://vuejs.org/v2/guide/components-registration.html>

```
<v-main>
  </v-main>
</v-app>
</template>

<script>
import AppBar from "../components/AppBar.vue"

export default {
  name: 'App',

  components: {
    AppBar
  },

  data: () => ({
    //
  }),
};
</script>
```

For more detailed information about components, please follow official documentation at <https://vuejs.org/v2/guide/components.html>.

Data and props in general

Each application works with data. In general, this data represents business logic, and usually, it is bind to nice templates. In the Vue.js framework, they are stored in the `data` property of the given instance. The `data` property is (in this case) a function that returns an object representing our data. The component that defines the data can do anything (change, remove etc.) with it (this data is accessible both in the component's template (directly without `this` reference – just `someData`) and in the JS via Vue.js magic - shortcut `this.someData` and not `this.data.someData`). We'll see it in action in a few moments.

In practice, some data, useful to the entire application, are usually stored outside of components, in the so-called store (for Vue.js applications most often Vuex²⁵). Vuex creates a global state and provides management around this state. Individual components have easy access to it and can mutate state in one place. As this is a more advanced approach, this tutorial will skip it.

For the components to be able to communicate with each other, the props concept is used in Vue.js. Props resemble the standard attributes of HTML elements. Each parent component can pass data in any format (string, number, array, function etc.) to the child component via props. The child component can pull them out of the props and use them. If the child component wants to pass information to the parent component, it must either trigger (emit) a custom event or call the passed callback. We'll see props in action in a few moments, but for more information, including simple examples, I highly recommend reading the documentation²⁶.

²⁵ <https://vuex.vuejs.org/>

²⁶ <https://vuejs.org/v2/guide/components-props.html>

Props and directives

Let's add a user to our application. The user will be stored in the data of the main component, which will distribute it further, e.g. to the `AppBar` component.

```
// App.vue

<template>
  <v-app>
    <app-bar :user="user" />
    <v-main> </v-main>
  </v-app>
</template>

<script>
...
  data: () => {
    return {
      user: {
        name: "Tomas"
      },
    };
  },
...
</script>
```

```
// AppBar.vue

<template>
  ...
  Hello {{ user.name }}
  ...
</template>

<script>
export default {
  props: {
    user: {
      type: Object,
      required: false,
    },
  },
};
</script>
```

What happened? We added mocked `user` to the data with `name` attribute and passed whole `user` object via props to `AppBar` component. The `AppBar` component has a defined interface (optional `props` key in the options of every Vue instance) that expects an optional prop named `user`, which has type `Object` (a user is an object because it has several properties such as first name, last name, id, etc.). There is also a shorthand notation of props (e.g. simple array of strings), but the presented declaration gives us more control and overview of how data is moved between components.

We used a special syntax (colon), which is the Vue.js directive, to pass `user` prop. This is shorthand for `v-bind` directive. We are saying, please, bind this attribute with some JS data. If we did not use the `v-bind` directive (and write just `user="user"`), we would pass the string "user" to the `AppBar` component. However, when we use `v-bind`, Vue.js will start interpreting the right side as

JS code. `v-bind` is just one of many Vue.js directives. We will also use more of them later, and the whole list (including examples) can be found at <https://vuejs.org/v2/guide/syntax.html>.

Also, in the `AppBar` component, we can see how the data from the Vue.js instance is used in the template. Special “Mustache” syntax is used for this task (double curly braces), and thanks to Vue, we have all props (as well as data, methods, computed properties etc.) automatically available in the template, and we can render their value or directly call some method.

We should now see the username displayed in the top bar next to three buttons. These buttons are always displayed regardless of whether the user is logged in or not. It would be strange if, for example, an unregistered user had the opportunity to log out, etc. Therefore, we need to hide the buttons in the template according to whether the user exists. To do this, we will use another directive `v-if`, which is used for conditional rendering of elements. Elements that do not meet the condition defined in `v-if` are completely removed (destroyed) from the DOM (on the contrary, another directive `v-show` only hides such elements using CSS, and thus these elements are not destroyed what is required in some cases).

```
// AppBar.vue

<template>
  ...
  <div v-if="user != null">
    Hello {{ user.name }}
    <v-btn outlined>
      Logout
    </v-btn>
  </div>
  <div v-else>
    <v-btn outlined>
      Login
    </v-btn>
    <v-btn outlined>
      Registration
    </v-btn>
  </div>
  ...
</template>
```

In the above code, we made the rendering of `div` elements conditional on `user != null`. The first `div` with the logout button is rendered if the `user` is defined, the second `div` with the login and register buttons is rendered otherwise (`v-else` directive).

Now let's implement logout logic. After the user logs off, we need to remove the user from the data in the `App` component. But how to do it when the logout button is in a different component than the original source of data? Simply! We will use props again, but this time the communication flow will have the opposite direction (from the child component to the parent component).

```
// App.vue

<template>
  <v-app>
    <app-bar :user="user" @logout="doLogout" />
    <v-main> </v-main>
  </v-app>
</template>
```

```

<script>
...
data: () => {
  return {
    user: {
      name: "Tomas",
    },
  };
},
methods: {
  doLogout() {
    this.user = null;
  },
},
...
</script>

```

```

// AppBar.vue

<template>
  <div v-if="user != null">
    Hello {{ user.name }}
    <v-btn outlined @click="logout">
      Logout
    </v-btn>
  </div>
</template>

<script>
...
props: {
  user: {
    type: Object,
    required: false,
  },
},
methods: {
  logout() {
    this.$emit("logout");
  },
},
...
</script>

```

Let's start with AppBar component. We added another special directive `v-on` (abbreviated to `@` only) to the logout button. We can use this directive to listen to DOM events²⁷ and run some JavaScript when they're triggered. When a user clicks on `v-btn` component from Vuetify, it creates `click` event and emits it upwards. Using the `v-on` directive, we subscribe (or listen) to this event, and when that happens, we call our `logout` function declared in `methods` (another part of Vue instance options, in which all functions of the component are defined).

Now we want to do the same here. Emit a new custom event so that the component that is above the AppBar component can listen to. To do this, we use the special function `$emit`, which is

²⁷ https://www.w3schools.com/jsref/dom_obj_event.asp

available in each component, and emit a custom event called `logout` (the name is completely arbitrary). `App` component listens to `logout` event and call method, which sets `user` to `null`. Great, now we are able to "logout" the user and adapt the UI accordingly.

Adding router

Cool, we've mastered the basic concept of props, data binding, directives, and now we can look at adding a router to our application. The router is very useful if there are several subpages (so-called routes) in the application. We already installed the `vue-router` using `npm` package manager at the beginning, now we are going to use it.

To have a reasonable project structure, we should create a new folder called `/router` (in `/src` directory) for the router and all route views. In the new `/src/router` directory, we create `router.js` file with the following content:

```
// router.js

import Vue from "vue";
import VueRouter from "vue-router";

Vue.use(VueRouter);

const routes = [];

const router = new VueRouter({
  routes,
  mode: "history",
});

export default router;
```

```
// main.js

import Vue from "vue";
import App from "./App.vue";
import vuetify from "./plugins/vuetify";
import router from "./router/router";

Vue.config.productionTip = false;

new Vue({
  vuetify,
  router,
  render: (h) => h(App),
}).$mount("#app");
```

In the given code, we register the `vue-router` plugin to the `Vue` ecosystem and create a new instance of the router. Then we import this instance into `main.js` and register it in the application (root `Vue` instance).

`routes` array is still empty; let's fill it! The route is a separate page that the router renders when the URL matches its path. Let's create a route component for login and registration.

Login and register forms

First, in the `/src/router` directory, create a new directory for all routes called `/views`. This directory will be used to store all route components. In this directory, create `Login.vue` file with the following template:

```
// Login.vue

<template>
  <v-container class="fill-height" fluid>
    <v-row align="center" justify="center">
      <v-col cols="12" sm="8" md="6" lg="4">
        <v-card class="elevation-2">
          <v-toolbar color="default" flat>
            <v-toolbar-title>Login</v-toolbar-title>
          </v-toolbar>
          <v-card-text>
            <v-form v-model="isFormValid">

              <v-text-field
                v-model="email"
                :rules="[rules.required, rules.email]"
                label="Email"
                type="email">
              </v-text-field>

              <v-text-field
                v-model="password"
                :rules="[rules.required]"
                label="Password"
                :append-icon="showPassEye ? 'mdi-eye' : 'mdi-eye-off'"
                :type="showPassEye ? 'text' : 'password'"
                @click:append="showPassEye = !showPassEye"
                class="mt-2">
              </v-text-field>

              <div class="d-flex justify-end mt-2">
                <v-btn
                  color="primary"
                  text
                  @click="doLogin"
                  :disabled="!isFormValid">
                  Login
                </v-btn>
              </div>
            </v-form>
          </v-card-text>
        </v-card>
      </v-col>
    </v-row>
  </v-container>
</template>
```

In this template, we basically create an HTML form using `v-form` component, two text fields using `v-text-field` (for email and password) and submit button at the bottom. This is a typical example of a Vuetify form. The form is wrapped into layout containers with responsive settings. Submit button is disabled if the form contains errors (see below). Props passing to layout

components are not very important now (their full description can be found in the Vuetify documentation). But let's look at `v-model`²⁸ attribute used in both text fields. This is not a typical prop but another important directive. It creates a two-way binding on a form input element (input, textarea, checkbox, radiobutton or select) and automatically picks the correct way to update the element based on the input type. If the value in the `data` is changed, the value rendered in the template is also changed automatically. The same goes in the opposite direction – that's why it's called two-way data binding. Now let's add a Vue instance for this template.

```
// Login.vue

<script>
export default {
  name: "Login",
  data() {
    return {
      email: "",
      password: "",
      showPassEye: false,
      isFormValid: false,
      rules: {
        required: (value) => !!value || "Required",
        email: (v) => /.+@.+/ .test(v) || "E-mail must be valid",
      },
    };
  },
  methods: {
    doLogin() {
      console.log("Clicked login");
    },
  },
};
</script>
```

The entered values from the user in input elements are stored in the `email` and `password` data properties, and they are updated via two-way data binding (`v-model` directive mentioned before). `showPassEye` tracks whether the user wants to show raw password, `rules` define the rules for `v-text-field` validation (the rules were created according to Vuetify docs), and `isFormValid` holds the current validation state of the whole form. The `doLogin` method has also been added in `methods` section. This function is called from the template after pressing the submit button. For now, it only prints the debug text to the console.

`Register.vue` component will be created in a very similar way - it is necessary to add a form with individual inputs, add a submit button and wrap it all in a responsive layout. For brevity, this step will be omitted in this text, and you can try to implement it by yourself. Of course, you can always find the solution in the folder with the finished project. When registering a new user, it will be necessary to fill in the following information: `email`, `name`, `surname`, `gender` ("male", "female", or "other") and `password`. All information is required, and without it, you will not be able to submit the form.

²⁸ <https://vuejs.org/v2/guide/forms.html>

Adding routes

Cool, I assume that in the previous step, you created two routes components for login and register. But you probably haven't seen these components in action yet because we haven't used them anywhere. Let's add them to the list of all routes.

```
// router.js

...
import Login from "../views/Login.vue";
import Register from "../views/Register.vue";
import Home from "../views/Home.vue";
import NotFound from "../views/NotFound.vue";
...
const routes = [
  {path: "/login", component: Login, name: "login"},
  {path: "/register", component: Register, name: "register"},
  {path: "", component: Home, name: "home"},
  {path: "*", component: NotFound, name: "notFound"},
];
...
export default router;
```

```
// App.vue

<template>
  <v-app>
    <app-bar ... />
    <v-main>
      <router-view></router-view>
    </v-main>
  </v-app>
</template>

...
```

Do you remember the empty array of routes? Now we have filled it with four routes. Each route has several configuration options. You can find a list of all in official docs²⁹, and we only need three so far:

- `path` – a string that equals the path of the current route and URL address in the browser,
- `component` – the component that the router should render if the path matches,
- `name` – sometimes, it is more convenient to identify a route with a name, especially when linking to a route or performing navigations.

Login and Register components created recently are probably known to you, but what about the others?

- `Home` – this component is rendered if there is no relative path in the URL (e.g. only <http://localhost:8080>). As it is basically the home page of your application (the user visits it first), I will leave the design entirely to your feelings. Of course, you may find inspiration in provided solution.
- `NotFound` – this component is rendered if no previous route is matched (note `*` in `path`). For these cases, I added a simple 404 page with the following template. In this template, you can see how to load a local image stored in the `/assets` folder and also how to apply

²⁹ <https://router.vuejs.org/guide/>

simple CSS styles that are scoped to a given component. Also, note that this component is a dummy - the `.vue` file does not contain a `script` section, so the component only displays HTML elements without JavaScript logic.

```
// NotFound.vue

<template>
  <div class="page-not-found">
    <v-container>
      <v-layout wrap row align-center>
        <v-flex class="mb-5">
          <v-img :src="require('../assets/404image.png')"
            contain height="250"></v-img>
        </v-flex>
        <v-flex class="text-xs-center">
          <div class="text-h3">
            Page not found.
          </div>
          <div class="text-subtitle-2 mt-2">
            The page you are trying to get never existed in this reality, or has
            migrated to a parallel universe. Try going back to home page and
            repeat your action.
          </div>
          <v-btn flat color="primary" exact class="mt-4" to="/">
            Homepage
          </v-btn>
        </v-flex>
      </v-layout>
    </v-container>
  </div>
</template>

<style scoped>
.page-not-found {
  padding-top: 10rem;
  margin-bottom: 5rem;
}
</style>
```

The router still does not render route components. We have to say exactly where these route components should be rendered. To do this, we have to use the `router-view` component and insert it in the appropriate place in the `App` component template (as mentioned above).

Great! Now we should have a completed router. If we put <http://localhost:8080/login> in the URL, we should see the login form (similarly for registration). If we just type <http://localhost:8080>, the home page should be rendered. If we write random text after the base URL (e.g. "abcd"), we should be redirected to a 404 page.

Adding links

Let's go back to our first `AppBar` component. There are login and register buttons, which are not working yet. And also the *logo*. Let's get them working!

Links within the application can be solved in several ways. In the `NotFound` component, we used props `to` on `v-btn` component from Vuetify, and we declare that we want to set location to `"/` after click (`v-btn` wraps the functionality of router when it is used with `to` attribute). In addition,

there is a `router-link` component from `vue-router` that gives us more options. The following code shows both ways in action.

```
// AppBar.vue

<template>
  <v-app-bar app dense color="primary" dark>
    <router-link :to="{ name: 'home' }" exact tag="button">
      <v-toolbar-title>Chat application</v-toolbar-title>
    </router-link>
    <v-spacer />
    <div v-if="user != null">
      Hello {{ user.name }}
      <v-btn outlined @click="logout">
        Logout
      </v-btn>
    </div>
    <div v-else>
      <v-btn class="mr-2" outlined :to="{ name: 'login' }"
        active-class="active" exact>
        Login
      </v-btn>
      <v-btn outlined :to="{ name: 'register' }" active-class="active" exact>
        Registration
      </v-btn>
    </div>
  </v-app-bar>
</template>

...
<style scoped>
.active {
  background: #6c08d1;
}
</style>
```

Props `to` get a JS object according to which the router decides. `active-class` defines the CSS class that should be set for a given element if the current URL matches the given path. Thanks to this, it is possible to beautifully highlight the currently displayed subpage in the navigation. `exact` props only specify that the match with the URL must be completely accurate (e.g. path `"/` partially matches all other paths and therefore, without using `exact`, the `active-class` would be applied every time in that case).

HTTP client setup

It's time to connect our frontend to the prepared backend. First, we have to start the server (according to the instructions in `readme.md` in `/server` directory). If the backend is up and running, we can continue.

We will create a new folder `/src/code` for some JavaScript code (helpers, constants etc.) shared across the entire application. In this folder, add file `http-common.js` and `constants.js` with the following content:

```
// http-common.js

import axios from "axios";
import { API_URL } from "../constants";
```

```
export const axiosInstance = axios.create({
  baseURL: API_URL,
});

// constants.js

export const API_URL = "http://localhost:3333";
```

In the file with constants, we defined the base API URL taken from the `readme.md` file in the `/server` folder. All HTTP requests made by the application will be directed to this address. In the second file, we created an axios instance. We installed the Axios library at the beginning, so now we can import it. This HTTP client is very popular, but of course, there are other great alternatives³⁰ as well as native APIs (e.g. `fetch`³¹ or `XMLHttpRequest`³²). Using a single instance is very useful because we can easily set in one place how requests are sent and responses are processed. In this case, we set the `baseURL`, which we import from `constants.js`.

Now we have to decide how we will use this axios instance. In general, we have two options: import this file at each necessary place (component) or register it on the Vue prototype and thus make it available globally to all components. Both solutions are suitable; I will choose the second one. It is enough to add the following two lines in the `main.js` file.

```
// main.js

...
import { axiosInstance } from "../code/http-common";
...
Vue.prototype.$http = axiosInstance;
...
```

Excellent! Now we can access this HTTP client via `this.$http` in each component.

HTTP requests

Let's go back to the login and register forms. Although they look amazing, they do not store data on the backend yet. Communication with the server and asynchronous operations, in general, will be the subject in this section.

The term asynchronous refers to two or more objects or events not existing or happening at the same time (or multiple related things happening without waiting for the previous one to complete)³³.

Many Web API features now use asynchronous code to run, especially those that access or fetch some kind of resource from an external device, such as fetching a file from the network, accessing a database and returning data from it, accessing a video stream from a web cam, or broadcasting the display to a VR headset. When you fetch an image from a server, you can't return the result immediately. That means that the following (pseudocode) wouldn't work:³⁴

```
let response = fetch('myImage.png'); // fetch is asynchronous
```

³⁰ <https://github.com/request/request/issues/3143>

³¹ https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

³² <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>

³³ <https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous>

³⁴ <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing>

```
let blob = response.blob();
// display your image blob in the UI somehow
```

That's because you don't know how long the image will take to download, so when you come to run the second line, it will throw an error (possibly intermittently, possibly every time) because the response is not yet available. Instead, you need your code to wait until the response is returned before it tries to do anything else to it. There are two main types of asynchronous code style you'll come across in JavaScript code, old-style *callbacks* and newer *promise*-style code. We will use promises.

Let's add an asynchronous request to the login endpoint. We have to modify `doLogin` function in Login component:

```
// Login.vue
...
<template>
...
  </v-form>
  </v-card-text>
  <v-alert v-if="error != null" type="error" dismissible>{{ error }}</v-alert>
</v-card>
...
</template>

<script>
...
  async doLogin() {
    this.error = null;
    const payload = { email: this.email, password: this.password };
    try {
      const response = await this.$http.post("/auth/login", payload);
      const { token } = response.data;
      console.log(token);
    } catch (e) {
      this.error = e?.response?.data?.message ?? "An unexpected error occurred.";
    }
  },
...
</script>
```

What happened here? We have prepared `payload` data in the format expected by BE (see Swagger documentation). Then we added a `try-catch` block to catch errors when the HTTP request is processed by the backend. Then using `this.$http` (our Axios instance), we created a POST request (the data is sent to BE and BE has an endpoint for the POST method), where we passed two parameters - endpoint URL (this string is pasted after the `baseUrl` defined when creating the Axios instance) and payload data. To handle the promise, we used the `async-await`³⁵ construct (ECMAScript 2017). This is a newer notation for promises that acts as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards. They make `async` code look more like old-school synchronous code, so they're well worth learning.

If the request is successful, BE sends a user token in the response, which is just written to the console for now. A token is a long string (e.g. `eyJhbGciOiJIUzI1NiI...Orvz26BV6PM8A`) that encodes

³⁵ https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

basic user information and serves to authorize and authenticate the user. This token is unique to the logged user (generated by BE), and after any token change, the token becomes invalid, and BE does not allow the user to read or write data anymore. That token is valid for two days. Specifically, it is a JWT token and can be decoded on the <http://jwt.io> page.

If the request fails, the code falls into the `catch` block. In this block, we try to read the error message sent from the server, and we also add a feedback message if BE does not return anything. We set this message to data property `error` (`error` is defined in the `data` object as well), and in the `template` we conditionally render the `v-alert` component from Vuetify with the error text using the `v-if` directive. This alert is displayed if the user has entered the wrong login credentials and immediately knows what is happening.

Use the same approach to create an HTTP request for registration in `Register` component. The request will differ by the URL of the endpoint (because we want to register and not log in) and payload data.

Token manager class

In the previous step, we received a token from the backend. From now we must attach this token to each (secured) endpoint in the request header. BE will always extract this token from the request header and verify its validity. So far, we are only writing the token to the console, but we are going to change it now and add a special `TokenManager` class.

Let's add a string constant `export const LS_TOKEN_KEY = "chat_token";` to `constants.js` that will be used as a key in LS. In the future, we can use this key to find a token in LS. Now let's create a new file in `/code/token-manager.js` and implement the following class:

```
// token-manager.js

import { axiosInstance } from "../http-common";
import { LS_TOKEN_KEY } from "../constants";

export class TokenManager {
  token = null;
  setToken(token) {
    this.token = token;
    axiosInstance.defaults.headers["Authorization"] = `Bearer ${token}`;
    localStorage.setItem(LS_TOKEN_KEY, token);
  }
  logout() {
    this.token = null;
    delete axiosInstance.defaults.headers["Authorization"];
    localStorage.removeItem(LS_TOKEN_KEY);
  }
  renew() {
    const token = localStorage.getItem(LS_TOKEN_KEY);
    if (token) {
      this.setToken(token);
    }
  }
  getPayload() {
    if (this.token) {
      const parts = this.token.split(".");
      const rawToken = decodeURIComponent(escape(atob(parts[1])));
      return JSON.parse(rawToken);
    }
  }
}
```

```

    return null;
  }
  isLoggedIn() {
    return this.token !== null;
  }
}

```

This class solves two things at the same time:

1. add/remove a token from request headers,
2. save/delete token to local storage of the browser.

The first is more or less clear - for authentication. We haven't solved the second feature yet. The local storage (LS)³⁶ interface provides access to a particular domain's storage data. It allows, for example, the addition, modification, or deletion of stored data items. While data stored in JavaScript is automatically lost after reloading, data stored in LS can only be deleted by an application (or manually by a user) and so usually persists even after closing the browser. We use this storage to store the received token. As soon as the application receives a new token from BE, it stores this token in LS. When the user visits the application again (or just refreshes the page), the token is read from the LS, and the user does not have to log in again.

The class has several methods on which you can notice how to work with Axios headers and `localStorage`. There is also a method for decoding a JWT token.

Finally, we need to instantiate this class. We will do this in the `main.js` file, from where we then export a new instance so that it can be used in other parts of the application as well. Do not forget to call the `renew()` function after creation, which will try to renew the user from the LS. This action must be performed at the application startup, and the `main.js` file is the first loaded JavaScript file.

```

// main.js

...
import { TokenManager } from "../code/token-manager";

Vue.prototype.$http = axiosInstance;

export const tokenManager = new TokenManager();
tokenManager.renew();
...

```

Login and logout flow

In the previous steps, we implemented a bunch of things in the background. Now let's implement the login and logout flow and use the created `TokenManager` class.

First, let's add a new route component `/router/views/Rooms`, which will be a dummy for now.

```

// Rooms.vue

<template>
  <div>Rooms component</div>
</template>

```

³⁶ <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

Let's add this route among the others.

```
// router.js

import Rooms from "../views/Rooms.vue";
const routes = [
  ...
  { path: "/rooms", component: Rooms, name: "rooms" },
  ...
];
```

Well done! Now let's go back to the `Login` component and complete the `doLogin` method as follows:

```
// Login.vue

<script>
import { tokenManager } from "../../main";
...
try {
  const response = await this.$http.post("/auth/login", payload);
  const { token } = response.data;
  tokenManager.setToken(token);
  const userData = tokenManager.getPayload();
  this.$emit("userLogged", userData);
  this.$router.push({ name: "rooms" });
} catch (e) {
  ...
}
```

After logging in, we will save obtained token in the `tokenManager` (imported at the beginning). Then we read the user data from the token and use Vue.js `$emit` function to create the `userLogged` event, where we pass user data as a second parameter. After logging in, we want to automatically redirect the user to the page with all rooms. To be able to do routing from JS, we have to use `this.$router` object, which is available globally and using its `push()` method, we will add a `rooms` route to the top of the history stack.

Now, if you try to log in with the correct credentials, you will be automatically redirected to the `Rooms` component, and you will see its template. Cool, isn't it?

Login works for us. Now let's go back to the `App` component and implement the following features:

- display the name of the current user in the `AppBar` component,
- logout logic.

As you can see in the code below, we modified the template of the `App` component. On the `router-view` component, we listen to `userLogged` event (emitted by the nested `Login` component) and at the same time pass the `user` props so that the user data is available in all view components rendered by the router (it will be useful later).

We set the data property `user` to `null` by default (no user exists at the time the application is started). Besides the fact that the `doLogout()` function resets the data property `user`, it has to do two more things:

- call `tokenManager.logout()` function to remove a token from LS and HTTP requests,
- redirect the user to the login page.

```
// App.vue

<template>
  <v-app>
    <app-bar :user="user" @logout="doLogout" />
    <v-main>
      <router-view @userLogged="onUserLog" :user="user"></router-view>
    </v-main>
  </v-app>
</template>

<script>
import { tokenManager } from "../main";
...
data: () => {
  return {
    user: null,
  };
},
methods: {
  doLogout() {
    this.user = null;
    tokenManager.logout();
    this.$router.push({ name: "login" });
  },
  onUserLog(userData) {
    this.user = userData;
  },
},
mounted() {
  this.user = tokenManager.getPayload();
},
...

```

Okay, the Login component performs the login, and the token is stored in the `tokenManager`. Now we need to store the user data also in the Vue application, specifically in the reactive `data` of this `App` component. That's why we added the `onUserLog()` function.

The last function, `mounted()`, is a special Vue.js function that is part of the life cycle³⁷ of every Vue component. Each Vue instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

Exactly when the component is `mounted`, we want reactive data property `user` to be automatically initialized with information from the non-reactive `tokenManager`. If the user exists in the LS, it was already renewed (the `renew()` function in `main.js`), and now the `App` component only reads this information and saves it to reactive `data`. If the user does not exist, the `tokenManager`

³⁷ <https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks>

returns `null`, so the `user` property will be reset to default. Note that lifecycle hooks are registered directly in the options of each Vue instance (and not in the `methods` section).

So the login and logout flow is finished, and you can test it. The user can now log in. After refreshing the page, he stays logged in (the token is in the LS of the browser), and after logging out, the user is redirected to the login page (the token is removed from the LS).

Room components

Now let's try to implement the design and functionality of rooms components. In total, we will add five new components. Main `Rooms` component fetches the list of all rooms from BE and renders them in the `template` after creation (lifecycle hook). The room list will be fetched at regular intervals to reflect changes on BE. The rooms can be searched by name and description (`RoomsFilters` component). This search will only be implemented on the FE side. We also want to be able to create a new room (`NewRoom` component). And finally, when a user clicks on a room in room list (`Room` component), he enters it (`RoomDetail` component).

```
// Rooms.vue

<template>
  <v-container fluid class="pt-16">
    <v-row align="center" justify="center">
      <v-col cols="12" sm="10" md="7" lg="6">
        <div class="text-h2 text--disabled mb-16">Chat rooms</div>
        <v-alert v-if="error !== null" type="error">
          <div>{{ error }}</div>
          <v-btn class="mt-6" @click="initFetchingInterval">Try again</v-btn>
        </v-alert>
        <div v-else>
          <rooms-filters :onFilterChange="setFilter"></rooms-filters>
          <v-expansion-panels multiple popout v-if="filteredRooms.length">
            <room v-for="room in filteredRooms" :key="room.id" :room="room"></room>
          </v-expansion-panels>
          <div class="text-subtitle-2" v-else>No room found.</div>
        </div>
        <new-room @onNewRoom="addNewRoom"></new-room>
      </v-col>
    </v-row>
  </v-container>
</template>

<script>
import RoomsFilters from "../../components/RoomsFilters.vue";
import Room from "../../components/Room.vue";
import NewRoom from "../../components/NewRoom.vue";
import { UPDATE_INTERVAL_MS } from "../../code/constants";

export default {
  components: {
    RoomsFilters,
    Room,
    NewRoom,
  },
  data() {
    return {
      rooms: [],
      timer: null,
    }
  }
}
```

```

    error: null,
    filter: "",
  };
},
computed: {
  filteredRooms() {
    const filter = this.filter.toLowerCase();
    return this.rooms.filter((room) => room.title.toLowerCase().includes(filter)
      || room.description.toLowerCase().includes(filter));
  },
},
methods: {
  async loadAllRooms() {
    this.error = null;

    try {
      const response = await this.$http.get("/rooms");
      this.rooms = response.data;
    } catch (e) {
      this.error = e?.response?.data?.message ?? "An unexpected error occurred.";
      clearInterval(this.timer);
    }
  },
  initFetchingInterval() {
    this.loadAllRooms();
    this.timer = setInterval(this.loadAllRooms, UPDATE_INTERVAL_MS * 5);
  },
  setFilter(filter) {
    this.filter = filter ?? "";
  },
  addNewRoom(newRoom) {
    this.rooms.push(newRoom);
  },
},
mounted() {
  this.initFetchingInterval();
},
beforeDestroy() {
  clearInterval(this.timer);
},
};
</script>

```

As you can see, the code is already quite long. The number of lines in the Vue.js component is often larger, which is due to the object syntax of the Vue instance. For a better view, I recommend looking at the source code in GIT.

In the `Rooms` component, we have added a `mounted` hook, which initializes the interval for regular fetching all rooms. This interval should then be cleared using the `clearInterval()` method, so this HTTP request is not called after the component is destroyed (`beforeDestroy` hook). Fetched rooms are stored in the data property `rooms`. There are two new features:

- `v-for` directive³⁸ – is used to iterate objects in Vue templates. The directive gets an array (or also a classical object) and will repeatedly render the element on which it was used (in

³⁸ <https://vuejs.org/v2/guide/list.html>

this case, the `Room` component – for each room from the database). When using this directive, the props `key` must also be set, which Vue uses to identify the element and optimize changes. This `key` should be unique in the given array (in our case, the room id).

- `computed properties`³⁹ – is a very handy concept that optimizes the execution of changes in a Vue instance. `computed properties` are used in the same way as classic data properties; the difference is in the background. `computed properties` enable you to create a property that can be used to modify, manipulate, and display data within your components in a readable and efficient manner. You can use `computed properties` to calculate and display values based on a value or set of values in the data model. It can also have some custom logic that is cached based on its dependencies, meaning it doesn't reload but does have a dependency that changes, allowing it to somewhat listen to changes and act accordingly.⁴⁰

Also, note how we imported other custom `Room` components and used them in the `template`.

```
// RoomsFilters.vue

<template>
  <v-row justify="end">
    <v-col sm="6" md="6">
      <v-text-field
        label="Rooms filter"
        placeholder="Filter by name or description"
        filled rounded dense clearable
        @input="onFilterChange"> </v-text-field>
    </v-col>
  </v-row>
</template>

<script>
export default {
  props: {
    onFilterChange: {
      type: Function,
      required: true,
    },
  },
};
</script>
```

`RoomsFilter` is a dummy component that has a `template` and expects one prop – a callback, which is called after each keyboard press in the `v-text-field` component (`input` event is used for this use case). The text that the user writes is passed via callback to the parent component, which further processes it (see `Rooms` component above).

