htblakaindorf
Leistung mit Menschlichkeit

informatik

Happiness-Metric

Diploma Thesis - 2019/2020

Lukas Holzmann, René Wagner

Supervisor:

Heinz Schiffermüller

Project partner:

Mattias Rauter

Statutory declaration

I declare that I have written the diploma thesis independently and without outside help. That I have not used any sources other than those specified and that I have stated every used asset in the bibliography.

Premstätten, 20.12.2019

Lukas Holzmann

(signature)

René Wagner

(signature)

Acknowledgement

First, we want to thank our diploma thesis supervisor Dipl.-Ing. Dr. Heinz Schiffermüller, who stood by our side whenever needed. He always motivated us and provided some ideas to make our project better and better.

Furthermore, we owe thanks to Mattias Rauter, who made our diploma thesis at Denovo possible and always supported us with great commitment. We are very pleased and grateful that we were able to gain experience at Denovo.

Thanks also goes to our parents, who always supported us and listened to us, no matter what problems we faced.

Prologue

Denovo and all its employees thank Lukas Holzmann and René Wagner for their great work on the Happiness Metric project. It was a pleasure for us to work with them and to see how they improved in SCRUM and general app development over time.

We gave the students a lot of freedom in designing the application and we are proud of what has been created.

We will integrate the Happiness Metric application into our everyday work life, and we wish them both the best of luck for their final exam

Abstract

The purpose of this diploma thesis was to create a system that, can track the satisfaction of SCRUM team members for our corporate partner, Denovo GmbH. The application was supposed to run on Android as well as iOS devices, to provide simplicity of use and accessibility. A flexible backend was an indispensable criterion for this project as was developing a system that, can handle users anonymously. Furthermore, we had to implement many features like sending notifications to the clients and uploading profile pictures to a storage location hosted by Amazon.

The basic design of the application was provided by Denovo GmbH and therefore we only had to create a few layouts on our own. The first requirement we defined in our sprint planning meeting was the user login system. It was decided that only employees of Denovo Gmbh should be able, to log into the application. Furthermore, the application should display a user's sprints in a diagram, and it should also be possible to view past sprints. Other than those requirements, we were given a decent amount of freedom in designing our own happiness tracking tool.

We used many new technologies to make this application possible. For example, we made use of JSON Web Tokens to ensure a safe connection between the application and the API, and we encrypted the stored passwords with a NodeJS library, to ensure maximal safety. Furthermore, we used AWS Buckets to store uploaded profile pictures and we utilized a service called Mailgun, which made it possible to send e-mails to registered user e-mail addresses.

To make the best use of the available time, we decided to use an agile approach to complete our project, which means that we had a structured workflow using sprints and SCRUM roles.

Kurzfassung

Ziel dieser Diplomarbeit war es, ein System zu schaffen, mit dem die Zufriedenheit der SCRUM-Teammitglieder für unseren Unternehmenspartner Denovo GmbH verfolgt werden kann. Die Anwendung sollte sowohl auf Android- als auch auf iOS-Geräten ausgeführt werden können. Ein flexibles Backend war ein unverzichtbares Kriterium für dieses Projekt, ebenso wie die Entwicklung eines Systems, das Benutzer anonym behandeln kann. Darüber hinaus mussten wir viele Funktionen implementieren, z. B. das Senden von App Benachrichtigungen und das Hochladen von Profilbildern an einen von Amazon gehosteten Speicherort.

Das grundlegende Design der Anwendung wurde von der Denovo GmbH bereitgestellt, weshalb wir nur wenige Layouts selbst erstellen mussten. Die erste Anforderung, die wir in unserem Sprint-Planungsmeeting definiert haben, war das Benutzeranmeldesystem. Es wurde beschlossen, dass sich nur Mitarbeiter der Denovo Gmbh in der Anwendung anmelden dürfen. Darüber hinaus sollte die Anwendung die Sprints eines Benutzers in einem Diagramm anzeigen und es sollte auch möglich sein, vergangene Sprints anzuzeigen. Abgesehen von diesen Anforderungen wurde uns ein angemessenes Maß an Freiheit bei der Entwicklung unseres eigenen Tools eingeräumt.

Wir haben viele neue Technologien verwendet, um diese Anwendung zu ermöglichen. Beispielsweise haben wir JSON-Web-Tokens verwendet, um eine sichere Verbindung zwischen der Anwendung und der API zu gewährleisten, und die gespeicherten Kennwörter mit einer NodeJS-Bibliothek verschlüsselt, um maximale Sicherheit zu gewährleisten. Darüber hinaus haben wir AWS Buckets zum Speichern hochgeladener Profilbilder benutzt und einen Dienst namens Mailgun verwendet, mit dem E-Mails an registrierte Benutzer-E-Mail-Adressen gesendet werden konnten.

Um die verfügbare Zeit optimal zu nutzen, haben wir uns für einen agilen Ansatz entschieden, um unser Projekt abzuschließen. Dies bedeutet, dass wir einen strukturierten Workflow mit Sprints und SCRUM-Rollen hatten.

Table of Content

1.1 Our Team

Figure 1 - Lukas Holzmann

Figure 2 - René Wagner

The project team consisted of two people. Lukas Holzmann, who was responsible for the diploma thesis and René Wagner. The division of labor was as follows: Lukas was responsible for the Android version and the backend of the application, as he already had experience with Android programming, while René was responsible for the iOS-version of the app. After the practical work, the team split the chapters of the written part of the diploma thesis, so both of the team members had about the same time expense.

1.2 Technologies used

Kotlin is a cross-platform programming language primarily used for mobile development on Android. This programming language is designed to have a high compile speed and as little code as possible.

Figure 1 - Kotlin icon

"Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. […] The design goals of XML emphasize simplicity, generality, and usability across the Internet." (XML Wikipedia, 2020)

Figure 2 - XML icon

"JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data object consisting of attribute-value pairs and array data types." (JSON Wikipedia, 2019)

Figure 3 - JSON icon

Node.js is a JavaScript runtime environment, which allows developers to write dynamic web pages and application programming interfaces (API).

Figure 4 - NodeJS icon

Figure 5 - Mailgun icon

Mailgun is a transactional Email API Service, which allows developers to send, receive and track emails effortlessly. The service is quickly scalable and has an easy Simple Mail Transport Protocol (SMTP) integration.

Figure 6 - AWS icon

Amazon Simple Storage Service Bucket is an object storage service offered by amazon. It has great security, performance, data availability and scalability. It is used to save and backup data.

Figure 7 - MongoDB icon

"MongoDB is a universal, documented, distributed database for modern application development and the cloud." (MongoDB, 2020)

Figure 8 - Firebase icon

Firebase Cloud Messaging (FCM) is a cloud messaging service provided by google. It is primarily used to send notifications to client applications.

1.3 Agile Software Development (SCRUM)

The project was processed with SCRUM, which is an agile process framework for managing complex knowledge work.

The project team choose SCRUM for this project, because it provides high flexibility, effectiveness and transparency.

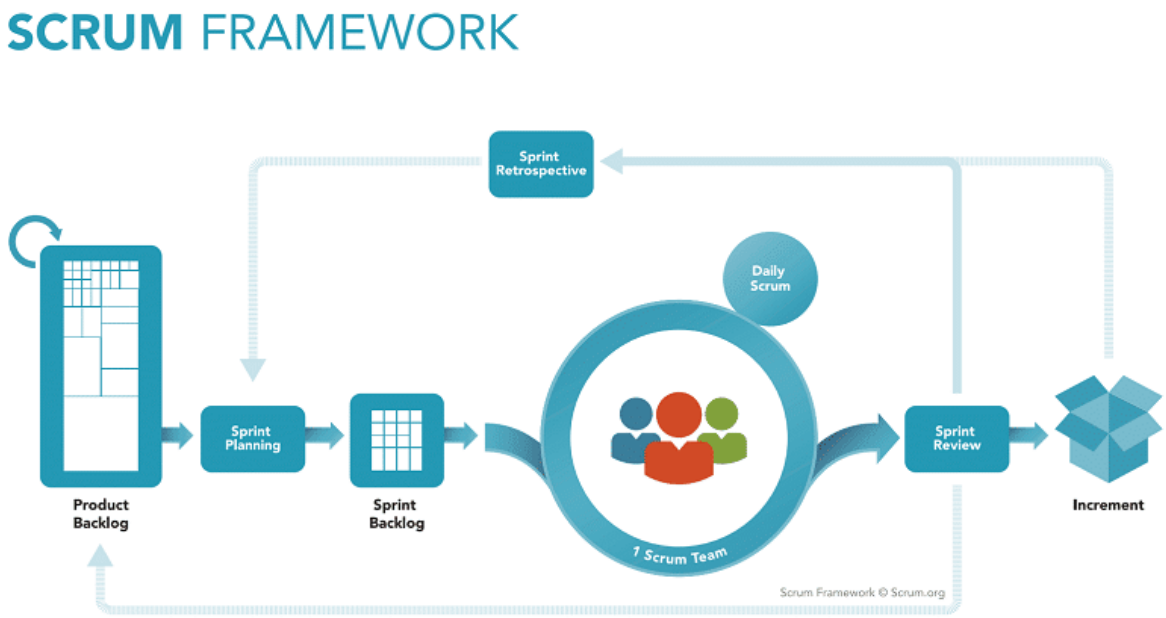


Figure 9 - SCRUM Framework

The Product Backlog is a sorted list of requirements defined by the customer, which is managed by the Product Owner. The Product Backlog is divided into Sprints during the Sprint Planning meeting. A Sprint is a repeatable workflow that achieves with a result at the end (Product-Increment). The goal of every sprint is to add more functionalities to the product. The Sprint Backlog defines the requirements for each sprint, and it is administrated by the Product Owner. The SCRUM Team works each individual Sprint of the project. The SCRUM Team meets every day (daily), to discuss which user stories have already been completed and which have not. A Sprint Review is a presentation held by the SCRUM Team at the end of every Sprint. Its goal is to inform the Product Owner and the Stakeholders about the work that has been done. At the beginning of a new Sprint a Retrospective is held. Its aim is to review the past Sprint and develop a better strategy to achieve the maximum possible performance.

1.3.1 How did we utilize SCRUM in our project?

The following roles were defined within the team:

- Product Owner: Mattias Rauter (Denovo GmbH)
- SCRUM Master: Lukas Holzmann
- SCRUM Team: Rene Wagner & Lukas Holzmann

The Product Backlog and Sprint Backlog were stored in Trello, which is a web-based project management software.

Customer requirements were created, which were later weighted in story points during the Sprint planning meeting.

Figure 10 - Trello Task

No fixed Sprint time span was defined, therefore Sprints ranged in length from 1 – 3 weeks. After every Sprint, the team had a Sprint review with the Product Owner, during which he reviewed what had been done and gave the team suggestions on how to enhance the current product. Because the SCRUM team was so small and everyone worked on a separate part of the project, the team chose not to do daily meetings. Instead, at the end of a Sprint the team held a retrospective, during which the outcome of the last Sprint and ways to improve were discussed

1.4 Program overview

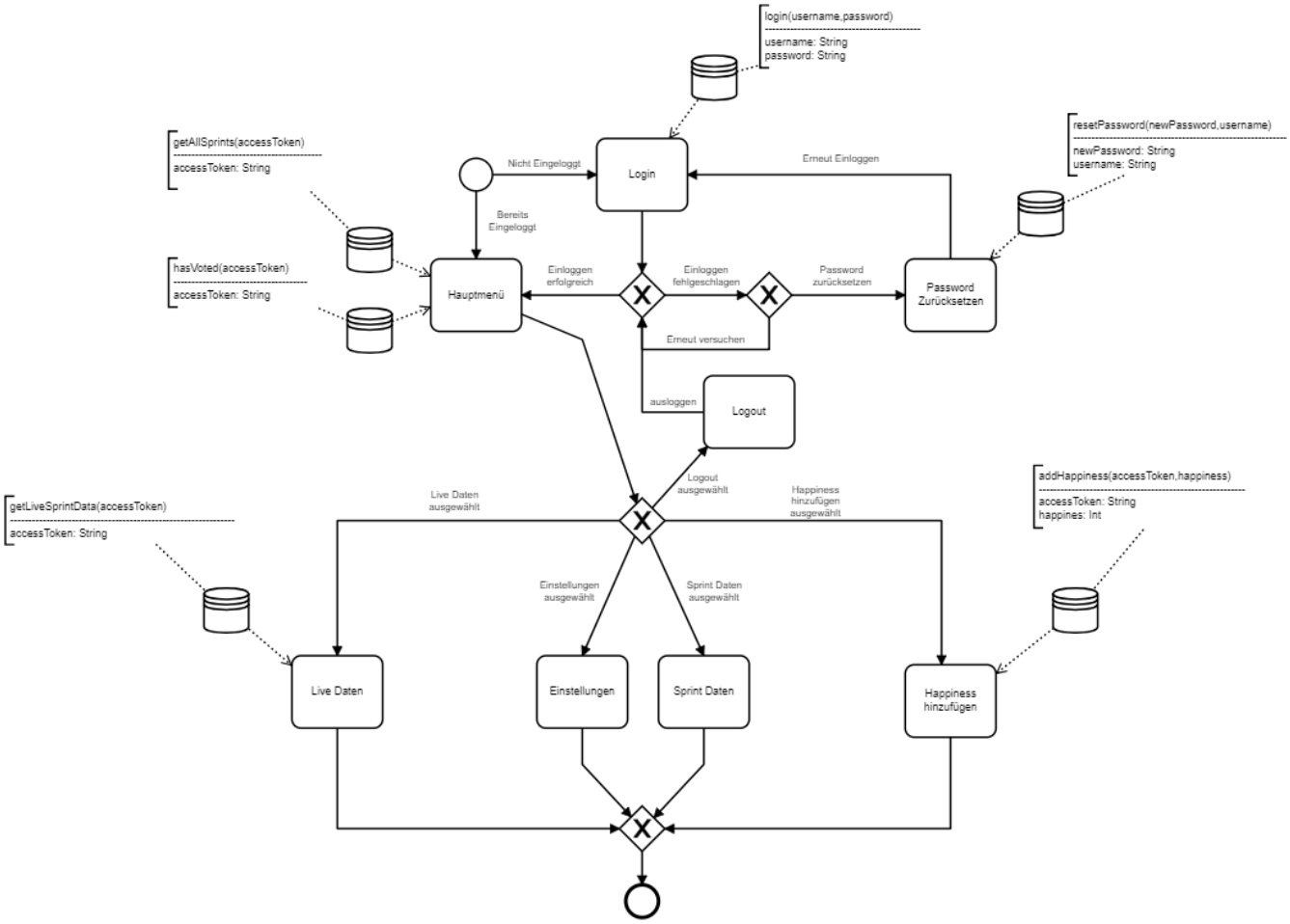


Figure 11 Program overview

Register

First, the idea was that only employees of the company Denovo should have access to this application. So, every employee is provided with a username and a password and cannot register on his own. This has the advantage of making it harder for someone, who does not work at Denovo, to log in.

Login

If the user has received his access data, he can probably log in. It is recommended that the user changes his password immediately. Also, if the user is already logged in, the first screen that appears is the Live Data screen, so the user does not have to login every time he starts the app. There is also the option to reset the password by email, if the user has forgotten his login data. To do this, the user only has to type in his email address.

Application

After the user is logged in, he can see the Live-Happiness level, which is a value calculated on the basis of all votes. The user also receives an access token, which he cannot see. The access token is essentially for all further activities the user wants to do. The user can change the view of how the data should be shown. If a user prefers a line chart, he can change to the chart view. In this view, the user will not only see the current happiness value, but also all past happiness values. From these two views the user can also go to the Settings screen or track their Happiness.

Track Happiness

If a user has not already voted about their happiness in the current Sprint, he is able to vote. Also, users can add a feedback, so the company knows what they have liked or not. The value a user has submitted will be calculated with all the other votes and added to the Live Happiness value results.

Settings

In the Settings there is an option to change the password or log out of the application. Also, a user can upload or change his profile picture or delete his account.

Change password

If the user wants to change his password, he has to type in his current password and the one he wants to set as the new password. Only if the current password is right, will the password be changed.

Happiness-Metric

Figure 12 - Android icon

Android & Backend development

1.5 Happiness-Metric Android Layout

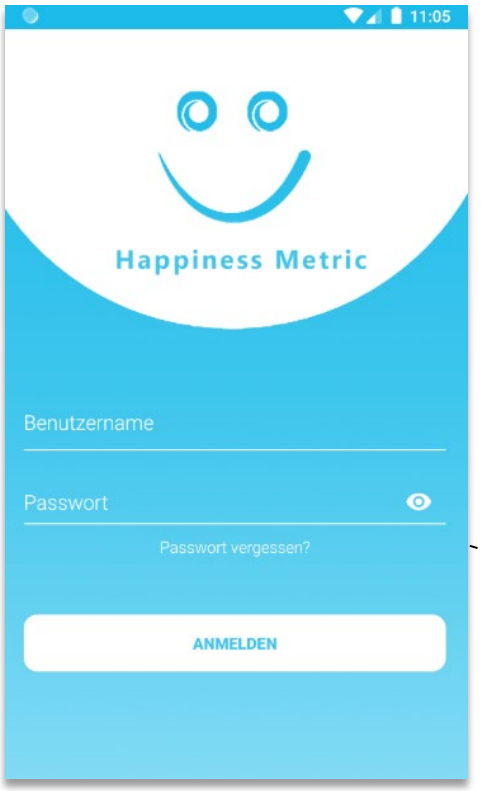


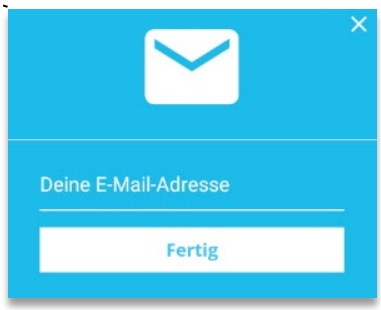
Figure 13 - Login screen frontend

1.5.1 Login Screen

This screen is called when the app is started for the first time. The user will be able to log in to the application using a username and a password.