And what does the `Room` component look like? Let's take a look!

```
// Room.vue

<template>
  <v-expansion-panel>
    <v-expansion-panel-header>
      <v-row no-gutters>
```

³⁹ <https://vuejs.org/v2/guide/computed.html>

⁴⁰ <https://blog.logrocket.com/understanding-computed-properties-in-vue-js/>

```

    <v-col cols="6">
      <strong>{{ room.title }}</strong>
    </v-col>
    <v-col cols="6" class="text--secondary">
      <v-fade-transition leave-absolute>
        <v-row no-gutters style="width: 100%">
          <v-col cols="6">
            {{ new Date(room.created).toLocaleDateString() }} </v-col>
          <v-col cols="6"> {{ room.totalUsers }} users </v-col>
        </v-row>
      </v-fade-transition>
    </v-col>
  </v-row>
</v-expansion-panel-header>
<v-expansion-panel-content>
  <div class="text--secondary">
    {{ room.description }}
  </div>
  <v-row justify="end">
    <v-btn text color="primary"
      :to="{ name: 'roomDetail', params: { id: room.id } }">
      Enter
    </v-btn>
  </v-row>
</v-expansion-panel-content>
</v-expansion-panel>
</template>

<script>
export default {
  props: {
    room: {
      type: Object,
      required: true,
    },
  },
};
</script>

```

The Room component is also dummy and display room information in a list of all rooms. The component gets room props, reads the necessary properties (title, description, created date, total number of active users) and displays them in the template. The component is rendered as an expansion panel from Vuetify, and there is a button to enter the room. Note that when routing to a specific room, we also pass one parameter – the room ID. This ID will appear in the URL. But before that, we need to add a new route component RoomDetail, to the router and specify dynamic parameter id. Don't forget to add the RoomDetail component to the /views directory (source code will be shown later in this text).

```

// router.js
...
{ path: "/rooms/:id", component: RoomDetail, name: "roomDetail" },
...

```


The `NewRoom` component displays a form to fill in information about the new room (its `title` and `description`). After confirmation, a POST request is sent to the server, the room is stored in the database, and we should see a newly created room among the other rooms.

```
// NewRoom.vue

<template>
  <div class="mt-8">
    <div v-if="!formVisible" class="text-center">
      <v-btn fab medium elevation="2" @click="formVisible = true">
        <v-icon dark>
          mdi-plus
        </v-icon>
      </v-btn>
    </div>
    <v-card v-else elevation="2">
      <v-card-title>Create new room</v-card-title>
      <v-card-subtitle>You will be the admin of the new room.</v-card-subtitle>
      <v-card-text>
        <v-form v-model="isFormValid">
          <v-text-field v-model="title"
            :rules="[rules.required]" label="Room title" outlined>
          </v-text-field>
          <v-text-field v-model="description"
            :rules="[rules.required]" label="Room description" outlined>
          </v-text-field>
        </v-form>
      </v-card-text>
      <v-card-actions>
        <v-btn text color="error" @click="resetForm">
          Cancel
        </v-btn>
        <v-btn text @click="createNewRoom" :disabled="!isFormValid">
          Create
        </v-btn>
      </v-card-actions>
    </v-card>
  </div>
</template>

<script>
export default {
  data: () => {
    return {
      title: "",
      description: "",
      formVisible: false,
      isFormValid: false,
      rules: {
        required: (value) => !!value || "Required.",
      },
    };
  },
  methods: {
    resetForm() {
      this.title = "";
      this.description = "";
      this.isFormValid = false;
      this.formVisible = false;
    }
  }
}
```

```

    },
    async createNewRoom() {
      const payload = { description: this.description, title: this.title };
      try {
        const response = await this.$http.post("/rooms", payload);
        const newRoom = response.data;
        this.$emit("onNewRoom", newRoom);
        this.resetForm();
      } catch (e) {
        // TODO: handle errors
      }
    },
  },
};
</script>

```

Secured routes

Whew, that was a lot of work, but we're getting close to the finals. Try the following experiment. Log the user out of the application, delete the token from LS and try entering <http://localhost:8080/rooms> or <http://localhost:8080/rooms/some-real-room-id> directly in the URL. The application will let you in, and the components will be loaded even though the user is logged out. Ok, the backend is secured and won't return data in this case, but what if it wasn't or there was static information (not from BE)? Anyone from the internet could access these pages and read our content without a login. We have to fix this now using secured routes. Let's go to the router configuration file one last time and make the following changes:

```

// router.js
...
const routes = [
  { path: "/login", component: Login, name: "login" },
  { path: "/register", component: Register, name: "register" },
  { path: "/rooms", component: Rooms, name: "rooms", meta: { requiresAuth: true } },
  { path: "/rooms/:id", component: RoomDetail, name: "roomDetail",
    meta: { requiresAuth: true } },
  { path: "", component: Home, name: "home" },
  { path: "*", component: NotFound, name: "notFound" },
];

const router = new VueRouter({
  routes,
  mode: "history",
});

router.beforeEach((to, from, next) => {
  if (to.meta && to.meta.requiresAuth) {
    if (tokenManager.isUserLoggedIn()) {
      next();
    } else {
      next({ name: "login" });
    }
  } else {
    next();
  }
});

```

As you can see, we added meta information to some routes that it is a route requiring authentication. Then we added a global router guard⁴¹ that works as follows. If the route to be displayed requires authentication and the `tokenManager` doesn't know about any logged user, the routing is redirected to the login page. In other cases, navigation is allowed (`next()` method without parameters). With this setup, we also secured the routes on the frontend. If you repeat the previous experiment again, you will be unsuccessful.

Message components

The last thing we have left is the chatting components – displaying messages (`Messages`) and users in the room (`RoomUsers`) and a component for sending a new message (`NewMessage`). Let's get right into it.

```
// RoomDetail.vue

<template>
  <v-container fluid class="pt-16">
    <v-row align="center" justify="center">
      <v-col cols="12" sm="10" md="7" lg="6">
        <div v-if="room != null">
          <div class="text-h2 text--disabled mb-16">
            <v-btn icon :to="{ name: 'rooms' }" class="mr-5 mb-2">
              <v-icon>mdi-arrow-left</v-icon>
            </v-btn>
            {{ room.title }}
          </div>
          <div>
            <messages :roomId="room.id" :userId="user.sub"></messages>
          </div>
        </div>
      </v-col>
    </v-row>
  </v-container>
</template>

<script>
import Messages from "../../components/Messages.vue";

export default {
  components: { Messages },
  props: {
    user: Object,
  },
  data() {
    return {
      room: null,
    };
  },
  methods: {
    async loadRoomDetail(id) {
      try {
        const response = await this.$http.get(`/rooms/${id}`);
        this.room = response.data;
      } catch (e) {
        // TODO: handle errors
      }
    }
  }
}
```

⁴¹ <https://router.vuejs.org/guide/advanced/navigation-guards.html>

```

    }
  },
},
mounted() {
  const id = this.$route.params.id;
  this.loadRoomDetail(id);
},
};
</script>

```

Message components are rendered in the room detail component (code above). As already several times, when the `RoomDetail` component is mounted, the information about the room is fetched from BE. Notice how the ID of the current room is obtained. If this route component could not be accessed directly via URL, then it would be enough for the component to get the room ID via props. However, since this is a route component, and the user can revive it by typing the URL directly into the browser (and not only via clicks in the application), it is necessary to get the ID from the URL. This is done using the `$route` object, which like `$router` is globally available. This component displays the room name and the `Messages` component, which displays the chat. See the following code.

```

// Messages.vue

<template>
  <div>
    <room-users :roomId="roomId" class="mb-5"></room-users>
    <v-card max-width="100%" max-height="480" min-height="480"
      class="mx-auto overflow-y-auto" ref="messages">
      <v-list v-if="messages.length" three-line>
        <template v-for="(message, i) in messages">
          <v-divider v-if="i !== 0" inset :key="i"></v-divider>

          <v-list-item :key="message.id">
            <v-list-item-avatar>
              <v-img
                :src="`https://avatars.dicebear.com/api/avataaars/${message.userId}.svg`">
              </v-img>
            </v-list-item-avatar>

            <v-list-item-content>
              <v-list-item-title>{{ message.message }}</v-list-item-title>
              <v-list-item-subtitle>
                <span class="text--primary text-caption">
                  {{ message.userFullName }}
                </span>
                <span class="text--disabled text-caption">
                  {{ new Date(message.created).toLocaleString() }}
                </span>
              </v-list-item-subtitle>
            </v-list-item-content>
          </v-list-item>
        </template>
      </v-list>
      <div v-else class="py-16">
        <div class="text--disabled text-center">
          No messages yet. Write something!
        </div>
      </div>
    </div>
  </template>

```

```

    </v-card>
    <new-message :roomId="roomId" @onNewMessage="addNewMessage"></new-message>
  </div>
</template>

<script>
import NewMessage from "./NewMessage.vue";
import RoomUsers from "./RoomUsers.vue";
import { UPDATE_INTERVAL_MS } from "../code/constants";

export default {
  components: { NewMessage, RoomUsers },
  props: {
    roomId: {
      type: String,
      required: true,
    },
    userId: {
      type: String,
      required: true,
    },
  },
  data() {
    return {
      messages: [],
      timer: null,
    };
  },
  methods: {
    async loadMessages() {
      try {
        const response = await this.$http.get(`/rooms/${this.roomId}/messages`);
        const hasNewMessages = response.data.length !== this.messages.length;
        this.messages = response.data;
        if (hasNewMessages) {
          this.scrollToBottom();
        }
      } catch (err) {
        // TODO: handle errors
      }
    },
    addNewMessage(newMessage) {
      this.messages.push(newMessage);
      this.scrollToBottom();
    },
    scrollToBottom() {
      const el = this.$refs.messages?.$el;
      if (el !== null) {
        setTimeout(() => el.scrollTo({ top: el.scrollHeight, behavior: "smooth", block: "end" }));
      }
    },
  },
  mounted() {
    this.loadMessages();
    this.timer = setInterval(this.loadMessages, UPDATE_INTERVAL_MS * 2);
  },
  beforeDestroy() {
    clearInterval(this.timer);
  }
}

```

```

    },
  };
</script>

```

The `messages` component fetches messages in the room at regular intervals. If a new message has been added in the meantime, the `scrollToBottom()` function scrolls the message container to the bottom so that the new message is visible. The messages are displayed using the Vuetify `v-list` component. The `messages` component also renders two of our components - the list of users (at the top) and input for writing messages (at the bottom).

`RoomUsers` component can look like:

```

// RoomUsers.vue

<template>
  <div>
    <v-sheet class="mx-auto" max-width="100%">
      <div class="text-subtitle-2 mb-2">Active users ({{ users.length }})</div>
      <v-slide-group multiple show-arrows>
        <v-slide-item v-for="user in users" :key="user.id">
          <div class="d-flex flex-column align-center">
            <v-img
              :src="`https://avatars.dicebear.com/api/avataaars/${user.id}.svg`"
              max-height="60" max-width="60">
            </v-img>
            <v-btn class="mx-2" active-class="purple white--text" depressed
              text small>
              {{ user.name }} {{ user.surname }}
            </v-btn>
          </div>
        </v-slide-item>
        <div v-if="!users.length">
          <div class="text-subtitle-1 text--disabled mb-2 text-center">
            Waiting for users...
          </div>
        </div>
      </v-slide-group>
    </v-sheet>
  </div>
</template>

<script>
import { UPDATE_INTERVAL_MS } from "../code/constants";

export default {
  props: {
    roomId: {
      type: String,
      required: true,
    },
  },
  data() {
    return {
      users: [],
      timer: null,
    };
  },
  methods: {

```

```

    async loadUsers() {
      try {
        const response = await this.$http.get(`/rooms/${this.roomId}/users`);
        this.users = response.data;
      } catch (err) {
        // TODO: handle errors
      }
    },
  },
  mounted() {
    this.loadUsers();
    this.timer = setInterval(this.loadUsers, UPDATE_INTERVAL_MS * 2);
  },
  beforeDestroy() {
    clearInterval(this.timer);
  },
};
</script>

```

The `RoomUsers` component works the same as the previous one. `mounted` hook sets the interval for fetching a list of users in the room. The users are rendered via the Vuetify component `v-slide-group`, which displays the items in a horizontal slide element.

We generate an avatar for each user via the freely available API <https://avatars.dicebear.com>. This service generates pretty nice images based on the seed. We send there a user ID that is unique, and therefore the service generates the same avatar for this ID every time.

And the last but not least component is `NewMessage`:

```

// NewMessage.vue

<template>
  <v-card max-width="100%" class="mx-auto">
    <v-text-field
      v-model="message" filled hide-details
      clear-icon="mdi-close-circle" clearable label="Message"
      type="text" append-icon="mdi-send"
      @click:append="sendMessage" @keypress.enter="sendMessage">
    </v-text-field>
  </v-card>
</template>

<script>
export default {
  props: {
    roomId: {
      type: String,
      required: true,
    },
  },
  data() {
    return {
      message: "",
    };
  },
  methods: {
    async sendMessage() {
      if (!this.message) {

```

```
        return;
    }
    const payload = {
        message: this.message,
    };
    try {
        const response =
            await this.$http.post(`/rooms/${this.roomId}/messages`, payload);
        this.$emit("onNewMessage", response.data);
        this.message = "";
    } catch (e) {
        // TODO: handle errors
    }
    },
},
};
</script>
```

This component renders one text field into which the user writes his message. After clicking on send icon (@click event) or pressing enter (@keypress event), the text is sent to BE, and the component emits the server response with the new message.

And that's all! You should now be able to register and login two different users (each in a different browser or via an incognito mode). Also, create a room or enter an already created one. Both users should be displayed in the list, and these users can communicate comfortably with each other using chat components.

Simple Forms

Create a simple HTML page with two inputs, button and element h1, similar to the next code:

```
First name: <input id="first_name" />
Last name: <input id="last_name" />
<button id="hello">Hello</button>
<h1 id = "result"></h1>
```

- Get input from the user (name and surname) and combine that with the string "Hello".
- Show a simple message that greeting you.
- Show the message like the content of element h1.

Create a simple HTML page with input Range and element h1, similar to the next code:

```
<input type="range" max="100" min="1" value="12" id="range">
<h1 id="heading">jQuery</h1>
```

- Create functionality for changing the font size of element h1. Font size will change after changing the value in Range input. Font size will depend on the value of Range input.
- Insert some images, and create functionality for image size changing (width and height). It will depend on the Range input.

JQuery Animation

The jQuery animate() method is used to create custom animations. The required params parameter defines the CSS properties to be animated.

Create a rectangle on the page and add two buttons – “move to the right” and “move to the left”, similar to the next code:

```
<style type="text/css">

div{
    background:#98bf21;
    height:100px;
    width:100px;
    position:absolute;}
</style>

<button>Start Animation</button>

<div></div>
```

- For the button "move to the right" create an effect for moving the rectangle to the right, and for the button "move to the left" create a movement to the left.

Hammer Hitting Turtle

Create an online "hammer hitting turtle" game with jQuery.

- Select and use the JavaScript method: `requestAnimationFrame()` or `setInterval()` for updating state of game before the next repaint.
- Hammer hit will be a mouse click event.
- Do not forget to create a "counter of the true hits" in your game.
- It is not necessary to create a beautiful graphical design for the game. Turtles and hammer could be coloured squares only.



MOBILE APPLICATIONS

To-do Application for Android in Java

The presented example shows a mobile application for Android. The mobile application serves as a productivity tool and allows users to insert their tasks. Users can associate tasks with dates and locations. Users can also modify the tasks and delete them. When they are done with the task, it can be marked as resolved using the checkbox.

Recommended Number of Developers

Individual.

Available Solutions

There are a lot of similar solutions. To name some:

- Todoist (<https://play.google.com/store/apps/details?id=com.todoist>)
- Google Tasks
(<https://play.google.com/store/apps/details?id=com.google.android.apps.tasks>)
- Any.do (<https://play.google.com/store/apps/details?id=com.anydo>)

However, these solutions are much more advanced.

Requirements

The application will be created in Android Studio. Here is an installation guide for Windows (<https://developer.android.com/studio/install>). The application is developed using the Kotlin programming language. The development requires a basic knowledge of Object-oriented programming.

Functional Requirements

- a) The application will show a list of current tasks of the user in the list.
- b) The user can add a new task.
- c) The user can modify an existing task.
- d) The user can delete an existing task.
- e) Each task has a description, date and image.
- f) Image can be captured by the internal camera, selected from the camera or selected from internal storage.

Non-functional Requirements

- a) Android 6.0 and higher.
- b) All tasks must be available after the application is closed and again opened.
- c) Java programming language

Application Design

Technology and Architecture Selection

The application is written using the Java programming language. It shows an older but still valid of creating a user interface using activities. The application also shows basic access to the database using the SQLiteOpenHelper class.

Data Model

Task	
id	Long
title	String
description	String
dueDate	String
image	String
done	boolean

Fig. 23 Data model

User Interfaces

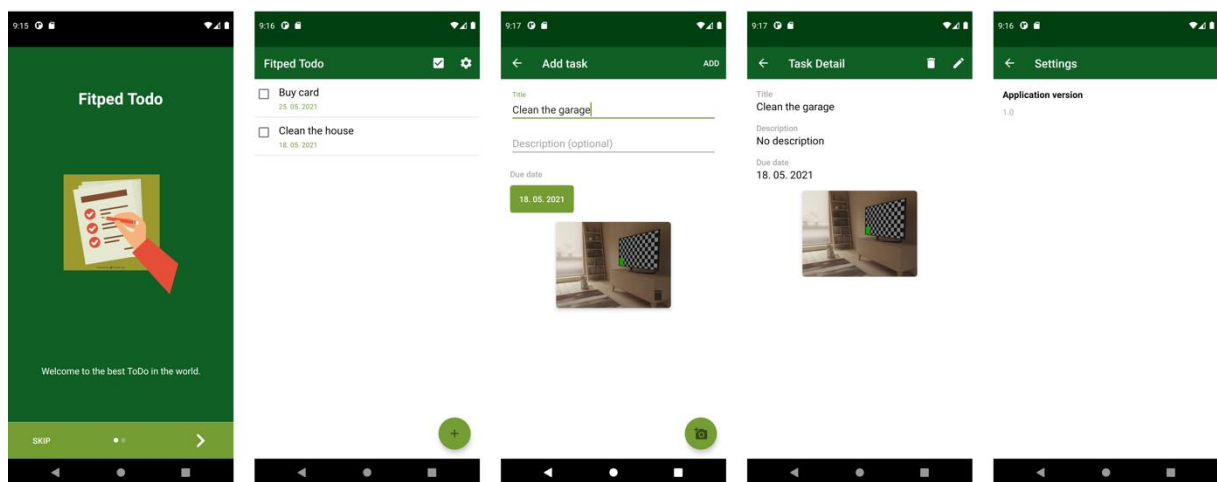


Fig. 24 Interface definition

Solution

Application database

The database is done using the *SQLiteOpenHelper* class. It is a basic class for defining the database structure. Nowadays, there are many libraries to provide better access to the database. The most popular is **Room**. However, it is still important to know how the database works.

On Android, we will be using the SQLite database. It is very similar to SQL. However, it does not have a full range of functions.

The database is created by inheriting from the *SQLiteOpenHelper* class. When we inherit from the *SQLiteOpenHelper* class, we need to *override* two methods:

- **onCreate** - called only once when the app is run for the first time.
- **onUpgrade** - called every time the database number increases.

```
public class TodoDatabaseHelper extends SQLiteOpenHelper {
    private static final int DATABASE_VERSION = 1;
```

```

private static final String DATABASE_NAME = "todo_db";

public TodoDatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(TaskDatabaseScheme.CREATE_TASKS_TABLE);
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL(TaskDatabaseScheme.DELETE_TASKS);
    onCreate(db);
}
}

```

In the constructor, we provide the name of the database and the database number. In the *onCreate* method, the *CREATE TABLE* query is performed.

In a previous code, we are using the *TaskDatabaseScheme* file. This file contains the definition of the operations over the database. It also contains the definition of the columns and the query to delete the entire table.

```

public interface TaskDatabaseScheme extends BaseColumns {
    String TABLE_NAME = "tasks";
    String COLUMN_TITLE = "title";
    String COLUMN_DESCRIPTION = "description";
    String COLUMN_TASK_DONE = "task_done";
    String COLUMN_DUE_DATE = "due_date";
    String COLUMN_IMAGE = "image";

    String CREATE_TASKS_TABLE =
        "CREATE TABLE " + TABLE_NAME + " (" +
            _ID + " INTEGER PRIMARY KEY," +
            COLUMN_TITLE + " TEXT," +
            COLUMN_DESCRIPTION + " TEXT," +
            COLUMN_DUE_DATE + " TEXT," +
            COLUMN_TASK_DONE + " INTEGER," +
            COLUMN_IMAGE + " TEXT" +
        ")";

    String DELETE_TASKS =
        "DROP TABLE IF EXISTS " + TABLE_NAME;
}

```

The access to the database, we will use the **Dao** pattern. Firstly, we define the interface representing the operations over the database.

```

public interface ITasksDao {
    void addTask(Task newTask);
    ArrayList<Task> getAllTasks();
    ArrayList<Task> getAllUndoneTasks();
    Task getTaskByID(long id);
    void updateTask(Task task);
}

```

```

void deleteTask(Task task);
void markTaskDone(long taskID, boolean done);
}

```

Next, we implement the interface. To save the task to the database, we need to convert it to the *ContentValues* object. The method *taskToContentValues* serves this purpose.

```

private ContentValues taskToContentValues(Task task) {
    ContentValues contentValues = new ContentValues();
    contentValues.put(TaskDatabaseScheme.COLUMN_TITLE, task.getTitle());
    contentValues.put(TaskDatabaseScheme.COLUMN_DESCRIPTION, task.getDescription());
    contentValues.put(TaskDatabaseScheme.COLUMN_DUE_DATE, task.getDueDate());
    contentValues.put(TaskDatabaseScheme.COLUMN_IMAGE, task.getImage());
    if (task.isDone()) {
        contentValues.put(TaskDatabaseScheme.COLUMN_TASK_DONE, 1);
    } else {
        contentValues.put(TaskDatabaseScheme.COLUMN_TASK_DONE, 0);
    }

    return contentValues;
}

```

To read the data from the database, we use the *Cursor* class. *Cursor* gives us a view of the result of the specific request. The method *cursorToTask* converts one row in the cursor to the task.

```

private Task cursorToTask(Cursor cursor) {
    Task task = new Task();
    task.setId(cursor.getLong(cursor.getColumnIndex(TaskDatabaseScheme._ID)));
    task.setTitle(cursor.getString(cursor.getColumnIndex(
        TaskDatabaseScheme.COLUMN_TITLE)));
    task.setDescription(cursor.getString(cursor.getColumnIndex(
        TaskDatabaseScheme.COLUMN_DESCRIPTION)));
    task.setDueDate(cursor.getString(cursor.getColumnIndex(
        TaskDatabaseScheme.COLUMN_DUE_DATE)));
    task.setImage(cursor.getString(cursor.getColumnIndex(
        TaskDatabaseScheme.COLUMN_IMAGE)));

    int taskDone =
        cursor.getInt(cursor.getColumnIndex(TaskDatabaseScheme.COLUMN_TASK_DONE));

    if (taskDone == 1) {
        task.setDone(true);
    } else {
        task.setDone(false);
    }

    return task;
}

```

The last step is the operations itself. Let's mention one as an example. The inserting of the new task is performed in *addTask* method. First, the instance of the database is created. Then, the writable object of the database is saved into the database variable. This object is then used to insert the *ContentValues* class into the database. The last operation is closing the database so we can later access it again.

```

public void addTask(Task newTask) {
    TodoDatabaseHelper databaseHelper = new TodoDatabaseHelper(context);
}

```



```

    if (databaseHelper != null) {
        SQLiteDatabase database = databaseHelper.getWritableDatabase();
        try {
            long id = database.insert(
                TaskDatabaseScheme.TABLE_NAME,
                null,
                taskToContentValues(newTask));
            newTask.setId(id);
        } finally {
            database.close();
        }
    }
}

```

SharedPreferences

The database is the best way to store large amounts of data. However, if we need to save just a single value, it is not necessary to create a database for it. One way to store single values is *SharedPreferences* class. The class saves the values to an internal XML file.

To initialize the *SharedPreferences* class, we need the name of the file. An application can have multiple *SharedPreferences* files.

Part of *SplashScreenActivity* is an introduction to the application. The introduction should be run only the first time the application is opened. Each time the application is opened, we check if it is run for the first time using the *isRunForFirstTime* method.

```

public class SharedPreferencesManager {
    private static final String FILENAME = "todosp";

    private static SharedPreferences getSharedPreferences(Context context) {
        return context.getSharedPreferences(FILENAME, Context.MODE_PRIVATE);
    }

    @SuppressWarnings("ApplySharedPref")
    public static void saveFirstRun(Context context) {
        SharedPreferences.Editor editor = getSharedPreferences(context).edit();
        editor.putBoolean(SharedPreferencesConstants.FIRST_RUN, false);
        editor.commit();
    }

    public static boolean isRunForFirstTime(Context context) {
        SharedPreferences sharedPreferences = getSharedPreferences(context);
        return sharedPreferences
            .getBoolean(SharedPreferencesConstants.FIRST_RUN, true);
    }
}

```

SplashScreenActivity

SplashScreenActivity is the first activity that is run. It consists of two parts. The first part shows the logo of the application. The second part is an application introduction. Showing a logo is done using the styles of the application. Firstly, we create a drawable file (*splash_screen_background.xml*). This file will represent the background of the application.

```
<?xml version="1.0" encoding="utf-8"?>

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:drawable="@color/colorWhite"/>
    <item
        android:gravity="center"
        android:drawable="@drawable/logo_round" />
</layer-list>
```

Next, we use this file in style defined specifically for this activity.

```
<style name="SplashScreenTheme" parent="Theme.AppCompat.NoActionBar">
    <item name="android:windowBackground">@drawable/splash_screen_background</item>
</style>
```

The next part is setting the style in the *AndroidManifest.xml* file.

```
<activity
    android:name=".activities.SplashScreenActivity"
    android:label="@string/app_name"
    android:theme="@style/SplashScreenTheme"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

And that is it. However, for the Splash screen to work properly, we do not use the method *setContentView* in the *onCreate* method of the activity.

The second part of the activity is the application introduction. We are going to use the *AppIntro* library (<https://github.com/apl-devs/AppIntro>). The *AppIntro* is implemented by inheriting from the *AppIntro* class. Then, in *onCreate* method, if the application is run for the first time, the appintro slides are configured and shown. Otherwise, we continue to the list of tasks.

```
if (SharedPreferencesManager.isRunForFirstTime(this)) {
    SliderPage page1 = new SliderPage();
    page1.setTitle(getString(R.string.app_name));
    page1.setDescription(getString(R.string.help_1));
    page1.setBgColor(ContextCompat.getColor(this, R.color.colorPrimary));
    page1.setImageDrawable(R.drawable.todo_1);

    SliderPage page2 = new SliderPage();
    page2.setTitle(getString(R.string.app_name));
    page2.setDescription(getString(R.string.help_2));
    page2.setBgColor(ContextCompat.getColor(this, R.color.colorPrimaryDark));
    page2.setImageDrawable(R.drawable.todo_2);

    addSlide(AppIntroFragment.newInstance(page1));
    addSlide(AppIntroFragment.newInstance(page2));
    setBarColor(ContextCompat.getColor(this, R.color.colorAccent));
    setSeparatorColor(ContextCompat.getColor(this, android.R.color.white));
    setDoneText(getString(R.string.done));
    setSkipText(getString(R.string.skip));
}
```

```

        showSkipButton(true);
        setProgressButtonEnabled(true);
    } else {
        continueToApp();
    }
}

```

TodoListActivity

The main activity of the application is the activity with the list of tasks. In the centre of this activity is a method for setting the list.

```

private void setList(ArrayList<Task> newListOfTasks) {
    listOfTasks.clear();
    listOfTasks.addAll(newListOfTasks);
    if (adapter == null) {
        adapter = new TasksListAdapter(listOfTasks);
        RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(this);
        recyclerView.setLayoutManager(layoutManager);
        DividerItemDecoration dividerItemDecoration
            = new DividerItemDecoration(recyclerView.getContext(),
                                       RecyclerView.VERTICAL);
        recyclerView.addItemDecoration(dividerItemDecoration);
        recyclerView.setAdapter(adapter);
    } else {
        adapter.notifyDataSetChanged();
    }
}

```

In this method, we clear an old list of tasks; then, if the adapter class was not previously created, we create the instance of the adapter and set the adapter and *LayoutManager* to the *RecyclerView*. The line as a divider is also added. If the adapter already exists, we just notify it to refresh.

Another important part of this activity is the *refreshList* method. It decides if all tasks should be shown. If yes, it loads all tasks from the database. If not, it loads only the tasks that are not done.

```

private void refreshList() {
    if (!showAllTasks) {
        setList(taskDao.getAllUndoneTasks());
    } else {
        setList(taskDao.getAllTasks());
    }
}

```

A typical usage can be seen in the *onActivityResult* method. If we are returning from the adding of a new task, then the list is refreshed. The list is also refreshed when returning from the task detail because, from the detail, we can navigate to the update of the task, or the task can be deleted.

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ADD_TASK_REQUEST_CODE && resultCode == RESULT_OK) {
        refreshList();
    }
    if (requestCode == TASK_DETAIL_REQUEST_CODE) {
        refreshList();
    }
}

```

```

    }
}

```

As mentioned in a previous part, the user is able to control what list of tasks will be shown. It can be either all tasks or just unfinished tasks. The control is done by button on the toolbar and its associated *onClick* method.

```

public void showDoneTasks(MenuItem item) {
    showAllTasks = !showAllTasks;
    refreshList();
    if (!showAllTasks) {
        showAllTasksMenuItem.setIcon(R.drawable.ic_check_box_white);
    } else {
        showAllTasksMenuItem.setIcon(R.drawable.ic_check_box_outline_white);
    }
}

```

The method refreshes the list and also changes the icon of the *MenuItem*.

AddEditTaskActivity

The most complex activity of the application is *AddEditTaskActivity*. The functions of the activity can be split into two parts—the management of the data and the work with images. The work with images will be described in the last part of this tutorial.