1.5.2 Password Reset Dialog

If a user has forgotten his password, he can reset it using his registered email address.



1.5.3 Password Reset Email

An email will be sent to a user who has forgotten his password. The user can reset his password by clicking the "Zurücksetzen" button in the email.

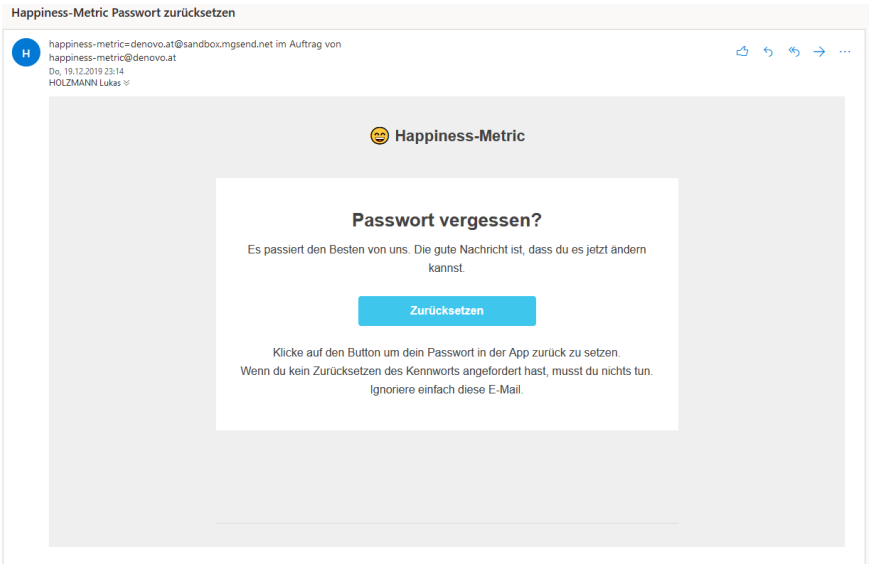


Figure 14 - Password reset e-mail

1.5.4 New Password Page

If a user clicks on the reset button in the email, he will be redirected to this page. A new password will be randomly generated for the user, and therefore he will be able to log in with the new user credentials.

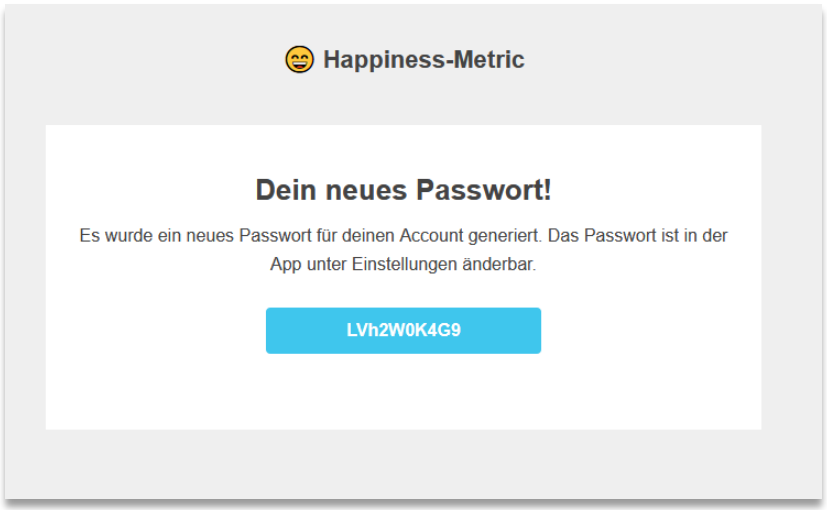


Figure 15 - New password page

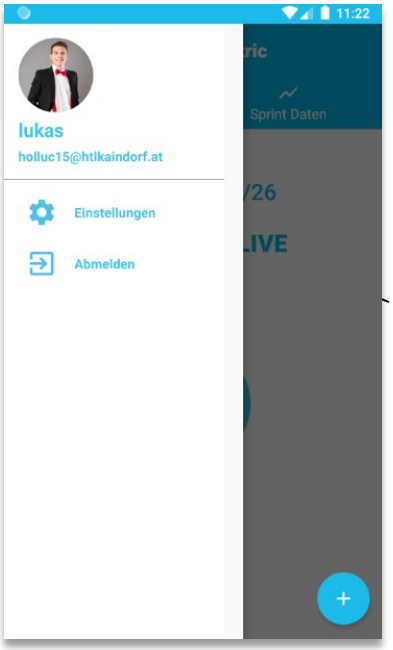
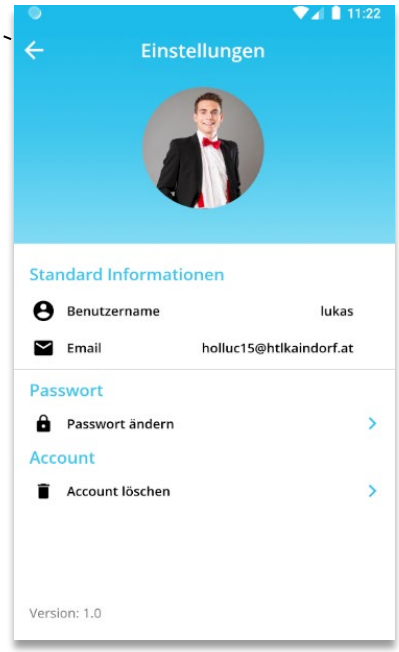


Figure 16 - Navigation drawer layout

1.5.5 Navigation Drawer

If the user swipes right, a navigation drawer opens. It will display information about the user and it gives the option to go to the Settings or to log out.



1.5.6 Main Menu Screen

Figure 17 - Main menu layout

1 Main Menu Screen

This screen displays the live happiness level, which is the average happiness of all SCRUM Team members.

2 Sprint Data Screen

The user can view past Sprints in the form of a diagram. It is possible to filter for certain sprints using the filter icon in the bottom right corner of the screen.

3 Filter Dialog

It is possible to filter for certain Sprints or to show complete years using this dialog. The chart will be updated with the new data as soon as the OK button is pressed.

4 Happiness Dialog

The user can input his happiness in the form of a value [0 – 10] and optional feedback. Each vote is saved anonymously in the database.

1.5.7 Settings Screen

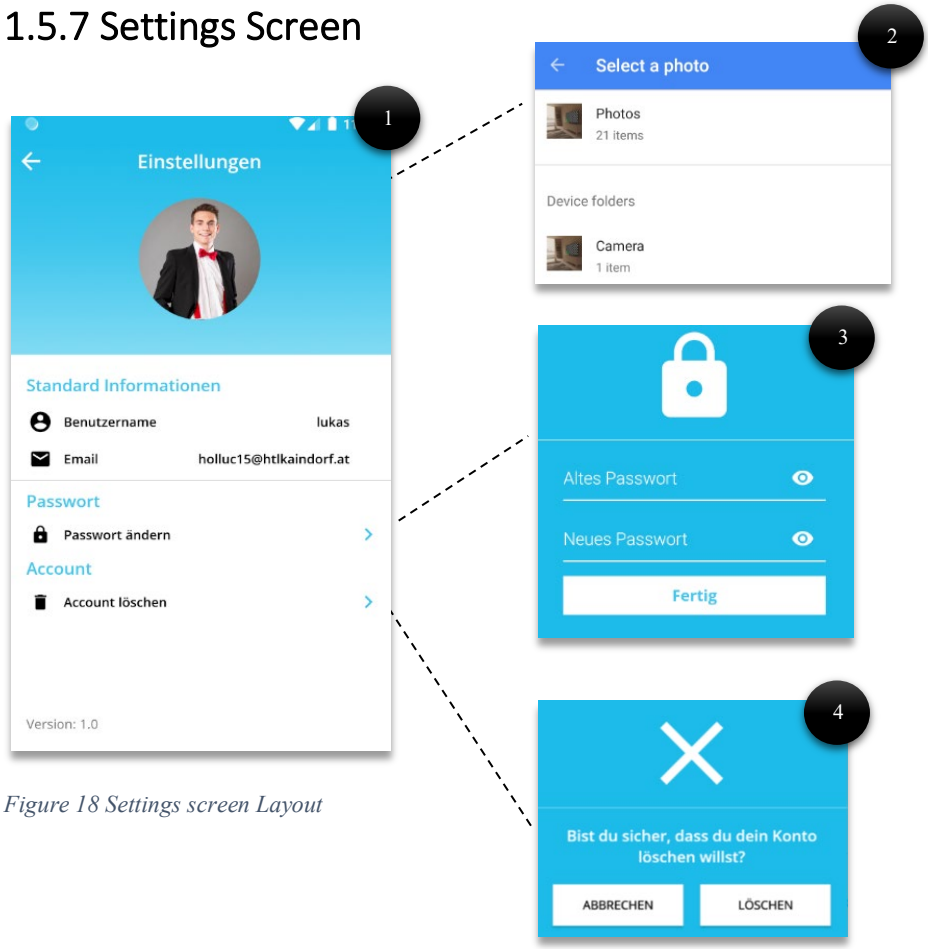


Figure 18 Settings screen Layout

1 Settings Screen

This screen displays basic information about the user, like username and email. It also shows the version number of the application at the bottom. If the user wishes to change his password, he can do this using the password reset option. He can also delete his account in the Settings menu, if needed.

2 Gallery Screen

This screen will be shown if the user clicks on his profile picture. It will show every picture in a sorted list and, if the user selects one it will be uploaded as a new profile picture.

3 Change Password Dialog

This dialog pops up if the user clicks on the change password button in the Settings menu. Here the user can change his password to a new one.

4 Delete Account Dialog

This dialog pops up if the user clicks on the delete account button in the Settings menu. If the user clicks on the delete button his account will be deleted and all data related to his account will be removed from the database.

Sprint 01 – First Frontend elements [13.08.2019 until 19.08.2019]

Tasks processed in this Sprint:

- 2.1.1 Project synchronization with GitLab
- 2.1.2 Frontend creation of the login screen
- 2.1.3 Frontend creation of the main menu

2.1.1 Project synchronization with GitLab

Figure 19 - GitLab

Firstly, a connection to Git, which is a Source Code Management System, was created. The team chose GitLab because of its usability and privacy. To synchronize a project with GitLab, a connection must be established. This is done by entering the following commands in a terminal.

Git needs the user's credentials, therefore these credentials were added to the git config using following commands:

```
git config --global user.name "Lukas Holzmann"
git config --global user.email "holzmann@denovo.at"
```

After setting the user credentials, a connection was established to the GitLab repository using following commands:

```
cd happiness-metric
git init
git remote add origin git@gitlab.com:denovo/happiness-metrics-android.git
```

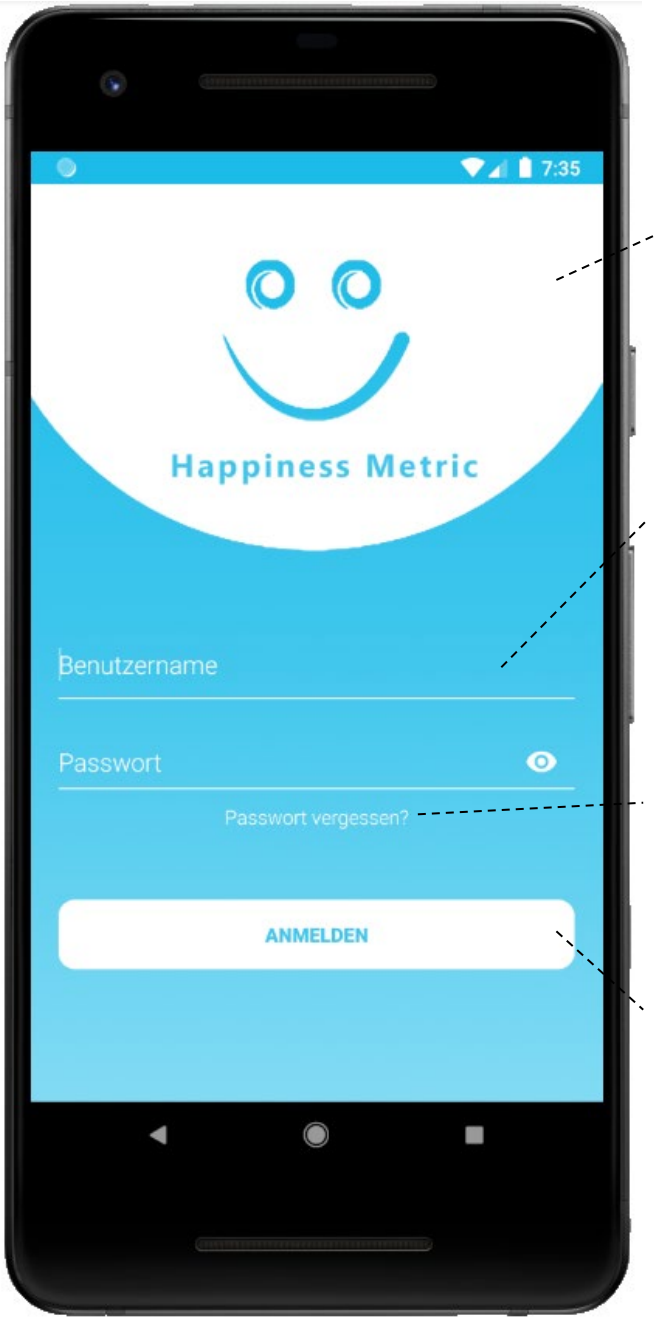
As a last step, the existing Android project had to be pushed to the GitLab repository. This was done by using following commands:

```
git add .
git commit -m "Initial commit"
git push -u origin master
```

The Android application was developed in Android Studio, which has an internal system to commit and then push to the repository. Navigating to VCS → Commit, commits the files to git, and VCS → Git → Push pushes the committed files to the repository.

2.1.2 Frontend creation of the login screen

In Android User Interface (UI) elements are created in Extensible Markup Language (XML). The purpose of this screen is to identify which users have access to the application and which do not. Access rights are established in the database and can be changed by an administrator. The login screen consists of following XML components:



An ImageView is used to display certain Images stored in the resources folder. It is used here to display the logo of the application.

The user can input text into this component, which the app can extract. This is how the app gets access to the user credentials of the user.

A TextView is a UI component, which can display text. It is also clickable, which means it can serve as a button. It is used to open the forgotten password dialog if clicked.

This UI component is used to detect user input. Which means if it gets clicked the app will check if the user credentials are stored in the database. If they are, the user will be forwarded to the main menu screen.

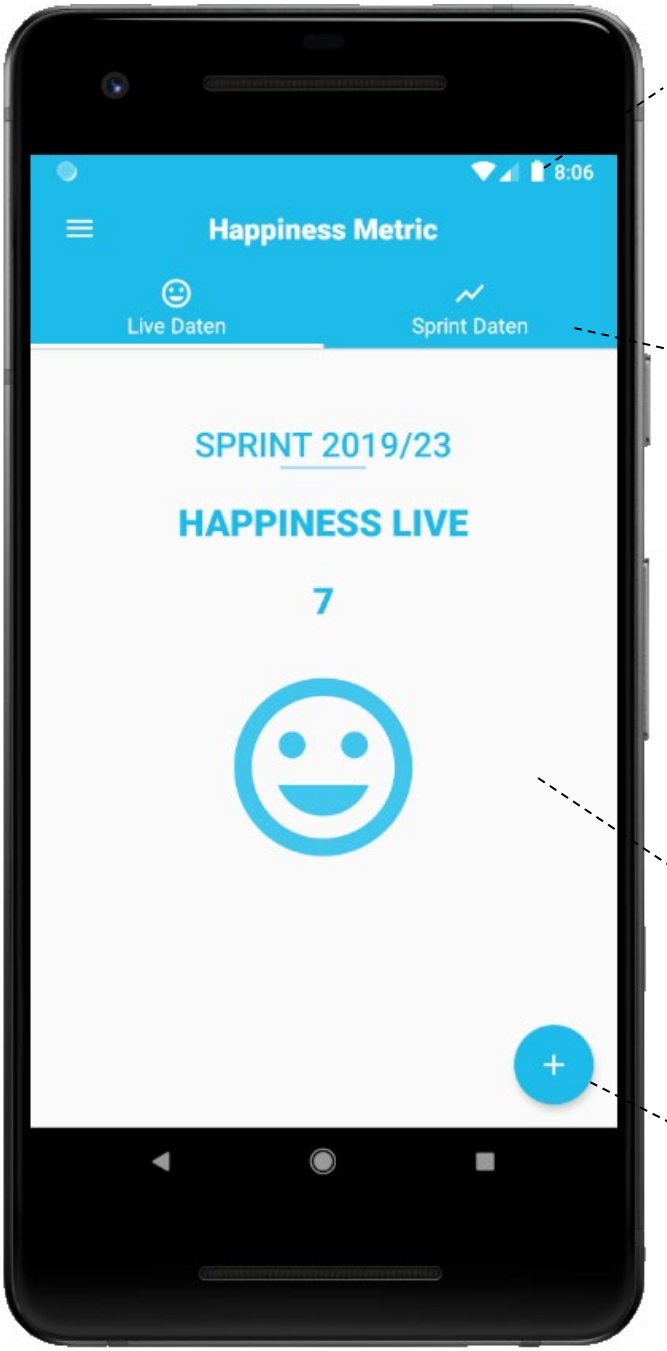
Figure 20 - Login screen

2.1.3 Frontend creation of the main menu

After creating the login screen, development of the main menu began. This screen displays the current happiness of a certain Sprint, which is calculated by using the arithmetic mean formula.

$$\boldsymbol{Sprint\ Happiness} = \frac{\boldsymbol{sum\ of\ each\ entered\ happiness}}{\boldsymbol{number\ of\ happiness\ entered}}$$

The main menu screen consists of the following XML components:



A TabLayout is a container that contains UI components, which are shown in tabs on the screen. The menu icon on the left side is a <Button>, which is used to display the <DrawerLayout> side menu if clicked.

This component was written from scratch, because Android has no default way to create a <ViewPager>, which disables gesture detection and therefore the ability to swipe between fragments. It is used to switch between the live fragment and the Sprint data fragment.

A LinearLayout is used to order components in a horizontal or vertical way. It contains <TextView> components, which are used to show the current Sprint number and the current Sprint happiness.

A FloatingActionButton is a button that is used for a special type of promoted action. If the button gets clicked, the Happiness Dialog pops up, allowing the user to input his current happiness.

Figure 21 - Main menu

Sprint 02 – Further development of the frontend [19.08.2019 until 01.09.2019]

Tasks processed in this Sprint:

2.2.1 Frontend creation of the navigation drawer

2.2.2 Frontend creation of the Sprint data fragment

2.2.3 Frontend creation of the live happiness dialog

2.2.4 Set up of the Happiness-Metric API project

2.2.1 Frontend creation of the navigation menu

A navigation drawer is a UI component, that provides access to destinations in the application. They can be controlled by a navigation menu icon or by swiping in the right direction.

A CircleImageView is an Android dependency written by Henning Dodenhof. It creates a circular border around an image displayed in an image view. It is used to display the profile picture of the user.

A View is the basic building block for UI components in Android. It is used here to display a 0.75dp thick line, which divides the user credentials and the menu options.

A NavigationView is the component that provides NavigationDrawer functionalities, such as gesture detection and event processing. It is used to create the NavigationDrawer.

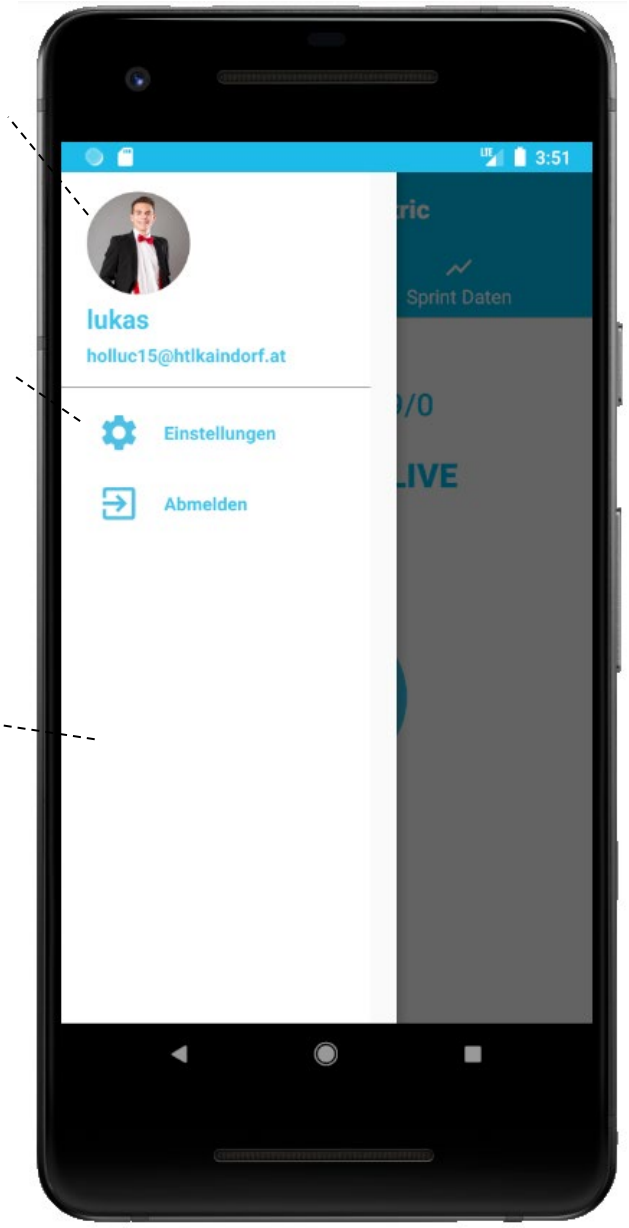
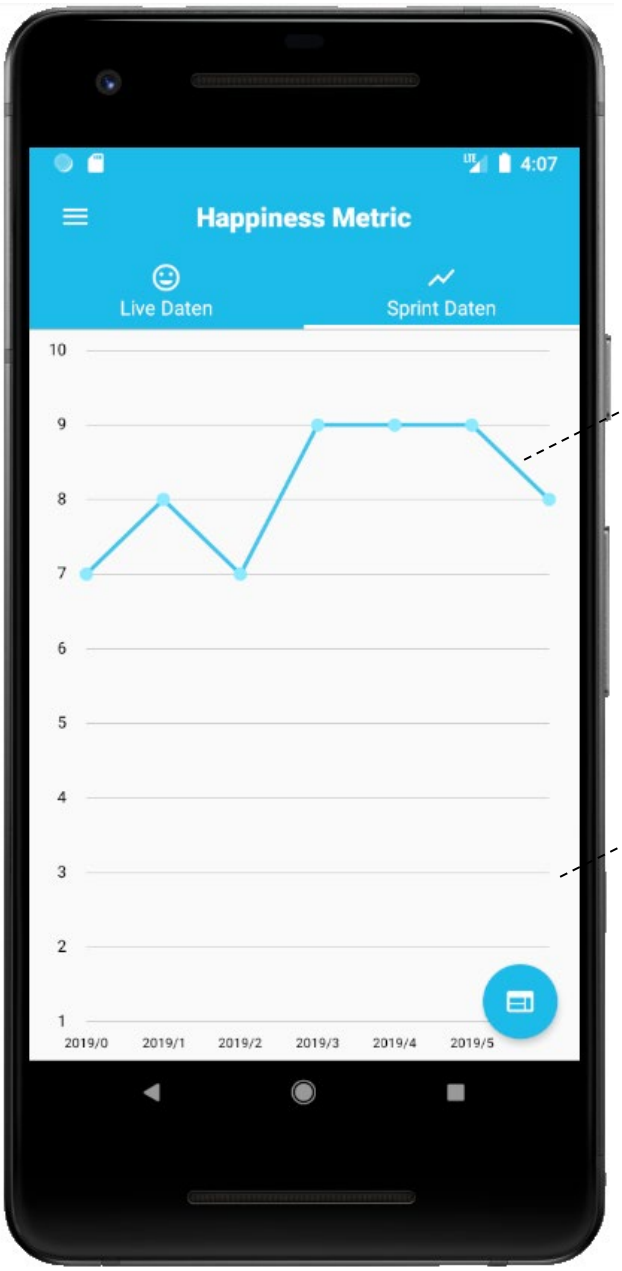


Figure 22 - Navigation drawer

2.2.2 Frontend creation of the sprint data fragment

The Sprint data fragment shows the average happiness of past Sprints. It displays these values in a linear diagram. The FloatingActionButton changes to a filter, that enables the user to filter by past Sprints or even by whole years.

The Sprint data screen consists of the following XML components:



A LineChart is a chart type from the MPAndroidChart library. It displays values in the form of a linear diagram. It is used to show the happiness values of Sprints.

This FloatingActionButton adapts its icon and functionality to the current fragment. It is used to display the filter popup if clicked.

Figure 23 - Sprint data frontend

2.2.3 Frontend creation of the Sprint data filter dialog

The purpose of this dialog is to give the user an option to filter by Sprint number and by year. This dialog uses a library called NumberPicker, developed by ShawnLin, that provides a simple and customizable NumberPicker. The dialog will automatically select the current Sprint and the current year.

The Sprint data filter dialog consists of the following XML components:

A NumberPicker is an Android library that allows the user to scroll through the number wheel and pick one. It was developed by ShawnLin.

A CardView is a layout that displays the content like a card. It is used here to add a slim shadow and a border to the dialog.

Figure 24 - Number pciker frontend

MPAndroidChart LineChart

> "This is a powerful Android chart view / graph view library, that supports line- bar- pie- radar- bubble- and candlestick charts as well as scaling, dragging and animations." (MPAndroidChart GitHub , 2020)

The LineChart must be formatted, therefore some code must be written in Kotlin. First, the chart needs to be initialized. This is done by getting the chart from the associated xml file.

```
//this is how a LineChart gets initialized
chart = view.findViewById(R.id.chart) as LineChart
```

Every Sprint name had to be displayed correctly, therefore some extra bottom margin was added and the pinch zoom was enabled using following commands:

```
// force pinch zoom along both axis
chart.setPinchZoom(true)
chart.extraBottomOffset = 10f
```

The legend and the description of the chart were disabled to ensure the cleanliness of the overall picture.

```
//disabling the legend of the chart
val legend = chart.legend
legend.isEnabled = false

//disabling the description of the chart
val description = chart.description
description.isEnabled = false
```

To add data to the chart a new LineDataset had to be created. This was done by transferring the list of each Sprint into a new LineDataSet object as shown below:

```
//creating a new sorted list object with the sprint entries
val sortedList = entries.sortedWith(compareBy { it.x })
dataSet = LineDataSet(sortedList, "Sprints")
```

This dataset was then formatted as required.

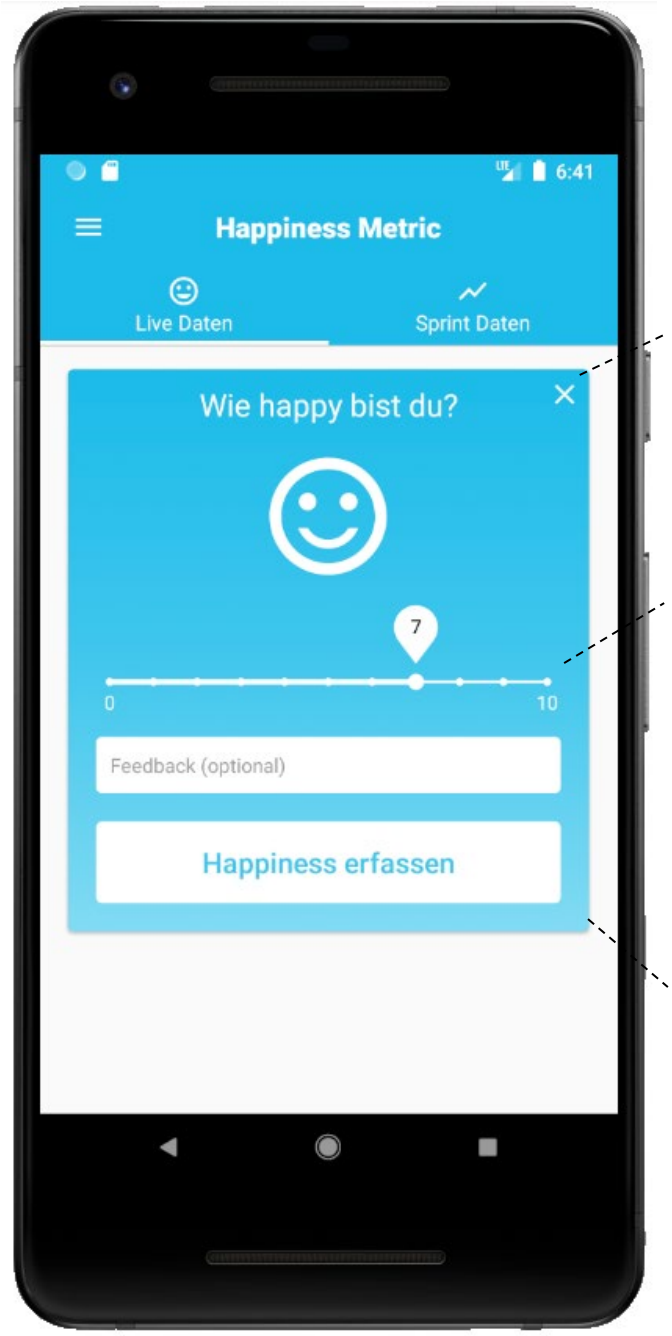
```
//setting the light blue color of the line
dataSet.color = R.color.colorPrimary
dataSet.highLightColor = ContextCompat.getColor(context!!, R.color.colorPrimary)
//changing the dot size
dataSet.setDrawCircleHole(false)
dataSet.setDrawValues(false)
dataSet.circleSize = 5f
//changing the line size and width
dataSet.lineWidth = 3f
dataSet.highlightLineWidth = 1f
```

Frontend creation of the live happiness dialog

This dialog's purpose is to collect the happiness value entered by a user and to send it to the database.

A user can input his happiness in the form of values. These range from 0 to 10, where 0 is the worst and 10 is the best. If a user desires, he can also provide feedback for the current Sprint.

The live happiness dialog consists of the following XML components:



An ImageButton is basically a button, that can display an image instead of the standard button design.

A BubbleSeekBar is an Android library that was developed by woxingxiao. This library helps users create custom seekbars in Android.
It is used to collect the current happiness of a user.

A CardView is a widget that is part of the v7 Support libraries. It is used to round off the button.

Figure 25 - Happiness tracking dialog

2.2.4 Set up of the Happiness-Metric API project

The Application Programming Interface (API) was developed in Visual Studio Code with the programming language NodeJS. First, the empty project was set up with Git using the following command:

```
git init
```

Then it was added to the API project using:

```
git add .
git commit -m "init"
```

As a last step, the API project was connected with a Git repository using the following commands:

```
git remote add origin remote <url>
git remote -v
git push -f origin master
```

Setting up a HTTP Server with NodeJS

Modules used to set up the API project:

- **Express**

  "Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications." (ExpressJS, 2017)

  Installing express:

  ```
  npm install express –save
  ```

- **Nodemon**

  "Nodemon is a utility that will monitor for any changes in the source and automatically restart the server." (Nodemon, 2020)

  Installing nodemon:

  ```
  npm install -g nodemon
  ```

- **Body-parser**

  "Body-parser is a module for NodeJS. It parses incoming request bodies in a middleware before moving them to handlers, available under the req.body property." (BodyParser NPM, 2014)

  Installing body-parser:

  ```
  npm install body-parser
  ```

First, a server.js file was created, in which I a server was created using the Hypertext Transfer Protocol (HTTP). To achieve this, following lines had to be written:

```
require('dotenv').config();

var http = require('http');
var app = require('./app');

//creating the server using the PORT, which is defined in the .env file
http.createServer(app).listen(process.env.PORT, () => {
  console.log('---------------------------------------');
  console.log('The Server is running on port ' + process.env.PORT);
  console.log('---------------------------------------');
});
```

Second, a .env file was created, in which the secret variables are stored. The .env file will not be published on GitLab and is therefore the perfect place to store secret variables, such as passwords and access keys. The port where the server will run was defined as shown below:

```
PORT=8080
```

To start the server using nodemon, the following was entered in the console.

```
nodemon start
```

The server will be started, and the following output will be displayed:

```
PS C:\Users\holluc15\Desktop\happiness-metric-backend> nodemon start
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node start server.js`
-----------------------------------------------
'The Server is running on port 8080
-----------------------------------------------
```

Sprint 03 – Finishing frontend development [01.09.2019 until 18.09.2019]

Tasks processed in this Sprint:

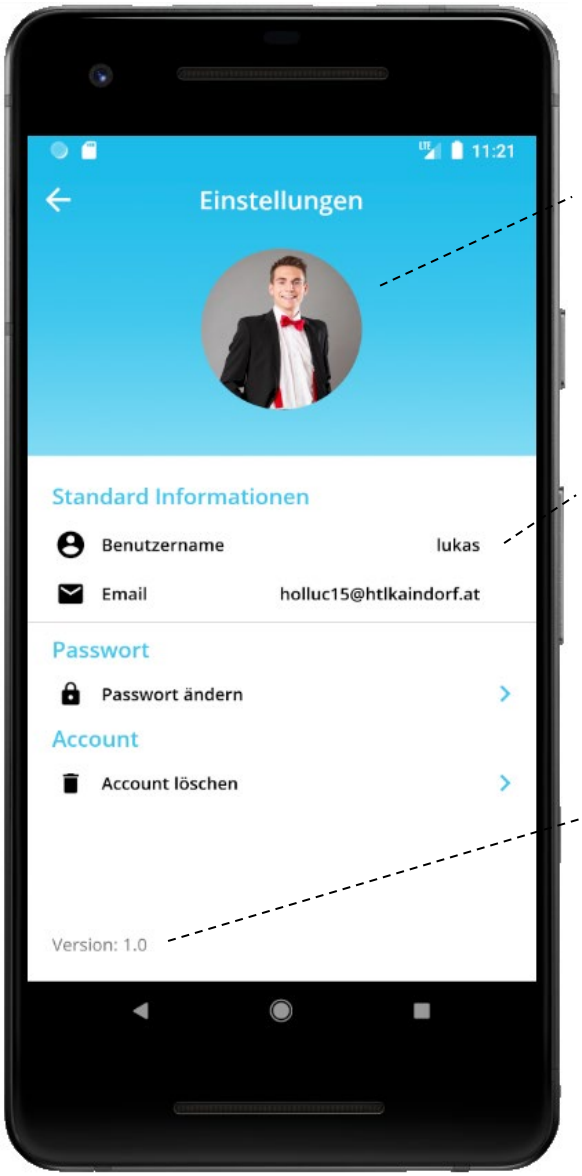
2.3.1 Frontend creation of the Settings Menu

2.3.2 Frontend creation of the account administration dialogs

2.3.3 Frontend creation of the error screen

2.3.1 Frontend creation of the Settings Menu

The Settings Menu is accessible by clicking the Settings button in the drawer menu. It shows the user his credentials and it allows the user to reset his password in the application. It also gives the user the option to delete his account and the option to upload a profile picture from his gallery. The Settings screen consists of following the XML components:



This CircleImageView displays the profile picture of the user. If a user does not have one, it will display a default instead. The user will be able to select a picture from his gallery, by clicking on it.