The first part is very straightforward. Firstly, we decide if the activity is for adding a new task or for updating an existing task. This is done in the *onCreate* method. Based on the value of the *id*, we determine the state and also set the title on the toolbar.

```

taskDao = new TaskDao(this);
id = getIntent().getLongExtra(IntentConstants.INTENT_ID, -1);
if (savedInstanceState != null) {
    task = (Task) savedInstanceState.getSerializable(TASK);
} else {
    if (id != -1) {
        task = taskDao.getTaskByID(id);
        getSupportActionBar().setTitle(
            getString(R.string.title_activity_edit_task));
    } else {
        task = new Task();
        task.setDueDate(DateUtility.getCurrentDate());
        getSupportActionBar().setTitle(
            getString(R.string.title_activity_add_task));
    }
}

```

The next part is setting the *View* values from the *Task* object loaded from the database. A very important part is using the *addTextChangedListener* method of the *TextView*. This way, the task title and description are updated at the moment when the user changes the text inputs.

```

private void setGUIValues() {
    dueDateButton.setText(task.getDueDate());
}

```

```

if (id != -1) {
    titleTextInputLayout.getEditText().setText(task.getTitle());
    descriptionTextInputLayout.getEditText().setText(task.getDescription());
    dueDateButton.setText(task.getDueDate());
}
titleTextInputLayout.getEditText().addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start,
                                int count, int after) {

    }

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
        task.setTitle(s.toString());
    }

    @Override
    public void afterTextChanged(Editable s) {
    }
});

descriptionTextInputLayout.getEditText().addTextChangedListener(
    new TextWatcher() {
        @Override
        public void beforeTextChanged(CharSequence s, int start,
                                    int count, int after) {

        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            task.setDescription(s.toString());
        }

        @Override
        public void afterTextChanged(Editable s) {
        }
    });
}

```

Once the activity is visible to the user and the user makes changes, we need to save those changes to the database. Before saving, we need to test if the task title was filled. If so, we can proceed to inserting a new task or updating existing ones. If a new task is inserted or existing updated, the *Toast* is shown to the user, and the activity is finished.

```

public void save(MenuItem item) {
    boolean everythingOK = true;
    if (titleTextInputLayout.getEditText().getText().toString().trim().equals("")) {
        titleTextInputLayout.setError(getString(R.string.no_title_hint));
        everythingOK = false;
    }

    if (everythingOK) {
        if (id == -1) {
            taskDao.addTask(task);
            Toast.makeText(AddEditTaskActivity.this, R.string.new_task_created,

```

```

        Toast.LENGTH_SHORT).show();
        setResult(RESULT_OK);
        finish();
    } else {
        task.setTitle(titleTextInputLayout.getEditText().getText().
            toString().trim());
        task.setDescription(descriptionTextInputLayout.getEditText().getText().
            toString().trim());
        taskDao.updateTask(task);
        Toast.makeText(AddEditTaskActivity.this, R.string.task_updated,
            Toast.LENGTH_SHORT).show();
        setResult(RESULT_OK);
        finish();
    }
}
}

```

TaskDetailActivity

The *TaskDetailActivity* provides basic information about the task. It shows all the properties. The first part of the activity is getting the task from the database.

```

taskDao = new TaskDao(this);
id = getIntent().getLongExtra(IntentConstants.INTENT_ID, -1);
task = taskDao.getTaskByID(id);
setGUIValues();

```

As a final step, we will set the task values to the views using the *setGUIValues()* method.

```

private void setGUIValues() {
    titleTextView.setText(task.getTitle());
    if (task.getDescription() != null && !task.getDescription().equals("")) {
        descriptionTextView.setText(task.getDescription());
    } else {
        descriptionTextView.setText(R.string.no_description);
    }
    dueDateTextView.setText(task.getDueDate());
    if (task.getImage() != null) {
        Picasso.get().load(new File(getFilesDir().toString(),
            task.getImage())).resize(1500, 1500).centerCrop().into(imageView);
    } else {
        imageContainer.setVisibility(View.GONE);
    }
}
}

```

The purpose of the detail activity is also to provide the user with an option to either delete the task or run the update of the task. The update is done using the *AddEditTaskActivity*. These two operations are performed by two menu onClick methods defined in *menu_task_detail.xml* file.

Delete operation removes the task from the database and finishes the current activity.

```

public void delete(MenuItem item) {
    taskDao.deleteTask(task);
    Toast.makeText(this, R.string.task_deleted, Toast.LENGTH_LONG).show();
    finish();
}

```

The edit runs the *AddEditTaskActivity* activity.

```
public void edit(MenuItem item) {
    Intent intent = AddEditTaskActivity.createIntent(this, task.getId());
    startActivityForResult(intent, EDIT_TASK_REQUEST_CODE);
}
```

Settings Activity

The *SettingsActivity* is the simplest activity of the entire application. The activity shows the version of the application.

```
public class SettingsActivity extends AppCompatActivity {

    private TextView appVersion;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_settings);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        toolbar.setNavigationOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                finish();
            }
        });

        appVersion = findViewById(R.id.app_version);

        try {
            String versionName = getPackageManager()
                .getPackageInfo(getPackageName(), 0).versionName;
            appVersion.setText(versionName);
        } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

The application version always consists of version code and version name. These are defined in the *build.gradle* file for the module.

```
versionCode 1

versionName "1.0"
```

VersionCode is an internal identification of the version. When we want to create a new version, we need to increase the code. On the other hand, the *versionName* is an external identification of the version. This is what will be visible to the user. It is a string, so it can contain basically anything.

Working with images

The last important part of this tutorial is working with images. In the *AddEditTaskActivity*, the user has the option to add an image to the *Task*. There are three options to get the image:

- Device camera
- Photo gallery
- Device storage

The choosing of the option is done in the dialogue created in *showSelectImageSourceDialog* method. Let's look at the first option, capturing the image from the camera.

```
private void captureImageFromCamera() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        try {
            photoFile = FileUtility.createImageFile(this);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        if (photoFile != null) {
            Uri photoURI = FileProvider.getUriForFile(this,
                "cz.mendelu.pef.fileprovider",
                photoFile);
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE);
        }
    }
}
```

The starting of the camera is done using the *Intent* class. At the beginning of the method, we need to test if there is any camera present in the device (*resolveActivity* method). Then, an empty image file is created. We then pass the *URI* of the image to the intent, and the camera will fill this image with the captured data. However, the image is not visible to external applications, such as the camera. We need to grant camera access to the file using *FileProvider*. In order for the File Provider to work, it needs to be specified in *AndroidManifest.xml* file.

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="cz.mendelu.pef.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/paths" />
</provider>
```

Once the image is captured with a camera, we need to process it in the *onActivityResult* method.

```
if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
    task.setImage(photoFile.getName());
    Picasso.get().load(photoFile).resize(1500, 1500).centerCrop().into(imageView);
    deleteImageButton.setVisibility(View.VISIBLE);
}
```


Image is set to the task and loaded to the *ImageView* class using *Picasso* library.

The second option is choosing the image from the gallery. The operation is again done using the *Intent* class; however, we also need to check for permission to access external storage. If we do not have permission, we need to request it.

```
private void selectImageFromGallery() {
    if (PermissionUtility.checkPermissions(this)) {
        Intent intent = new Intent();
        intent.setType("image/*");
        intent.putExtra(Intent.EXTRA_LOCAL_ONLY, true);
        intent.setAction(Intent.ACTION_GET_CONTENT);
        startActivityForResult(Intent.createChooser(intent,
            getString(R.string.select_image)), GALLERY_IMAGE_REQUEST_CODE);
    } else {
        PermissionUtility.requestPermissions(AddEditTaskActivity.this,
            PERMISSION_SELECT_FROM_GALLERY_REQUEST_CODE);
    }
}
```

Once we open the gallery, we need to process the result in the *onActivityResult* method. However, it is much more difficult than with the image from the camera. The biggest problem is getting the correct address of the image. For this purpose, we need to use the *FileUtility* class. In this class, the method *getRealPath* returns us the path of the image that can be saved to the database but, more importantly, accessed in code. The next part is copying the image to our own internal storage and finally loading it into the *ImageView*.

```
if (requestCode == GALLERY_IMAGE_REQUEST_CODE && resultCode == RESULT_OK) {
    Uri uri = data.getData();
    String path = FileUtility.getRealPath(this, uri);
    File sourceFile = new File(path);
    File destinationFile = new File(getFilesDir(), sourceFile.getName());
    try {
        FileUtility.copy(sourceFile, destinationFile);
    } catch (IOException e) {
        e.printStackTrace();
    }
    task.setImage(destinationFile.getName());
    // load the image with picasso
    Picasso.get().load(uri).resize(1500, 1500).centerCrop().into(imageView);
    // show delete button.
    deleteImageButton.setVisibility(View.VISIBLE);
}
```

The last option is selecting an image from the external storage. This option is almost the same as the previous one. The only main difference is in the usage of *Intent* class. The *Intent* class is created with a parameter.

```
Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
```

Google Map Application Template

The presented application serves as a template for working with a Google Map on the Android operating system. Many applications contain the map, so it is a very important part of the development.

Recommended Number of Developers

Individual.

Available Solutions

There are many common map applications. However, in the context of this application, it is not necessary to mention them. The presented application is for study purposes only.

Requirements

The application allows users to add places in the form of markers to the map. The place can be added either by clicking on the map, clicking on a button or by moving the device in the real world.

Functional Requirements

- a) The application will show a list of places on the map.
- b) Each place has latitude, longitude and an id.
- c) The user can add a new place by touching the map.
- d) The user can add a new place by clicking on a FloatingActionButton.
- e) The user can add a new place by moving the device. The place will be added every 10 seconds.
- f) The user can update the existing place by dragging a marker.
- g) The user can delete an existing place by clicking on the marker.
- h) The user can zoom to all places by clicking on the FloatingActionButton.
- i) The application will save and restore the map camera position.

Non-functional Requirements

- a) Android 6.0 and higher.
- b) All places must be available after an application is closed and again opened.
- c) MVVM architecture.
- d) Use of Navigation Component for navigation.
- e) Kotlin programming language.

Application Design

Technology and Architecture Selection

The application is written in MVVM architecture, which is a recommended architecture for Android application development. The application uses Navigation Component with fragments to show the user interface. As a database, the Room library is used.

Data Model

The data model consists of a single class Place. It is defined by its latitude and longitude.

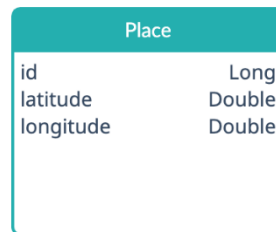


Fig. 25 Simple data model definition

User Interfaces

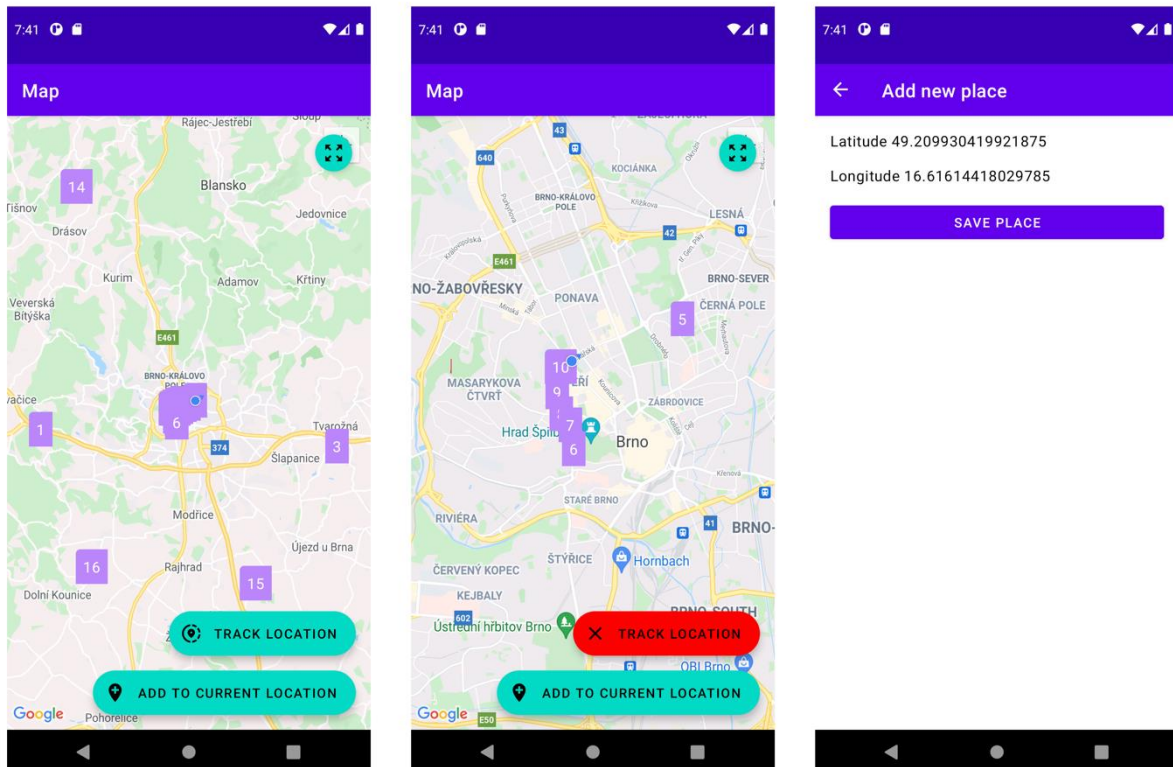


Fig. 26 Interface definition

Solution

The core of the application consists of 3 classes that take care of the operations around the map. The classes are:

- **LocationManager**
- **MapManager**
- **MarkerManager**

LocationManager

LocationManager class serves as a utility for accessing the device location. It uses the *FusedLocationProviderClient*, which allows developers to access location data. The location manager provides two main ways to access the location. The first is getting the current location. The second is

through location updates. Every X millisecond, the location of the device is provided and passed to *LocationCallback* class. The location updates also need to be stopped when the user does not need the location updates anymore.

```
class LocationManager(private val activity: FragmentActivity) {

    private var fusedLocationProviderClient: FusedLocationProviderClient

    init {
        fusedLocationProviderClient = FusedLocationProviderClient(activity)
    }

    @SuppressWarnings("MissingPermission")
    fun getCurrentLocation(listener: OnSuccessListener<Location>) {
        fusedLocationProviderClient.lastLocation
            .addOnSuccessListener(listener)
    }

    @SuppressWarnings("MissingPermission")
    fun startLocationUpdates(interval: Long, locationCallBack: LocationCallback) {
        val locationRequest = LocationRequest.create()
        locationRequest.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
        locationRequest.fastestInterval = interval
        locationRequest.interval = interval

        fusedLocationProviderClient.requestLocationUpdates(
            locationRequest,
            locationCallBack,
            Looper.getMainLooper()
        )
    }

    fun stopLocationUpdates(locationCallBack: LocationCallback) {
        fusedLocationProviderClient.removeLocationUpdates(locationCallBack)
    }
}
```

The method *startLocationUpdates* allows us to start location updates in the specified interval. Method *stopLocationUpdates* stop them.

MapManager

The second important class is *MapManager*. This class provides some basic operations with a map and also allows us to retrieve the last known map state using *DataStoreManager* class.

```
class MapManager {
    private val datastoreManager = DataStoreManager(MyApplication.appContext)
    fun getSavedMapPosition() = datastoreManager.cameraPosition

    suspend fun saveMapPosition(cameraPosition: CameraPosition){
        datastoreManager.saveMapState(cameraPosition)
    }
}
```

```

fun zoomToAllPlaces(googleMap: GoogleMap, places: List<Place>){
    val builder = LatLngBounds.Builder()
    for (place in places){
        builder.include(LatLng(place.latitude!!, place.longitude!!))
    }
    googleMap.animateCamera(
        CameraUpdateFactory.newLatLngBounds(builder.build(), 100))
}

fun zoomToLocation(googleMap: GoogleMap, location: LatLng){
    googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(location, 16.0f))
}
}

```

It contains two methods for the manipulation of the camera in the virtual map. The method *zoomToLocation* takes a specific location and animates the camera to that location. On the other hand, the method *zoomToAllPlaces* takes a list of places as a parameter and zooms the camera so that all the places are visible in the visible region of the map.

The second part of the class is retrieving and saving the map state. The virtual map camera has its position. The position is defined by the *CameraPosition* class. Method *getSavedMapPosition* returns the camera position previously saved. Method *saveMapPosition* save the camera position when the user leaves the *MapFragment* to navigate elsewhere.

MarkerManager

The last important class is *MarkerManager*. The first crucial part of the class is this line:

```
private val markers: HashMap<Long, Marker> = hashMapOf()
```

It allows us to save the markers added to the map. The important part about the objects added to the map is that once we add any object to the map, we cannot retrieve it later. So, the only way to work with them later is to save their instances. Saving them into the *HashMap* has many advantages, e.g. an easy way to retrieve a specific object based on its id.

The method *addMarkerToMap* is responsible for creating a *Marker* based on the *Place* object. Firstly, the *MarkerOptions* is created. *MarkerOptions* class defines the properties of the Marker. Each object added to the map has its options class, e.g. *Polygon* has *PolygonOptions*.

```

fun addMarkerToMap(context: Activity, map: GoogleMap, place: Place) {
    if (!markers.containsKey(place.id!!)) {
        val options = MarkerOptions()
        options.position(LatLng(place.latitude!!, place.longitude!!))
        options.icon(
            BitmapDescriptorFactory.fromBitmap(createCustomMarkerBitmap(
                context, place.id!!))
        )
        options.anchor(0.5f, 0.5f)
        options.draggable(true)
        val marker = map.addMarker(options)
        marker!!.tag = place.id
        markers.put(place.id!!, marker)
    }
}

```

```

    }
}

```

Once the *MarkerOptions* is defined, we add the marker to the map using the *addMarker* method. Now comes the most important part. We need to be able to distinguish markers from each other to identify them.

```
marker!!.tag = place.id
```

We can save the id of the place to the marker tag.

The last operation is saving the marker to our Hashmap.

```
markers.put(place.id!!, marker)
```

One very common operation with a marker is creating the marker from a custom layout. It means that the marker can look like anything we want. We can accomplish it using the *createCustomMarkerBitmap* method. The method inflates our layout and converts it to the Bitmap needed for marker creation.

```
private fun createCustomMarkerBitmap(context: Activity, id: Long): Bitmap {
    val markerView = LayoutInflater.from(context).inflate(
        R.layout.marker_layout, null)

    val textView = markerView.findViewById<TextView>(R.id.markerIdTV)
    textView.text = id.toString()

    val displayMetrics = DisplayMetrics()
    context.windowManager.defaultDisplay.getMetrics(displayMetrics)
    markerView.measure(displayMetrics.widthPixels, displayMetrics.heightPixels)
    markerView.layout(0, 0, displayMetrics.widthPixels,
        displayMetrics.heightPixels)

    markerView.buildDrawingCache()
    val bitmap = Bitmap.createBitmap(
        markerView.getMeasuredWidth(),
        markerView.getMeasuredHeight(),
        Bitmap.Config.ARGB_8888
    )
    val canvas = Canvas(bitmap)
    markerView.draw(canvas)
    return bitmap
}
```

The last method of the *MarkerManager* class is *removeMarker*. This method removes a specific marker from the map and also from our map of markers.

```
fun removeMarker(id: Long) {
    if (markers.containsKey(id)) {
        val marker = markers.get(id)
        marker!!.remove()
        markers.remove(id)
    }
}
```

MapFragment

The most important fragment of the application is *MapFragment*. The *MapFragment* contains an instance of the map, which is loaded at the fragmented startup. It all starts with the *initViews* method.

```
override fun initViews() {
    locationManager = LocationManager(requireActivity())
    mapManager = MapManager()
    markerManager = MarkerManager()

    val mapFragment = childFragmentManager.findFragmentById(R.id.map)
                                                    as SupportMapFragment?
    mapFragment?.getMapAsync(callback)
    binding.currentLocationFAB.setOnClickListener {
        if (PermissionUtility.checkLocationPermission(requireActivity())) {
            addPointToCurrentLocation()
        } else {
            PermissionUtility.requestLocationPermission(requireActivity(),
                                                         LOCATION_PERMISSION_REQUEST_CODE)
        }
    }

    binding.locationUpdatesFAB.setOnClickListener {
        if (!viewModel.trackingLocation) {
            if (PermissionUtility.checkLocationPermission(requireActivity())) {
                startLocationUpdates()
            } else {
                PermissionUtility.requestLocationPermission(requireActivity(),
                                                             LOCATION_UPDATES_PERMISSION_REQUEST_CODE)
            }
        } else {
            viewModel.trackingLocation = false
            setLocationUpdatesFAB()
            locationManager.stopLocationUpdates(locationCallback)
        }
    }
    binding.zoomToAllFab.setOnClickListener {
        if (viewModel.places.size > 0) {
            mapManager.zoomToAllPlaces(googleMap, viewModel.places)
        }
    }
}
```

Firstly, the manager classes are initialized. Then the map fragment is initialized. The map fragment is loaded asynchronously, which means we have to wait for it to load. Next, all the floating action buttons have their *onClick* methods set. The fragment contains three buttons:

- **currentLocationFAB** – places a marker at the current user location.
- **locationUpdatesFAB** – starts or stops location updates.
- **zoomToAllFab** – moves the map so that all the places are visible.

The next important part of the method is the *OnMapReadyCallback*.

```
@SuppressLint("MissingPermission")

private val callback = OnMapReadyCallback { googleMap ->
    this.googleMap = googleMap
    lifecycleScope.launch {
        mapManager.getSavedMapPosition().collect {
            googleMap.moveCamera(CameraUpdateFactory.newCameraPosition(it))
        }
    }

    googleMap.setOnMarkerDragListener(this)
    googleMap.setOnMarkerClickListener(this)
    googleMap.setOnMapClickListener(this)

    if (PermissionUtility.checkLocationPermission(requireActivity())) {
        googleMap.isMyLocationEnabled = true
    }

    viewModel.getAll().observe(viewLifecycleOwner, Observer {
        viewModel.places.clear()
        viewModel.places.addAll(it)
        for (place in it) {
            markerManager.addMarkerToMap(requireActivity(), googleMap, place)
        }
    })
}
```

In the beginning, when the map is loaded, it sets the previous position of the virtual camera. This way, the user never loses the camera position, and the map is zoomed in at the last position. Next, the appropriate listeners are set. We are using three of them:

- **setOnMarkerDragListener** – detects the drag event of the marker.
- **setOnMarkerClickListener** – detects the touch event on the marker.
- **setOnMapClickListener** – detect the touch event on the map.

The next part allows the map to use the current user location, and in the end, we load all places from the database. Please note that we are loading the places after the map is initialized.

In the previous step, we have set three listeners to the map. Let's look at their methods.

When the user finishes dragging the marker, it saves the new marker position to the database.

```
override fun onMarkerDragEnd(p0: Marker) {
    lifecycleScope.launch {
        val id = p0.tag as Long
        val place = viewModel.findById(id)
        place.latitude = p0.position.latitude
        place.longitude = p0.position.longitude
        viewModel.update(place)
    }
}
```

When the user clicks on the marker, it opens an *AlertDialog*, which ask the user to delete the marker. If the user agrees to delete it, it removes the marker from the map and deletes it from the database.


```

override fun onMarkerClick(p0: Marker): Boolean {
    val builder = AlertDialog.Builder(requireContext())
    val id = p0!!.tag as Long
    val dialog = builder.setTitle(getString(R.string.delete_dialog_title))
        .setMessage(getString(R.string.delete_dialog_message))
        .setPositiveButton(getString(R.string.delete), object :
            DialogInterface.OnClickListener{
                override fun onClick(dialog: DialogInterface?, which: Int) {
                    dialog?.dismiss()
                    lifecycleScope.launch {
                        viewModel.delete(id)
                    }
                    markerManager.removeMarker(id)
                }
            })
        .setNegativeButton(getString(R.string.cancel), object :
            DialogInterface.OnClickListener{
                override fun onClick(dialog: DialogInterface?, which: Int) {
                    dialog?.dismiss()
                }
            })
        .create()
    dialog.show()
    return true
}

```

The last `onClick` method is `onMapClick`. When the user touches the map, it opens a new fragment to add a new place to the database.

```

override fun onMapClick(p0: LatLng) {
    val action = MapFragmentDirections.actionMapToAddPlace(
        p0!!.latitude.toFloat(), p0.longitude.toFloat())
    findNavController().navigate(action)
}

```

Now, let's look at other methods in the *MapFragment*. When the view is destroyed, the map position is saved so it can be retrieved the next time the *MapFragment* is active.

```

override fun onDestroyView() {
    super.onDestroyView()
    lifecycleScope.launch {
        mapManager.saveMapPosition(googleMap.cameraPosition)
    }
}

```

The method starts location updates and also manages the state of the Floating action button.

```

private fun startLocationUpdates() {
    viewModel.trackingLocation = true
    setLocationUpdatesFAB()
    locationManager.startLocationUpdates(10000, locationCallback)
}

```

Opens the *AddPlaceFragment* so that a user can add a place to his current position.

```

private fun addPointToCurrentLocation() {
    locationManager.getCurrentLocation(object : OnSuccessListener<Location>{

```

```
        override fun onSuccess(p0: Location?) {
            val action = MapFragmentDirections.actionMapToAddPlace(
                p0!!.latitude.toFloat(), p0.longitude.toFloat()
            )
            findNavController().navigate(action)
        }
    })
}
```

The two last important methods are *onPause* and *onResume*. In *onPause*, when the fragment is going to the background, the location updates, if they are performed, are stopped. In *onResume*, if the location updates were active, they are started again. This way, the application behaves consistently during location updates.

```
override fun onPause() {
    super.onPause()
    if (viewModel.trackingLocation) {
        locationManager.stopLocationUpdates(locationCallback)
    }
}

override fun onResume() {
    super.onResume()
    if (viewModel.trackingLocation) {
        startLocationUpdates()
    }
}
```

To-do Application with Maps

The presented example shows a mobile application for Android. The mobile application serves as a productivity tool and allows users to insert their tasks. Users can associate tasks with dates and locations. Users can also modify the tasks and delete them. When the user is done with the task, it can be marked as resolved using the checkbox.

Recommended Number of Developers

Individual.

Available Solutions

There are a lot of similar solutions. To name some:

- Todoist (<https://play.google.com/store/apps/details?id=com.todoist>)
- Google Tasks (<https://play.google.com/store/apps/details?id=com.google.android.apps.tasks>)
- Any.do (<https://play.google.com/store/apps/details?id=com.anydo>)

However, these solutions are much more advanced and use synchronization with the server.

Requirements

The application will be created in Android Studio. Here is an installation guide for Windows (<https://developer.android.com/studio/install>). The application is developed using the Kotlin programming language. The development requires a basic knowledge of Object-oriented programming.

Functional Requirements

- a) The application will show a list of current tasks of the user in the list.
- b) The user can add a new task.
- c) The user can modify the existing task.
- d) The user can delete the existing task.
- e) Each task has a description, date and location.
- f) The location is selected using Google Maps Android SDK.

Non-functional Requirements

- a) Android 6.0 and higher.
- b) All tasks must be available after the application is closed and again opened.
- c) MVVM architecture.
- d) Use of Navigation Component for navigation.

Application Design

Technology and Architecture Selection

The application is written in MVVM architecture, which is a recommended architecture for Android application development. The application uses Navigation Component with fragments to show the user interface. As a database, the Room library is used.

Data Model

The data model for the application is very simple. It contains one class *Task*. The task has id, text, date, done indicator and latitude and longitude.

Task	
id	Long
text	String
date	Long
done	Boolean
latitude	Double
longitude	Double

Fig. 27 Data model

User Interfaces

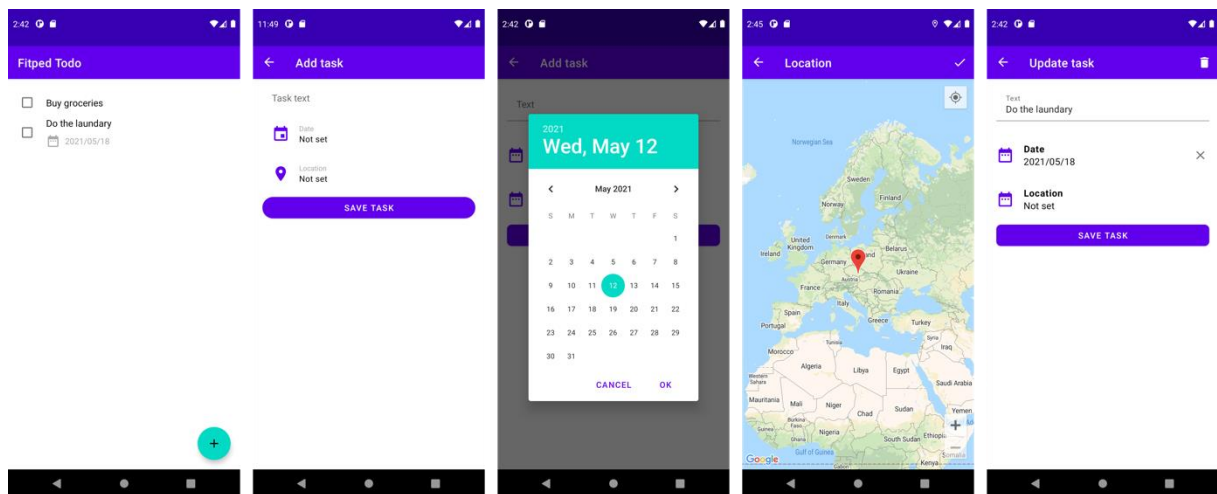


Fig. 28 Interface definition

Solution

Design definition

The first step is the creation of a user interface with navigation. The user interface uses Navigation Component for the navigation.

The basis of navigation is several elements, which together create smooth navigation. It consists of three parts:

- **element fragment in layout** - It serves as a container for other fragments.
- **navigation chart** - NavGraph. Defines from where the user is navigated and which fragment will be displayed, for example, after clicking on the button.
- **fragments** - specific classes of fragments, which we will navigate between.

Fragment

When creating a new project, we can find the fragment element in the *content_main.xml* file. This element serves as a container for all fragments displayed in this activity.

```
<fragment
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />
```

It has two important features:

- **name** - the name of the class that serves as a container for fragments.
- **navGraph** - a link to the navigation graph, i.e. a file with the definition of navigation.

Navigation chart

Another element is the navigation chart. You can find it in the *layout/navigation* folder. You will usually find one file (*nav_graph.xml*) in this folder, but there may be more, especially if you use multiple navigation sources.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/taskListFragment">

    <fragment
        android:id="@+id/taskListFragment"
        android:name="sk.example.fitped.todo.fitpedtodo.ui.fragments.TaskListFragment"
        android:label="@string/app_name"
        tools:layout="@layout/fragment_task_list">

        <action
            android:id="@+id/action_list_to_add"
            app:destination="@id/addTaskFragment" />

    </fragment>

    <fragment
        android:id="@+id/addTaskFragment"
        android:name="sk.example.fitped.todo.fitpedtodo.ui.fragments.AddTaskFragment"
        tools:layout="@layout/fragment_add_task">
        <action
            android:id="@+id/action_add_to_map"
            app:destination="@id/mapsFragment" />
    </fragment>
```

```

<fragment
    android:id="@+id/mapsFragment"
    android:name="sk.example.fitped.todo.fitpedtodo.ui.fragments.MapsFragment"
    android:label="@string/location"
    tools:layout="@layout/fragment_maps">
</fragment>
</navigation>

```

You can see that:

- There are three fragments, *TaskListFragment*, *AddTaskFragment* and *MapsFragment*. Each fragment has a name, a label, an id, and a layout (which layout to create when creating the fragment).
- Two fragments have actions. The action represents the action to take, that is, the navigation that can be performed. *TaskListFragment* can navigate to *AddTaskFragment*, and *AddTaskFragment* can navigate to *MapsFragment*.
- Action always contains its id, destination.

Caution: If you want to navigate somewhere where it is not defined, the application will crash.

To avoid casting problems with navigation, it is a good idea to use the *Safe Args Gradle* plugin. This will ensure the generation of special classes that will be used for navigation. The generation and transmission of arguments are automated, and errors can no longer occur.

The first step is to define the attributes directly in the navigation chart. The *AddTaskFragment* will look like this:

```

<fragment
    android:id="@+id/addTaskFragment"
    android:name="sk.example.fitped.todo.fitpedtodo.ui.fragments.AddTaskFragment"
    tools:layout="@layout/fragment_add_task">
    <argument
        android:name="id"
        app:argType="long"
        android:defaultValue="-1L"
    />

    <action
        android:id="@+id/action_add_to_map"
        app:destination="@id/mapsFragment" />
</fragment>

```

In the code, the navigation to adding will look like this:

```

val action = TaskListFragmentDirections.actionListToAdd()
findNavController().navigate(action)

```

and the navigation to update will look like this:

```

val action = TaskListFragmentDirections.actionListToAdd(
    taskList.get(holder.adapterPosition).id!!)
findNavController().navigate(action)

```

A class with a *Directions* suffix was generated. It contains a method representing action, and it requires the task id as an argument.

Be careful; if the *Directions* class is not generated, try the Build/Rebuild project. If you want more arguments, just add them to the navigation chart.

Creation of the database

There are several ways to store data on a mobile device, and a database is one of them. Android uses the SQLite database. SQLite is basically a traditional SQL database but has a smaller number of functions. This means that some more advanced operations cannot be performed in it.

In the past, when it was necessary to work with a database, it was accessed directly. This means that SELECTs were performed directly on the database, data were manually INSERTED, etc....

Fortunately, these times are gone. The disadvantage was that great emphasis was placed on the accuracy of the queries. A small mistake in the name was enough, and everything was wrong. It often took a very long time to find such a mistake.

Room library

The Room library has become a lifeline for developers. The library represents an abstract layer above the database and allows you to access the database much more efficiently and, above all, more easily.

The database using the Room library consists of three parts:

- **Database class** - this class inherits from the *RoomDatabase* class and contains a list of tables (database entities). It also contains a version of the database. More on that later.
- **Entities** - an entity is basically a class that we store in a database, e.g. Person, Car.
- **Dao** - an interface that contains methods for working with a specific entity.

Entity

An entity represents a single database table. The table has columns that have their own data types. Let's have the *Task* class. The task has a name, id and a couple more columns. The id is the primary key. We mark the entity only with the annotation *@Entity*. The individual columns are then annotated using *@ColumnInfo*. Annotation *@ColumnInfo* may not even be there. In this case, the variable name is named differently.