A RelativeLayout allows the programmer to align views inside the layout how he wants them. It can also work as a button, if a click listener is applied. It displays the password reset dialog if clicked.

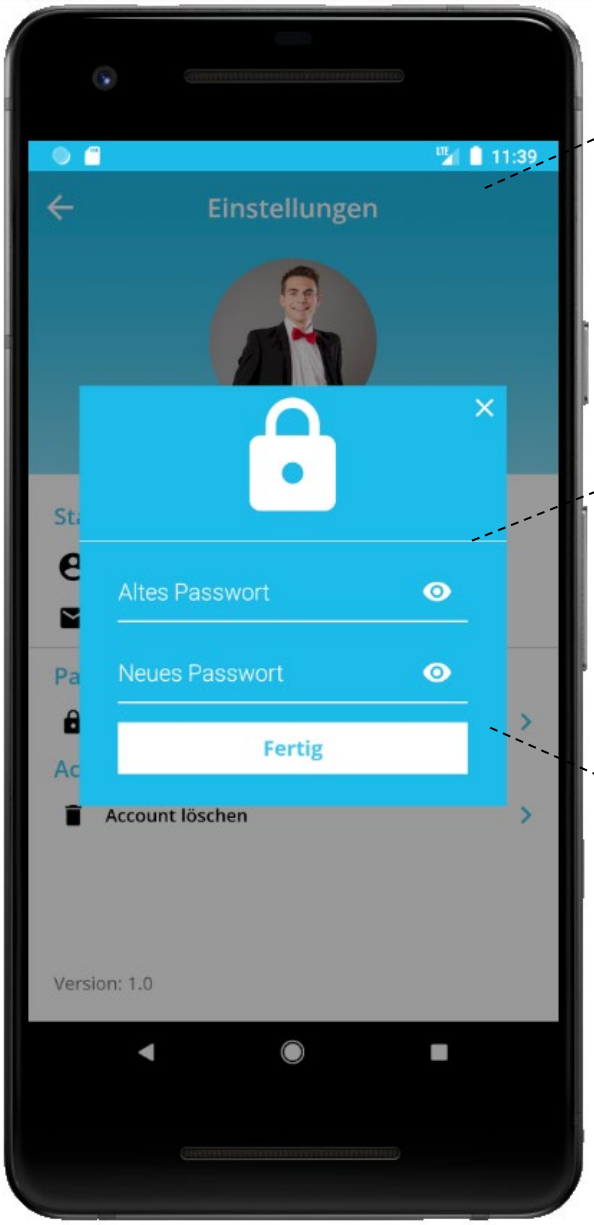
A TextView can show text. It is used to display the version number of the application, which is accessed by using the BuildConfig.VERSION_CODE variable.

Figure 26 - Settings screen

2.3.2 Frontend creation of the account administration dialogs

The password reset dialog's purpose is to give the user the option to reset his password. This is done by entering the old password in the first text field and the new password in the second text field. The user's new password will be hashed by a NodeJS module called bcrypt and if his old password is correct, will be set as a new password in the database.

The password reset dialog consists of the following XML components:



The background consists of a RelativeLayout, which changes to be slightly darker if the password reset dialog is active.

A TextInputLayout is used to give the user the option to show his entered password, to avoid spelling errors. It stores an <EditText> component, which handles the text input.

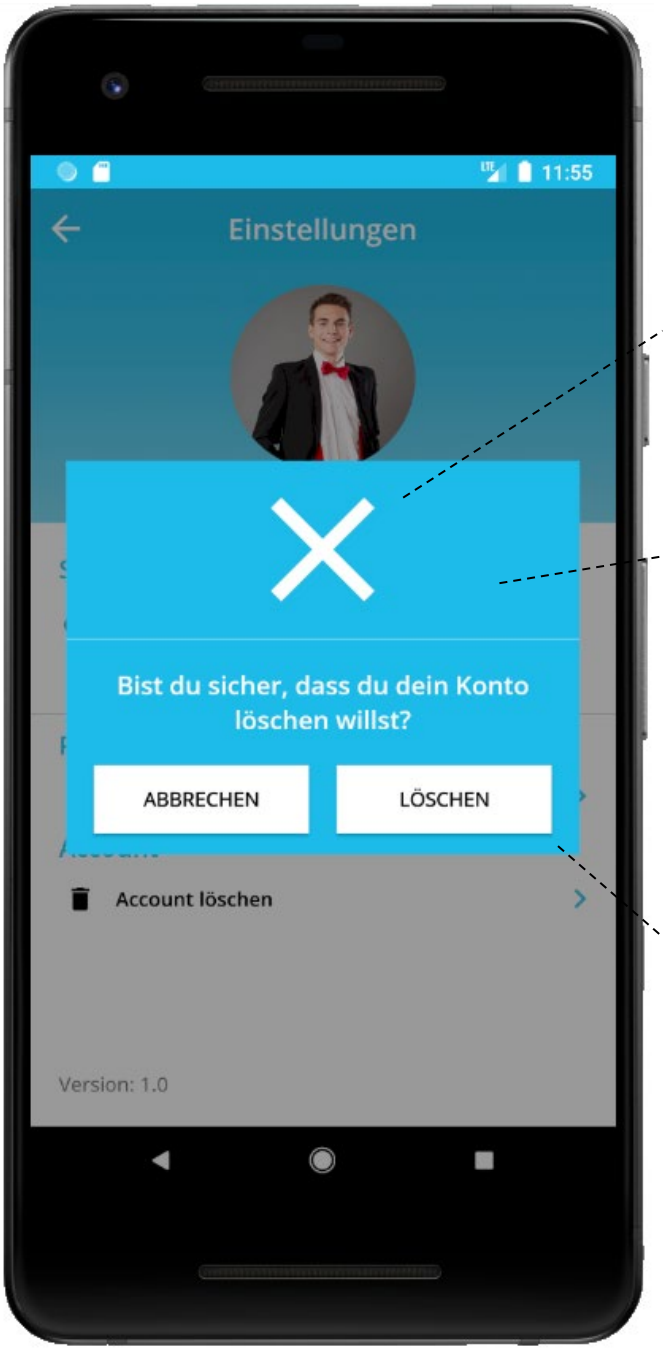
A FrameLayout is used around this <ImageButton>, to give it the ability to display a <ProgressBar>, instead of the text.

Figure 27 - Password reset dialog

Frontend creation of the account deletion dialog

This dialog is used to delete a user's account. If the user wishes, all data associated with the account will be deleted and the user will be sent back to the login screen.

The account deletion dialog consists of the following XML components:



This ImageView displays an icon from the Android library, which is free to use.

The root layout of this dialog is a CardView, because it has a shadow. The thickness of the shadow can be changed by using the cardElevation attribute.

This LinearLayout is directed horizontally, to make it possible to order two buttons right next to each other.

Figure 28 - Delete account dialog

Frontend creation of the “password reset over email” dialog

This dialog is used to send a password reset email to an entered email address, if the address is connected to an account stored in the database. If the user tries to enter an invalid email address, an error popup is shown.

The password reset over email dialog consists of the following XML components:

The color of the TitleBar can be changed by using a custom Theme.

This is the same RelativeLayout used by the “password reset” dialog. The application knows where the user accesses this dialog from and changes the values of the dialog to match the request.

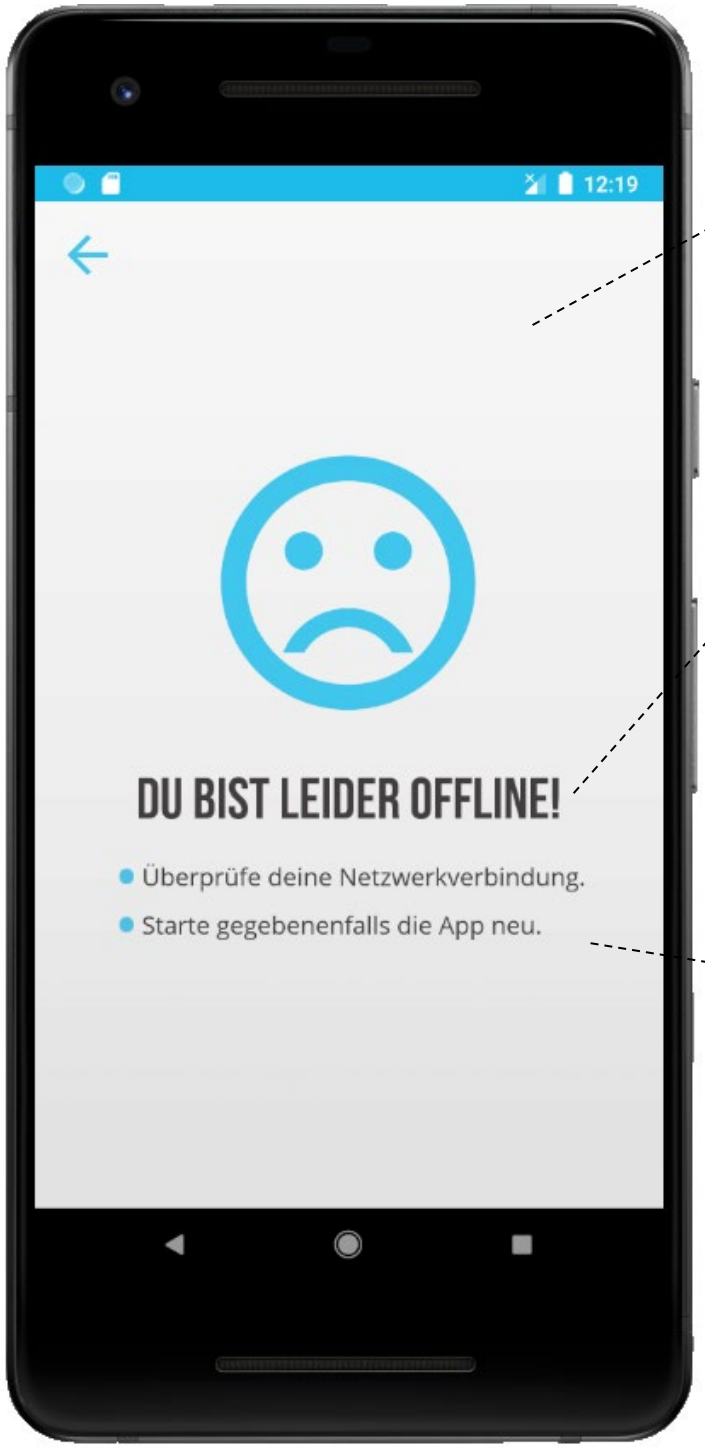
A thin line is created by using a view with a background color. The view’s size can be changed by using the width and height attributes.

Figure 29 - Password reset dialog frontend

2.3.3 Frontend creation of the error screen

The user must be notified if an error has occurred. Therefore, the app needs an error screen. This screen will pop up if the user is offline or if an API request times out.

The error screen consists of the following XML components:



This LinearLayout has a custom background. It displays a light gradient from white to black.

This TextView displays the error message, to give the user some feedback on what went wrong.

This layout stores text views, which display a suggestion for how to fix the error.

Figure 30 - Error screen frontend

Sprint 04 – Setting up the connection to the API [18.09.2019 until 02.10.2019]

Tasks processed in this Sprint:

2.4.1 Setting up the connection to the API

2.4.2 Creating a functional login

2.4.1 Setting up the connection to the API

To set up a connection between the API and the Android application, the following things must be done:

API

Creating a new route

First, a test call was added, to test if the connection between the Android app and the API was successful. This was done by creating a new file named app.js, which sets up the routes of the API.

```
app.use(morgan('dev'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.use('/user', userRoute);

//if the route is not found the API returns a 404 Not found error
app.use((req, res, next) => {
    const error = new Error('Error 404');
    error.status = 404;
    next(error);
});

//if the route is not found the API returns a 404 Not found error
app.use((error, req, res, next) => {
    res.status(error.status || 500).json({
        message: error.message
    })
});

//exports the app, which allows for further usage
module.exports = app;
```

Creating a connection to the mongodb

The first thing that must be done is creating an account on the mongodb website.
Then a new project must be created. This is done by clicking the “Projects” button under the organization tab. After this, a new cluster must be created, which is done by simply clicking on Clusters → Build a new Cluster tab. Then a region and a cluster storage size must be selected.

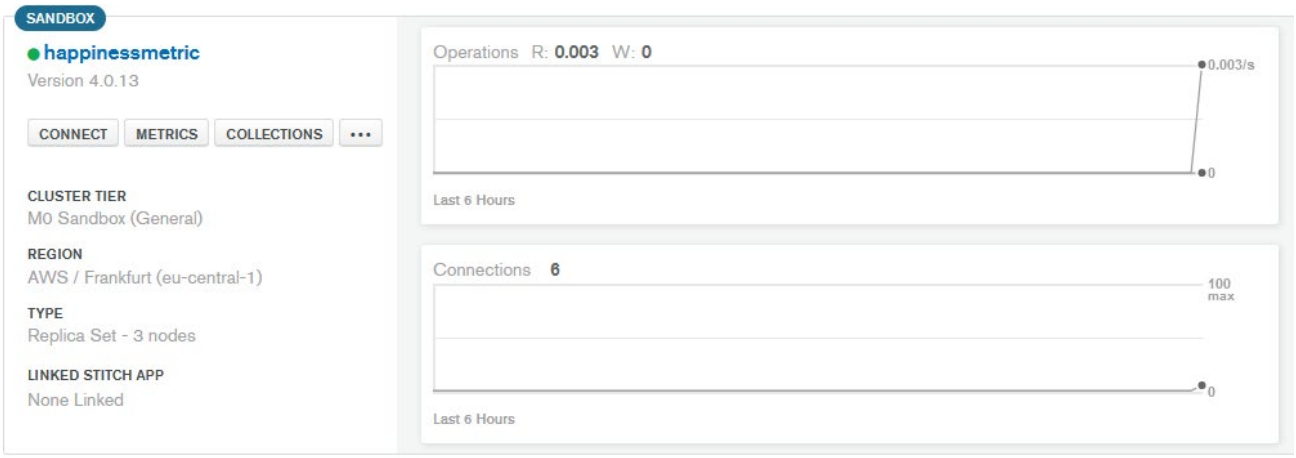


Figure 31 - MongoDB Cluster info

The connection string can be accessed by clicking on connect → Connect your application → Connection String only. The <password> string must be replaced with the password the user created earlier. This was done by creating a new variable in the .env file called MONGODB_PASSWORD.

```
//implementing the mongoose and morgan module
const mongoose = require('mongoose');
const morgan = require('morgan');

//creating the connection to the mongodb using the connection string
mongoose.connect('mongodb+srv://holluc15:' + process.env.MONGODB_PASSWORD + '@ha
ppinessmetric-z8wvn.mongodb.net/happinessmetric?retryWrites=true&w=majority',
{ useNewUrlParser: true, useUnifiedTopology: true })
```

2.4.2 Creating a functional login

First of all, a new file has to be created called “routes”.

To create a new route, the method must first be specialized.

```
router.post('/login', (req, res, next) => { }
```

After this a test must be run, to check if a username and a password are contained in the body of the call.

```
if (req.body.username === undefined || req.body.password === undefined) {
    //it will return an error with the status code of 400
    return res.status(400).json({
        message: 'Username or password is missing'
    });
}
//searches for a user with the username provided in the body of the call
User.find({ username: req.body.username })
    .exec()
    .then(user => {});
        if (user.length < 1) {
            return res.status(401).json({
                message: 'User does not exist'
            });
        }
  //compares the entered password to the hashed password of the user
  bcrypt.compare(req.body.password, user[0].password, (err, result) => {
            if (err) {
                return res.status(500).json({
                    message: 'Comparing failed'
                });
            }

            if (result) {
               /*if they are the same a new access token is created, which
               stores the username and the email of the user*/
                const accessToken = jwt.sign(
                    {
                        username: req.body.username,
                        email: user[0].email

                    }, process.env.SECRET_ACCESS_TOKEN);


//if everything went right, a 200 code will be returned with the following
parameter: message,accesstoken and email
              return res.status(200).json({
                    message: 'Successfully logged in',
                    accessToken: accessToken,
                    email: user[0].email });
```

To create a new User Collection, which stores the user information in the mongodb, the following file must be created: modules/user.js.