```
@Entity(tableName = "tasks")
data class Task(@ColumnInfo(name = "text") var text: String) {

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    var id: Long? = null

    @ColumnInfo(name = "date")
    var date: Long? = null

    @ColumnInfo(name = "done")
    var done: Boolean = false

    @ColumnInfo(name = "latitude")
```

```

var latitude: Double? = null

@ColumnInfo(name = "longitude")
var longitude: Double? = null

fun hasLocation(): Boolean = latitude != null && longitude != null
}

```

Dao (Data Access Object)

Dao is a design pattern used to specify a unified interface through which we access the database. With the help of the Room library, this means that we can easily work with the database. All you have to do is mark the interface with the `@Dao` annotation. Note that the interface has no implementation. The room takes care of that for you.

```

@Dao
interface TasksDao {

    @Query("SELECT * FROM tasks")
    fun getAll(): LiveData<MutableList<Task>>

    @Query("SELECT * FROM tasks WHERE id = :id")
    suspend fun findById(id : Long): Task

    @Insert
    suspend fun insert(task: Task): Long

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)

    @Query("UPDATE tasks SET done = :done WHERE id = :id")
    suspend fun markAsDone(id: Long, done: Boolean)
}

```

Database class

The last part consists of the database class itself. It is responsible for defining the database. The Singleton design pattern is commonly used to access the database. Thanks to it, we always access one single instance of the database.

```

@Database(entities = [Task::class], version = 1, exportSchema = true)
abstract class TasksDatabase : RoomDatabase() {
    abstract fun tasksDao(): TasksDao
    companion object {
        private var INSTANCE: TasksDatabase? = null

        fun getDatabase(context: Context): TasksDatabase {
            if (INSTANCE == null) {
                synchronized(TasksDatabase::class.java) {
                    if (INSTANCE == null) {
                        INSTANCE = Room.databaseBuilder(
                            context.applicationContext,
                            TasksDatabase::class.java, "tasks_database"
                        )
                    }
                }
            }
            return INSTANCE!!
        }
    }
}

```



```

        }.build()
    }
}
return INSTANCE!!
}
}
}

```

Each database has a version. The version tells you what specific state the database is in this time. If the model class changes, the migration needs to be performed. More here:

<https://developer.android.com/training/data-storage/room/migrating-db-versions>.

Because we are using the MVVM architecture, we also need to define the Repository classes, which serves as a mediator between ViewModel and Dao class.

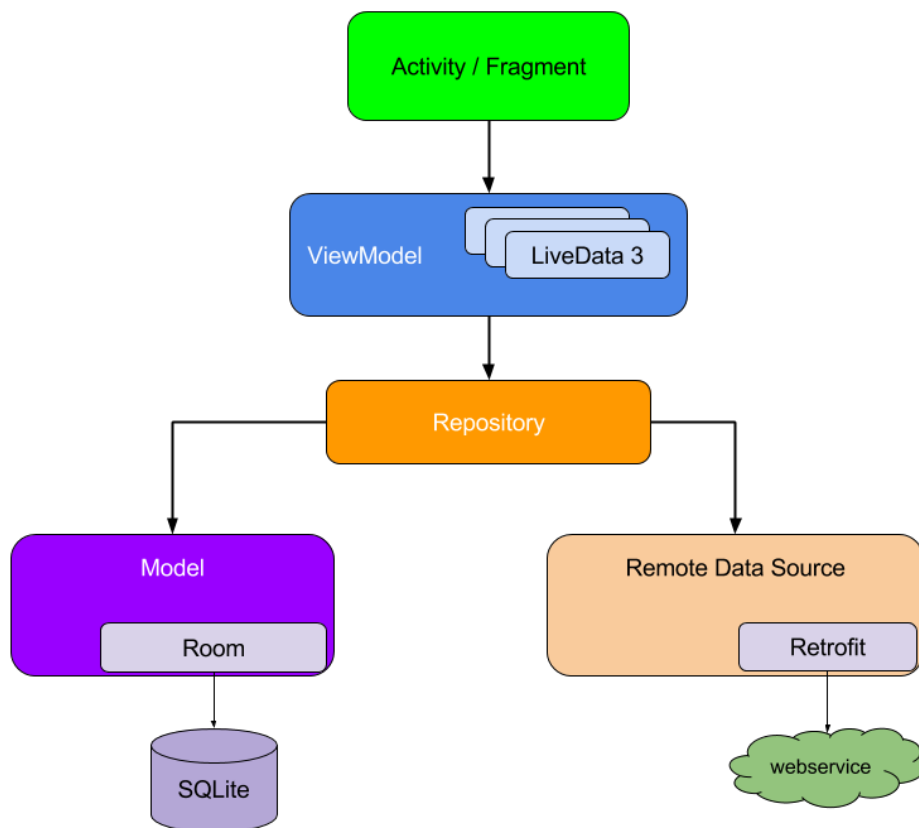


Fig. 29 MVVM architecture. Source: <https://developer.android.com/jetpack/guide>

The first part is the repository interface. It defines public methods which will be used to access the database.

```

interface ITasksLocalRepository {
    fun getAll(): LiveData<MutableList<Task>>
    suspend fun findById(id : Long): Task
    suspend fun insert(task: Task): Long
    suspend fun update(task: Task)
    suspend fun delete(task: Task)
}

```

```

suspend fun markAsDone(id: Long, done: Boolean)
}

```

The second part is the implementation of the interface.

```

class TasksLocalRepositoryImpl (private val tasksDao: TasksDao) :
ITasksLocalRepository {

    private var getAllLiveData: LiveData<MutableList<Task>> = tasksDao.getAll()

    override fun getAll(): LiveData<MutableList<Task>> {
        return getAllLiveData
    }

    override suspend fun findById(id: Long): Task {
        return tasksDao.findById(id)
    }

    override suspend fun insert(task: Task): Long {
        return tasksDao.insert(task)
    }

    override suspend fun update(task: Task) {
        tasksDao.update(task)
    }

    override suspend fun delete(task: Task) {
        tasksDao.delete(task)
    }

    override suspend fun markAsDone(id: Long, done: Boolean) {
        tasksDao.markAsDone(id, done)
    }
}

```

The repository class is then used in the ViewModel class.

```

class TaskListViewModel(private val taskRepository: TasksLocalRepositoryImpl) :
BaseViewModel() {

    fun getAll(): LiveData<MutableList<Task>> {
        return taskRepository.getAll()
    }

    suspend fun markAsDone(id: Long, done: Boolean){
        taskRepository.markAsDone(id, done)
    }
}

```

This allows us complete access to the database through the repository design pattern.

Implementation of the list of tasks

If we want to work with lists, we basically have two options:

- *ListView*
- *RecyclerView*

These are classes in the OS that can display a list of items. *ListView* is an older implementation, which we will not deal with. Although it still has its use (e.g. for creating Widgets), it is not as powerful as *RecyclerView*.

What do we need for the list?

1. *RecyclerView*. We define the class directly in the XML layout.
2. One list item. A class contains data.
3. Definition of the appearance of each line (layout)
4. List of elements
5. *ViewHolder* - class containing links to View (e.g. *TextView*)
6. *Adapter* - the class responsible for displaying the list.
7. *LayoutManager* - class responsible for the way the list is displayed (vertical, horizontal)
8. Put everything together

Class to display the list

Insert the *RecyclerView* class directly in the place where we want to display the list. E.g. to the *content_main.xml* file. The class has an id, thanks to which we identify it in the code. That's enough for now.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/base_indentation"
    android:paddingBottom="@dimen/base_indentation"
    android:clipToPadding="true"/>
```

One list item

This is a class that maintains data. For example, our *Task* class.

Defines the appearance of each line

In the *res/layout* folder we will create a new file *row_task_list.xml*.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <CheckBox
        android:id="@+id/checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginStart="@dimen/base_indentation"
```

```

/>

<LinearLayout
    android:id="@+id/taskRowContent"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@id/checkbox"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginEnd="@dimen/base_indentation"
>

    <TextView
        android:id="@+id/taskName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ellipsize="end"
        android:maxLines="2"
        android:textColor="@android:color/black"
        android:textAppearance="@style/TextAppearance.AppCompat.Subhead"/>

    <LinearLayout
        android:id="@+id/dateContainer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_marginTop="@dimen/half_indentation"
    >

        <ImageView
            android:layout_width="24dp"
            android:layout_height="24dp"
            android:contentDescription="@null"
            android:layout_gravity="center_vertical"
            android:src="@drawable/ic_date"
            app:tint="@android:color/darker_gray" />

        <TextView
            android:id="@+id/taskDate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:layout_marginStart="@dimen/half_indentation"
            android:textAppearance="@style/TextAppearance.AppCompat.Body1"
            android:textColor="@android:color/darker_gray"
        />

    </LinearLayout>
</LinearLayout>
</androidx.constraintlayout.widget.ConstraintLayout>

```

List of elements

This is a specific list of elements. Directly as an attribute of our fragment. Note two details:

- There is only one list. We will use it everywhere for this list of items.

- The list is initially initialized. It can only be in the empty/with elements state. Thanks to that, we won't have to deal with NPE (*NullPointerException*)

```
private val taskList: MutableList<Task> = mutableListOf()
private lateinit var layoutManager: LinearLayoutManager
private lateinit var tasksAdapter: TasksAdapter
```

ViewHolder

ViewHolder is a class serving as a container for our Views, which are defined in the layout line. We must keep all the Views that we declare for the row in the *ViewHolder*.

```
inner class TaskViewHolder(val binding: RowTaskListBinding) :
    RecyclerView.ViewHolder(binding.root)
```

ViewHolder is often declared as part of an adapter. But it can also be in a separate file. Each ViewHolder inherits from the RecyclerView.ViewHolder class, which needs a View to initialize. This represents a single line (the entire initialized layout).

Adapter

The adapter is the most important class for the list. It is a class where we declare what should happen when rendering a list.

```
inner class TasksAdapter : RecyclerView.Adapter<TasksAdapter.TaskViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup,
                                    viewType: Int): TaskViewHolder {
        return TaskViewHolder(RowTaskListBinding.inflate(
            LayoutInflater.from(parent.context), parent, false))
    }

    override fun onBindViewHolder(holder: TaskViewHolder, position: Int) {
        val task = taskList.get(position)
        holder.binding.taskName.text = task.text
        task.date?.let {
            holder.binding.dateContainer.visibility = View.VISIBLE
            holder.binding.taskDate.text = DateUtils.getDateString(it)
        }?:kotlin.run {
            holder.binding.dateContainer.visibility = View.GONE
        }
        holder.binding.checkbox.isChecked = task.done
        holder.binding.checkbox.setOnCheckedChangeListener(
            object : CompoundButton.OnCheckedChangeListener {
                override fun onCheckedChanged(p0: CompoundButton?, p1: Boolean) {
                    lifecycleScope.launch {
                        viewModel.markAsDone(taskList.get(holder.adapterPosition).id!!,
                            !task.done)
                    }
                }
            })
        holder.binding.root.setOnClickListener {
            val action = TaskListFragmentDirections.actionListToAdd(
                taskList.get(holder.adapterPosition).id!!)
            findNavController().navigate(action)
        }
    }
}
```

```

    }
}

override fun getItemCount() = taskList.size
inner class TaskViewHolder(val binding: RowTaskListBinding) :
    RecyclerView.ViewHolder(binding.root)
}

```

The adapter has three methods:

- *onCreateViewHolder* - creates an instance of the *ViewHolder* class. It's called once.
- *getItemCount* - returns the number of list items. The adapter must always have a finite number of elements. It is not possible not to know how many elements the list currently has.
- *onBindViewHolder* - the most important method. Here we influence what each line will look like. We display text, set View states. This method is called for each line whenever it is displayed. Even if we move to the end of the list and return to the beginning.

LayoutManager

The layout manager is responsible for calculating the space that each row will have on the screen. The basic implementation is the *LinearLayoutManager* class, which is either a vertical or horizontal list.

Put everything together

The last step is putting everything together. What we need to do:

- initialize the adapter
- initialize the *LayoutManager*
- fill the field (we can do it only after initialization, basically at any time).

Let's see what our fragment will look like:

```

tasksAdapter = TasksAdapter()
layoutManager = LinearLayoutManager(requireContext())
binding.recyclerView.layoutManager = layoutManager
binding.recyclerView.adapter = tasksAdapter

viewModel.getAll().observe(viewLifecycleOwner, object : Observer<MutableList<Task>>
{
    override fun onChanged(t: MutableList<Task>?) {
        t?.let {
            val diffCallback = TaskDiffUtils(taskList, t)
            val diffResult = DiffUtil.calculateDiff(diffCallback)
            diffResult.dispatchUpdatesTo(tasksAdapter)
            taskList.clear()
            taskList.addAll(t)
        }
    }
})

```

DiffUtils

But what if we need to refresh the list. The best way is using the *DiffUtils*. In the past, programmers created this themselves. Google responded by creating the *DiffUtils* class. This class automatically detects what has changed and refreshes the list.

```

inner class TaskDiffUtils(private val oldList: MutableList<Task>,
    private val newList: MutableList<Task>) : DiffUtil.Callback() {
    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int):
        Boolean {
        return oldList[oldItemPosition].id == newList[newItemPosition].id
    }

    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int):
        Boolean {
        return oldList[oldItemPosition].text == newList[newItemPosition].text
            && oldList[oldItemPosition].date == newList[newItemPosition].date
    }

    override fun getOldListSize() = oldList.size
    override fun getNewListSize() = newList.size
}

```

Using the *areContentsTheSame* and *areItemsTheSame* methods, we say whether the lists or the items at the same position are identical. If so, it is not necessary to refresh them. All you have to do is identify each line; for example, we will use the name here.

Adding and modifying the task

The fragment for *AddTaskFragment* will be used to add a new task but also to update existing. The reason is simple. Most of the code will be the same. The basic declaration of the fragment looks like this.

```

class AddTaskFragment : BaseFragment<FragmentAddTaskBinding,
    AddTaskViewModel>(AddTaskViewModel::class) {

    private val arguments: AddTaskFragmentArgs by navArgs()
    override val bindingInflater: (LayoutInflater) -> FragmentAddTaskBinding
        get() = FragmentAddTaskBinding::inflate
    override fun initView() {
    }
}

```

However, it also contains a lot of methods. Let's go through them one by one.

The method *initViews* is responsible for the initialization of the fragment. In the beginning, the id is loaded from arguments. If the id is not null, it means we are updating the task. The data are loaded from the database, and the layout views are filled. If the id is null, we are adding a new task.

The last two parts of the method process the location send from the *MapsFragment*.

```

override fun initView() {
    viewModel.id = if (arguments.id != -1L) arguments.id else null

    viewModel.id?.let {
        setToolbarTitle(getString(R.string.update_task))
        setHasOptionsMenu(true)
        lifecycleScope.launch {
            viewModel.task = viewModel.findById(it)
        }.invokeOnCompletion {
            fillLayout()
        }
    }
}

```

```

}?: kotlin.run {
    setToolbarTitle(getString(R.string.add_task))
    fillLayout()
}
setInteractionListeners()

binding.saveButton.setOnClickListener {
    saveTask()
}

findNavController().currentBackStackEntry?.savedStateHandle
    ?.getLiveData<Double>(LiveDataConstants.LATITUDE)?.observe(
        viewLifecycleOwner, androidx.lifecycle.Observer {
            viewModel.task.latitude = it
            setLocation()
            findNavController().currentBackStackEntry?.savedStateHandle?.
                remove<Double>(LiveDataConstants.LATITUDE)
        })

findNavController().currentBackStackEntry?.savedStateHandle
    ?.getLiveData<Double>(LiveDataConstants.LONGITUDE)?.observe(
        viewLifecycleOwner, androidx.lifecycle.Observer {
            viewModel.task.longitude = it
            setLocation()
            findNavController().currentBackStackEntry?.savedStateHandle?.
                remove<Double>(LiveDataConstants.LONGITUDE)
        })
}

```

Method *fillLayout* is responsible for setting all values to the Views. We set the text, the date and the location.

```

private fun fillLayout() {
    viewModel.task.text.let {
        binding.taskName.text = it
    }
    setDate()
    setLocation()
}

```

The next part is setting interaction listeners. It means managing the buttons and their clicks.

```

private fun setInteractionListeners() {
    binding.dateInfoView.setOnClickListener(object : View.OnClickListener {
        override fun onClick(p0: View?) {
            openDatePicker()
        }
    })

    binding.dateInfoView.setOnClearButtonListener(object : View.OnClickListener {
        override fun onClick(p0: View?) {
            binding.dateInfoView.setValue(getString(R.string.not_set))
            binding.dateInfoView.hideClearButton()
            viewModel.task.date = null
        }
    })

    binding.mapInfoView.setOnClearButtonListener(object : View.OnClickListener {

```



```

        override fun onClick(v: View?) {
            viewModel.task.latitude = null
            viewModel.task.longitude = null
            setLocation()
        }

    })

    binding.mapInfoView.setOnClickListener({
        var direction: NavDirections? = null
        if (viewModel.task.hasLocation()) {
            direction = AddTaskFragmentDirections.actionAddToMap(
                viewModel.task.latitude!!.toFloat(),
                viewModel.task.longitude!!.toFloat()
            )
        } else {
            direction = AddTaskFragmentDirections.actionAddToMap()
        }
        findNavController().navigate(direction)
    })
}

```

A very important part is saving the task to the database. We are either inserting a new task or updating the existing one. First, the input text is checked. If it is not empty, the saving can proceed.

```

private fun saveTask() {
    val text = binding.taskName.text
    if (!text.isEmpty()) {
        lifecycleScope.launch {
            viewModel.task.text = text
            viewModel.id?.let {
                viewModel.update(viewModel.task)
            }?: kotlin.run {
                viewModel.insert(viewModel.task)
            }
        }.invokeOnCompletion {
            finishCurrentFragment()
        }
    } else {
        binding.taskName.setError(getString(R.string.fill_in_the_text))
    }
}

```

The last important method is *openDatePicker*. This method allows users to choose the date. Firstly, the currently selected date is sent to the dialogue using the *Calendar* class. Then, the dialogue is opened, and the result date is saved to the *ViewModel*.

```

private fun openDatePicker() {
    val calendar = Calendar.getInstance()
    viewModel.task.date?.let {
        calendar.timeInMillis = it
    }

    val y = calendar.get(Calendar.YEAR)
    val m = calendar.get(Calendar.MONTH)
    val d = calendar.get(Calendar.DAY_OF_MONTH)

    val datePickerDialog = DatePickerDialog(requireContext(),

```

```

        object : DatePickerDialog.OnDateSetListener {
            override fun onDateSet(view: DatePicker?, year: Int,
                                   monthOfYear: Int, dayOfMonth: Int) {
                viewModel.task.date = DateUtils.getUnixTime(year, monthOfYear, dayOfMonth)
                setDate()
            }
        }, y, m, d)
        datePickerDialog.show()
    }
}

```

Custom Views for the adding of tasks

The best way to optimize the user interface is to create your own *View*. It means making a child of the *View* class (or a child of a child of the *View* class).

Basically, we make our own element for Layouts.

Creating your own *View* is the key to creating an effective UI. Imagine that you have an element that will be repeated over and over in many places in the application. It is not effective to copy the same element to different places. Much better is to encapsulate it inside the view.

When creating our own *View*, we need three things:

- **Layout** - layout definition. What the element will look like.
- **Class** - code of our *View*.
- **Attribute definitions** - elements in layouts have attributes. We can also add our own elements.

Layout

The layout is simple. This is a separate layout file.

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/textInputLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:errorEnabled="true"
        >

        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/textInputEditText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:maxLength="500"
            android:inputType="text"
            android:maxLines="1"
            android:background="@android:color/transparent"

```

```

        />

        </com.google.android.material.textfield.TextInputLayout>
    </androidx.constraintlayout.widget.ConstraintLayout>

```

Definition of attributes

The definition of attributes is another part of creating a custom view. In the *values* folder, we will create a new file *values_text_input.xml*. We define attributes in it:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="TextInputView">
        <attr name="hint" format="string"/>
    </declare-styleable>
</resources>

```

The attribute always has a name and a format. The format, in this case, is a string, but it can also be colour, reference and more. The name is the name of the class.

Class

Creating a class is just about creating a new class file. Our *View* inherits in *FrameLayout*.

```

class TextInputView @JvmOverloads constructor(
    context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0
) : FrameLayout(context, attrs, defStyleAttr) {

    init {
        init(context, attrs, defStyleAttr)
    }

    private lateinit var binding: ViewTextInputBinding
    internal var text: String
        get() = binding.textInputEditText.text.toString()
        set(text) {
            binding.textInputEditText.setText(text)
        }

    private fun init(context: Context, attrs: AttributeSet?, defStyle: Int?) {
        binding = ViewTextInputBinding.inflate(LayoutInflater.from(context), this,
            true)

        if (attrs != null && defStyle != null) {
            loadAttributes(attrs, defStyle)
        }
    }

    private fun loadAttributes(attrs: AttributeSet, defStyle: Int?) {
        val a = context.obtainStyledAttributes(
            attrs,
            R.styleable.TextInputView, 0, 0
        )
        val hint = a.getString(R.styleable.TextInputView_hint)
        binding.textInputLayout.setHint(hint)
        a.recycle()
    }
}

```

```
fun setError(error: String?) {
    binding.textInputLayout.error = error
}
}
```

Notice a few things in the code:

- **Constructors** - View is initialized via three constructors. These are created automatically using `@JvmOverloads`.
- **We use ViewBinding.**
- In the `loadAttributes()` method, we load the attributes. Attributes can be of different types by definition.
- After finishing the work with attributes, we have to call the `recycle()` method.

Use in the application

When we finish our own *View*, we can use it in the layout file.

```
<sk.example.fitped.todo.fitpedtodo.views.TextInputView
    android:id="@+id/taskName"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/base_indentation"
    android:layout_marginEnd="@dimen/base_indentation"
    android:layout_marginTop="@dimen/base_indentation"
    app:hint="Text"/>
```

Creating your own *View* has one undeniable advantage. You are preparing for the future. In many cases, you know exactly what the UI will look like, but often not. It can happen that you have a text box in many places in the application, and suddenly both the appearance and its logic change. Thanks to its own *View*, making changes is very simple and fast.

BaseClasses

One of the principles of development is code reusability. If we keep doing the same operation over and over again, it's easier to put them in a separate class, classes and then just use those.

On Android, you will come across an approach that advises you to define the base classes of common objects, so-called Base Classes. There are usually some operations that are repeating. In our application, they are:

- **Fragment** - fragment initialization, ViewBinding initialization, ViewModel initialization
- **Activity** - initialization of ViewBinding, initialization of the navigation component
- **ViewHolder** - initialization of ViewBinding
- **ViewModel** - maintaining the state of activity after rotating the device
- **View** - initialization of own View

The point of these classes is simple. If something happens again, let's put it in them. Imagine that you always need to set a toolbar title in a snippet. Therefore, you will make a base fragment, others will inherit from it, and each fragment will be able to control the toolbar.

There are many forms of these classes. We will look at one specific, namely a fragment.

```

abstract class BaseFragment<B : ViewBinding,
    VM : ViewModel>(viewModelClass: KClass<VM>) : Fragment(){

    protected abstract val bindingInflater: (LayoutInflater) -> B
    private var baseBinding: ViewBinding? = null

    protected val binding: B
        get() = baseBinding as B

    val viewModel: VM by lazy { getViewModel(null, viewModelClass) }

    abstract fun initViews()

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?): View? {
        baseBinding = bindingInflater(inflater)
        initViews()
        return baseBinding!!.root
    }

    override fun onDestroy() {
        super.onDestroy()
        baseBinding = null
    }

    fun finishCurrentFragment(){
        requireActivity().runOnUiThread {
            hideKeyboard()
            Navigation.findNavController(binding.root).popBackStack()
        }
    }

    fun hideKeyboard() {
        activity?.let {
            val inputManager: InputMethodManager = it.getSystemService(
                Context.INPUT_METHOD_SERVICE) as InputMethodManager
            val currentFocusedView: View? = requireActivity().currentFocus
            if (currentFocusedView != null) {
                inputManager.hideSoftInputFromWindow(
                    currentFocusedView.getWindowToken(),
                    InputMethodManager.HIDE_NOT_ALWAYS
                )
            }
        }
    }

    fun setToolbarTitle(title: String){
        (requireActivity() as AppCompatActivity?)!!.supportActionBar!!.
            title = title
    }
}

```

Attention: This class uses the concept of Dependency injection in the form of the `getViewModel` method.

You always need *ViewBinding* and *ViewModel* in your own fragments. *BaseFragment* is a generic class to which we can send any *ViewModel* and any *ViewBinding*, and it always initializes them. It also has an abstract *initViews* method that will be called when *onCreateView()* is called.

And last but not least, it will take care of the correct removal of the *ViewBinding*.

Use in code

When we create a new fragment, we just inherit from the *BaseFragment*. The huge advantage of *BaseFragment* lies in its reusability, code clarity and, last but not least, extensibility.

```
class AddTaskFragment : BaseFragment<FragmentAddTaskBinding,
                                AddTaskViewModel>(AddTaskViewModel::class) {

    override val bindingInflater: (LayoutInflater) -> FragmentAddTaskBinding
        get() = FragmentAddTaskBinding::inflate

    override fun initViewViews() {
    }
}
```

Dependency injection

Dependency injection is a design pattern that allows us to instantiate a class outside of where we need it and then inject it into that place.

Dependency injection has a huge number of benefits. Thanks to it:

- the code becomes cleaner,
- the code becomes reusable,
- is easy to refactor and make changes,
- and is easier to test.

Dependency injection extremely simplifies the entire code. We no longer have to create new instances of various other classes in each class, but we simply pass them where the instance is needed. This beautifully solves various dependency issues.

There are many frameworks for DI. We will use Koin.

Koin

The Koin framework works with modules. Let's look at the basic use. According to the MVVM architecture, we have *ViewModels*. *ViewModels* contain *Repositories* and are in fragments. *Repositories* contain *Dao objects*. To get to *Dao*, we need a database object. As can be seen,

We will create four modules:

- **DatabaseModule** - a module that returns an instance of the database.
- **DaoModule** - a module that returns an instance of Dao classes.
- **RepositoryModule** - module responsible for returning Repository classes.
- **ViewModelModule** - a module that returns instances of ViewModels.

For Koin to work, we need to add it in the gradle file:

```
implementation "org.koin:koin-android:2.2.2"
implementation "org.koin:koin-android-scope:2.2.2"
implementation "org.koin:koin-android-viewmodel:2.2.2"
implementation "org.koin:koin-android-ext:2.2.2"
```

Subsequently, we can create modules. We always create an ordinary file without any class declaration.

DatabaseModule

```
val databaseModule = module {
    fun provideDatabase(): TasksDatabase =
        TasksDatabase.getDatabase(TaskApplication.appContext)

    single {
        provideDatabase()
    }
}
```

In the database module, we create a method that returns an instance of the database. In order to inject it somewhere, we define a scope *single* in which the method we created is called. And now, if we want an instance of the database somewhere, we can inject it.

For DI to work, we need to define it in our Application class:

```
class TaskApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        startKoin {
            appContext = applicationContext!!
            androidLogger(Level.ERROR)
            androidContext(appContext)
            modules(
                databaseModule,
                viewModelModule,
                daoModule,
                repositoryModule
            )
        }
    }

    companion object {
        @SuppressWarnings("StaticFieldLeak")
        lateinit var appContext: Context
        private set
    }
}
```

DaoModule

Dao module works with a database. We have a method for obtaining the Dao class. However, we need a database instance in the constructor. The *get()* method solves this for us. This is really enough, and the entire database instance is sent to the *provideTasksDao()* method.

```
val daoModule = module {
    fun provideTasksDao(database: TasksDatabase): TasksDao = database.tasksDao()
    single {
        provideTasksDao(get())
    }
}
```

RepositoryModule

Another module is the Repository. It is responsible for instantiating the repository class.

```
val repositoryModule = module {
    fun provideLocalTaskRepository(dao: TasksDao): TasksLocalRepositoryImpl {
        return TasksLocalRepositoryImpl(dao)
    }
    single { provideLocalTaskRepository(get()) }
}
```

The repository needs Dao. Dao needs a database. Thanks to DI, everything is created automatically.

As you can see, Dao is sent directly to the repository:

```
class TasksLocalRepositoryImpl(private val tasksDao: TasksDao) :
    ITasksLocalRepository {}
```

ViewModelModule

```
val viewModelModule = module {
    viewModel { TaskListViewModel(get()) }
    viewModel { AddTaskViewModel(get()) }
    viewModel { MapsViewModel() }
}
```

The last module is ViewModelModule. It is responsible for creating *ViewModels*. *ViewModel* needs a Repository, so it has it in the constructor:

```
class AddTaskViewModel(val taskRepository: TasksLocalRepositoryImpl) :
    BaseViewModel()
```

Options menu

There are a lot of menus in Android OS. The most common use is for the buttons on the right side of the Toolbar, or for bottom navigation (*BottomNavigationView*) or for the so-called hamburger menu.

Let's take a look at the menu on the Toolbar. The menu can be either part of the activity or part of a fragment.

Menu in fragment with NavigationComponent

If you use *NavigationComponent* in your project, using the menu is the same as for the activity. The menu as part of the fragment consists of three parts. The first is an XML file defining menu items. The second is menu initialization, and the last is the method for detecting clicks on a menu item.

The XML file can be found in the *res/menu* folder.

For each menu item, we can define several properties:

- **id** - identifier.
- **title** - item text. It will be displayed if no icon is specified.

- **showAsAction** - how the menu will be displayed. Whether always, or whenever there is a space, or never, and thus hides under the familiar three dots in the right corner of the Toolbar.
- **icon** - menu item icon.

The next part is the use of the menu in the code, its initialization:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    requireActivity().menuInflater.inflate(R.menu.menu_location, menu)
    super.onCreateOptionsMenu(menu, inflater)
}
```

The last part is then to verify if the item was clicked.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_done -> {
            // do the magic
            return true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

For the menu to work in a fragment, it is important to call one method in the `onCreateView` method:

```
setHasOptionsMenu(true)
```

If we only have a static menu that does not change, we only need the methods we have already used. However, if we need to change the menu dynamically based on how the fragment behaves (e.g. hide some items), we cannot do it in the `onCreateOptionsMenu` method, but we must do it in the `onPrepareOptionsMenu` method. Only after calling the `onPrepareOptionsMenu` method do we know that the menu is initialized and can edit it.