This file tells the database which parameters a table has and which datatype they are.

```
//creates a schema in the database with the variables provided
const userSchema = mongoose.Schema({
    username: String, password: String, email: String });
```

Creating an API Software Development Kit (SDK)

An SDK is required to create a connection to the API. First, a new file called rest/HappinessSDK.kt was created. The connection to the API will be built in this file using an Android library called retrofit.

First, an object must be created that, which stores the routes to the API

```
var service: GetDataService
```

After this, a logger is needed, that outputs the calls and their results to the console

```
val logging = HttpLoggingInterceptor()
logging.level = HttpLoggingInterceptor.Level.BASIC

//applies the logger
okBuilder.addInterceptor(logging)
```

Then moshi has to be implemented, which parses json into Java objects.

```
val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()
```

> "A converter that changes strings as well as both primitives and their boxed types to text/plain bodies must be added to the builder object." (Java Documentation, 1993)

```
builder.addConverterFactory(MoshiConverterFactory.create(moshi))
builder.addCallAdapterFactory(RxJavaCallAdapterFactory.create())

    okBuilder.addInterceptor { chain ->

        val original = chain.request()
        val request = original.newBuilder()
            //This changes the content type to "application/json"
            .header("Content-Type", "application/json;charset=utf-8")
            .header("Accept", "application/json;charset=utf-8")
            .method(original.method, original.body)
            .build()
        chain.proceed(request)
    }
```

Creating a Data Service which stores the API calls

A new Interface must be created, that stores every single API call and their parameters. I created Therefore, a new file was created: rest/GetDataService.kt

```
interface GetDataService {

    /*Creates a POST call with the route user/login and the parameters:
      username and password. It returns a LoginUser object which stores the
      returned parameters of the API*/
    @FormUrlEncoded
    @POST("user/login")
    fun login(@Field(value = "username") username: String,
              @Field(value =    "password") password: String):
    Observable<Response<LoginUser>>
}
```

A new file called beans/User.kt was created, that stores the object schemas needed to process API call returns.

This creates a LoginUser object schema.

Every parameter stored here must match the returns of the API.

```
data class LoginUser(
    var accessToken: String,
    var email: String
)
```

Invoke the login call

This Android project was set up to be a single activity project, which means it only has a MainActivity and each different screen is a fragment. Therefore, a fragment called fragments/LoginFragment.kt was created which contains the logic behind the login process.

Fragments contain two override methods: the onCreateView() method, which sets the displayed .xml layout and the onViewCreated() method, which is basically a constructor.

```
class LoginFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
         //displays the login layout at startup of the fragment
        return inflater.inflate(R.layout.login_layout, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        /*This method will be called if the fragment gets added to the
          fragment manager. Therefore, initialization will be done here*/

        super.onViewCreated(view, savedInstanceState)
    }
}
```

First, an instance of the login button must be created and then a click listener must be set in order to detect if the button gets clicked. This is done with the following lines of code:

```
val loginButton = view.login_button
loginButton.setOnClickListener {
 //get entered username and password
 val username = view.username_field.text.trim().toString()
 val password = view.password_field.text.trim().toString()

 }
```

If the login button gets clicked, the login API call must be invoked. This is done by calling the login method from the data service. The service must be initiated as a global variable, like shown below:

```
private val service = HappinessSDK.service
private var sharedPreferences: SharedPreferences
```

To send an API call, the following lines of code must be implemented in the click listener of the login button:

```
//this is how the login call is called
service.login(username,password)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe({ response ->
       if (response.isSuccessful) {
           val body = response.body()!!
           //user successfully logged in

       }
   }, { error ->
      //prints out the error if one occurred
      error.printStackTrace()
   })
```

Now the fragment must be called at the startup of the application. This is done by first checking if the user has a stable internet connection and then adding it onto the fragment manager.

A ConnectivityManager was used, to check if a user is online.

```
private fun isOnline(): Boolean {
    val cm: ConnectivityManager =
    getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    return if (cm.activeNetworkInfo != null) {
        val netInfo: NetworkInfo = cm.activeNetworkInfo
        netInfo.isConnected
    } else {
        false
    }

}
```

By adding the following lines to the onCreate() method of the MainActivity, the login fragment will be displayed, if the user has an internet connection.

```
if (isOnline()) {
  supportFragmentManager.beginTransaction()
                        .add(R.id.main_frame,LoginFragment())
                        .addToBackStack("LoginFragment").commit()}
```

Sprint 05 – Creation of the remaining API calls [02.10.2019 until 31.10.2019]

Tasks processed in this Sprint:

2.5.1 Creating an anonymous happiness tracking system

2.5.2 Displaying the Sprint data in the MPAndroidChart

2.5.3 Creating a functional profile info menu

2.5.1 Creating an anonymous happiness tracking system

The happiness tracking system had to track users anonymously, while still being able to distinguish between users who already voted in a certain Sprint and users who did not vote. To meet those requirements, the team used a database model, which does exactly that. This works by implementing a table that stores which users voted in which Sprints but does not store what happiness values the users entered.

To recreate this Entity Relationship Diagram (ERD) in the mongo-database, schemas were created first.

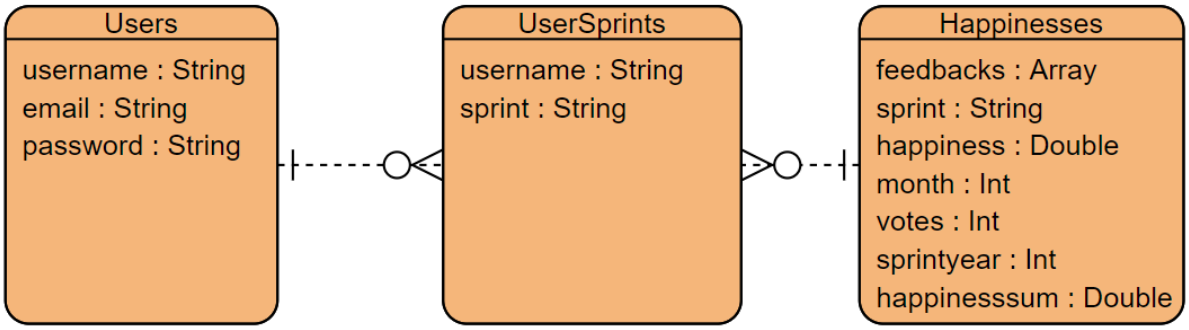


Figure 32 - Entity Relationship Diagram

```
//this is the schema for the Happiness table
const happinessSchema = mongoose.Schema({
    sprint: String,
    happiness: Number,
    feedbacks: [String],
    sprintyear: Number,
    votes: Number,
    happinesssum: Number});
//this is the schema for the UserSprint table
const usersprintSchema = mongoose.Schema({
    username: String,
    sprint: String});
//this is the schema for the User table
const userSchema = mongoose.Schema({
    username: String,
    password: String,
    email: String});
```

After this, a new route, called “happiness”, was implemented which is responsible for each voting call.

To create a new route, the following lines had to be added to the app.js file.

```
const happinessRoute = require('./routes/happiness');
app.use('/happiness', happinessRoute);
```

Therefore, a new file called route/happiness.js was created, which stores the API calls.
To restrict the API calls to users, who are permitted to use it, a system had to be implemented to detect if an access token is provided and whether it is valid. This was achieved, by creating a new file called check-access-token.js.
This file verifies the access token and returns an error code if it is not valid.

```
const jwt = require('jsonwebtoken');
const User = require('./models/user');

module.exports = async (req, res, next) => {
    try {
        /*decodes the access-token with the SECRET_ACCESS_TOKEN provided in
          the .env file*/
        const decoded = jwt.verify(req.headers.authorization,
                                   process.env.SECRET_ACCESS_TOKEN);
        req.tokenData = decoded;

    /*searches for a user, with the username, which was stored in the
      access-token*/
    User.find({ username: req.tokenData.username})
    .exec()
    .then(user => {
        if (user.length === 0) {
            return res.status(401).json({
                message: 'Auth failed'
            });
        }
        next();
        });
    } catch (error) {
        return res.status(401).json({
            message: 'Auth failed'
        });
    }
};
```

To apply the check-access-token file to a call, it must simply be called as shown below:

```
const checkAccessToken = require('../check-access-token');
router.post('/track', checkAccessToken, (req, res, next) => {…}
```

Creation of the happiness tracking call

Nodejs

This call will store the entered happiness value of a user in the database and, if the Sprint in which he voted does not yet exist, it will be created.

First, each parameter is checked to see if it is correct. If not, a 900-error code with a message will be returned.

```
if(req.body.happiness > 10 || req.body.happiness < 0) {
    return res.status(900).json({
        message: 'Incorrect Happiness'
    });
}
```

Then the program checks whether the user already voted. This is done by searching in the database for an entry of matching that user in that particular Sprint.

```
Usersprint.find({ username: req.tokenData.username, sprint: req.body.sprint })
        .exec()
        .then(result => { … });
```

If the result is greater than 1, the user has already voted once. To check for this, an “if statement” was used.

```
if (result.length >= 1) {
                return res.status(400).json({
                    message: "Already voted!"
                });
}
```

If the user has not yet voted, a new user-sprint entry will be created.

```
const usersprint = new Usersprint({
                    username: req.tokenData.username,
                    sprint: req.body.sprint
                })

                usersprint.save()
```

Then, the current happiness entry must be found. If no other entry exists, it is the first entry.

```
Happiness.find({ sprint: req.body.sprint })
                    .exec()
                    .then(result => {
                        if (result.length == 0) {
                            //First entry
                        }
```

If it is the first entry, a new happiness dataset has to be created.

```
const happiness = new Happiness({
 sprint: req.body.sprint,
 happiness: req.body.happiness,
 feedbacks: [req.body.feedback],
 votes: 1,
 sprintyear: parseInt(req.body.sprint.split("/")[0]),
 happinesssum: req.body.happiness });
```

If it is not the first entry, the new happiness level has to be calculated using following formula:

$$\boldsymbol{Sprint\ Happiness} = \frac{\boldsymbol{sum\ of\ each\ entered\ happiness}}{\boldsymbol{number\ of\ happiness\ entered}}$$

The calculation was implemented in NodeJS as shown below:

```
var newVotes = result[0].votes +1
var newHappinessSum = parseInt(result[0].happinesssum)
newHappinessSum += parseInt(req.body.happiness)
var newhappiness = newHappinessSum/newVotes
```

After calculating the happiness level, the feedback is entered into an array.

```
var newFeedback = result[0].feedbacks
newFeedback.push(req.body.feedback)
```

Then the database entry is updated as shown below:

```
Happiness.updateOne({ sprint: req.body.sprint },
{ $set: {
     happiness: newhappiness,
     votes: newVotes,
     happinesssum: newHappinessSum ,
     feedbacks : newFeedback }
},function(err,res) {
     if (err) throw err;
});
```

If everything worked correctly, it will now be possible to view the database entry on the mongodb website.

First, a usersprint entry is generated, which shows who has already voted and who has not.

```
_id: ObjectId("5ddec7c6b2cdc9001719cecf")
username:"denovo"
sprint:"2019/24"
__v:0
```

Secondly a happiness entry is created. These show the application the average happiness of a Sprint and they store the values to recalculate the average happiness if a new user votes in this Sprint. Following variables are stored in the database:

```
_id: ObjectId("5dae23fb8ea4490c749c1798")
feedbacks: Array
sprint:"2019/21"
happiness:5.5
month:9
votes:2
sprintyear:2019
happinesssum:11
__v:0
```

The application must know whether the current user has already voted or not. This is done by calling the /hasVoted call. This call returns a Boolean value that provides information as to whether the user has already voted or not. The call is structured like this:

```
router.get('/hasVoted', checkAccessToken, (req, res, next) => { }
```

The call needs two critical information, to determine if a user has voted. Firstly, the username is needed, and secondly the Sprint is needed.

```
Usersprint.find({ username: req.tokenData.username, sprint: req.query.sprint })
        .exec()
        .then(result => { });
```

If a user is found, it will return the value “true”, and if not, it will return “false”.

```
if (result.length >= 1) {
                return res.status(200).json({
                    voted: true
                });
            } else {
                return res.status(200).json({
                    voted: false
                });
```

Android

Now the application can tell users who have not yet voted apart from those who have already voted. The only thing that must be done in Android is to call the API call as soon as the app starts.

This is the call which is defined in the GetDataService class.

```
@GET("happiness/hasVoted")
fun hasUserVoted(@Header(value = "authorization") authorization: String,
@Query(value = "sprint") sprint: String): Observable<Response<Voted>>
```

To invoke a call as soon as the app starts it must be placed in the onViewCreated() method of the main fragment.

This is how it gets called in the fragment:

```
HappinessSDK.service.hasUserVoted(
    MainActivity.USER.accessToken!!,
    getCurrentSprint())
    .subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
    .subscribe({ response -> })
```

The getCurrentSprint() method returns the current Sprint as a string value.

```
fun getCurrentSprint(): String {
    val calendar: Calendar = Calendar.getInstance()
    return String.format("%d/%d",calendar.get(Calendar.YEAR),
    calendar.get(Calendar.WEEK_OF_YEAR) / 2 }}
```

2.5.2 Displaying the Sprint data in the MPAndroidChart

NodeJS

A call was created, that returns all data needed to show it in an MPAndroidChart. To start, the call that, provides the Sprint data. The route is called /sprints and it will return all data stored from one Sprint year.

```
router.get('/sprints', checkAccessToken, (req, res, next) => { }
```

The call simply searches for happiness entries with the provided Sprint year.

```
Happiness.find({ sprintyear: req.query.sprintyear })
         .exec()
         .then(data => { });
```

If at least one sprint is found in the Sprint year, the call will return an array with all Sprints found.

```
return res.status(200).json({
                sprints: data
            });
```

Android

To get the data stored in the database, the /sprint call must be invoked. This is done by firstly creating a new call in the GetDataService class, and secondly calling it when the app launches. To create a new call in the GetDataService class, the following lines had to be written. They define the route of the call and the parameter types.

```
@GET("happiness/sprints")
fun getSprintsFromYear(@Header(value = "authorization") authorization: String,
@Query(value = "sprintyear") sprintyear: Int): Observable<Response<SprintData>>
```

Then the call can be invoked like this:

```
HappinessSDK.service.getSprintsFromYear(MainActivity.USER.accessToken!!, year)
    .subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
    .subscribe({ response -> })
```

Next, the entries for the chart had to be created. This was done by creating a new ArrayList, which contained every value returned from the getSprintsFormYear call.

```
val entries = ArrayList<Entry>()
//After adding every entry to the array list, it had to be sorted by the x-value
val sortedList = entries.sortedWith(compareBy { it.x })
dataSet = LineDataSet(sortedList, "Label")
```

2.5.3 Creating a functional profile info menu

To add functionalities to the already designed Profile Info menu, it first had to be ensured that everything was stored safely in the database. Therefore, the calls were checker and an issue was found. The issue was that a user with no access rights could invoke the profile info call and gain insights into a certain user. This issue was quickly fixed by creating a method that takes the username stored in the access token to search for users in the database.

```
if (req.tokenData.username === undefined) {
        return res.status(401).json({
            message: 'Auth failed'
        });
    }

User.find({ username: req.tokenData.username })
        .exec()
        .then(user => { }
```

Another call had to be created to enable a user to delete his account. This was quite challenging, because it entailed writing a delete call.

```
router.delete('/deleteAccount', checkAccessToken, (req, res, next) => { }
```

It was important to ensure that only the owner of the account could delete it. This was done by checking the username stored in the access token. If a user was found, and it was made sure that the account belongs to him, then it can safely be deleted. Every entry that was made with this account must also be deleted. This can be achieved by using the deleteMany() option given by mongodb.

```
User.deleteOne({ username: req.tokenData.username }, function(err, res) {
                        if (err) throw err;

                    });
UserSprint.deleteMany({ username: req.tokenData.username }, function(err, res) {
                        if (err) throw err; });
```

Android

To be able to upload a profile picture, the application required access the photo library of the user's device. This was done by using the following lines of code:

```
private fun pickImages () {
    val intent = Intent(Intent.ACTION_PICK)
        intent.type = "image/*"
    val mimeTypes = arrayOf("image/jpeg", "image/png")
    intent.putExtra(Intent.EXTRA_MIME_TYPES, mimeTypes)
    startActivityForResult(intent, PROFILE_REQUEST_CODE)
}
```

If you start an activity for a result, it means that something will be returned.