Working with map

If you want to display something on the map in Android OS, there is no better choice than Google Maps. Other SDKs usually build on a standard Google map, and the result is not always usable. However, I would also mention alternatives:

- Mapbox
- ArcGIS

Let's take a look at how to add Google Maps to our application. The easiest way is to add it via the wizard. We use the NavigationComponent, so let's create a new fragment with a map. However, the new map activity will work the same way.

When you create a new map snippet, one additional file is created at the same time - `google_maps_api.xml` in the values folder.

```
<resources>
    <string name="google_maps_key" templateMergeStrategy="preserve"
        translatable="false">YOUR KEY HERE</string>
</resources>
```

For a map to work, three things are important:

- **SHA-1** - Certificate for your instance of Android Studio.
- **Package name** - your package.
- **YOUR_CODE_HERE** - here we insert the generated API key from the console (see below).

Getting the key from the console

You'll find everything you need on the Google API Administration page. Two operations are required to get the map up and running. First, you need to create a key, and then you need to activate Maps SDK for Android.

In the APIs & Services - Credentials section, click on **CREATE CREDENTIALS** and then select the API Key here.

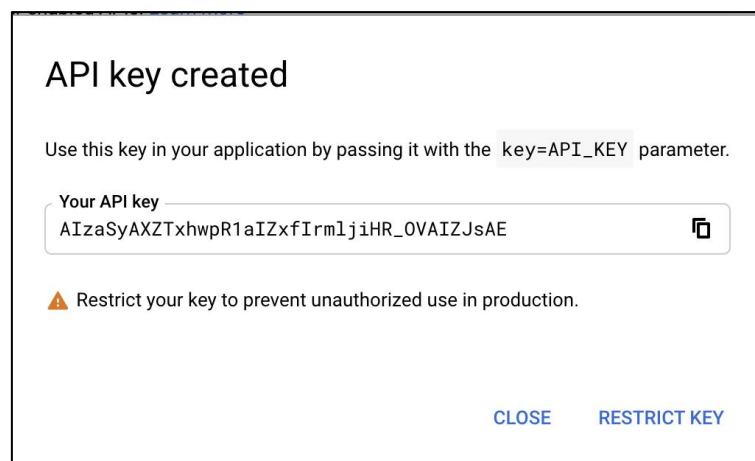


Fig. 30 API key initialisation

A new key will be created for you then you need to click on **RESTRICT KEY**. Now select Android apps and enter your package and SHA-1 in the Restrict usage to your Android apps field.

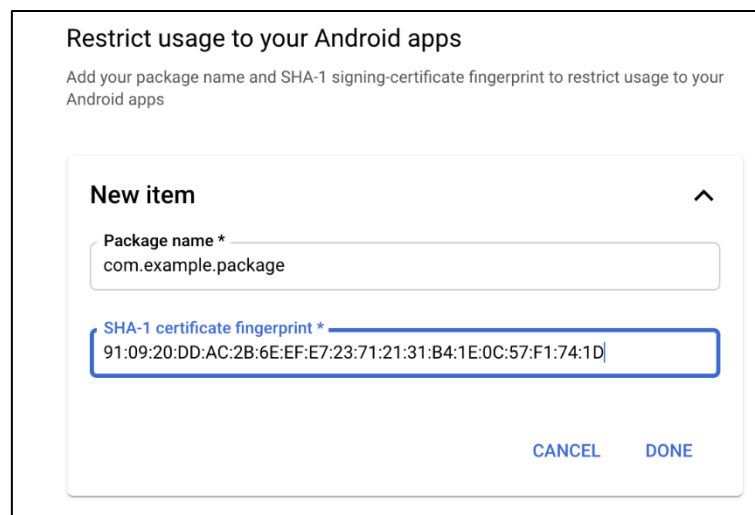


Fig. 31 Restrict key

Click on **DONE** and **SAVE**. Then paste the generated key into the `google_maps_api.xml` file.

Then, in the Library section, find Maps SDK for Android and click on **ENABLE**.

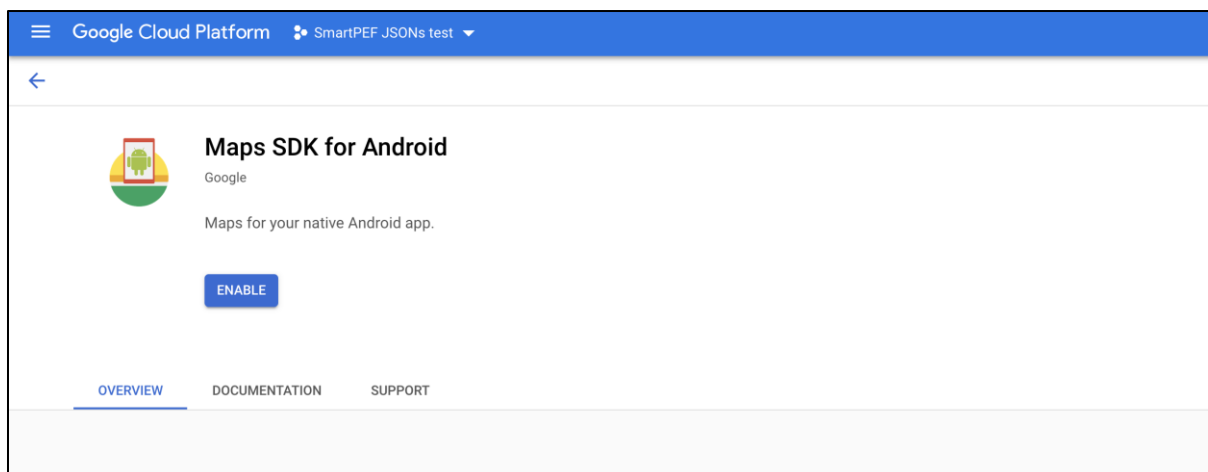


Fig. 32 Final step

Map in fragment

The generated fragment is only the basic fragment. But remember that we use our fragments as descendants of the *BaseFragment*. Therefore, we need to create a *ViewModel* for the map and make *MapsFragment* a child of the *BaseFragment* class.

```
class MapsFragment: BaseFragment <FragmentMapsBinding, MapsViewModel>
    (MapsViewModel :: class), GoogleMap.OnMarkerDragListener,
    GoogleMap.OnMarkerClickListener
{
    private val arguments: MapsFragmentArgs by navArgs ()

    private lateinit var map: GoogleMap

    private val callback = OnMapReadyCallback {googleMap ->
        map = googleMap
    }

    override fun onCreateView (view: View, savedInstanceState: Bundle?) {
        super.onCreateView (view, savedInstanceState)
        val mapFragment = childFragmentManager.findFragmentById (R.id.map)
                                                                    as SupportMapFragment?
        mapFragment?.getMapAsync (callback)
    }

    override val bindingInflater: (LayoutInflater) -> FragmentMapsBinding
        get () = FragmentMapsBinding :: inflate

    override fun initView () {
    }
}
```

The map is loaded asynchronously using the *getMapAsync* method. Once the map is loaded, a callback, the *onMapReady* method, is called.

Attention: You cannot add objects to it until the map is loaded.

It is now possible to add more operations. In general, you can do the following things with a map:

- add objects,
- change the map type,
- move with a virtual camera,
- add an object to the map.

The easiest way is to add a *Marker*, i.e., a point from the map. Let's look at a method to add a marker to the map.

```
private fun loadMapData(){
    val position: LatLng
    if (viewModel.latitude != null && viewModel.longitude != null){
        position = LatLng(viewModel.latitude!!.toDouble(),
                           viewModel.longitude!!.toDouble())
    } else {
        position = LatLng(DEFAULT_LATITUDE, DEFAULT_LONGITUDE)
    }

    val markerOptions: MarkerOptions = MarkerOptions().position(position).
                                                draggable(true).title("Experiment")
    var marker: Marker = map.addMarker(markerOptions)
    map.moveCamera(CameraUpdateFactory.newLatLng(position))
}
```

To create a *Marker*, you need to create a *MarkerOptions* class. It controls where the point is, but also how it looks. Subsequently, the *Marker* is created and returned by the method.

Caution: save the instance returned by the *addMarker* method. It can be useful for later processing.

Change the map background

Changing the background map is easy. Just call the method:

```
map.setMapType (GoogleMap.MAP_TYPE_HYBRID);
```

The variants are:

```
public static final int MAP_TYPE_NONE = 0;
public static final int MAP_TYPE_NORMAL = 1;
public static final int MAP_TYPE_SATELLITE = 2;
public static final int MAP_TYPE_TERRAIN = 3;
```

Map Box Application Template

The presented application serves as a template for working with a MapBox on the Android operating system using Java language.

Recommended Number of Developers

Individual.

Available Solutions

There are many common map applications; **MapBox** is a suitable component for many applications with a better price-performance ratio than Google Maps. The application presents a simple implementation of this technology and its features.

Requirements

Functional Requirements

- a) The application will show a list of typical map functionalities.
- b) The user can run different features from starting the activity.
- c) The user can use all implemented user features in the MapBox environment.
- d) The application presents the Geo-json approach.

Non-functional Requirements

- a) Android 5.0 and higher.
- b) Java programming language.

Application Design

Technology and Architecture Selection

Google Maps is just one of the existing services for providing map content. It may be the majority, but the charging policy in recent years has forced programmers to look for alternative solutions based on the different rules.

The offer of services for map reading and processing in March 2020 was as follows:

	OsmAnd	Mapbox	JawgMaps	HERE	GraphHopper	Google Maps
Free request/month	unlimited	50,000	50,000	250,000	15,000	28,000
Paid plans (from)	\$1,63	\$5	\$250	\$449	\$48	\$7/each 1000
Types of maps	tile, vector	tile, vector	tile, vector	tile, vector	vector only	tile, vector, satellite
Open-source data	✓	✓	✓	✗	✓	✗
Business geocoding	✓	✗	✓	✓	✓	✓
Navigation	✓	✓	✓	✓	✓	✓
Voice guidance	✓	✗	✗	✓	✗	✓
Traffic Insights	✗	✓	✗	✓	✗	✓
Layers	✓	✓	✗	✓	✓	✓
GPX tracks	✓	✓	✗	✓	✓	✓
Street view	✓	✗	✗	✓	✗	✓

Fig. 33 Policy of Map component developers

In addition to choosing from existing operators, there are also options for hosting your own "streaming" servers with open-source map sources.

Solution

Connection

As an alternative service, we chose MapBox for the purpose of creating mobile applications. Reasons to choose this service were:

- one of the cheapest solutions for small projects
- support for satellite and map display
- Android, iOS, web, Unity support
- popularity
- easy integration into the application
- evolving platform
- well-prepared tutorial + examples

The principle of using all external services is very similar. As with Google Maps, we also need to get an API key first with MapBox.

We can start by <https://account.mapbox.com/auth/signup/> and gradually go through the individual steps:

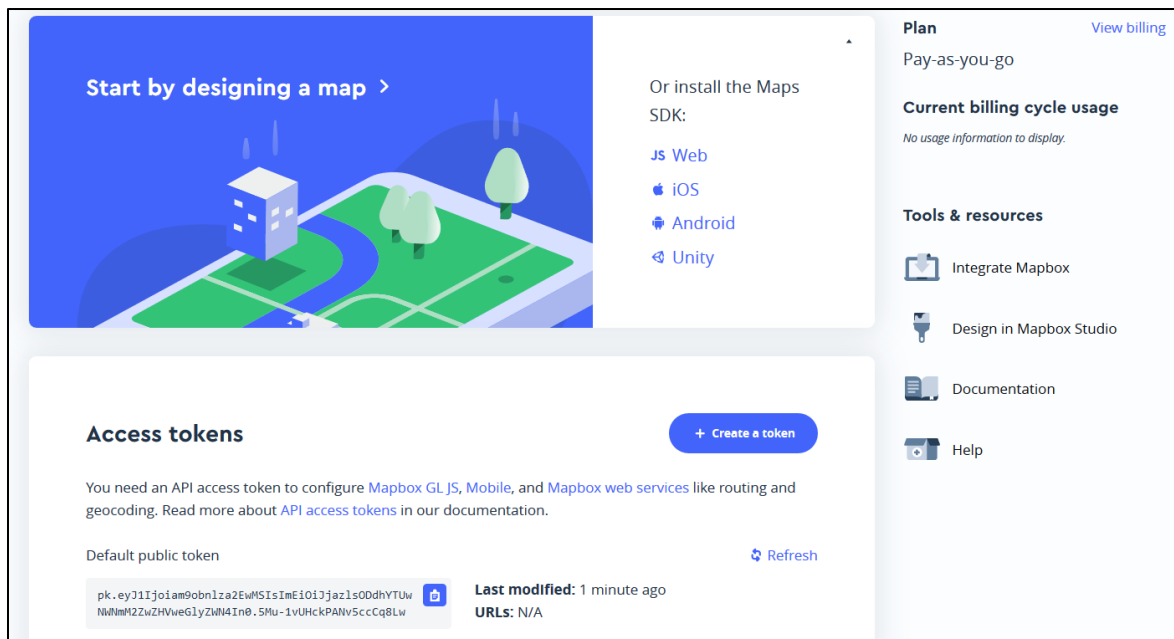


Fig. 34 Welcome screen to create a token

In the beginning, we create a project and define the required functionality with the ability to limit the use of the token to a single URL.

Create an access token

Token name
Choose a name to help associate it with a project.

Name
Andro_proj 10 / 128

Token scopes
All tokens, regardless of the scopes included, are able to view styles, tilesets, and geocode locations for the token's owner. [Learn more.](#)

Public scopes

<input checked="" type="checkbox"/> STYLES:TILES	<input checked="" type="checkbox"/> STYLES:READ	<input checked="" type="checkbox"/> FONTS:READ	<input checked="" type="checkbox"/> DATASETS:READ
<input checked="" type="checkbox"/> VISION:READ			

Secret scopes

<input type="checkbox"/> SCOPES:LIST	<input type="checkbox"/> MAP:READ	<input type="checkbox"/> MAP:WRITE	<input type="checkbox"/> USER:READ
<input type="checkbox"/> USER:WRITE	<input type="checkbox"/> UPLOADS:READ	<input type="checkbox"/> UPLOADS:LIST	<input type="checkbox"/> UPLOADS:WRITE
<input type="checkbox"/> FONTS:LIST	<input type="checkbox"/> FONTS:WRITE	<input type="checkbox"/> STYLES:WRITE	<input type="checkbox"/> STYLES:LIST
<input type="checkbox"/> TOKENS:READ	<input type="checkbox"/> TOKENS:WRITE	<input type="checkbox"/> DATASETS:LIST	<input type="checkbox"/> DATASETS:WRITE
<input type="checkbox"/> TILESETS:LIST	<input type="checkbox"/> TILESETS:READ	<input type="checkbox"/> TILESETS:WRITE	<input type="checkbox"/> VISION:DOWNLOAD

Token restrictions
Make your access tokens more secure by adding URL restrictions. When you add a URL restriction to a token, that

Fig. 35 Definition of functionalities

The created token can then be copied to applications.

mapbox | Account

Dashboard Tokens Statistics Invoices Settings

Access tokens

You need an API access token to configure [Mapbox GL JS](#), [Mobile](#), and [Mapbox web services](#) like routing and geocoding. Read more about [API access tokens](#) in our documentation.

[+ Create a token](#)

Name	Token	Last modified	URLs
Default public token	pk.eyJ1Ijoiam9obn1za2EwMSIsImE1OiJjczlsODdhYTUwNmhmM2ZwZHVveGlyZWhN4In0.5Mu-1vUHckPANv5ccCq8Lw	8 minutes ago	N/A Refresh
Andro_proj	pk.eyJ1Ijoiam9obn1za2EwMSIsImE1OiJjczlsODdhYTUwNmhmM2ZwZHVveGlyZWhN4In0.5Mu-1vUHckPANv5ccCq8Lw	less than a minute ago	0

Fig. 36 List of created tokens

After obtaining a token for use in Android, three application steps/modifications are required. You can also find the current description at <https://docs.mapbox.com/android/maps/overview/>. In the text, we use images from this description page to show that the process is really very simple.

The first step is to connect the project to an address with MapBox. We connect by inserting a pair of dependencies into the gradle files. We can see the placement of the text in the sections in the picture.

1401MapBox build.gradle

Project: 1401MapBox

- app
 - manifests
 - java
 - com.example.a1401mapbox
 - com.example.a1401mapbox (androidTest)
 - com.example.a1401mapbox (test)
 - java (generated)
 - res
 - drawable
 - layout
 - mipmap
 - values
- Gradle Scripts
 - build.gradle (Project: 1401MapBox)
 - build.gradle (Module: app)
 - gradle-wrapper.properties (Gradle Version)
 - proguard-rules.pro (ProGuard Rules for app)
 - gradle.properties (Project Properties)
 - settings.gradle (Project Settings)
 - local.properties (SDK Location)

Add Maps SDK for Android dependency

Add the following to your `build.gradle`.

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.mapbox.mapboxsdk:mapbox-android-sdk:9.1.0'
}
```

[Next >](#)

Fig. 37 Writing dependencies to build files (settings)

We will add the permissions to access the location to the manifest:

Set up permissions

Add the following permissions to your `AndroidManifest.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    ...
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    ...
</manifest>
```

[Next >](#)

Fig. 38 Adding permissions to the manifest

Let's take the last step of the procedure as a guide only - there are different ways to use MapBox, and we will show other procedures as well.

Add a map view class and declare layout

Add the following code to `MainActivity.java` to create a map view activity class.

```
import android.os.Bundle;
import com.mapbox.mapboxsdk.Mapbox;
import com.mapbox.mapboxsdk.maps.MapView;
import com.mapbox.mapboxsdk.maps.MapboxMap;
import com.mapbox.mapboxsdk.maps.OnMapReadyCallback;
import com.mapbox.mapboxsdk.maps.Style;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private MapView mapView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

Then in `res > layout > activity_main.xml`, add the following code to declare your layout.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.mapbox.mapboxsdk.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

[Next >](#)

Fig. 39 Offer ways to use MapBox in a project on the provider's website

Activation

Task:

Create an application that allows you to run Mapbox in independent activity.



The map display element is the `MapView` element, which we can place in the activity. We set the size for the entire width and height of the activity in which it is placed and named it so that it can be identified in the java code.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MapActivity1">

    <com.mapbox.mapboxsdk.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

We operate on a very similar principle to Google Maps. In this case, we will add the callback directly to the `onCreate()` method, and we will not define it as a separate method - we do not need to use the interface defined in the activity.

```
public class MapActivity1 extends AppCompatActivity {
    // component for working with the map
    private MapView mapView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // before loading xml, an instance is created to work with the map -
        // the API token / key is small
        Mapbox.getInstance(this, "pk.eyJJIjoiaxXXXXXXXXMu-1vUHckPANv5ccCq8Lw");
    }
}
```

```

// loading design
setContentView(R.layout.activity_map1);
mapView = findViewById(R.id.mapView);
// call the event when creating the map
mapView.onCreate(savedInstanceState);
// similar to an async task that loads a map
// and calls onMapReady when finished
mapView.getMapAsync(new OnMapReadyCallback() {
    @Override
    public void onMapReady(@NonNull MapboxMap mapboxMap) {
        mapboxMap.setStyle(Style.MAPBOX_STREETS,
            new Style.OnStyleLoaded() {
                @Override
                // similar to an async task, for loading a style,
                // executes the body code when finished
                public void onStyleLoaded(@NonNull Style style) {
                    // The map is ready, the map layout is loaded
                }
            }
        );
    }
});
}
}
}

```



The first application is ready and works in the emulator without the need to install system images with Google Play (which Google Maps requires).

Position on the map

Task:

Present different ways to display data on a map. Set to the Constantine the Philosopher University in Nitra (CPU or UKF) position.

Once again, we will create the main activity, from which we will run individual tasks, which will be added as part of the presentation of **MapBox** functionalities.

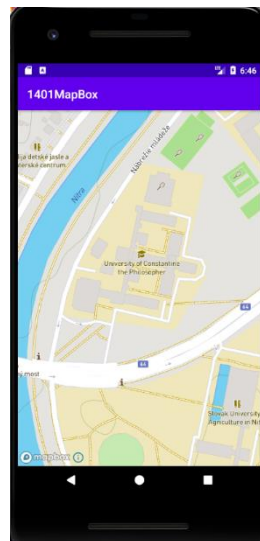




Fig. 40 Example of an introduction activity

For the first task, we will present three different appearance styles. We send these styles selected using radiobuttons to the imaging activity. Based on the style, the type of map data is selected and send to the user view in the application.

Clicking on the MAP button starts the process of preparing the content, which is then downloaded to the map activity, and the content is adapted accordingly.

```
public void onMapClick2(View view) {
    Intent i = new Intent(this, MapActivity2.class);
    String mapType = Style.MAPBOX_STREETS;
    RadioButton rb = (RadioButton) findViewById(R.id.radioButton2);
    if (rb.isChecked()) mapType = Style.OUTDOORS;
    rb = (RadioButton) findViewById(R.id.radioButton3);
    if (rb.isChecked()) mapType = Style.SATELLITE;
    i.putExtra("style", mapType);
    startActivity(i);
}
```

Style is a class that defines individual styles as constants of type **String**. We will send the selected style and, therefore, only read it in the map activity. In this case, we will use dynamic content creation of the activity - similarly to the creation of **Canvas** => xml activity in principle we do not even need.

```
public class MapActivity2 extends AppCompatActivity {
    private MapView mapView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Mapbox.getInstance(this, "pk.xxx");
        // we set the camera to our position and zoom in
        // we only prepare the settings
        MapboxMapOptions options = MapboxMapOptions.createFromAttributes(
            this, null) // používame builder a ""
            .camera(new CameraPosition.Builder()
                .target(new LatLng(48.308526, 18.091698))
```

```

        .zoom(16)
        .build());
// create a mapview based on the settings
mapView = new MapView(this, options);
mapView.onCreate(savedInstanceState);
// we load the contents of the map with an asynchronous task
mapView.getMapAsync(new OnMapReadyCallback() {
    @Override
    public void onMapReady(@NonNull MapboxMap mapboxMap) {
        Intent i = getIntent();
        String mapStyle = i.getStringExtra("style");

        mapboxMap.setStyle(mapStyle, new Style.OnStyleLoaded() {
            @Override
            public void onStyleLoaded(@NonNull Style style) {
                // we can perform other operations after loading
            }
        });
    }
});
// we will set the map as the content of the created activity
setContentView(mapView);
}

```

We will send the look (style) of the map to the activity via intent. Our goal is to set and zoom the map to a specific position. We can make this setting before loading the map and apply it when loading - the result will be a more efficient (= faster) loading of the required content.

Again, we have asynchronous methods in which we respond to:

- the situation when the map is ready (*onMapReady*)
- the situation when a style is recorded / ready (*onStyleLoaded*)

Markers

Task:

Use markers on the map to identify important places.

The simplicity of using markers in **MapBox** has ended in previous versions of Android. Currently, there is a group of Managers to provide displaying various objects and graphics in different layers:



- positive: we can do anything on individual layers (custom icons, custom graphics),
- negative: the code is a bit longer and sometimes a little bit difficult to understand.

In order to use a more modern approach, it is necessary to add a library with annotations to the dependencies in **build.gradle(Module.app)**.

```
...
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    implementation 'com.mapbox.mapboxsdk:mapbox-android-sdk:9.1.0'
    implementation 'com.mapbox.mapboxsdk:mapbox-android-plugin-annotation-v9:0.8.0'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

In this case, we will not use dynamic creation of the layout in the code but in the xml file. We insert the view into the xml to display the map (**com.mapbox.mapboxsdk.maps.MapView**), and we will access it from the activity - we can also set the position and zoom of the map.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:mapbox="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity3">

    <com.mapbox.mapboxsdk.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        mapbox:mapbox_cameraTargetLat="48.308526"
        mapbox:mapbox_cameraTargetLng="18.091698"
        mapbox:mapbox_cameraZoom="14"
    />

</androidx.constraintlayout.widget.ConstraintLayout>
```

If we use in the component settings belonging to **MapBox** - not **android**: but **mapbox**: prefix, it is necessary to specify the definition of the **MapBox** in the layout settings (line 5). We will also show an alternative approach to writing java code:

- we will not insert the method for callback into **onCreate()**; we will create it as a separate one somewhere in the activity code,
- an interface needs to be implemented on the activity **OnMapReadyCallback**.

```
public class MainActivity3 extends AppCompatActivity implements OnMapReadyCallback {
    private MapView mapView;
    // private MapboxMap mapboxMap;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Mapbox.getInstance(this, "pk.xxx");
    }
}
```

```

        setContentView(R.layout.activity_map3);

        // Initialisation of MapView
        mapView = findViewById(R.id.mapView);
        mapView.onCreate(savedInstanceState);
        mapView.getMapAsync(this);
    }

    @Override
    public void onMapReady(@NonNull final MapboxMap mapboxMap) {
        ...
    }
}

```

After preparing the map, we can:

- set the style and after setting the style
- ...use an outdated/deprecated method **addMarker()**
- to display a marker with a CPU (UKF) caption at the CPU position

```

@Override
public void onMapReady(@NonNull final MapboxMap mapboxMap) {

    mapboxMap.setStyle(Style.MAPBOX_STREETS, new Style.OnStyleLoaded() {
        @Override
        public void onStyleLoaded(@NonNull Style style) {
            mapboxMap.addMarker(new MarkerOptions()
                .position(new LatLng(48.308526, 18.091698))
                .title("CPU"));
        }
    })
}

```

Or in a new way: we will use **SymbolManager** to create icons on the map. When created, the icon will be placed on the map to the position that we entered when creating the manager.

```

public void onMapReady(@NonNull final MapboxMap mapboxMap) {
    mapboxMap.setStyle(Style.MAPBOX_STREETS, new Style.OnStyleLoaded() {
        @Override
        public void onStyleLoaded(@NonNull Style style) {
            SymbolManager symbolManager = new SymbolManager(
                mapView, mapboxMap, style);

            // we set the icons so that they do not interfere with each other
            symbolManager.setIconAllowOverlap(true);
            symbolManager.setIconIgnorePlacement(true);

            // we will add the fire station icon to the position with the zoom
            Symbol symbol = symbolManager.create(new SymbolOptions()
                .withLatLng(new LatLng(48.308520,
18.093))
                .withIconImage("fire-station-15")
                .withIconSize(2.0f));

```



```

    }
  });
}

```

Area boundary

Task:

Define the boundary of the area (e.g. with a rectangle).



The map view in the activity will remain set and zoomed to our central point.

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:mapbox="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MapActivity4">

    <com.mapbox.mapboxsdk.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        mapbox:mapbox_cameraTargetLat="48.308526"
        mapbox:mapbox_cameraTargetLng="18.091698"
        mapbox:mapbox_cameraZoom="14"
        />

</androidx.constraintlayout.widget.ConstraintLayout>

```

The initialization of the activity is the same as in the previous case, and we only add a list of boundary points.

```

public class MapActivity4 extends AppCompatActivity implements OnMapReadyCallback
{
    private MapView mapView;
    List<Point> myArea;
}

```



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Mapbox.getInstance(this, "pk.xxx");

    setContentView(R.layout.activity_map4);
    naplnBody();
    // Initialisation of mapView
    mapView = findViewById(R.id.mapView);
    mapView.onCreate(savedInstanceState);
    mapView.getMapAsync(this);
}

private void naplnBody() {
    // create list
    myArea = new ArrayList();
    myArea.add(Point.fromLngLat(18.088, 48.307));
    myArea.add(Point.fromLngLat(18.088, 48.3095));
    myArea.add(Point.fromLngLat(18.095, 48.3095));
    myArea.add(Point.fromLngLat(18.095, 48.307));
    myArea.add(Point.fromLngLat(18.088, 48.307));
}

```

After the style loading, a new layer will be created in the callback. We will set the content properties for it.

```

@Override
public void onMapReady(@NonNull final MapboxMap mapboxMap) {
    mapboxMap.setStyle(Style.MAPBOX_STREETS, new Style.OnStyleLoaded() {
        @Override
        public void onStyleLoaded(@NonNull Style style) {
            // new layer with dashed line
            style.addLayer(new LineLayer
                ("linelayer", "line-source").withProperties(
                    PropertyFactory.lineDasharray(new Float[] {0.1f, 2f}),
                    PropertyFactory.lineCap(Property.LINE_CAP_ROUND),
                    PropertyFactory.lineJoin(Property.LINE_JOIN_ROUND),
                    PropertyFactory.lineWidth(5f),
                    PropertyFactory.lineColor(Color.parseColor("#e55e5e"))
                ));
            ...
        }
    });
}

```

We will place lines in this layer - we will create an object **GeoJsonSource**, which is the source for objects plotted on the map - the created field is transformed into geo-data.

```

...
// array transformation to geodata
style.addSource(new GeoJsonSource("line-source",
    FeatureCollection.fromFeatures(new Feature[]
        {Feature.fromGeometry(
            LineString.fromLngLats(myArea)
        })
    }
));
}
});

```

Directions

Task:

Find the path between two points.



Fig. 41 From start position to the destination

Path finder is one of the advanced features that require machine time to count and generate a result according to the user's current requirements. It is available in the **Directions** package, which uses a different API set, so you need to add it to the dependencies in **build.gradle(Module.app)**.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    implementation 'com.mapbox.mapboxsdk:mapbox-android-sdk:9.1.0'
    implementation 'com.mapbox.mapboxsdk:mapbox-android-plugin-annotation-v9:0.8.0'
    implementation 'com.mapbox.mapboxsdk:mapbox-sdk-services:5.1.0'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

When creating an application, you may encounter a problem with the versions of SDKs and the Java version, and it has been stated that only versions from Android N above are supported. By adding compiler settings (in **build.gradle(Module.app)**) is the problem solved.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.3"

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

In the first step we define constants for layers and basic objects for working with the map.

```
public class MapActivity5 extends AppCompatActivity implements OnMapReadyCallback {
    private static final String ROUTE_LAYER_ID = "route-layer-id";
    private static final String ROUTE_SOURCE_ID = "route-source-id";
    private static final String ICON_SOURCE_ID = "icon-source-id";

    private MapView mapView;
    private DirectionsRoute currentRoute;
    private MapboxDirections client;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Mapbox.getInstance(this, "pk.xxx");
        setContentView(R.layout.activity_map5);
        // map view
        mapView = findViewById(R.id.mapView);
        mapView.onCreate(savedInstanceState);
        mapView.getMapAsync(this);
    }
}
```

After obtaining the map, we define the beginning and end of the route. The documentation states that more than 20 transition points can be defined. We will prepare a format for the query and a layer for rendering

```
@Override
public void onMapReady(@NonNull final MapboxMap mapboxMap) {
    mapboxMap.setStyle(Style.MAPBOX_STREETS, new Style.OnStyleLoaded() {
        @Override
        public void onStyleLoaded(@NonNull Style style) {
            // start a end
            Point origin = Point.fromLngLat(18.090272, 48.307561);
            Point destination = Point.fromLngLat(18.094539, 48.312398);
            // prepare the geodata to get the path
            initSource(style, origin, destination);
            // prepare layer for drawing
            initLayers(style);
            // get directions from the Mapbox Directions API
            getRoute(mapboxMap, origin, destination);
        }
    });
}
```

The **initSource()** method ensures that the start and end position data settings are placed in the **loadedMapStyle** variable (object).