To see if the user finished the selecting process, the RESULT_OK flag was used.

```
if (resultCode == AppCompatActivity.RESULT_OK
 && requestCode == PROFILE_REQUEST_CODE) {}
```

To send an image over http, a multipart body call had to be created to convert the image into a byte array.

```
val selectedImage: Uri = data!!.data!!
val imageStream: InputStream =
    activity?.contentResolver?.openInputStream(selectedImage)!!
val bitmap: Bitmap = BitmapFactory.decodeStream(imageStream)

bitmap.compress(Bitmap.CompressFormat.JPEG, 100, ByteArrayOutputStream())
```

A new call in the GetDataService class was created that must be able to send a multipart body. Therefore, the call was annotated with the @Multipart annotation, as shown below.

```
@Multipart
@POST("user/uploadProfilePicture")
fun uploadProfilePicture(@Header(value = "authorization") authorization: String,
@Part profilePicture: MultipartBody.Part): Observable<Response<ProfilePicture>>
```

Then, the selected image was parsed from the photo gallery into a file using its URI. An Extension was written which retrieves the Path from a URI.

```
fun Context.getPathFromURI(contentUri: Uri): String? {
    val loader = CursorLoader(applicationContext, contentUri,
arrayOf(MediaStore.Images.Media.DATA), null, null, null)
    val cursor = loader.loadInBackground()
    val columnIndex =
cursor?.getColumnIndexOrThrow(MediaStore.Images.Media.DATA)
    cursor?.moveToFirst()
    cursor?.getString(columnIndex!!).let {
        return it!!
    }
}
```

After this, the extension is called using the context variable.

```
val file = File(context!!.getPathFromURI(selectedImage))
```

At the end, a multipart request body, had to be created to be able to send the file over http.

```
// Create a request body with file and image media type
val fileReqBody = file.asRequestBody("image/jpg".toMediaTypeOrNull())
// Create MultipartBody.Part using file request-body,file name and part name
val part = MultipartBody.Part.createFormData("image", file.name, fileReqBody)
```

Sprint 06 – Setting up of the AWS Bucket System [31.10.2019 until 14.11.2019]

Tasks processed in this Sprint:

2.6.1 Setting up the Amazon S3 service

2.6.2 Creating a connection to the bucket using NodeJS

2.6.3 Uploading a profile picture

2.6.1 Setting up the Amazon S3 service

An AWS Bucket is basically a storage system, which can be accessed by the API using NodeJS. Buckets provide many advantages, like low latency and 99.99% uptime. Buckets was used in this project to store the profile pictures, which are uploaded by the users.
First, an account had to be registered on the Amazon s3 website. To create a new AWS-Bucket, a region and a name must be entered. The latency will depend on where the bucket is located.

2.6.2 Creating a connection to the bucket using NodeJS

After this, the Bucket was connected using NodeJS. This was done by, first configuring the variables.

```
// Set the region
AWS.config.setPromisesDependency();
AWS.config.update({
    accessKeyId: process.env.ACCESS_KEY_ID,
    secretAccessKey: process.env.SECRET_ACCESS_KEY,
    region: 'eu-west-3'
});

const s3 = new AWS.S3();
const response = s3.listObjectsV2({
    Bucket: 'happinessmetric',
    Prefix: 'profile_pictures'
}).promise();

var params = {
    Bucket: 'happinessmetric',
    Key: "profile_pictures/" + req.tokenData.username + ".png"
};
```

If everything was configured correctly, there should be no problem connecting to the Bucket using following lines of code.

```
response.then(function (result) {
        // Using callbacks
        s3.headObject(params, function (err, metadata) {
              const url = s3.getSignedUrl('getObject', {
                    Bucket: 'happinessmetric',
                    Key: "profile_pictures/" + "user_icon" + ".png"})
        });}
```

2.6.3 Uploading a profile picture

While creating the /uploadProfilePicture call, attention had to be paid to the filetype of the image, because if someone tries to upload a .gif file or something else, an exception would occur.

```
const fileFilter = (req, file, cb) => {
     if (file.mimetype == 'image/jpeg'
         || file.mimetype == 'image/jpg'
         || file.mimetype == 'image/png') {
            cb(null, true)
        } else {
            cb(new Error('Invalid Mime Type, only JPG and PNG'), false);
        }
    }
```

To upload an image, multer which is a middleware for handling multipart/form-data, was used to create a file out of the multipart body and upload it to the AWS-Bucket.

```
const upload = multer({
        fileFilter: fileFilter,
        storage: multerS3({
            s3: s3,
            bucket: 'happinessmetric',
            acl: 'public-read',
            metadata: function (req, file, cb) {
                cb(null, { fieldName: 'MetaData' });
            },
            key: function (req, file, cb) {
                cb(null, 'profile_pictures/' + req.tokenData.username + '.png')
            }
        })
    })

    const singleUpload = upload.single('image');
```

To upload a file, the singleUpload() function was called. If everything works correctly, the API will respond with the link of the previously uploaded profile picture, which will be displayed in Android using the Glide library.

```
return res.status(200).json({
            profilePictureLink: req.file.location
        })
```

Glide is an image loading library, with significant performance and usability.

```
Glide
    .with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .into(myImageView);
```

Sprint 07 – Creating Firebase notifications [14.11.2019 until 28.11.2019]

Tasks processed in this Sprint:

2.7.1 Setting up the Firebase service

2.7.2 Creating a connection to the Firebase Cloud Messaging service

2.7.3 Creating weekly notifications

2.7.1 Setting up the Firebase service

First, a new project on the Google Firebase website was created. It had to be ensured that the app-id and project id were valid, because if they were not, they would need to be changed. After finishing this verification, it was possible to add the cloud messaging service to the project by clicking on the cloud messaging tab.

2.7.2 Creating a connection to the Firebase Cloud Messaging service

To add Firebase to an existing Android project, it had to be ensured that it targets API level 16 or higher and that it uses Gradle 4.1 or higher. It is possible to view the selected API level in the build.gradle file.

```
defaultConfig {
     minSdkVersion 26
     targetSdkVersion 28
}
```

Then, the project had to be registered with Firebase.
This was done by entering the package name into the Firebase console. At the end, a Firebase configuration file and the Google-services plugin had to be added to the project.

```
// Add the Firebase SDK for Google Analytics
implementation 'com.google.firebase:firebase-analytics:17.2.2'

// Firebase Authentication and Cloud Firestore
implementation 'com.google.firebase:firebase-auth:19.2.0'
implementation 'com.google.firebase:firebase-firestore:21.3.1'
```

2.7.3 Creating weekly notifications

A Firebase Messaging Service must be created to be able to receive notifications. This class extends from FirebaseMessagingService() and its only purpose is to receive, and display notifications sent by the Firebase Cloud Messaging service.

```
val notificationManager: NotificationManager =
    getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager


val builder = NotificationCompat.Builder(this, CHANNEL_ID)
builder.setAutoCancel(true)
    .setWhen(System.currentTimeMillis())
    .setSmallIcon(at.denovo.happinessmetric.R.drawable.ic_notification_face)
    .setContentTitle("Bitte vergiss nicht deine Glücklichkeit abzugeben!")
    .setStyle(bigText)

notificationManager.notify(1, builder.build())
```

If the user clicks on the notification, he will be forwarded to the voting menu in the happiness-metric application.

A received notification looks like this:

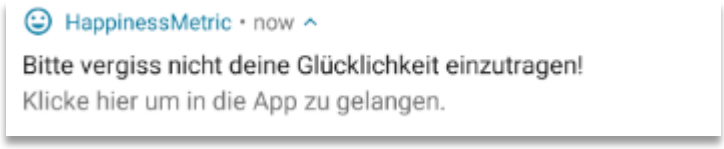


Figure 33 - Notification

To create weekly notifications, a simple notification plan must be created. It is possible to select the time and date on which notifications will be sent out. For this project, this was set to a Monday every two weeks, because it's always in a new Sprint.

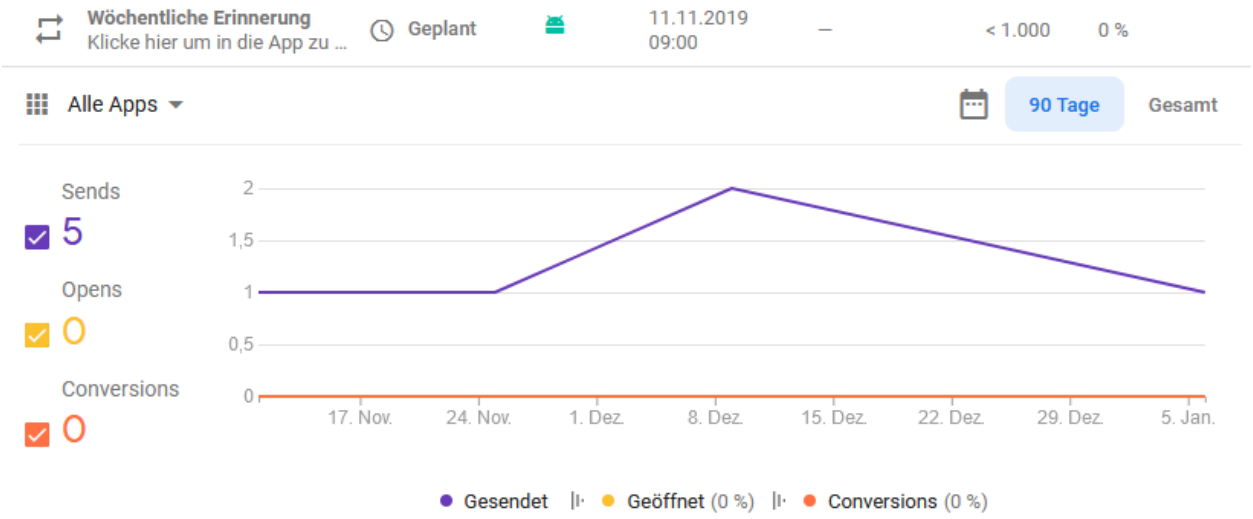


Figure 34 - Firebase statistics

Sprint 08 – Creating a mail service [28.11.2019 until 12.12.2019]

Tasks processed in this Sprint:

2.8.1 Setting up the Mailgun service

2.8.2 Creating a connection to Mailgun service

2.8.3 Creating an email template

2.8.4 Resetting the password using a button in the e-mail

2.8.1 Setting up the Mailgun service

To set up a Mailgun service, a new Mailgun account must be created using its homepage.
To be able to send a mail to a user, the user must be added to the authorized recipients list on the Mailgun website. After adding test users, a sandbox must be created. This is possible under the sending tab in the side bar menu.

Figure 35 - Verified e-mails

2.8.2 Creating a connection to the Mailgun service

Now the Mailgun service must be connected with the application. This is done by adding a new call that sends an email to an entered email address, but only if the address is on the authorized recipients list. First it has to be checked to see whether the entered email address is valid.

```
var emailError = validator.invalidEmail(req.query.email);
```

If it is, then the user it belongs to has to be found, using following lines.

```
User.find({ email: req.query.email })
        .exec()
        .then(user => { });

const smtp = nodemailer.createTransport(mg(mailgunAuth))

const template = handlebars.compile(emailTemplateSource)

const htmlToSend = template({ updateLink: link })
```

2.8.3 Creating an email template

An email template has to be created using the Hypertext Markup Language (HTML).

```
const html = fs.readFileSync(path.join(__dirname, "../template.hbs"), "utf8")
```

Now the Mailgun needs to authorize the request.

```
const mailgunAuth = {
                auth: {
                    api_key: process.env.MAILGUN_KEY,
                    domain: process.env.MAILGUN_DOMAIN
                }
```

Now the email can be put together and be sent to the destination address.

```
const mailOptions = {
                from: "happiness-metric@denovo.at",
                to: req.query.email,
                subject: "Happiness-Metric Passwort zurücksetzen",
                html: htmlToSend
            }
```

To send an email, `sendMail()` from the Simple Mail Transport Protocol (SMTP) is used.

```
smtp.sendMail(mailOptions, function (error, response) { });
```

2.8.4 Resetting the password using a button in the e-mail

If the button in the email gets clicked a link will be called that redirects the user to an HTML page where he can view his new temporarily password. The new password is randomly generated and has 10 digits.

```
var newPassword = '';
var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
var charactersLength = characters.length;
for (var i = 0; i < 10; i++) {
     newPassword += chars.charAt(Math.floor(Math.random() * charactersLength));
}
```

To display the newly generated password in a reasonably way, the call has to return HTML code. This is done by simply using the response.write() method.

```
res.writeHead(200, {'Content-Type': 'text/html'});
res.write('…html code…');
res.end()
```

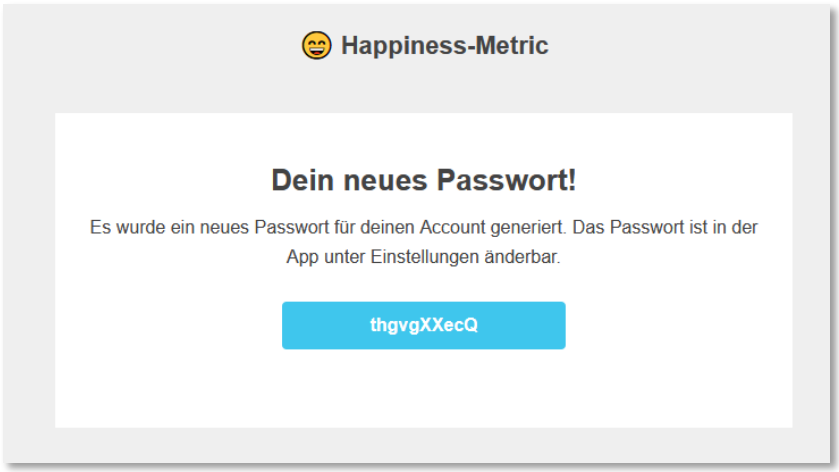


Figure 36 - New Password Page

Happiness-Metric
iOS development

Figure 37 - Apple icon

3.1 Happiness-Metric – iOS Layout

This chapter shows, how the iOS-version of Happiness-Metric looks like and where the differences between the programming in Kotlin (for Android) and Swift (for iOS) are.

3.1.1 Loading screen

This screen appears while the app is loading. While the user can see this screen on his phone, the app will check whether the phone is connected to the internet or not. If it is connected, the user will be redirected to the login screen or even to the Live Data screen.

Figure 38 - iOS Loading Screen

Figure 40 - iOS Login Screen

3.1.2 Login screen

After the app has loaded, this screen appears first. The user has to type in his username or email and his password. If his credentials match with the data in the API, the user will be logged in and the next screen will be visible

Figure 39 - iOS Password forgotten Screen

3.1.3 Password forgotten screen

If the user forgets his password, it is not a problem. He only has to tap on the „Passwort vergessen?“ button and type in his email-address. Then the user will be sent an email with a new password. By using his username and the new, randomly created password, the user will be able to log in again.

3.1.4 Live-Data screen

After the user has successfully logged in, the Live Data screen appears. This screen provides information about the current happiness level of the staff. The smiley graphic, which is shown, changes its facial expression depending on the Live Happiness level. From this screen, users can also access the Sprint-Data screen or the Settings.

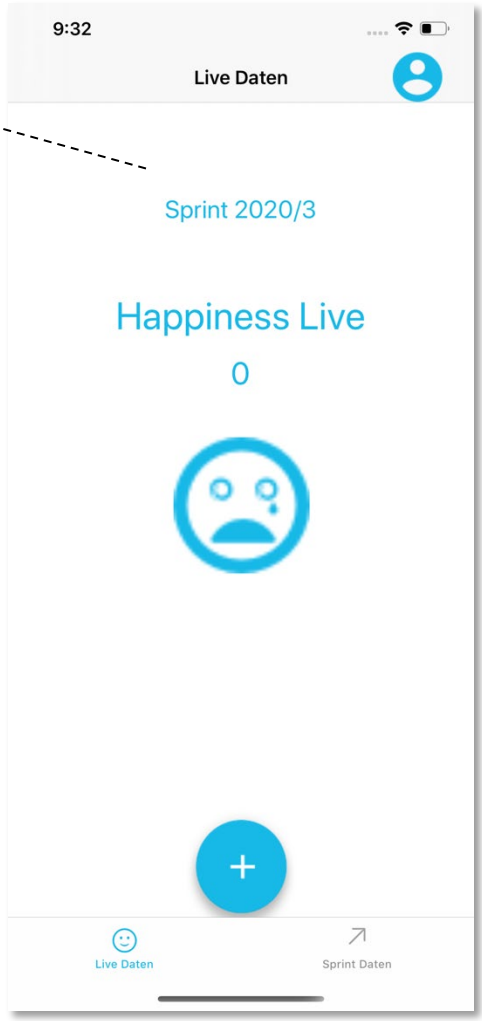


Figure 42 - iOS Live Data Screen

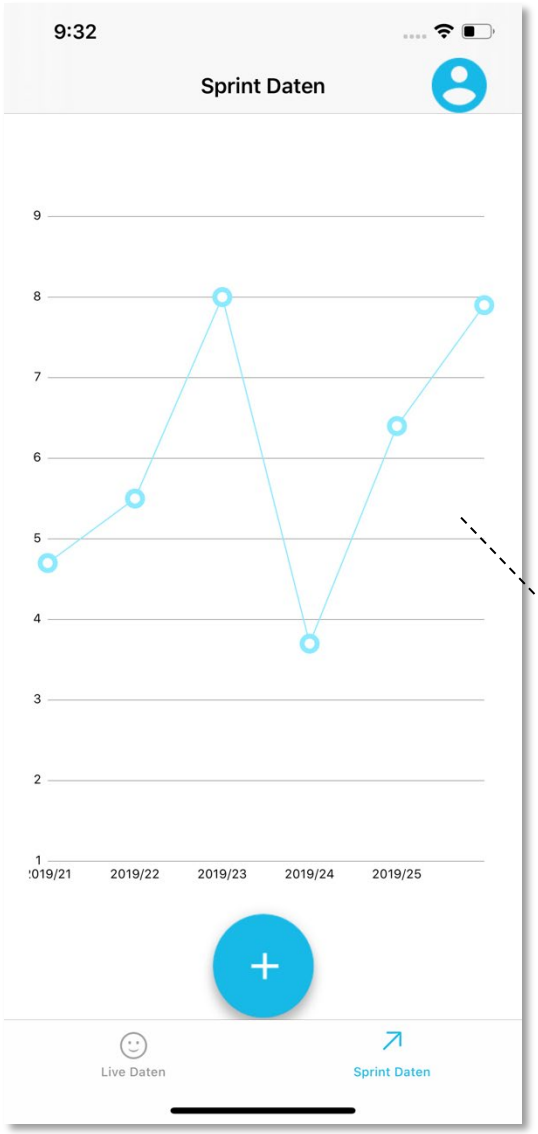


Figure 41 - iOS Sprint Data Screen

3.1.5 Sprint-Data screen

The Sprint-Data screen provides a nice overview of the past Sprints. Every point in the chart stands for the happiness value of a certain Sprint. Toward the bottom of the screen, users can see a button with a plus, which is the button that enables users to input and track their own happiness.