```
private void initSource(@NonNull Style loadedMapStyle, Point origin, Point
destination) {
    loadedMapStyle.addSource(new GeoJsonSource(ROUTE_SOURCE_ID));

    GeoJsonSource iconGeoJsonSource = new GeoJsonSource(ICON_SOURCE_ID,
```

```

        FeatureCollection.fromFeatures(new Feature[] {
            Feature.fromGeometry(Point.fromLngLat(origin.longitude(),
                                                origin.latitude())),
            Feature.fromGeometry(Point.fromLngLat(destination.longitude(),
                                                destination.latitude()))));
loadedMapStyle.addSource(iconGeoJsonSource);
}

```

The **initLayers()** method defines a new layer and in it the parameters for the drawn line, respectively layer of lines. The information goes back to the map settings via the **loadedMapStyle** object.

```

private void initLayers(@NonNull Style loadedMapStyle) {
    LineLayer routeLayer = new LineLayer(ROUTE_LAYER_ID, ROUTE_SOURCE_ID);

    // Adds LineLayer to the maps. This layer displays the directions route.
    routeLayer.setProperties(
        lineCap(Property.LINE_CAP_ROUND),
        lineJoin(Property.LINE_JOIN_ROUND),
        lineWidth(5f),
        lineColor(Color.parseColor("#FF0000"))
    );
    loadedMapStyle.addLayer(routeLayer);
}

```

The **getRoute()** method is key to getting the information you need. It creates a request for data return, and at this point, it is possible to configure whether it should be a route for pedestrians, cyclists or cars.

```

private void getRoute(final MapboxMap mapboxMap, Point origin, Point destination) {
    client = MapboxDirections.builder()
        .origin(origin)
        .destination(destination)
        .overview(DirectionsCriteria.OVERVIEW_FULL)
        .profile(DirectionsCriteria.PROFILE_DRIVING)
        .accessToken(getString(R.string.access_token))
        .build();
    ...
}

```

It asks for data with an asynchronous request. If there is an empty result or a result without positions, we will terminate; otherwise, the result is a path – **currentRoute**, in which we can find out, for example, length.

```

client.enqueueCall(new Callback<DirectionsResponse>() {
    @Override
    public void onResponse(Call<DirectionsResponse> call,
                          Response<DirectionsResponse> response) {
        // if it does not return any result, the termination
        if (response.body() == null) {
            return;
        } else if (response.body().routes().size() < 1) {
            return;
        }

        // starting point
        currentRoute = response.body().routes().get(0);
    }
}

```

```
// for illustration - distance
Toast.makeText(MapActivity5.this, "distance: " + currentRoute.distance(),
    Toast.LENGTH_SHORT).show();
```

If the process is finished correctly, **mapboxMap** still exists, so after loading the style, we get access to the source for drawing lines and send lines from **CurrentRoute** to the drawing process/method.

```
if (mapboxMap != null) {
    mapboxMap.getStyle(new Style.OnStyleLoaded() {
        @Override
        public void onStyleLoaded(@NonNull Style style) {

            // Loads and updates the source showing the route
            GeoJsonSource source = style.getSourceAs(ROUTE_SOURCE_ID);

            // Creates a LineString sequence with route geometry
            // resets the GeoJSON source for the LineLayer route source
            if (source != null) {
                source.setGeoJson(LineString.fromPolyline(
                    currentRoute.geometry(),
                    PRECISION_6));
            }
        }
    });
}
```

Finally, we need to implement the method **onFailure** for **client.enqueueCall**.

```
@Override
public void onFailure(Call<DirectionsResponse> call,
    Throwable throwable) {
    Toast.makeText(MapActivity5.this, "Error: " +
        throwable.getMessage(), Toast.LENGTH_SHORT).show();
}
});
}
```

Conclusion

Using **MapBox**, we can create the same applications as using **Google Maps**, and especially in the case of finding a route, the procedure was much simpler.

MapBox provides a number of other features; in many ways, it is currently easier to work with than with **Google Maps**. Among other things, it allows you to create and insert into the map source various types of objects that can be shared with the community.

MapBox also supports the web, and so it is a suitable tool for creating your own applications on multiple platforms.

Compass

Create a compass for the smartphone.

The application presents the basic principles of using sensors, reading data from sensors and following processing. An integrated mathematical apparatus is also used to calculate the triaxial rotation of a mobile device, on the basis of which it is possible to identify not only the position but also the orientation of the device.

This approach is widely used in a variety of map applications, from simple geolocation games to augmented reality applications.

Recommended Number of Developers

Individual.

Available Solutions

There are a few simple applications focused on geomagnetic sensor use. This chapter presents the complexity of the problem solution.

Requirements

Functional Requirements

- a) The application will show a sensor list of the smartphone and identify the existence and type of sensors.
- b) The user can identify the north pole position using graphical elements in the system.

Non-functional Requirements

- a) Android 5.0 and higher.
- b) Java programming language.

Sensors

Sensors are a communication element by which devices can sense the parameters of the environment or manipulation the device. Today, thanks to the expansion of the IoT area, sensors are an affordable and inexpensive tool for extending the functionality of any device. Thanks to the minimization of their size, they can be integrated into devices such as watches, pieces of jewellery, markers or used in their own device designs (robot, toy car, drone, etc.).

The sensors can send the acquired data directly via the elements of the device in which they are integrated, or they can transmit them using one of the wireless technologies.

A common requirement of practice is to create a network of sensors that collect data at multiple locations and send it to one collection location.

The quality, types and capabilities of sensors integrated into mobile devices are gradually improving, and today a common smartphone is equipped with a number of miniature sensors, many of which required a separate single-purpose device a few years ago.

Communication with the environment

Perception of the environment is an essential element of the survival of living things. People (or more precisely, animals) is able to use all five of their senses:

- sight - allows you to read and then decode textual and/or pictorial information
- hearing - allows you to recognize sounds,
- touch - provides feelings represented by touches, allows identification of shapes and perception of temperature or cold,
- smell - allows you to perceive the smell in the environment in which the animal is currently,
- taste - allows identifying on the basis of taste receptors the characteristics of perception of the surface or interior of the object.



The Android operating system is able to obtain information from the environment that people cannot perceive. Typical sensed characteristics are, in addition to image and sound, e.g., temperature, humidity, movement, position of the north magnetic pole, position within the GPS system.



Access to sensors or, in general, to elements that read or convey information from the environment or provide communication in the environment can be obtained:

- using existing resources providing the necessary interfaces - we use ready-made activities that we call via intents (mail, camera),
- creating our own functions and interfaces, where we need to gain access to the relevant devices and capture (stream) messages with the content, they convey (camera, microphone, accelerometer, pedometer, etc.).

Sensor categorization

Most Android devices have built-in sensors capable of measuring movement, orientation or environmental parameters (pressure, temperature, etc.).

Sensors as devices are generally divided into:

- **motion** – sensors identifying changes due to motion (hardware: accelerometer, gyroscope; software: gravity, linear acceleration, rotation vector),
- **environmental** – sensors identifying changes in the environment (light sensor, humidity, temperature, pressure sensor), where the integration of sensors existence depends on the device,
- **position** – position sensors (hardware: geomagnetic field sensor/compass).

List of sensors in the device

Task:

Create an application that displays a list of sensors on your device.

To process data from sensors, the application uses **SensorManager**, which makes available the system service to initialise various sensors in devices. By default, it is defined in the activity for which we want to make sensor data available.

The list of sensors is provided by the **getSensorList()** method, which returns a list of sensors with name, version, manufacturer and possibly other parameters.

In a simple program, we just have them written into a text component:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    SensorManager mSensorManager =
        (SensorManager) getSystemService(SENSOR_SERVICE);
    List<Sensor> mList = mSensorManager.getSensorList(Sensor.TYPE_ALL);

    TextView tv1 = (TextView) findViewById(R.id.textView);
    for (int i = 1; i < mList.size(); i++) {
        tv1.append("\n" + mList.get(i).getName() + "\n" +
            mList.get(i).getVendor() + "\n" + mList.get(i).getVersion());
    }
}
```

Identifying the existence of sensors during the run (at startup) of the application can modify the behaviour of the application - add new functions, faster control, or, conversely, the absence of sensors replace their functionality with "manual" (non-sensors) methods.

In the case of missing sensors, we can also decide not to run the application or make a completely different alternative functionality and visually available instead of the full version.

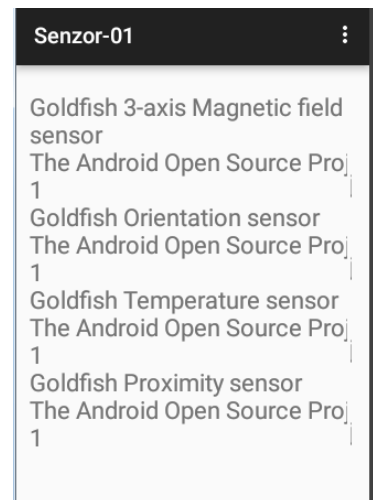


Fig. 42 List of sensors

Accelerometer - acquisition of data from sensors

We will show the way of working with sensors and the principle of data acquisition on the accelerometer. The accelerometer is the basic and the most used device when working with sensors.

The accelerometer responds to acceleration - a change in movement in any direction. To identify this ability, it uses a piezoelectric phenomenon, where it measures the voltage generated by the microscopic crystals, which moves with any movement, creating pressure between them that generates a voltage (in each direction).

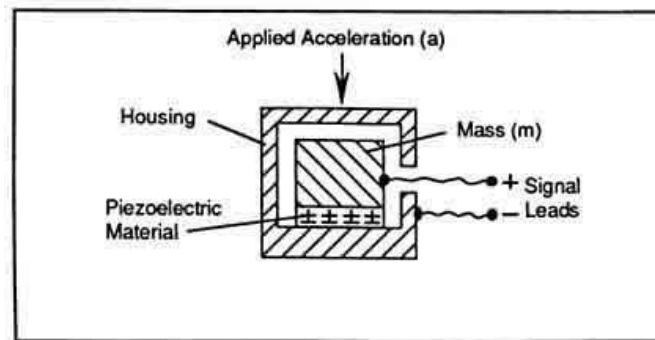


Fig. 43 Accelerometer principle (source: <https://www.pc-control.co.uk/accelerometers.htm>)

Coordinate system

Many motion sensors express their state relative to a three-dimensional coordinate system. The coordinate system is fixed for each device regardless of its angle of rotation.

Smartphones have a coordinate system defined as is presented in Figure 44; tablets usually rotated 90°. The systems are thus adapted to the most frequently used rotation of the device.

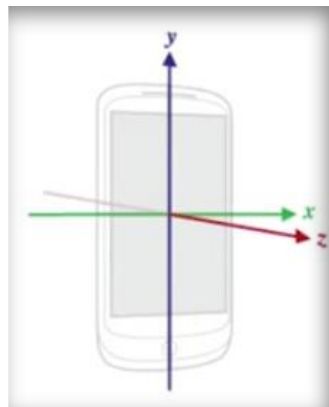


Fig. 44 Coordinate system for a smartphone

Be careful: the coordinate system does not change even when the device is rotated.

Principle of accelerometer operation

The accelerometer is constantly affected by gravitational acceleration

- if we point it to the centre of the earth in the y-direction, this value is theoretically 9.81, and the other values are 0
- if we turn the device upwards, the gravitational acceleration acts in the z-direction
- if we ideally rotate (place) it on the side edge, the acceleration acts in the x-direction.



Fig. 45 The direction of gravitational acceleration at different positions of the smartphone

All other positions distribute g between the individual directions; the value in each direction is non-zero and less than 9.81.

Any movement, more precisely its change, changes the actual values returned by the accelerometer, and thus it can be identified that the device is in motion:

- if I grab the phone and move it, there is an acceleration detected
- If I run at a constant speed with the phone (after starting up), no change occurs
- only when I stop will the opposite acceleration value be identified

Earth gravity in the form of gravitational acceleration (g) acts on the device constantly, which is used in various applications and games.

Capturing values

Task:

View the values provided by the accelerometer and observe their changes at rest and as you move the device.

The structure of all applications using sensors is practically identical:

- the application needs to implement an interface **SensorEventListener** to be able to read data from sensors
- this interface requires the implementation of methods:
 - **onSensorChanged** – when the sensor data changes,
 - **onAccuracyChanged** – if the accuracy changes, it is used for some types of sensors that are calibrated to return the result with better accuracy (usually, the method is empty).

```
public class MainActivity extends AppCompatActivity
implements SensorEventListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
```

```
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {

    }
}
```

In order for the application to obtain data from the sensor, it needs to register its listener.

- Registration ensures constant reception of data from the sensor to the application.
- Unregister ensures termination of data reception by the application/activity.

It is good practice to receive data from sensors only when the activity is in the foreground and when data is overlapped by another activity, pause data reception.

- In the case of simultaneous reception of data by different activities, the system could very easily become overloaded,
- however, there are also cases where it is appropriate to choose a different approach.

The activity life cycle provides us several places where the registration to receive data from the sensor can be turned on and off.

The best time to register is **onResume()**, which takes place when the activity becomes active and comes to the forefront of the activity stack.

An event **onPause()** is appropriate to stop receiving data for any reason that causes the activity to go into the background.

The subsequent return of activity to the foreground invokes **onResume()** again and thus ensures that data reception is switched on again (registration with the listener).

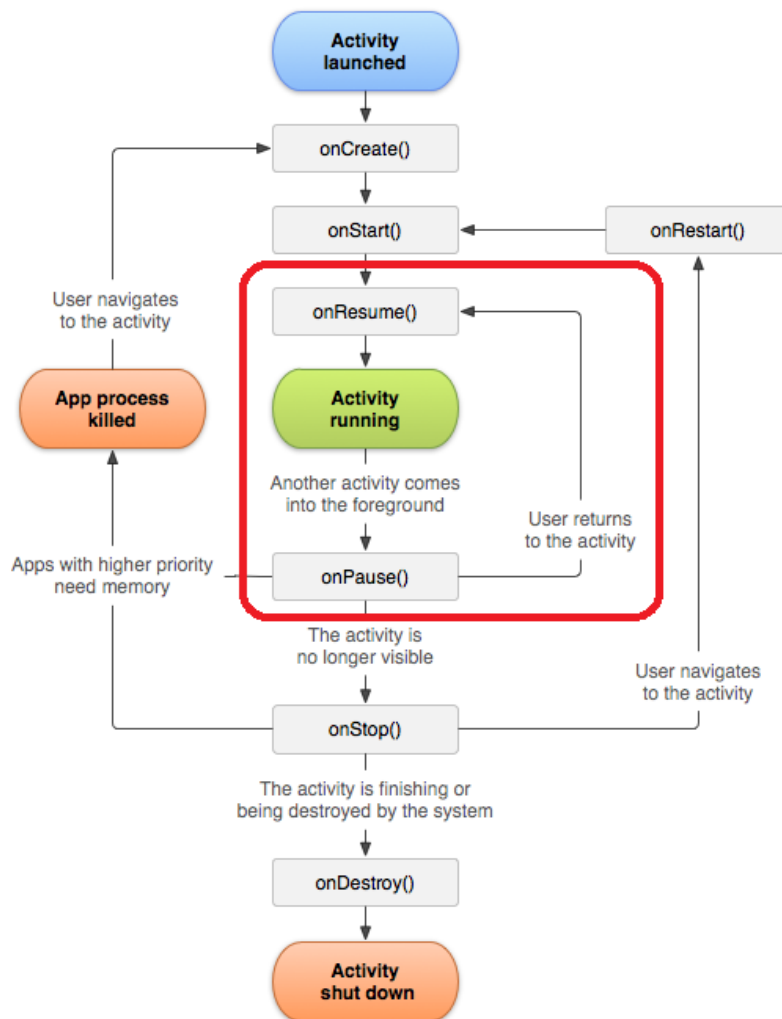


Fig. 46 Activity life cycle (source: <https://www.geeksforgeeks.org/activity-lifecycle-in-android-with-demo-app/>)

Registration and unregistration

When registering the sensor, we enter the listener, sensor and frequency of reading data. The frequency is optimized for different types of applications and represents the speed (frequency) of querying the data (e.g., for `SENSOR_DELAY_UI`, `SENSOR_DELAY_GAME`)

We already know that we need to work with sensors **SensorManager** and its initialization.

Then in the **onCreate()** method, we get access to the accelerometer and insert it into the variable with the aim to access it further.

In the **onResume()** and **onPause()** methods, we provide registration and unregistration for receiving data from the sensor.

```

public class MainActivity extends AppCompatActivity
implements SensorEventListener {
    private SensorManager sManager; // sensor access manager
    Sensor accelerometer; // my sensor - now accelerometer
  
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // getting a manager
    sManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    // getting a sensor
    accelerometer =
        sManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
}

protected void onResume() { // registration for sensor data collection
    super.onResume();
    sManager.registerListener(this, accelerometer,
        SensorManager.SENSOR_DELAY_UI);
}

protected void onPause() { // end data collection from the sensor
    super.onPause();
    sManager.unregisterListener(this);
}
}

```

Reading data from the sensor

At this point, the system is ready to read data from the sensor. In the **onSensorChanged()** method, we provide to print a list of values at each data delivery. You can find out from the manual for the sensor what data comes from the sensor and in what structure it is. It is true that they are stored in the array **values**.

```

@Override
public void onSensorChanged(SensorEvent event) {
    float ax, ay, az; // zrýchlenie v každom smere

    // we only monitor the accelerometer
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        // the expected values come as the first three values in the variable
        // representing the event
        ax = event.values[0];
        ay = event.values[1];
        az = event.values[2];

        // we just print them
        TextView tv = (TextView) findViewById(R.id.textView);
        tv.setText("X = " + ax + "\n Y = " + ay + "\n Z = " + az);
    }
}

```

While when using an emulator, the values are somewhere around the expected ones, in the case of a physical device, the values also change constantly when the device is at rest.



Fig. 47 Values obtained from the accelerometer in the emulator

In the case of a physical device, and if the device is oriented by default, the values are somewhere around:

- $x = 0 \text{ m/s}^2$
- $y = 9,81 \text{ m/s}^2$
- $z = 0 \text{ m/s}^2$

but they are constantly changing due to natural movements, uneven surfaces, noise, etc.

Data interpretation

The data we obtain from the accelerometer represents the current state of the device in all directions and usually does not reflect changes compared to the device at rest. If we stop moving after moving the device by a constant speed, we cannot identify whether we stopped or moved in the opposite direction (the accelerometer in the direction of the x-axis only returns a negative value).

In addition, the data that comes from the sensors is often skewed by various influences and can sometimes be inaccurate. The obtained values are often harmonised (or filtered) to prepare trustworthy data:

- often averages a certain number of changes or
- low-pass filter - goes through low values and reduces the amplitude of values higher than the defined value
- high-pass filter - goes through high values and reduces the amplitude of values lower than the defined value.

Geomagnetic sensor

A magnetic field sensor or compass is a sensor that can identify the position of the north magnetic pole.

The magnetic field sensor usually does not work exactly inside the building because it is very sensitive to any metal objects or electromagnetic fields in its nearby. To verify the correct functionality of the application, we recommend using an open space without ambient interference.

The sensor is included in most current smartphones and can be found under various names: magnetometer, magnetic field sensor, compass, etc.

While an accelerometer measures acceleration, a magnetometer measures the impact or the intensity of the magnetic field in μT , and it is a necessary element in determining the orientation of the device.

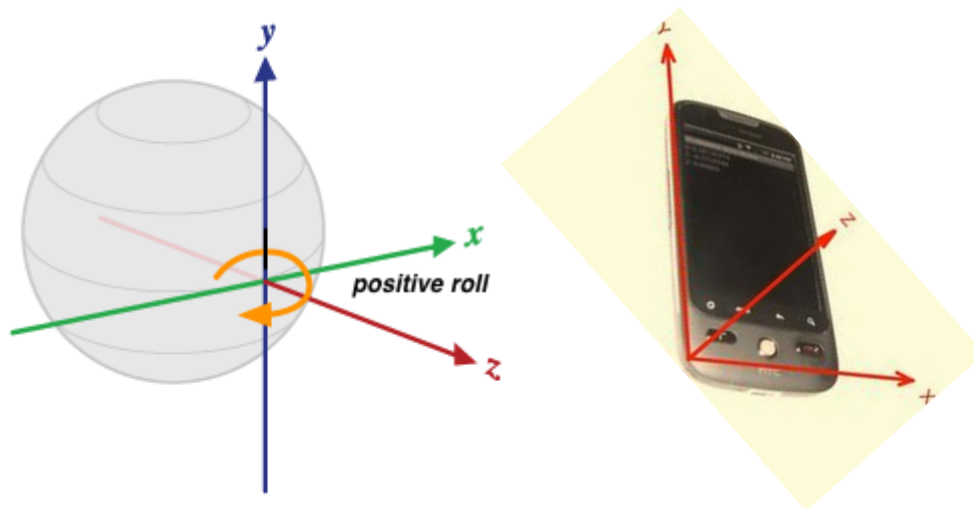


Fig. 48 Device orientation and direction of the magnetic field at the equator

Compass

Task:

Create an application in which the compass image will show the user where the north is.



To be able to identify the direction of the magnetic field, it is also necessary to have information about the position of the device towards gravity because the position of the north in the viewed image is also determined by the rotation of the device.

We can already get information about the rotation of the device with respect to gravity from the accelerometer.

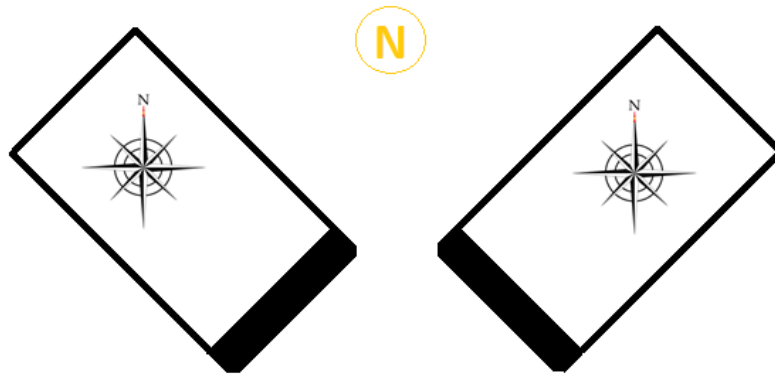
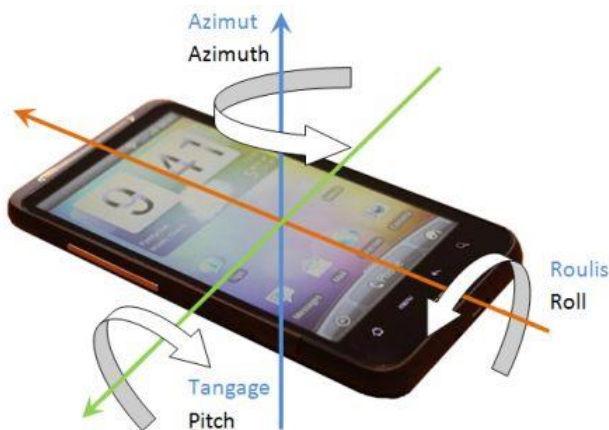


Fig. 49 Different rotation of the device and only one north

Thus, a functional and accurate compass can only be constructed as a combination of data from a magnetometer and an accelerometer.

A rotary matrix is used to processing the measured data. The rotary matrix is obtained via the built-in function **SensorManager.getRotationMatrix()**, into which the data from both sensors is entered. If the matrix generation finishes successfully, we gain data representing all directions of rotation in the first three values of the matrix: azimuth, pitch, and roll. Finally, we can display these values graphically on the smartphone.

Each subsequent rotation of the device will cause a subsequent recalculation so that our compass will always process the current values.



$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 50 Device rotation options and rotary matrix for individual axes

So we will create an activity in which we will place the image. To display the image, we will use the **ImageView** component and add a descriptive **TextView**. This **TextView** will display the loaded and calculated angle.

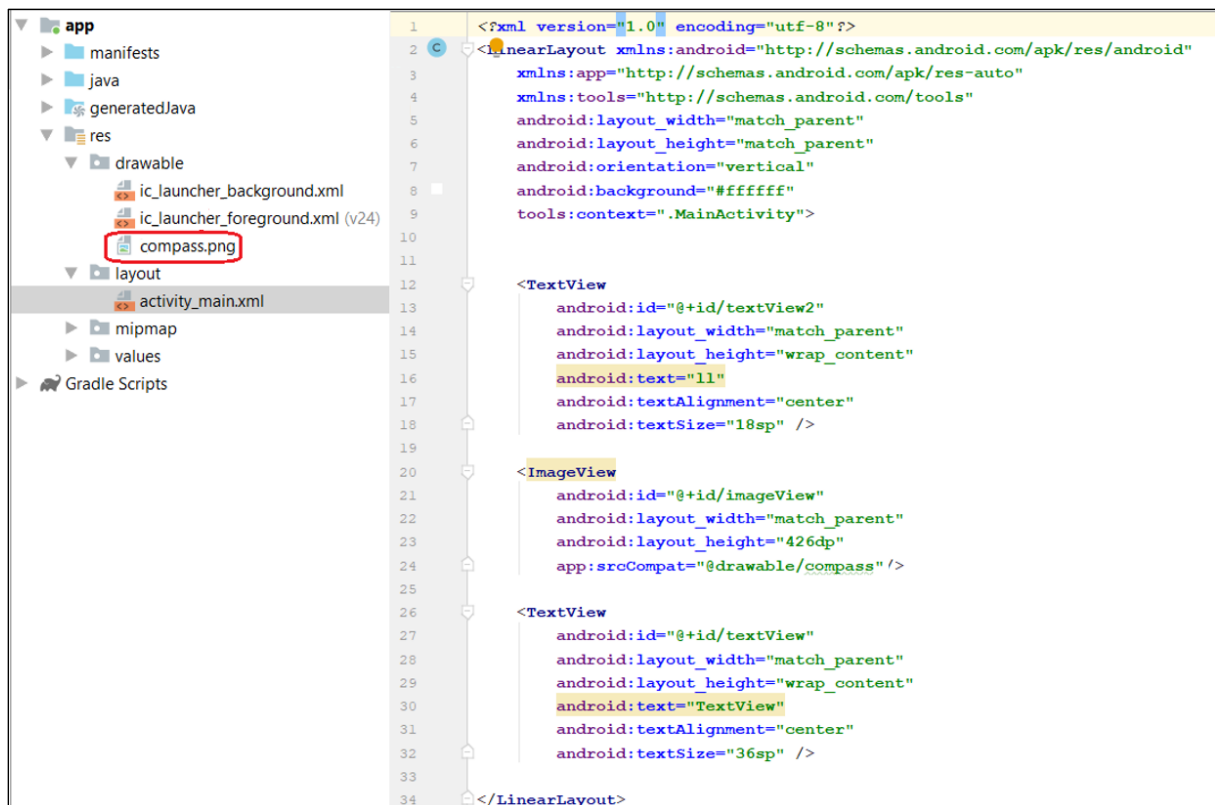


Fig. 51 Placing an image into resources and code

We will rotate the image in the code so that the marked arrow points north. We will remember the current rotation angle in the variable **currentDegree** with the goal to animate the rotation from the original to the new rotation angle.

In the activity, we define the necessary attributes, and in the **onCreate()** method, we initialize them.

```
public class MainActivity extends AppCompatActivity
    implements SensorEventListener {

    private ImageView image; // view for image
    // current rotation value
    private float currentDegree = 0f;
    private SensorManager mSensorManager; // sensor manager
    Sensor accelerometer;
    Sensor magnetometer;
    TextView tvStupne; // view
    // array for an accelometer
    float[] mGravity = null;
    // array for a magnetometer
    float[] mGeomagnetic = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.imageView);
        tvStupne = (TextView) findViewById(R.id.textView); // TextView
        mSensorManager = (SensorManager)
            getSystemService(SENSOR_SERVICE);

        accelerometer =
            mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        magnetometer =
```

```

        mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
    }

```

We perform the following operations in the **onSensorChanged()** method:

- we read data from the accelerometer
- we read data from the magnetometer
- since the data does not arrive at once, we must read it and remember it in a variable (array) defined at the class level; when we have read both values, we start the conversion and display the result.

The conversion consists in filling the rotation matrix (R) and a tilt matrix (I) based on the three-dimensional vectors we obtained from the sensors. It is used for filling

```

SensorManager.getRotationMatrix(R, I, mGravity, mGeomagnetic);

```

From the obtained matrices, we will use only a three-dimensional orientation vector, which we convert to the angle of rotation of the compass image and then we rotate this image. Finally, we remember the current angle of rotation. We always express the angle as a positive value, so we add 360 degrees to it and convert it to the basic angle (% 360).

```

public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
        mGravity = event.values;
    if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
        mGeomagnetic = event.values;
    if (mGravity != null && mGeomagnetic != null) {
        float R[] = new float[9];
        float I[] = new float[9];
        boolean success =
            SensorManager.getRotationMatrix(R, I, mGravity, mGeomagnetic);
        if (success) {
            float orientation[] = new float[3];
            SensorManager.getOrientation(R, orientation);
            float azimuth = orientation[0]; // contains: azimuth, pitch, roll
            float degree = (float) (Math.toDegrees(azimuth) + 360) % 360;
            tvStupne.setText(""+degree);

            RotateAnimation ra = new RotateAnimation(
                currentDegree,
                -degree,
                Animation.RELATIVE_TO_SELF, 0.5f,
                Animation.RELATIVE_TO_SELF, 0.5f);

            // duration in milliseconds
            ra.setDuration(210);
            ra.setFillAfter(true);
            // start animation
            image.startAnimation(ra);
            currentDegree = -degree;
        }
    }
}

```

An interesting element of the operating system is animation. There are several types of pre-built animation templates in the Android OS. We will use rotating animation for our needs –

RotateAnimation. The animation is always created first, and its parameters are defined. After setting, it is applied to the object and run.

```
RotateAnimation ra = new RotateAnimation(  
    currentDegree,  
    -degree,  
    Animation.RELATIVE_TO_SELF, 0.5f,  
    Animation.RELATIVE_TO_SELF, 0.5f);  
  
// duration in milliseconds  
ra.setDuration(210);  
ra.setFillAfter(true);  
// start animation  
image.startAnimation(ra);
```

The rotating animation starts the animation from one angle of rotation to the other. Our animation starts at the angle to which the compass image was last rotated (**currentDegree**) and ended at an angle that has been calculated based on the tilt of the device (**-degree**). The image is currently rotated to the angle defined in **currentDegree**, so the change gives the impression that it is a rotation from the current rotation.

The last four parameters define the position of the point around which it will rotate:

RELATIVE_TO_SELF means with respect to the object we are rotating, and 0.5 represents in both cases the centre of the object.