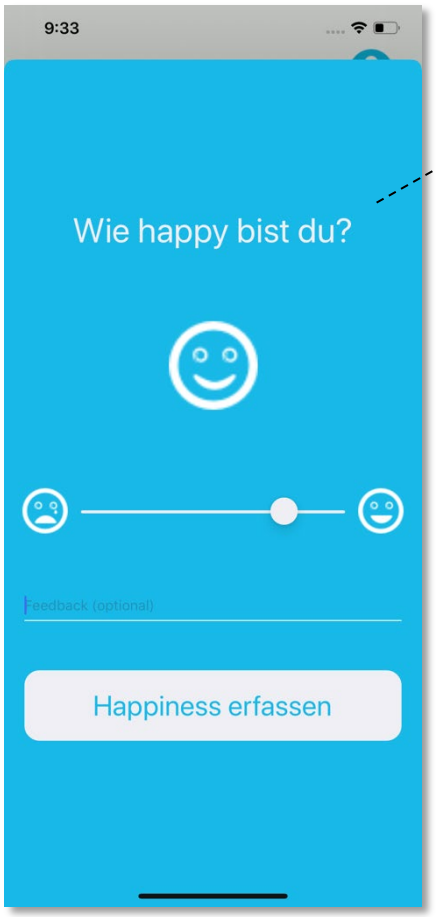


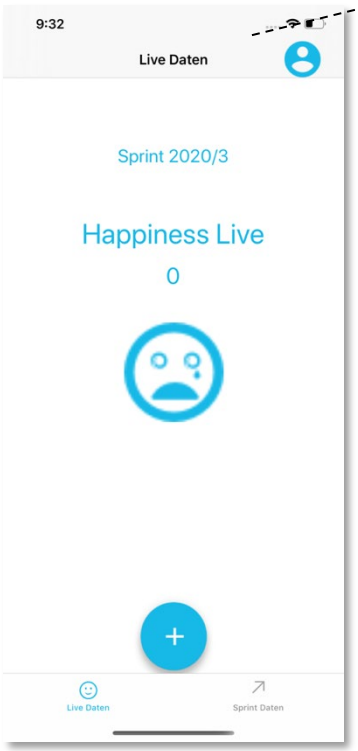
Figure 43 - iOS Track Happiness Screen

3.1.6 Track-Happiness screen

As said above, this is the screen where a user can track his happiness for the current Sprint. Additionally, it is possible for the users to write a feedback about why they are happy or why they are not happy. Note: users can only track their happiness once per Sprint so that the results cannot be manipulated.

3.1.7 Menu screen

If the user clicks on the user icon, he accesses the menu screen. Here the user can decide whether he makes some Settings or log out.



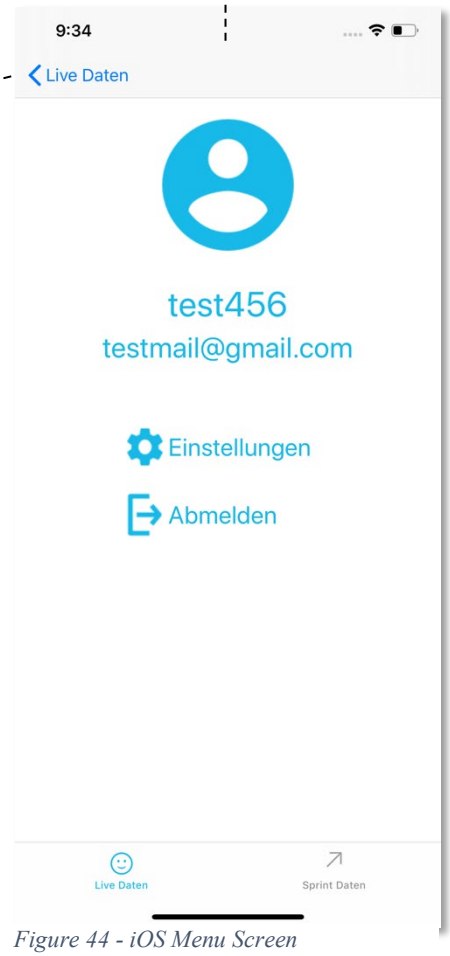


Figure 44 - iOS Menu Screen

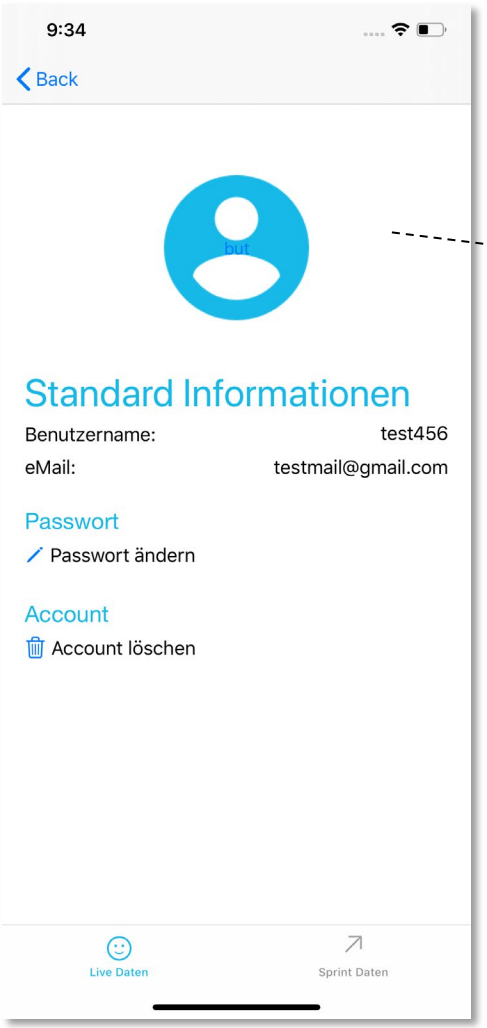


Figure 46 - iOS Setting Screen

3.1.8 Settings screen

The Settings screen provides two options: one, is to change a user's password, and, two, to delete a user's account.

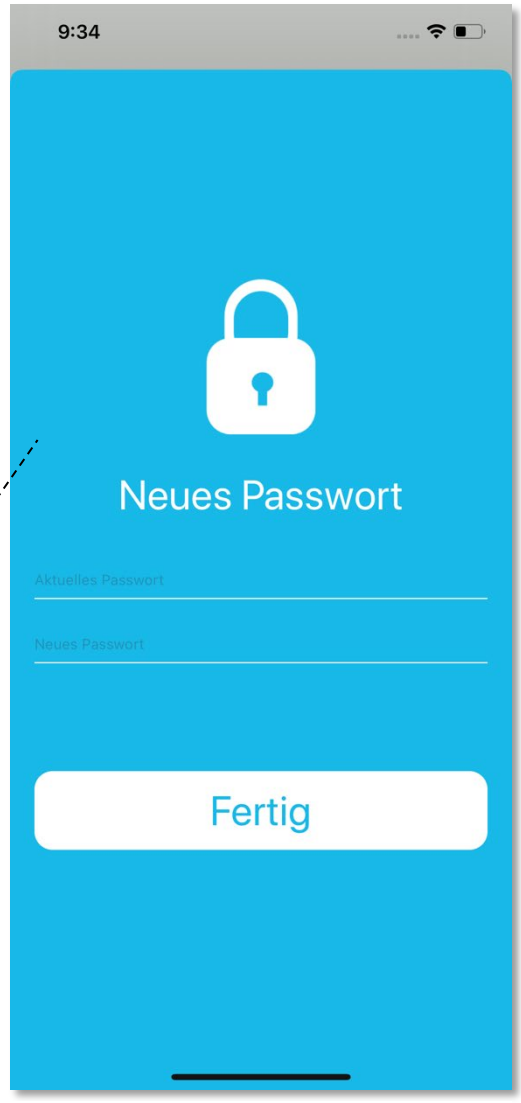


Figure 45 - iOS Password Update Screen

3.1.9 Change Password screen

To change his password, a user has to type in his current password and then the desired new password. If the current password matches with the data in the API, the new password will be set.

Sprint 01 – Getting familiar with swift

During the planning phase of the diploma thesis project, the decision was made for René Wagner to take over the iOS-developing. Although he had no prior knowledge of Swift or even app programming, he was motivated to create the iOS-Version of Happiness-Metric, because of his interest in Apple and their apps. So, for René Wagner, the first Sprint consisted of watching videos and reading websites to learn how to use Swift and xCode. Furthermore, he did some small projects to gain experience in Swift-programming.

Figure 47 xCode icon

xCode
xCode is an IDE (integrated development environment), in which a developer can write programs for different Apple devices. Mainly it is used for Objective-C and Swift programming, but it can also be used for Java, Ruby or C/C++ for example.

Sprint 02 – Setting up the project and first Frontend elements

After completing some practice projects, work began on the diploma thesis project, Happiness Metric. A project was created in Swift and then pushed to the GitLab.

4.2.1 Using GitLab

First, developer user credentials were configured in the Git-config:

```
git config --global user.name "Rene Wagner"
git config --global user.email wagred15@htlkaindorf.at
```

After setting the user credentials, the developer should be able to connect to the GitLab repository, and open the project folder in the terminal with:

```
cd happiness-metric
```

followed by:

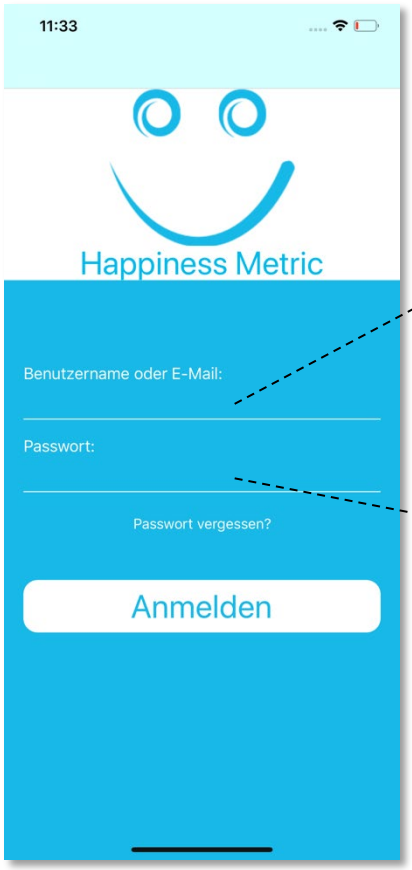
```
git init
git remote add origin git@gitlab.com:denovo/happiness-metrics-ios.git
```

To push the existing project, you have to use following commands:

```
git add .
git commit -m "Initial commit"
git push -u origin master
```

4.2.2 First frontend elements

To create frontend elements, elements can be easily dragged and dropped into the Main.storyboard, which is automatically created when creating a project. First, the login screen was created. Adding special designs to the screen must be done programmatically, just like the textfields.




```
func setBottomBorderToTextFields()  {

        let bottomLine = CALayer()
        bottomLine.frame = CGRect(x: 0, y:
usernameTextField.frame.height - 1, width:
usernameTextField.frame.width, height: 1)
        bottomLine.backgroundColor =  colorLiteral(red:
1.0, green: 1.0, blue: 1.0, alpha: 1.0)
        usernameTextField.borderStyle =
UITextField.BorderStyle.none
        usernameTextField.layer.addSublayer(bottomLine)

        let bottomLine1 = CALayer()
        bottomLine1.frame = CGRect(x: 0, y:
passwordTextField.frame.height - 1, width:
passwordTextField.frame.width, height: 1)
        bottomLine1.backgroundColor =
colorLiteral(red: 1.0, green: 1.0, blue: 1.0, alpha:
1.0)
        passwordTextField.borderStyle =
UITextField.BorderStyle.none

passwordTextField.layer.addSublayer(bottomLine1)

}
```

Figure 48 Login Screen

What does this code?
This code removes the typical textfield layout elements, like the frame and the white background, and adds a white border line.

Also, a class was created to round the corners of the buttons, which looks like the following:

```
import UIKit

@IBDesignable
class RoundButton: UIButton {

    @IBInspectable var cornerRadius: CGFloat = 0{
        didSet{
            self.layer.cornerRadius = cornerRadius
        }
    }

    @IBInspectable var borderWidth: CGFloat = 0{
        didSet{
            self.layer.borderWidth = borderWidth
        }
    }

    @IBInspectable var borderColor: UIColor =
UIColor.clear{
        didSet{
            self.layer.borderColor = borderColor.cgColor
        }
    }
}
```

This code changes the radius of the corners of the selected buttons. This results in rounded buttons.

Sprint 03 – Visual Frontend

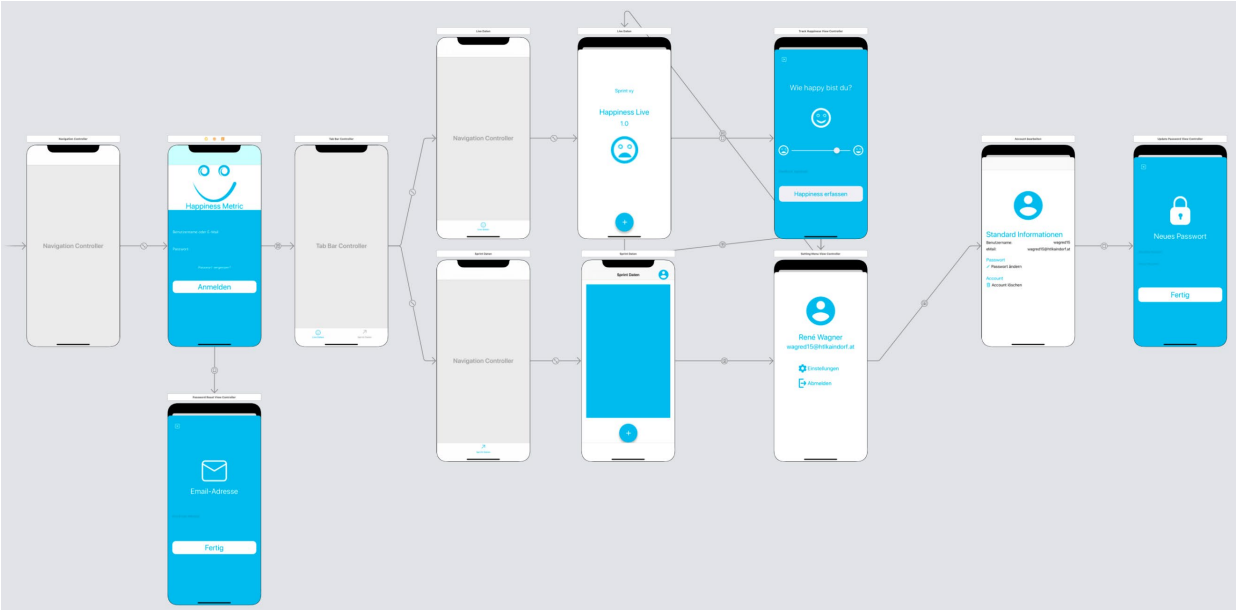


Figure 49 - iOS Happiness Metric Layout

The image shows the whole iOS graphical layout. Here the View Controller, Tab Bar Controller and Navigation Controller can be seen.

4.3.1 View Controller

Every screen that can be seen in the iOS-Layout part, is a View Controller. A View Controller manages different parts of the content of an application. So, if the content does not fit on one screen, more than one View Controller must be used, each of which can, in turn, have again many subviews.

4.3.2 Tab Bar Controller

The Tab Bar Controller is responsible for the Tab Bar, as its name already says. The bar that can be seen on the bottom of the application, when a user is on the Live Data or Sprint Data screen, is a tab bar. So, users can switch between these two screens if they tap on the button in the bar.

4.3.3 Navigation Controller

The navigation controller is responsible for, amongst other things, the back buttons, such as the “Live Daten” in the top left corner for example. It is a kind of container view controller, because it manages child views in a navigation interface, but only one child view is visible at the time.

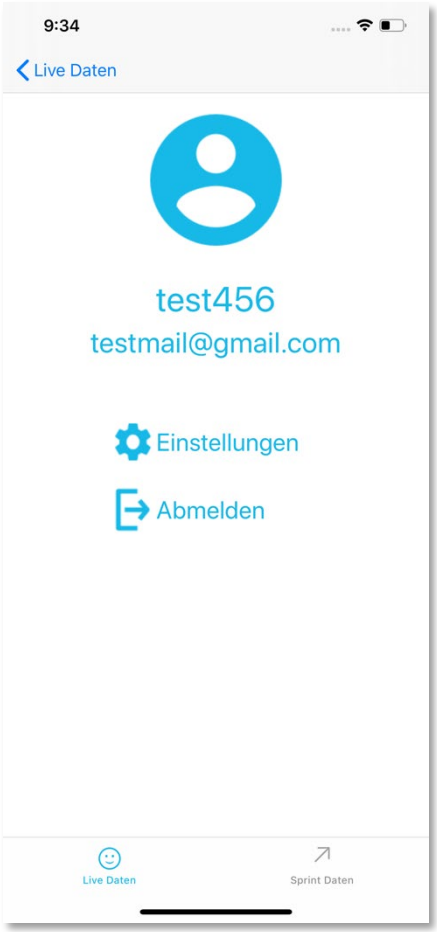


Figure 50 iOS Settings Menu

Labels, textfields, buttons, sliders and image views were also used.