Using **setDuration()** we set the animation duration in milliseconds.

Parameter **setFillAfter()** defines whether the final form of the object should remain displayed after rotation or not.

Finally, we run the created animation over any object (view), in our case, of course, over the image.

Light sensor

The luminance sensor is one of the basic sensors included in most common devices.

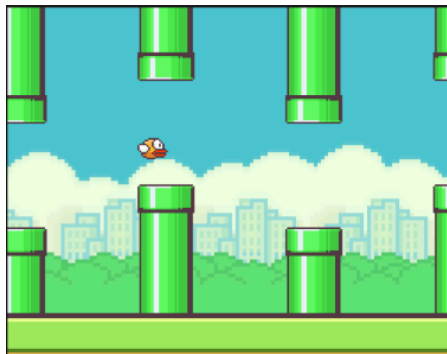
Returns the value of luminosity in lux, and access to it is the same as for other sensors. It is identified by the **TYPE_LIGHT** constant.

```
public void onSensorChanged(SensorEvent event) {  
    Sensor mySensor = event.sensor;  
  
    if (mySensor.getType() == Sensor.TYPE_LIGHT) {  
        TextView t = (TextView) findViewById(R.id.textView);  
        float newLight = event.values[0];  
    }  
}
```

Create an application that notifies the smartphone with a sound (triggered an alarm) in the event of a large immediate change in lighting, either an increase or decrease in brightness.

Simple Game with Accelerometer

Create an application that will be a variant of the flappy-bird application.



The player's goal is to fly through a changing environment consisting of obstacles displayed from above and below.

The flying object passes through an environment in which it moves up and down to avoid obstacles by tilting the device.

Step Detector

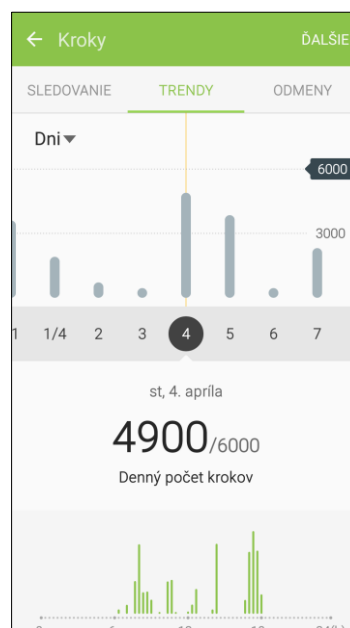
The accelerometer can be used not only to determine the current acceleration of the device, but also more complex operations can be identified based on the sequence of changes in the accelerometer values.

An example is the identification of a smartphone shake or the function of a pedometer, which since version 4.4 behaves as a separate sensor.

Using the sensors, create a pedometer capable of tracking device user activity:

- within a day
- within a week
- within days of the month
- the best day of a year
- average results of the week, month, year, etc.

Alternatives to Google Play may be a good example of an app (e.g. Samsung Health):



A suitable sensor for solving the problem is **StepDetector**.

This sensor counts the steps for the application since the first user registration. It resets only by the boot of the device. The big advantage is that we can read it at any time, and the application does not have to be run while the sensor is running. Each application can access it after launching and then display the current status (number of identified steps).

```
@Override
public void onSensorChanged(SensorEvent event) {
    Sensor sensor = event.sensor;
    float[] values = event.values;
    int value = -1;
```

```
    if (values.length > 0) {  
        value = (int) values[0];  
    }  
  
    if (sensor.getType() == Sensor.TYPE_STEP_COUNTER)  
        k1.setText("Step Counter: " + value);  
}
```

DATA PROCESSING

Customer-Products Data Model Development

Every second there is an incredible increase of data today. More data were generated for 2014 and 2015 than for the entire previous duration of humanity (Marr, 2015). This, of course, entails great demands on their storage and processing. The increase in the volume of data is unimaginable for the average person. Nowadays, we are used to the concept of Big Data. There are many interpretations of this concept. But in a way, it follows that it is the amount of data that makes it difficult to process it in real-time.

Where does all that data come from? Is it because companies are storing more and more data about their activities? A paper dealing with exponential data growth was published on the insidebigdata.com server in February 2017 (insidebigdata.com, 2017). Among other things, this article deals with the estimation of how the growth of data and their structure will develop from 2020, as you can see in Figure 1. So it is not mainly company data, but data from various sensors, but also data on people.



Fig. 52 Estimation of data growth and their structure (insidebigdata.com, 2017)

The issue of data processing in the field of information technology is certainly no newcomer, but it is quite clear that there are still many opportunities to find employment in this area. Maybe even more than in previous years. The term Data Science is one of the other modern concepts that are already common today. But before you can call yourself a data scientist and be able to process unimaginable volumes of data, you need to focus on the basics on which you can build your data career.

One of the basics is to understand data models and gain the skill to design a database to store all the necessary data.

Recommended Number of Developers

To create a database model, you can be alone; however, a small team of two or three members can be useful. It's always good to discuss your ideas with someone else.

Available Solutions

There are a lot of tutorials that can help you to learn how to develop a database model. Just use youtube or google and write a Data modelling tutorial. You will find many possibilities.

Requirements

The creation of the data model is based on the requirements, which are usually entered in the form of a scenario. The scenario describes in plain text what data needs to be stored. The challenge for data model designers is to understand what is important in the text. When designing a database, it is not customary to set functional requirements, as is often the case with other types of applications. The following scenario is an example of input from a customer.

Nowadays, according to some authorities, it is necessary to register everything electronically, so we have no choice but to start recording everything consistently. When I do something as a business owner, I do it properly. This means that I want more than just sales records from the newly created system. I would like the system to focus on products, suppliers, employees, customers, and everything related to our tavern. I will try to describe here how it works with us:

We buy input raw materials from various suppliers. We only take first-class quality. Some of the purchased products are only resold with the appropriate margin; some are used for the production of our products. For all of them, we want to know who is the supplier, when it was delivered, in what quantity and how much it cost. The amount of raw materials is measured in units (meat in kg, milk in l, rolls in pieces). We know the name and address of the suppliers, and for some, we also know the phone, email, and contact person (name and surname). We can also write a note for each supplier.

From what we buy, we cook meals, however, also just put something for sale. Our basic food and beverage menu are quite unchangeable, but I would like every change to be recorded and traceable, including the price. On weekdays we prepare a lunch menu for our guests. It is only served from 11:00 until the stock is sold out for the day. I emphasize once again that I want to be able to look at what the offer was that day, maybe two years back. The food and beverage menu is divided into categories. These categories can be nested (eg food -> main -> meat -> pork). We also distinguish the type of offer, i.e. whether it is a permanent offer, special, or menu.

According to the law, for everything we sell, we have to record what allergens it contains, so it should also be included there.

Several chefs take part in the preparation of meals for each shift. There is one main one for each shift. In the production of food, it is necessary to record how much of a given shift was consumed raw materials and how many individual meals were sold. It would be ideal for determining how many ingredients were needed to produce a particular food, but in my experience, this is unrealistic.

Other employees are involved in the service. He/she always holds one of the functions within a given shift. He/she is either a waiter, or he/she's behind a bar. The worker behind the bar prepares drinks for the waiters but can also sell them directly to customers. The waiter is always assigned certain tables for a given shift, which he/she has to take care of and from which he then receives money to spend. Spending at the bar is cashed by the bartender. I always want to know who received the money.

Regular customers have their account with us, and if they spend more than two hundred euros in the previous month, they have a ten per cent discount the following month. That is why we want to record with regular customers what they bought from us. Of course, I need to record all our sales, regardless of whether or not it is a regular customer.

I'm thinking about introducing happy hours. For example, that Monday from 16:00 to 19:00 would be a discount. Are you able to figure out how to record it for me?

Data modelling

Data modelling is a process that aims to create a data model. The data model describes the data and its structure. Data can be viewed from three levels.

The first is the external view. It is sometimes referred to as an application or user view of data. It is the view of the regular user who is not interested in deeper connections between data. He/she is usually just their consumer. As an example, imagine an online store customer. He/she sees a range of products. When he/she chooses one and adds it to the cart and then pays, he/she does not care at all how the data is organized in the e-shop, where it is stored, etc. It only consumes their content. An example of such a model is the above scenario.

The second view is called logical or conceptual. Some authors define a difference between the logical and conceptual views. The logical one is more detailed. In both cases, however, it is a matter of identifying important objects of interest and the relationships between them. It does not solve its own implementation. It covers current needs with the possibility of further development. For example, with an online store, it would be a matter of identifying important objects, such as products, categories, customers, orders, and more. In addition, it is necessary to recognize the interrelationships between these objects.

The third view is the physical view. This view looks at how data is stored. It is a design proposal for implementation in a specific database (or other) system. It contains tables, object structures, and integrity constraints. For example, a saving process in Excel uses workbooks, sheets, columns, cells, etc., in a relational database system using tables, indexes, integrity constraints, etc. Perhaps there is no need for an example with an online store.

This case study will focus on logical modelling. For the modelling, Barker's notation will be used.

Entity identification

Before embarking on identifying entities, it is important to understand what entity is. An entity can be likened to a class in object-oriented programming. It generalizes the real-world elements about which data needs to be stored. Specific occurrences are referred to as instances. An example of an entity can be sports and instances of football, archery, volleyball, etc.

Are you able to recognize an instance and an entity? How about a "dog"? Is it an instance of an entity? What do you think?

You cannot tell from one word. It always depends on the context. If you are creating a data model for a dog shelter, it is very likely that the dog will be an entity and will have its instances (specific dogs in the shelter). A dog, on the other hand, maybe an instance of an animal entity in some other model. It is, therefore, necessary to look at the problem as a whole.

Entities are characterized by their attributes. For example, an entity sport can have the attributes name, number of players, and whether it takes place outdoors or indoors.

Attributes are used to specify entities in more detail and thus allow the individual instances of a given entity to be distinguished from each other. Attributes describe the properties of entities in more detail. Each attribute quantifies, qualifies, classifies, or refines the relevant entity.

Attributes may also be related to integrity constraints. These are additional rules for ensuring the conformity of the model with the modelled reality. More about integrity constraints will be written later.

In data modelling, the first task is to identify entities. These act as nouns. However, nouns can also be instances or attributes. So when identifying entities, it is important to think carefully about what an entity is, what an instance is, and what an attribute is. Before you continue reading, try to identify entities in this scenario. It can help you highlight nouns.

Here is a possible list of entities in alphabetical order:

- address,
- allergen,
- category,
- customer,
- discount,
- employee,
- ingredient,
- job position,
- offer,
- offer type,
- product,
- raw material,
- raw material consumption,
- raw materials purchase,
- sale,
- sales item,
- shift,
- supplier,
- table,
- unit,
- work classification.

If you have it differently, it does not mean that it is wrong. This is one of the possibilities. Entities are usually named in a single number. In the model, entities are represented by rectangles.

Relationship identification

Another important part of the data model represented by an entity-relationship diagram is the relationships between entities. Identifying relationships may not be easy, especially when there are many entities. It is easy to forget a relationship. To prevent this situation, there is a tool called a matrix diagram. A matrix diagram is a table that has entity names in both row and column headers.

You can see this in the following example.

	address	allergen	category	customer	discount	employee	ingredient	job position	offer	offer type	product	raw material	raw material consumption	raw material purchase	sale	sale item	shift	supplier	table	unit	work classification
address																					
allergen																					
category																					
customer																					
discount																					
employee																					
ingredient																					
job position																					
offer																					
offer type																					
product																					
raw material																					
raw material consumption																					
raw material purchase																					
sale																					
sale item																					
shift																					
supplier																					
table																					
unit																					
work classification																					

When you identify relationships between entities using a matrix diagram, you go through the individual cells and ask questions. Is there a relationship between the entity in the row and the entity in the column? If so, name it. Relationships are usually expressed by verbs. It is important to realize that a relationship has two names. Entity A has a relationship with Entity B, but Entity B has a relationship with Entity A as well. Both may have different names. Let's look at an example. We will not go through the whole table now, but we will show it on a subset of it to make it clearer. Let's use entities product, raw material, category, and ingredients. The empty matrix diagram with these entities looks like this table.

	product	raw material	category	ingredient
product				

	product	raw material	category	ingredient
raw material				
category				
ingredient				

Let's start asking! Does a product have a relationship with another product? There is no such relationship so leave this cell empty.

Next question: Does a product have a relationship with raw material? Now we could say yes. The product consists of raw materials. However, you must be really careful. There is also the entity ingredient. This entity states that raw material is an ingredient for a product. So even there is a relationship between a product and a raw material, we will not put it in the matrix diagram because we want to have only direct relationships.

Let's continue! Is there a relationship between a product and a category? Finally, we can say yes. How should the relationship be named? What about a product is categorized by category? As was already stated, we need two names for each relationship. We can use a category to categorize a product.

The last question involving the entity product is: Does a product have a relationship with an ingredient? It is obvious that it does. A product consists of ingredients, and an ingredient is a part of a product.

After these four questions, the matrix diagram will look like this.

	product	raw material	category	ingredient
product			is categorized by	consists of
raw material				
category	categorizes			
ingredient	is a part of			

Now, we do not need to care about the entity product. We solved all its relationships. As you can see, only three entities left to take care of them. So, does a raw material have a relationship with a raw material? The answer is no.

Is there a relationship between raw material and a category? Again no. The last question, including the entity raw material, is: Does a raw material have a relationship with an ingredient? Now, we can

say yes. The raw material is an ingredient, and an ingredient is a raw material. The word "is" is not a very good name for a relationship. If you can come up with something else, use it. However, sometimes there is no other meaningful possibility.

We move forward with this matrix diagram.

	product	raw material	category	ingredient
product			is categorized by	consists of
raw material				is an
category	categorizes			
ingredient	is a part of	is a		

Let's finish it up. Does a category have a relationship with a category? From previous entities, it might seem that entity does not have a relationship with itself. Usually, it doesn't. However, there are some cases when it does. In the scenario, you can read: "The food and beverage menu is divided into categories. These categories can be nested (eg food -> main -> meat -> pork)." If categories can be nested, that means there can be subcategories. So, there is a relationship between the two categories. One is superior, and the other subordinate to the first. There is a little problem. How to write two names into one cell? You can either split the cell or use a slash.

Is there a relationship between a category and an ingredient? No, there isn't. Is there a relationship between an ingredient and an ingredient? No, there isn't.

	product	raw material	category	ingredient
product			is categorized by	consists of
raw material				is an
category	categorizes		is superior to subordinates to	
ingredient	is a part of	is a		

We finished our small matrix diagram. Easy, isn't it? Now you can practice on the big one.

To finish the big one will take a little bit of time. However, you'll surely make it. Here is an example of how it can be done. Instead of the names of the constraints in the matrix, you will find only X. The matrix with all the names of the constraints would not fit here.

	address	allergen	category	customer	discount	employee	ingredients	job position	offer	offer type	product	raw material	raw material consumption	raw material purchase	sale	sale item	shift	supplier	table	unit	work classification
address						X												X			
allergen												X									
category			X								X										
customer					X										X						
discount				X																	
employee	X																				X
ingredients											X	X									
job position																					X
offer										X	X										
offer type									X												
product			X				X		X							X					
raw material		X					X						X	X						X	
raw material consumption												X					X				
raw material purchase												X						X			
sale				X												X					X
sale item											X				X						
shift													X								X
supplier	X													X							
table																					X
unit												X									
work classification						X		X							X		X		X		

Relationships between entities are specified outside their names by whether they are mandatory or optional for instances of those entities. This is called relationship optionality. Next, for instances of entities in a relationship, it is determined by how many instances of other entities they can be in a relationship. This is called the cardinality of the relationship.

It is, therefore, necessary to determine cardinality and selectivity for each relationship. This can be achieved through questions. For optionality, we ask: Must the instance of entity A have a relationship with the instance of entity B? The obligation must be determined from the opposite side. Does the entity B entity have to relate to the entity A instance? Using the example from the scenario, let's determine the selectivity of the relationship between the product and category entities.

- Does each product have to be categorized by category?
- Does each category have to categorize a product?

Let's say that each product must be a categorized category, and a category can categorize a product.

In the resulting model, the relationships are represented by lines. If the relationship is mandatory for instances of the entity, the line is solid. If the relationship is optional, then the line is dashed.

When determining cardinalities, there are three options to choose from:

- 1: 1,
- 1: N,
- N: M.

You can find out cardinality with suitable questions.

- How many instances of entity B can be in a relationship with an instance of entity A? With one or more?
- How many instances of entity A can be in a relationship with an instance of entity B? With one or more?

In the case of relationships between product and category entities, we ask the following questions:

- How many categories can one product be categorized? One or more?
- How many products can one category categorize? One or more?

The scenario shows that a product is included in one category, but a category can include multiple products. This is a 1: N cardinality.

In the model, the cardinality is represented by a single toe or crow's foot.



For a better understanding, a language called ERDish was created. The two sentences clearly describe the relationship between the two entities. The syntax is as follows:

1. Each
2. Entity A
3. Optionality (must be/may be)
4. Relationship name
5. Cardinality (one and only one/ one or more)
6. Entity B

and also in the opposite direction.

Let's create ERDish sentences for the relationship between product and category entities.

1. Each
2. product
3. must be
4. categorized by
5. one and only one
6. category.

And from the other side.

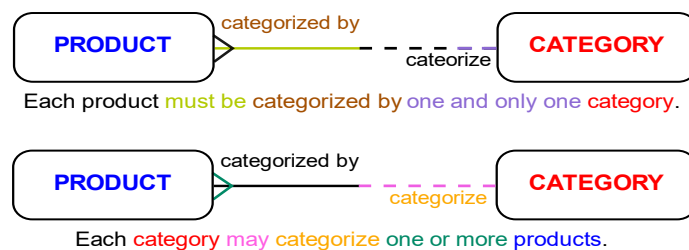
1. Each
2. category
3. may
4. categorize
5. one or more
6. products.

So now we have two sentences that describe the relationship between two entities, product and category.

Each product must be categorized by one and only one category.

Each category may categorize one or more products.

With the help of ERDish sentences, you will clarify everything you need about a given relationship, in addition to a form that almost everyone will understand. Everything important is clarified, so it is possible to bring the relationship into the model.



Let's create ERDish sentences for restore of relations from our small matrix diagram.

PRODUCT – INGREDIENT

Each product must consist of one or more ingredients.

Each ingredient must be part of one and only one product.

RAW MATERIAL – INGREDIENT

Each raw material must be one or more ingredients.

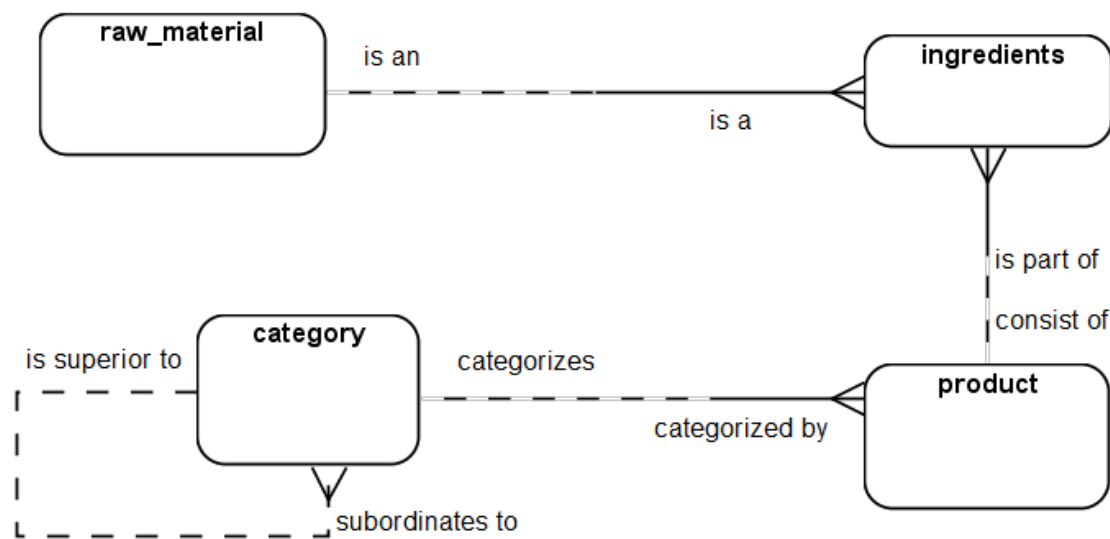
Each ingredient must be one and only one raw material.

CATEGORY – CATEGORY

Each category may be superior to one or more categories.

Each category may be subordinated to one and only one category.

Now we can transfer these sentences into the model.



Attributes

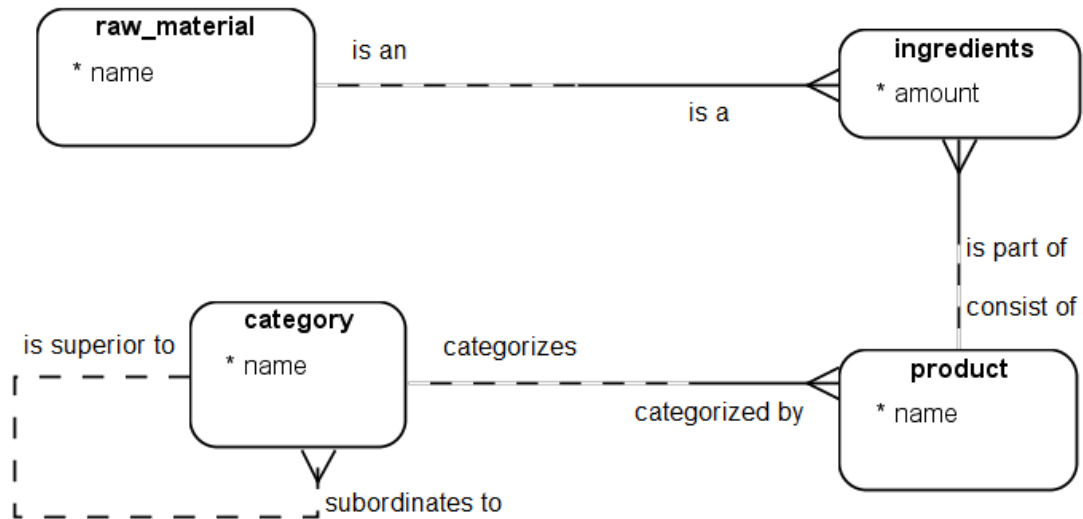
In order to be able to specify entities in more detail, and thus to enable the individual instances of a given entity to be distinguished from each other, the entities have their attributes. Attributes describe the properties of entities in more detail. The individual attributes quantify, qualify, classify or refine the relevant entity.

The attribute takes exactly one value (number, character string, date, image, sound, etc.) of the respective domain (a subset of values of a certain data type) - e.g. age is an integer from 0 to 120. However, the logical model does not deal with the domain much. Attributes may also be related to integrity constraints. These are additional rules to ensure the compliance of the model with the modelled reality (e.g. the date of order fulfilment must be greater than or equal to the date of order acceptance).

However, attributes are distinguished not only by the domain but also by whether each instance of the entity must have a value for the attribute. It is talked about whether the attribute is mandatory or optional. The optionality of individual attributes again depends on the input scenario. When you look for attributes in the scenario, you again need to choose between nouns. For nouns, it will be decided here whether it is an entity or an attribute. However, some nouns may also represent instances or may not play any role in the model.

When modelling, it is necessary to agree on the rules of what the model will look like so that everyone understands its meaning. The notation is called notation. There are several notations for creating data models, and it cannot be said that one of them would play a primary role. As mentioned in the introduction, this case study will use Baker's notation.

As we already know, in this notation, entities are denoted as rectangles. The first line contains the name of the entity. It is usually written in a singular number. The following lines list the attributes, including their optionality. If the attribute is required, it has an asterisk in front of it. If it is optional, it has a circle in front of it. So let's add attributes to the four entities from our small diagram.



That wasn't so difficult, was it? However, don't you miss anything? Each instance of a given entity needs to be uniquely identified in order to work with it. Identifiers are used for this.

An identifier is an attribute or combination of attributes that uniquely distinguish one instance from another. It is a unique identifier for just one instance. It is often used the abbreviation UID. If the identifier is composed, it is a combination of multiple attributes. This is the case when one attribute is not sufficient for identification.

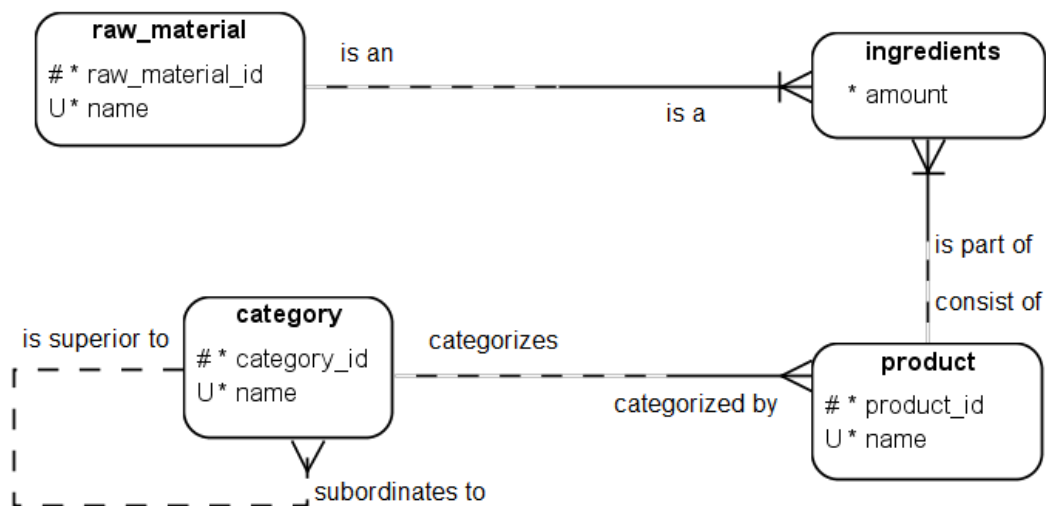
In some cases, it is not possible to create an identifier even by combining all the attributes of an entity. In such cases, an artificial identifier comes into play. A numeric attribute (natural number) is usually chosen as an artificial identifier. Such an attribute is most often named as an id or in combination with the name of an entity.

There are also cases where the entity has more identifiers. One of them is chosen as the primary. The others are referred to as candidate (secondary) identifiers. An artificial identifier is created even in cases where another identifier exists, but for certain reasons, it is not suitable as a primary identifier.

When an entity is created, it must always have a primary identifier specified. It is necessary to be able to unambiguously distinguish one instance from another in order to be able to work with the given instance (select, edit, delete). The primary identifier must therefore be a mandatory attribute. If it is composed of several attributes, all of them must be mandatory. Secondary attributes can also be marked as optional.

As part of the design of the data model, it is necessary to record the facts about identifiers. A grid is used for the primary identifier. U is used for other unique identifiers.

So let's add identifiers.



All name attributes can be marked as unique. What good would two categories (raw materials, products) of the same name be for? Nevertheless, it is advisable to introduce an artificial primary identifier here. It is from a practical point of view. Relationships are represented by transferring the primary identifier of the source entity to the destination. If, for example, the product name changed, a change would have to be made to all instances of the ingredients entity. This way, the change can be made in one place.

It seems to you that the ingredient entity does not have a primary identifier. If you look closely, you will see that vertical lines have been added to the lines representing the relationships. These determine that the transmitted foreign identifiers become part of the primary identifier. Thus, the ingredient entity has a primary identifier composed of a raw material identifier and a product identifier. It does not make sense to have a given raw material twice in one product. Just determine the amount.

Unlike other notations, this notation does not display transmitted foreign identifiers. It may be confusing at first, but you get used to it. For example, an ingredient entity has three attributes (`raw_material_id`, `product_id`, and `amount`).

So now it's up to you to try to complete the whole model. The principle has already been explained. Then you can compare your solution with model one. Just because it doesn't match one hundred per cent doesn't mean you're wrong. There may be more suitable solutions. This is just one of them.

The experience takes you to be able to design a data model correctly. The more models you create and put into practice, the better and easier they will be for you to create.

This cannot be said to be the only way to proceed. Everyone has to figure out for themselves what suits them. Someone first assigns attributes to entities and then resolves relationships. That is also one of the possibilities. It is ideal for working in a team and discussing the design. When two or more agree, they are more likely to make sense. By designing the model itself, you will not know if it is badly designed. You will find out this only during its implementation. Designing it correctly will save teachers a lot of time repairing it.



For designing a data model, it is important to have abstract thinking and a good imagination. But even so, you will not achieve the championship until after several attempts. Experience is very important here.

Knowledge Discovery from Log File

The presented example shows the complex process that has to be done to discover knowledge from a log file. The log file came from a university web server and was used as source data. The log file is in standard log file format. The source web server contained information on any logged event on a website. The results of this task will be interesting rules of behaviour of web users obtained using association analysis.

Recommended Number of Developers

Individual.

Requirements

The log file was obtained from a university web server. This dataset has to be preprocessed and analysed using data mining methods, for this will be used the Python programming language using the Jupyter Notebook. We will require to import the pandas data analysis library for preprocessing and the MLxtend library for association analysis.

Methodology

The application was made following several steps:

1. Data acquisition – defining the observed variables in the log file from the point of viewing the necessary data (IP address, date and time of access, URL, UserAgent);
2. Data preprocessing – consists of multiple steps to obtain data matrix without ambiguous data:
 - a. Data cleaning – removing the access to images, videos, javascript, css and similar files. Also removing the accesses of the robots of search engines;
 - b. User identification – identifying users based on the IP address and UserAgent;
 - c. Session identification – identifying sessions using a fixed time window – an estimate based on a quartile range that is not affected by extreme values, e.g. $Q3 + 1.5Q$, where $Q3$ is the upper quartile (75th percentile), and Q is the quartile range (mean 50% of the values). In other words, if we consider the time spent on the site to be a remote value, a new session begins;
3. Data analysis – searching for behavioural patterns of web users using association analysis.
4. Understanding the data.

Solution

The work will be done according to the methodology and supplemented with the source code.

The first step is to import all of the libraries that will be used during the process.

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

Data cleaning

Next will be the examination of the log file. We load the log file to the pandas dataframe and create a header. The standardized log file consists of IP address, cookie, user information, datetime and time zone, requested site, request code, bytes, referrer and user agent.

```
columns = ['IP', 'Cookie', 'User', 'Datetime', 'TMZ', 'Request', 'ReturnCode',
           'Bytes', 'Referrer', 'UserAgent']
df = pd.read_csv('Log_file.log', header=None, names=columns, sep=' ')
```

Using the **head** function of the dataframe we can see the first five rows of the dataframe and correct any mistakes done in the header.

```
df.head()
```

Out[3]:

	IP	Cookie	User	Datetime	TMZ	Request	ReturnCode	Bytes	Referrer	UserAgent
0	193.87.12.30	-	-	[19/Feb/2020:06:25:50	+0100]	GET /navbar/navbar-ukf.html HTTP/1.0	200	7584	-	-
1	193.87.12.30	-	-	[19/Feb/2020:06:25:55	+0100]	GET /navbar/navbar-ukf.html HTTP/1.0	200	7584	-	-
2	193.87.12.30	-	-	[19/Feb/2020:06:25:56	+0100]	GET /navbar/navbar-ukf.html HTTP/1.0	200	7584	-	-
3	193.87.12.30	-	-	[19/Feb/2020:06:25:57	+0100]	GET /navbar/navbar-ukf.html HTTP/1.0	200	7584	-	-
4	193.87.12.30	-	-	[19/Feb/2020:06:25:49	+0100]	GET / HTTP/1.1	200	20925	-	libwww-perl/6.08

Following that is the most important part, to clean the data and remove unnecessary access to the web portal. Log files are typical in that they contain a considerable amount of irrelevant data that can corrupt the analysis of the data, so it is necessary to delete this data already in the data preparation phase. Data cleaning aims to delete records, i.e. links that are not essential to the behaviour of web users. Such links mainly include approaches to:

- picture,
- flash videos,
- cursor icons
- javascript,
- style.

The usual procedure for identifying such records involves identification based on the extension (*.jpg, *.jpeg, *.bmp, *.png, *.gif, *.css, *.js, *.flw, *.swf, *.cur, *.rss, *.ico, *.xml, fonts and the like). Even if only one page is loaded, all these requests are written to the log file.

```
suffix = (".jpg", ".jpeg", ".png", ".gif", ".bmp", ".css", ".flv", ".ico", ".swf",
          ".rss", ".xml", ".cur", ".js", ".json", ".svg", ".woff", ".eot", ".font", "POST",
          "HEAD", ".JPG")
for s in suffix:
    df.drop(df[df['Request'].str.contains(s)].index, inplace=True)
```

In addition to the GET request, other HTTP protocol requests are written to the log file, such as 4xx / 5xx return or status codes, which identify the client/server error that needs to be cleared.

The HTTP status code is part of the server response header for the client request. Specifies how the response was processed by the server - whether the request was processed positively, negatively, or an error occurred. The next step is for the client to interpret and respond to the response status code.

The status code is three decimal numbers, where the first number specifies the category of the answer and the remaining numbers specify it in more detail:

- 1xx Information,
- 2xx Successful,
- 3xx Redirect,
- 4xx Client error,
- 5xx Server error.

```
df.drop(df.loc[df.ReturnCode >= 400].index, inplace = True)
df.drop(df.loc[df.ReturnCode < 200].index, inplace = True)
```

The next step in data preparation is to clean the data from accesses by search engine robots such as Google, Yahoo, Bing, etc. Because robots access the web portal sequentially, it is not appropriate to include their activity in the study of user behaviour.

The robots are detected either based on their identification in the User-Agent field or based on an IP address that can be compared with the robot database, which can be found, for example, at www.robotstxt.org. Identification of search engine robots can be performed using:

- keyword bot, spider, crawl, robot,
- hidden link access,
- robots.txt accesses.

First, we will identify the IP addresses that accessed the robots.txt file. We save the IP addresses to a list and use it in another iteration to remove these accesses from the log file.

```
robotstxt = df[df['Request'].str.contains("robots.txt")]
ips = robotstxt['IP']

for ip in ips:
    df.drop(df[df['IP'].str.contains(ip)].index, inplace=True)
```

Next, we can use again the keywords used by the search engine robots to identify robots in the User-Agent column. This column contains the information about the used web browser and operating system. The search engines often use their names in this column.

```
df.drop(df[df['UserAgent'].str.contains('bot')].index, inplace=True)
df.drop(df[df['UserAgent'].str.contains('crawl')].index, inplace=True)
df.drop(df[df['UserAgent'].str.contains('spider')].index, inplace=True)
```

Here we could end with the data cleaning. However, it is recommended to check the cleaned log file to whether it does not contain some unnecessary data that is not usually cleaned. We will use a simple frequency table and look at the Request column. Pandas offers a crosstabulation function where if we input one column, it will create a frequency table.

```
req_tab = pd.crosstab(index=df['Request'], columns="count")
req_tab = req_tab.sort_values(by=["count"], ascending=False)
req_tab.head(15)
```

We will look at the 15 most frequent requests.

Out [10]:

	col_0	count
Request		
GET /navbar/navbar-ukf.html HTTP/1.0		24854
GET / HTTP/1.1		5534
GET /index.php?option=com_acymailing&ctrl=cron HTTP/1.1		1440
GET /univerzita/kontakt/adresar-osob HTTP/1.1		495
GET /verejnost/aktuality/foto/image?view=image&format=raw&type=orig&id=16556 HTTP/1.1		319
GET /prijimacie-konanie/prihlasovanie-sa-na-studium HTTP/1.1		297
GET /verejnost/aktuality/foto/predaj20 HTTP/1.1		281
GET /images/oznamy/2020_Oznamy/Univerzita_Kon%C5%A1tant%C3%ADna_Filozofa_-_ponuka_na_predajauta_NR_120_GL_-_final.pdf HTTP/1.1		250
GET /verejnost/aktuality/foto/image?view=image&format=raw&type=orig&id=16557 HTTP/1.1		238
GET /fakulty-a-sucasti/filozoficka-fakulta HTTP/1.1		215
GET /verejnost/aktuality/foto/image?view=image&format=raw&type=orig&id=16558 HTTP/1.1		213
GET /fakulty-a-sucasti/pedagogicka-fakulta HTTP/1.1		190
GET /studium/organizacia-studia/harmonogram-akademickeho-roka HTTP/1.1		180
GET /verejnost/aktuality/foto/image?view=image&format=raw&type=orig&id=16560 HTTP/1.1		174
GET /fakulty-a-sucasti HTTP/1.1		171

The table shows that there are two strange requests. The most frequent is a navbar page. This page seems like it is loaded by each web page as part of a build-up. This is an access that we do not want to have in our analysis, so we will also remove all the accesses to the navbar. Next on the third place is a request that seems like a cron job. This could be an automated script for backup or antivirus that checks the web portal. These accesses can also be removed as it is very unlikely that these accesses were made by web visitors.

```
df.drop(df[df['Request'].str.contains('navbar')].index, inplace=True)
df.drop(df[df['Request'].str.contains('cron')].index, inplace=True)
```

User/session identification

Firstly, we will focus on the **dummy variables** that we will use in the next phases of data preparation. These are mainly time-based variables or other variables that are needed for the later phase of data analysis (for example, distinguishing user access, etc.). The first essential variable is a variable representing the date and time of access to the web portal. Thanks to this variable, we can later identify individual user sessions. It usually has a date and time format, and because the webserver can handle different format settings, the formats may differ. For example, the source data date fields are in the format YYYY / MM / DD, and the target date fields are in the format MM-DD-YYYY. It is necessary to unify these formats. To unify the formats, we will use the so-called UNIXtime and use the transformation to convert the date field of the source data to the corresponding target format.

UNIXtime (also known as Epoch time, POSIX time, seconds since Epoch, or UNIX Epoch time) is a time point description system. It is the number of seconds that have elapsed since the Unix era, minus the leap seconds; the Unix epoch is dated January 1, 1970 00:00:00 UTC; leap seconds are ignored, with a leap second having the same Unix time as the second before it, and each day is considered to be exactly 86 400 seconds long. Thanks to this approach, Unix time is not a true expression of UTC.

We will create two user functions *getMonthNum* and *parseDateToUnix*, to obtain the UNIXtime from the Datetime column.

```
import datetime
```

```
def getMonthNum(month):
    return {
        'Jan': 1,
        'Feb': 2,
        'Mar': 3,
        'Apr': 4,
        'May': 5,
        'Jun': 6,
        'Jul': 7,
        'Aug': 8,
        'Sep': 9,
        'Oct': 10,
        'Nov': 11,
        'Dec': 12
    }.get(month, -1)

def parseDateToUnix(dat):
    day = dat[1:3]
    month = getMonthNum(dat[4:7])
    year = dat[8:12]
    hour = dat[13:15]
    minute = dat[16:18]
    second = dat[19:21]
    return datetime.datetime(int(year), int(month), int(day), int(hour),
int(minute), int(second)).timestamp()
```

Now we can use a lambda function and create a new column (variable) called UNIXtime in our dataframe.

```
df['Unixtime'] = df.apply(lambda row: parseDateToUnix(row['Datetime']), axis=1)
```

Another dummy variable that we will need is the time spent on the page, i.e. length of time spent on the page. It is mainly used to identify sessions, and it is used to refer to this variable as length. When creating the length variable, it is necessary to start from the Unix time stamp and have a log file sorted according to the following fields:

- IP address,
- UserAgent,
- UNIXtime.

```
df = df.sort_values(by=["IP", "UserAgent", "Unixtime"])
```

This will ensure sequential follow-up of the approaches of individual visitors. The log file primarily records anonymous user data, but there is a problem with uniquely identifying the site visitor. In the analysis, it is not necessary to know the specific identity of the user but to be able to distinguish between individual users. The assumption that an IP address is sufficient to identify a user is incorrect because there can be multiple users behind one IP address. Because the IP address is not a sufficient parameter to identify the user, it is necessary to combine several methods, such as using the Cookie field or a combination of the IP address with the User-Agent field. Several heuristic methods mainly use a combination of an IP address with a User-Agent field. If the IP address changes, it is clear that it is a new user. If the IP address is the same, the User-Agent field is compared, and if there is a change, a new user is identified. Otherwise, it is the same user.

```

user = []
ip_before = "null"
agent_before = "null"
usid = 1
for ip,agent in zip(df['IP'], df['UserAgent']):
    if ip_before!="null":
        if ip_before==ip and agent_before==agent:
            user.append(usid)
        else:
            usid+=1
            user.append(usid)
    else:
        user.append(usid)
    ip_before = ip
    agent_before = agent

df['UserID'] = user

```

We will create the length variable by going through the whole log file and comparing two consecutive records. If there are equal IP addresses and also a User Agent in two consecutive records, we can read the Unix access times between these two records. In this way, we get the time spent on the page of the first record.

The time spent on the site is always positive! It is advisable to choose the upper limit of the so-called time window, and we assume that if the time between two visited pages is greater than, for example, 1 hour, then it will be a new visit. We can choose the size of the time window according to the needs of the web portal.

```

length = []
unxtm_before = -1
usr_before = -1
for unxtm,usr in zip(df['Unixtime'],df['UserID']):
    if usr_before!=-1:
        if usr==usr_before:
            unx_dif = unxtm-unxtm_before
            if unx_dif<=3600:
                length.append(unx_dif)
            else:
                length.append(None)
        else:
            length.append(None)
    unxtm_before = unxtm
    usr_before = usr

length.append(None)
df['Length'] = length

```

A user can visit a specific page multiple times, in which case a multiple session (visit) is recorded for each user in the log file. However, to work with data, we need to distinguish individual sessions, to divide the individual approaches of each user into separate sessions. This is done by session identification, which is one of the most important steps in data preprocessing. Sessions can also be distinguished by time. The simplest method is if we consider a session to be a series of clicks over

some time - a time window, e.g. in 10 minutes, 30 minutes, etc. The duration of the session must not exceed the value of the time window.

An alternative to a fixed time window is an estimate based on a quartile range that is not affected by extreme values, e.g. $Q3 + 1.5Q$ where $Q3$ is the upper quartile (75th percentile) and Q is the quartile range (mean 50% of the values). In other words, if we consider the time spent on the site to be a remote value, a new session begins.

```
uppQ = sdf['Length'].quantile(0.75)
lowQ = sdf['Length'].quantile(0.25)
Q = uppQ + 1.5 * (uppQ-lowQ)
```

When we have calculated the quartile range, we can use it as a cutoff time of the time window and identify sessions for the users.

```
sttQ = []
usr_before = -1
length_before = -1
stQ = 1
for length,usr in zip(df['Length'],df['UserID']):
    if usr_before!=-1:
        if usr==usr_before:
            if length_before<=Q:
                sttQ.append(stQ)
            else:
                stQ+=1
                sttQ.append(stQ)
        else:
            stQ+=1
            sttQ.append(stQ)
    else:
        sttQ.append(stQ)
    usr_before = usr
    length_before = length

df['STT_Q'] = sttQ
```

The raw log file started with more than 250 000 rows. After the data cleaning phase remained in the dataframe only 20 000 rows, this shows the importance of data preprocessing as the raw log file contained a lot of unnecessary data for our analysis. We have identified around 4 000 users that accessed the web portal and around 8 000 sessions.

Data transformation

Before we can obtain any relevant knowledge from our log file, we need to transform the data into a more appropriate format. We want to examine the accessed web pages, but there are many various pages and sub-pages. This would result in small frequencies of accesses, and nothing interesting would come out of the analysis. We need to create a new variable (column) that will represent a web category. The logic can be simple: we take the name of the page after the first slash in the request variable. If there is a request only to a slash, that means it is the homepage of the web portal.

```
def parseCategory(req):
    if req == "GET / HTTP/1.1":
        return "home"
    strng = req.split("/")
```

```

    if len(strng)>2:
        out = strng[1].replace(" HTTP", "")
        if out=="en":
            return "home"
        else:
            return out

df['Category']= df.apply(lambda row: parseCategory(row['Request']), axis=1)

```

Once again we should examine the created variable and for that, we use a frequency table.

```

req_tab = pd.crosstab(index=df['Category'], columns="count")
req_tab = req_tab.sort_values(by="count", ascending=False)
req_tab.head(15)

```

From the table, we can see that there is a maximum of 10 categories that have more access.

```

Out[23]:

```

col_0	count
Category	
verejnost	6966
home	5947
fakulty-a-sucasti	1283
univerzita	1269
images	997
prijimacie-konanie	897
studium	833
media-a-marketing	246
administrator	227
component	217
helpdesk	112
oznamy	107
tz	107
mojaukf	104
ubytovanie	99

The other categories can be joined into one category named other.

```

def updateCategory(cat):
    if (cat=="verejnost" or cat=="home" or cat=="fakulty-a-sucasti" or
        cat=="univerzita" or cat=="images" or cat=="prijimacie-konanie" or
        cat=="studium" or cat=="media-a-marketing" or cat=="administrator" or
        cat=="component"):
        return cat
    else:
        return "other"

df['Category'] = df.apply(lambda row: updateCategory(row['Category']), axis=1)

```

Now we can check how many unique categories we have using a unique function.

```
df.Category.unique()
```

We can see that now we have only 11 categories that we will analyse.

```
Out[33]: array(['home', 'univerzita', 'other', 'fakulty-a-sucasti', 'studium',
               'verejnost', 'images', 'component', 'prijimacie-konanie',
               'media-a-marketing', 'administrator'], dtype=object)
```

Finally, we are getting closer to the data analysis. We need to consolidate the items into 1 transaction per row with each web category one-hot encoded. First, we will transpose the accessed web categories into columns. This way we get the accessed web categories for each session in one row.

```
sess = df.sort_values('STT_Q').groupby('STT_Q')['Category'].apply(lambda df:
    df.reset_index(drop=True)).unstack()
```

```
Out[29]:
```

	0	1	2	3	4	5	6	7	8	9	...	194	195	196	197	198	199	200	201	202	203
STT_Q																					
1	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...
8002	other	prijimacie-konanie	fakulty-a-sucasti	other	other	fakulty-a-sucasti	other	fakulty-a-sucasti	fakulty-a-sucasti	fakulty-a-sucasti	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8003	home	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8004	home	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8005	verejnost	other	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8006	prijimacie-konanie	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

8006 rows × 204 columns

Next, we take the unique web categories that will be used to create the one-hot encoded items.

```
items = df.Category.unique()
```

Now we are ready to create the one-hot encoding that will be needed for the association analysis.

```
itemset = set(items)
encoded_vals = []
for index, row in sess.iterrows():
    rowset = set(row)
    labels = {}
    uncommons = list(itemset - rowset)
    commons = list(itemset.intersection(rowset))
    for uc in uncommons:
        labels[uc] = 0
    for com in commons:
        labels[com] = 1
    encoded_vals.append(labels)
encoded_vals[0]

ohe_df = pd.DataFrame(encoded_vals)
```

Data analysis

Association analysis is relatively light on the math concepts and easy to explain to non-technical people. It is a good start for certain cases of data exploration and can point the way for a deeper dive into the data using other approaches.

Support is the relative frequency that the rules show up. In many instances, you may want to look for high support to make sure it is a useful relationship. However, there may be instances where low support is useful if you are trying to find “hidden” relationships.

Confidence is a measure of the reliability of the rule. Confidence of 0.5 would mean that in 50% of the cases where two categories were accessed, the session also included other categories. For product recommendations, a 50% confidence may be perfectly acceptable, but in a medical situation, this level may not be high enough.

Lift is the ratio of the observed support to that expected if the two rules were independent. The basic rule of thumb is that a lift value close to 1 means the rules were completely independent. Lift values > 1 are generally more “interesting” and could be indicative of a useful rule pattern.

One final note related to the data: this analysis requires that all the data for a session be included in 1 row, and the items should be one-hot encoded.

We have everything ready from our log file and only need to import the library used for association analysis.

```
from mlxtend.frequent_patterns import apriori, association_rules
```

Apriori is an algorithm for frequent itemset mining and association rule learning over relational databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets as long as those item sets appear sufficiently often in the database. The frequent itemsets determined by Apriori can be used to determine association rules which highlight general trends in the database: this has applications in domains such as market basket analysis. Apriori algorithm is the perfect algorithm to start with association analysis as it is not just easy to understand and interpret but also to implement. Python has many libraries for apriori implementation. One can also implement the algorithm from scratch. But as there are many solutions, we will use the library MLxtend. This library has a beautiful implementation of apriori, and it also allows to extract the association rules from the result.

Apriori module from MLxtend library provides fast and efficient apriori implementation. The model will generate frequent itemsets. The output is a data frame with support for each itemset.

```
freq_items = apriori(ohe_df, min_support=0.01, use_colnames=True, verbose=1)
freq_items = freq_items.sort_values(by=["support"], ascending=False)
freq_items.head()
```


We generated the frequent itemsets and sorted them based on the support to see the most frequent web categories.

Out[37]:

	support	itemsets
9	0.545966	(home)
0	0.125781	(verejnost)
4	0.119910	(other)
7	0.111167	(univerzita)
5	0.102798	(fakulty-a-sucasti)

The most accessed web category is the home page, and then the next most accessed are categories public (verejnost), other, university (univerzita) and faculty (fakulty-a-sucasti). In the final step, we will find the association rules for the frequent itemsets. Let us look at the rules based on the lift, where we would like to see the rules that have the minimum support of 0.01. This will generate rules that can be interesting for our interpretation.

```
rules = association_rules(freq_items, metric="support", min_threshold=0.01)
rules = rules.sort_values(by=["support"], ascending=False)
rules.head()
```

Out[41]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(fakulty-a-sucasti)	(home)	0.102798	0.545966	0.038346	0.373026	0.683240	-0.017778	0.724167
1	(home)	(fakulty-a-sucasti)	0.545966	0.102798	0.038346	0.070236	0.683240	-0.017778	0.964978
2	(home)	(univerzita)	0.545966	0.111167	0.032975	0.060398	0.543311	-0.027718	0.945968
3	(univerzita)	(home)	0.111167	0.545966	0.032975	0.296629	0.543311	-0.027718	0.645512
4	(home)	(verejnost)	0.545966	0.125781	0.031476	0.057653	0.458359	-0.037196	0.927704

We obtained 28 rules and selected the 5 with the highest support. As we can see, the support of the rules is not so high. Despite that, we can see that there is a pattern. Home page and faculty web category are the most accessed combination of web categories. Also, the visitors tend to access in their sessions more often the web category university with the combination of the home page.

On the other hand, if we consider another metric of the lift and sort the rules based on the lift value, we can see different rules.

Out[45]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
6	(images)	(verejnost)	0.076817	0.125781	0.023982	0.312195	2.482060	0.014320	1.271028
7	(verejnost)	(images)	0.125781	0.076817	0.023982	0.190665	2.482060	0.014320	1.140669
21	(studium)	(images)	0.072571	0.076817	0.011491	0.158348	2.061352	0.005917	1.096869
20	(images)	(studium)	0.076817	0.072571	0.011491	0.149593	2.061352	0.005917	1.090572
19	(fakulty-a-sucasti)	(prijimacie-konanie)	0.102798	0.076443	0.012990	0.126367	1.653094	0.005132	1.057146

The higher the lift value, the more likely are the two categories together in the session. We can see that the images category is often with the public category. This can mean that the public category contains information about some activities or events that lead to pages with image galleries.

The association analysis offers fast results and a nice insight into the examined data. For further analysis, we could have compared two types of users based on the IP addresses and look for differences in their behaviour. The nice aspect of association analysis is that it is easy to run and relatively easy to interpret. If we did not have access to MLxtend and this association analysis, it would be exceedingly difficult to find these patterns using basic Excel analysis. With python and MLxtend, the analysis process is relatively straightforward, and since you are in python, you have access to all the additional visualization techniques and data analysis tools in the python ecosystem.

Knowledge Discovery using Sequence Pattern Mining

Analyse the behaviour of web users using another data mining method – sequence pattern mining. The source web server contains information that is stored in a log file that has to be preprocessed based on the previous assignment (data cleaning, user/session identification, data transformation). Use the sequential analysis (sequence pattern mining) to obtain rules of behaviour of web users. The difference of this data analysis method is that sequential analysis looks for time-ordered association patterns. The accessed web categories are ordered based on time sequence resulting in a sequence rule. This way, you can discover the most used paths of the web user on the webserver. The obtained results can help the web administrator to reorganize the structure of the web portal to improve the user experience.

This assignment should be solved using the Python programming language using the Jupyter Notebook. You will require to import the pandas data analysis library for preprocessing and the scikit-learn library for sequence pattern mining.

Knowledge Discovery using Cluster Analysis

Analyse the behaviour of web users using another data mining method – cluster analysis. Use the log file from the previous assignments and prepare it in a similar way (data cleaning, user/session identification, data transformation). The aim is to identify web categories that are often accessed in the sessions by web visitors. First, use the Elbow method or the Silhouette score to identify the number of clusters needed for the algorithm (you can choose one of the mentioned methods to obtain the results). Next, use the KMeans algorithm to create the clusters and divide the web categories into clusters. The last step is the clusters visualization using the dendrogram.

This assignment should be solved using the Python programming language using the Jupyter Notebook. You will require to import the pandas data analysis library for preprocessing and the scikit-learn library for cluster analysis.

Source Codes

WEB DEVELOPMENT

Web Game

- Web_development/WebGame.zip

Authentication And Menu Based on User's Roles for a Web App In PHP

- Web_development/Web_PHP_login_roles-solution.zip

Create, Retrieve, Update, Delete, and List Users for a Web App in PHP

- Web_development/PHP_Web_App_Users_CRUD-solution.zip
- Web_development/PHP+JavaScript_Web_App_Users_CRUD-solution.zip

Chat in VueJS

- Web_development/fitped-chat-app-8c685fbe94a2.zip

MOBILE APPLICATIONS

To-Do Application for Android in Java

- Mobile_applications/fitped-todo-java-master.zip

Google Map Application Template

- Mobile_applications/fitped-map-master.zip

To-Do Application with Maps

- Mobile_applications/fitped-todo-kotlin-master.zip

Map Box Application Template

- Mobile_applications/fitped-MapBox-java.zip

Compass

- Mobile_applications/sensor_list.zip
- Mobile_applications/Accelerator.zip
- Mobile_applications/Compass.zip

DATA PROCESSING

Customer-Products Data Model Development

- N/A

Knowledge Discovery from Log File

- /Datamining/Fitped-data-mining-main.zip

Bibliography

- [1] http://androidexample.com/LISTVIEW/index.php?view=article_discription&aid=65&aaid=90
- [2] <http://blog.bawa.com/2013/11/create-your-own-simple-pedometer.html>
- [3] <http://coderzpassion.com/android-working-camera2-api/>
- [4] <http://lucasar.org/2012/04/05/performance-tips-for-androids-listview/>
- [5] <http://luugiathuy.com/2011/02/android-java-bluetooth/>
- [6] <http://manojprasaddevelopers.blogspot.sk/2012/02/bluetooth-data-transfer-example.html>
- [7] <http://slideplayer.com/slide/11642897/>
- [8] <http://startandroid.ru/en/lessons/complete-list/241-lesson-28-extras-passing-data-using-intent.html>
- [9] <http://toomanytutorials.blogspot.sk/2015/03/scanning-for-bluetooth-devices-in.html>
- [10] <http://tutlane.com/tutorial/android/android-bluetooth-with-examples>
- [11] <http://www.androidhive.info/2011/10/android-listview-tutorial/>
- [12] <http://www.codepool.biz/take-a-photo-from-android-camera-and-upload-it-to-a-remote-php-server.html>
- [13] <http://www.devexchanges.info/2016/10/simple-bluetooth-chat-application-in.html>
- [14] http://www.dis.uniroma1.it/~beraldi/PSD_014/slides/6_Services_OK.pdf
- [15] <http://www.i-programmer.info/programming/android/7849-android-adventures-listview-and-adapters.html>
- [16] <http://www.javacodegeeks.com/2013/09/android-listview-with-adapter-example.html>
- [17] <http://www.londatiga.net/it/programming/android/how-to-programmatically-pair-or-unpair-android-bluetooth-device/>
- [18] <http://www.londatiga.net/it/programming/android/how-to-programmatically-scan-or-discover-android-bluetooth-device/>
- [19] <http://www.outware.com.au/insights/which-direction-am-i-facing-using-the-sensors-on-your-android-phone-to-record-where-you-are-facing/>
- [20] <http://www.posterus.sk/?p=10653>
- [21] <http://www.raywenderlich.com/78574/android-tutorial-for-beginners-part-1>
- [22] <http://www.softengine.sk/android/doc/clanok19.doc>
- [23] <http://www.theappguruz.com/blog/to-allow-two-way-text-chat-over-bluetooth-in-android>
- [24] <http://www.vogella.com/articles/AndroidListView/article.html#androidlists>
- [25] <http://www.vogella.com/tutorials/AndroidBroadcastReceiver/article.html>
- [26] <http://www.vogella.com/tutorials/AndroidCamera/article.html>
- [27] <http://www.zdrojak.cz/clanky/vyvijime-pro-android-prvni-krucky/>
- [28] <http://www.zdrojak.cz/clanky/vyvijime-pro-android-zaciname/>
- [29] <http://zattackcoder.com/android-camera-2-api-example-without-preview/>
- [30] <https://android.googlesource.com/platform/development/+25b6aed7b2e01ce7bdc0dfa1a79eaf009ad178fe/samples/BluetoothChat/src/com/example/android/BluetoothChat>
- [31] <https://androidkennel.org/android-camera-access-tutorial/>
- [32] <https://angular.io/>
- [33] <https://athemes.com/collections/vue-ui-component-libraries/> or <https://www.codeinwp.com/blog/vue-ui-component-libraries/>
- [34] <https://blog.logrocket.com/understanding-computed-properties-in-vue-js/>
- [35] <https://cli.vuejs.org/>
- [36] <https://cli.vuejs.org/guide/css.html>
- [37] <https://code.tutsplus.com/tutorials/android-quick-look-bluetoothadapter--mobile-7813>
- [38] <https://code.visualstudio.com/>

- [39] <https://developer.android.com/>
- [40] <https://developer.android.com/guide/components/broadcast-exceptions.html>
- [41] <https://developer.android.com/guide/platform/index.html>
- [42] <https://developer.android.com/guide/topics/connectivity/bluetooth.html>
- [43] <https://developer.android.com/guide/topics/manifest/intent-filter-element.html>
- [44] <https://developer.android.com/guide/topics/manifest/manifest-intro.html#ifs>
- [45] <https://developer.android.com/guide/topics/permissions/overview.html#normal-dangerous>
- [46] <https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html>
- [47] <https://developer.android.com/reference/android/hardware/Camera.html>
- [48] <https://developer.android.com/reference/android/hardware/camera2/package-summary.html>
- [49] <https://developer.android.com/reference/android/hardware/Sensor.html>
- [50] <https://developer.android.com/training/basics/firstapp/creating-project.html>
- [51] <https://developer.android.com/training/permissions/requesting>
- [52] <https://developer.android.com/training/permissions/requesting.html#normal-dangerous>
- [53] <https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous>
- [54] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await
- [55] <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducin>
- [56] <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing>
- [57] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- [58] <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>
- [59] <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>
- [60] <https://docs.mapbox.com/android/maps/>
- [61] <https://docs.mapbox.com/android/maps/examples/symbol-layer-info-window/>
- [62] <https://docs.mapbox.com/android/plugins/overview/annotation/>
- [63] <https://eslint.org/>
- [64] <https://github.com/axios/axios>
- [65] <https://github.com/botyourbusiness/android-camera2-secret-picture-taker>
- [66] <https://github.com/kevalpatel2106/android-hidden-camera>
- [67] <https://github.com/request/request/issues/3143>
- [68] https://github.com/stereoboy/android_samples/tree/master/SimpleCamera
- [69] <https://nodejs.org/en/>
- [70] <https://reactjs.org/>
- [71] <https://router.vuejs.org/>
- [72] <https://router.vuejs.org/guide/>
- [73] https://source.android.com/devices/sensors/sensor-types#rotation_vector
- [74] <https://stackoverflow.com/questions/21752637/how-to-capture-an-image-in-background-without-using-the-camera-application>
- [75] <https://stackoverflow.com/questions/36936914/list-of-android-permissions-normal-permissions-and-dangerous-permissions-in-api>
- [76] <https://swagger.io/>
- [77] <https://vuejs.org/>
- [78] <https://vuejs.org/v2/guide/components-props.html>
- [79] <https://vuejs.org/v2/guide/components-registration.html>
- [80] <https://vuejs.org/v2/guide/computed.html>
- [81] <https://vuejs.org/v2/guide/forms.html>
- [82] <https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks>
- [83] <https://vuejs.org/v2/guide/list.html>

- [84] <https://vuejs.org/v2/guide/single-file-components.html>
- [85] <https://vuetifyjs.com/en/>
- [86] <https://vuex.vuejs.org/>
- [87] <https://webpack.js.org/>
- [88] <https://www.androidauthority.com/adding-bluetooth-to-your-app-742538/>
- [89] <https://www.codeproject.com/Articles/814814/Android-Connectivity#bluetooth>
- [90] <https://www.codespeedy.com/simple-compass-code-with-android-studio/>
- [91] <https://www.grokkingandroid.com/android-tutorial-broadcastreceiver/>
- [92] <https://www.javatpoint.com/android-bluetooth-list-paired-devices-example>
- [93] <https://www.javatpoint.com/android-bluetooth-tutorial>
- [94] <https://www.npmjs.com/>
- [95] <https://www.php.net>
- [96] <https://www.php.net/manual/en/book.pdo.php>
- [97] <https://www.php.net/manual/en/faq.passwords.php>
- [98] <https://www.sitepoint.com/get-started-vuetify/>
- [99] <https://www.slideshare.net/boochlin/camera2-how-tocreate>
- [100] <https://www.sqlite.org/index.html>
- [101] https://www.tutorialspoint.com/android/android_broadcast_receivers.htm
- [102] <https://www.udemy.com/course/the-complete-android-oreo-developer-course/>
- [103] https://www.w3schools.com/jsref/dom_obj_event.asp
- [104] <https://www.youtube.com/watch?v=ls1cjNcgdFI>
- [105] <https://www.youtube.com/watch?v=nOQxq2YpEjQ>
- [106] <https://www.youtube.com/watch?v=pDz8y5B8GsE>
- [107] <https://www.zdrojak.cz/clanky/android-studio-nove-vyvojove-prostredi/>
- [108] insidebigdata.com The Exponential Growth of Data, 17. 2. 2017 online
<https://insidebigdata.com/2017/02/16/the-exponential-growth-of-data>
- [109] Marr Bernard: Big Data: 20 Mind-Boggling Facts Everyone Must Read,
<https://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read/#6e7dfc7317b1> Sep 30, 2015



PRISCILLA



priscilla.fitped.eu