4.3.4 Label

A label is mostly used to write fixed text. The text in the label cannot be changed by the user, but can be changed programmatically by a developer.

The text, “Sprint 2020/3”, is an example of a label. While “Sprint” is always part of the text, “2020/3” can change depending on which sprint it is.

4.3.5 Textfield

The textfield is mostly used for a user input. The login screen for example requires a text input for the username and the password textfields, so that the data can be loaded.

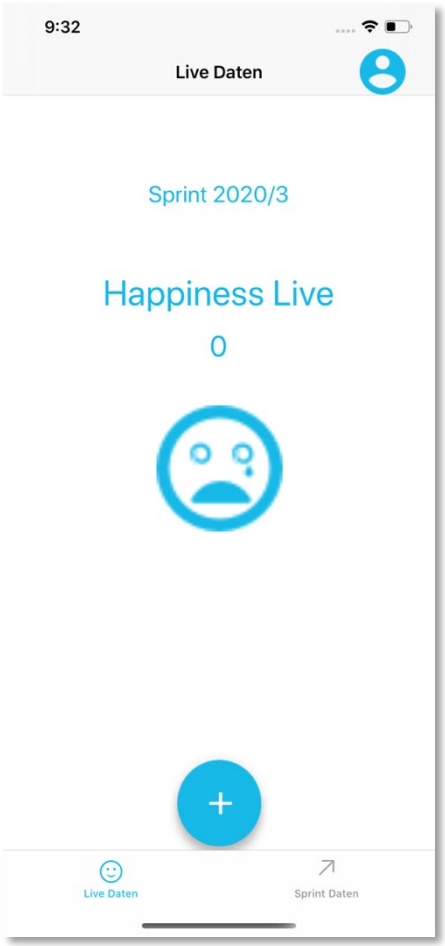


Figure 51 iOS Main Screen

Sprint 04 – Create the chart

4.4.1 CocoaPods

Use of the charts in Swift requires the use of CocoaPods. CocoaPods is a dependency manager for Swift and Objective-C, that makes it possible to easily add and remove libraries. CocoaPods can be installed in the terminal with the following command:

```
sudo gem install cocoapods
```

Then, redirect to the project folder and install a Podfile with:

```
cd path/to/the/projectFolder
```

```
pod init
```

With these commands, the Podfile will have been created in the project directory. Now, the Podfile has to be modified. In the case of this project, the Podfile should have the following written into it:

```
platform: ios, '10.0'
use_frameworks!
target 'HappMetr' do
      pod 'Charts'
end;
```

Finally, type:

```
pod install
```

into the terminal, to finish installing the Charts library.

The charts library can now be used in the Main.Storyboard, but first a Container View must be imported. If you go into Module Charts can now be chosen from the settings of the Container View.

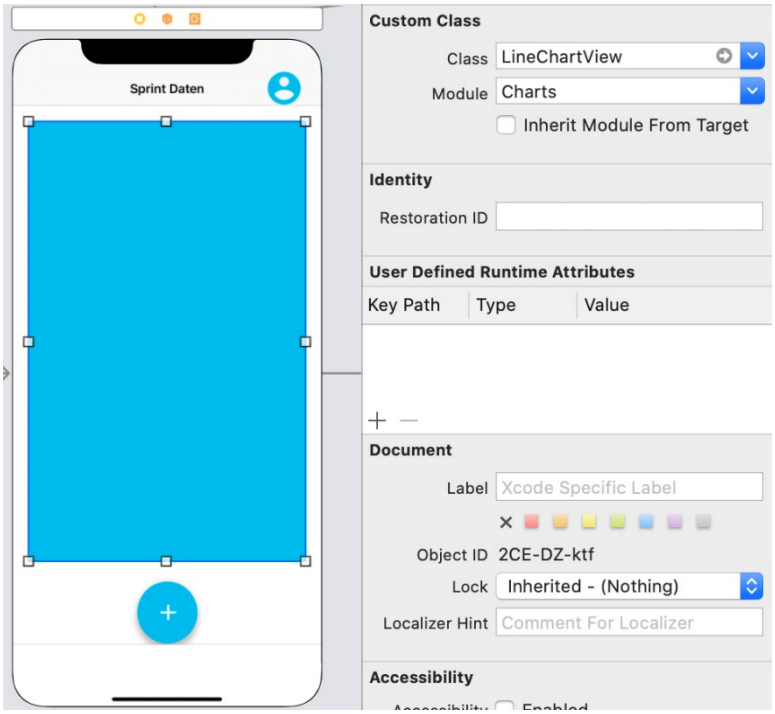


Figure 52 - xCode Chart Import

4.4.2 Axis configuration

The following code shows the y-Axis configuration. The label count sets the maximum value of the y-Axis, while the axisMinimum shows the minimum value of the y-Axis. The labelCount shows the steps between the values on the y-Axis, which, in this project, is always +1 from 1 to 10.

```
//y-Axis configuration
        yAxis.drawAxisLineEnabled = false
        yAxis.setLabelCount(10, force:
true)
        yAxis.axisMinimum = 1
        yAxis.labelCount = Int(10 - 1)


yRight.drawAxisLineEnabled = false
        yRight.drawGridLinesEnabled = false
        yRight.drawLabelsEnabled = false
```

For the x-Axis the labeling was set to the bottom.

```
//x-Axis configuration
        xAxis.granularityEnabled = true
        xAxis.granularity = 1
        xAxis.drawAxisLineEnabled = false
        xAxis.drawGridLinesEnabled = false
        xAxis.labelPosition = .bottom
```

Furthermore, the chart had to be formatted in general. For example, it was formatted so as to not include a zoom in function, a legend or an additional description.

```
//Chart formatting
        chtChart.highlightPerDragEnabled =
false
        chtChart.chartDescription?.text = ""
        chtChart.legend.enabled = false
        chtChart.pinchZoomEnabled = false
        chtChart.doubleTapToZoomEnabled =
false
        chtChart.scaleYEnabled = false
```

Finally, data was set into the chart. This was done with a for-loop, as seen in the following coding example. Also, a range was set for how many x-values should be shown.

```
//Chart Data Import
for n in 0...(s.count)-1{

    chtData.append(s[n]["happiness"] as! Double)
    sprData.append(s[n]["sprint"] as! String)

}

DispatchQueue.main.async {

    self.createGraph(dataForX: sprData, dataForY:
chtData)

}
```

Sprint 05 – Connect Frontend with Backend

For the frontend – backend connection a class called NetworkHandler was created. The NetworkHandler includes all API calls. So, to have a GET or POST request in a view controller, the NetworkHandler and the associated function must be called. The Android and the iOS version are using the same MongoDB database.

4.5.1 An example (Reset Password per Mail)

The API call in the NetworkHandler is as follows:

```
func sendPasswordResetMail(mail: String, completion: @escaping (_
message: String?, _ success: Bool)->()) {

        guard let url = URL(string: Config.url +
"user/sendPasswordResetMail?email=" + mail) else
        {
                return
        }

        var request = URLRequest(url: url, cachePolicy:
.useProtocolCachePolicy, timeoutInterval: Config.timeout)
        request.httpMethod = "GET"

        let dataTask = session.dataTask(with: request) { (data, response,
err) in
            guard let data = data,
                let responseDict = try?
JSONSerialization.jsonObject(with: data, options: []) as? [String: Any?],
                (response as? HTTPURLResponse)?.statusCode == 200 else {
                    completion(nil, false)
                    return
            }

            print("send password request: \(responseDict["message"] ??
"<no message set>")")

            completion(responseDict["message"] as? String, true)

        }

        dataTask.resume()

    }
```

With the responseDict function, variable of the API request can be accessed. The parameters in the completion handler provides the requested values. For this step, the desired information was only a message indicating whether or not everything worked.

The following code calls the NetworkHandler from the PasswordResetViewController:

```
NetworkHandler.shared.sendPasswordResetMail(mail: mail) { (message,
success) in

            if(success == true){

                print("password reset mail --SUCCESSFUL--")

            } else {

                print("password reset mail --FAILED--")

            }

            OperationQueue.main.addOperation {

                let alert = UIAlertController(title: "Passwort
zurücksetzen", message: message, preferredStyle:
UIAlertController.Style.alert)

                alert.addAction(UIAlertAction(title: "O.k.", style:
UIAlertAction.Style.default, handler: { _ in

                    self.dismiss(animated: true, completion: nil)

                }))

                self.present(alert, animated: true, completion: nil)

            }
```

Because the NetworkHandler is called, the API call is triggered and, in the best case, the email with the new password will be sent. However, for safety's sake, the message with the information about whether it worked or not, will also be alerted. The OperationQueue.main.addOperation, which can be seen in the code snippet, is used to add an operation to the main thread, and in some cases it is required. In this OperationQueue Iit was also defined, that the reset password by email screen disappears after the button is clicked.

Sprint 06 – Using keychain

4.6.1 Install Locksmith library

As well as with the Charts, the Cocoapods were again used for the keychain. As the Cocoapods were already installed and the Podfile was already configured in the directory, where the project was also located, only the content of the Podfile had to be changed. As the keychain method is from Locksmith, the library had to be imported..

At this point, the Podfile should look like the following:

```
platform: ios, '10.0'

use_frameworks!

target 'HappMetr' do

      pod 'Charts',

      pod 'Locksmith'

end;
```

4.6.2 Why Keychain?

Keychain is a tool, that helps you to store login data and passwords, and reduces the number of passwords a user has to remember by heart. It also provides security, as it is saves the data to the user's device.

4.6.3 Use Locksmith

For this project, Keychain was used to store the access token. The access token is a unique key that is required to access one's data from the database. It gets stored into the Keychain when a user logs in to his account.

```
try Locksmith.updateData(data: ["accessToken" : key], forUserAccount:
"key")
```

It can be seen in the code snippet, that the key, which is the accessToken, gets stored with the ID "accessToken". To make sure, only authorized people have access to the data, it has to be stored for a user account.

```
let authorization = Locksmith.loadDataForUserAccount(userAccount:
"key")
```

To access the stored key again, which is needed for several API calls, the load data function has to be used. This function returns the stored value and saves it to the authorization variable.

Sprint 07 – Using alerts

Using alerts is a good way to tell the user of the application what is happening within the app or why something is not working. The Login is a good example here, because if the user is not able to log in, he will probably want to know why.

So, if the user types an incorrect password, he will get an alert like:

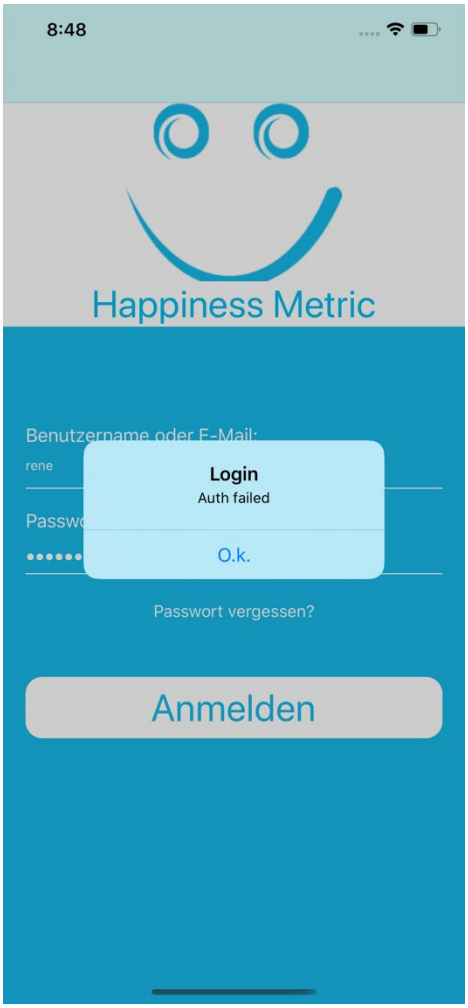


Figure 53 - iOS Login alert

Additionally, if the user, for example, votes about his happiness, he would be pleased to get a feedback about whether it worked or not.

Furthermore, it is useful, to use alerts for risky actions, just like deleting an account or logging out of the application. For these situations, the user should be given two possible alert actions, one to let the user confirm that he really wants to delete his account or log out and one to cancel the action if he tapped the wrong button.

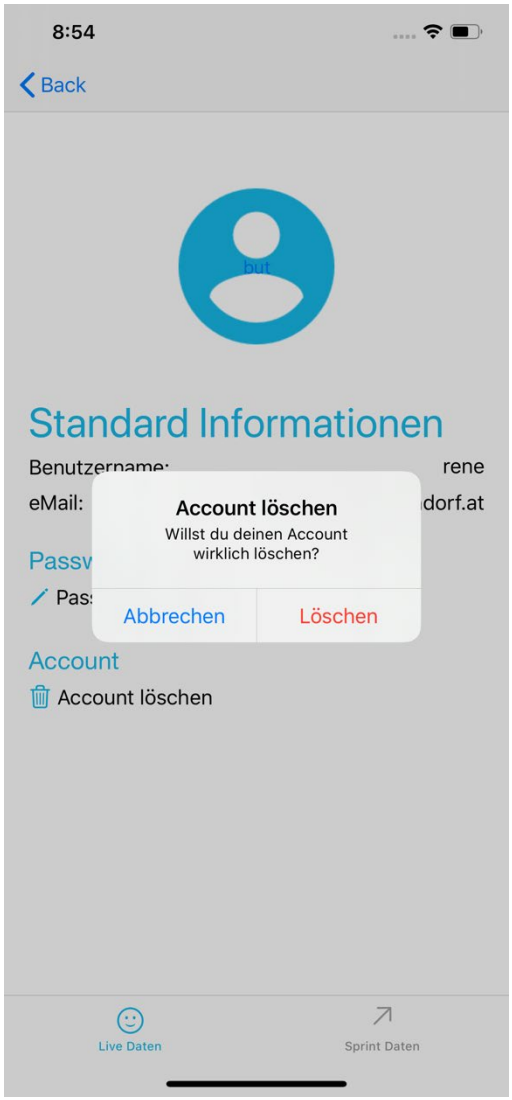


Figure 55 - iOS Delete account alert

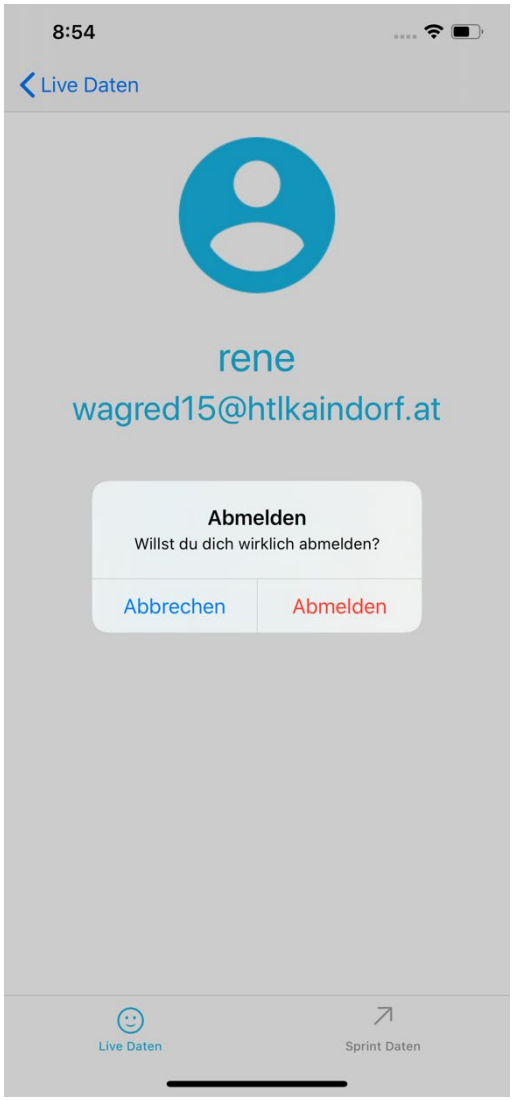


Figure 54 - iOS logout alert

Sprint 08 – Upload profile Picture

For this project, another library was needed to upload or set a profile picture. This library is called Nuke. Just like the other libraries, Nuke had to be installed with Cocoapods. First, the library was added to the podfile, which looked like the following:

```
platform: ios, '10.0'

use_frameworks!

target 'HappMetr' do

      pod 'Charts',

      pod 'Locksmith',

      pod 'Nuke'

end;
```

Then `pod install` was typed into the command line, to install the podfile with the libraries into the application.

After installing the libraries, the Nuke library could be used simply with:

```
import Nuke
```

in the xCode project.

After installing the library, the API-request checks whether a profile picture has been uploaded or not. If none has been uploaded, the application uses a default image as the profile picture, but if one has been uploaded, it uses the uploaded image. Therefore, Nuke has to be used in the following way:

```
Nuke.loadImage(with: URL(fileURLWithPath: profilePictureLink ?? ""), into:
self.imageView)
```

This code loads an image, which is saved on the server and reachable with a link into the UIImageView.

If a user does not want the default image or his uploaded image anymore, he can easily change it, by clicking on the profile image on the "Account bearbeiten" screen.

Figure 56 – Change Profile Image

If the button is clicked, the uploadImage function is called. This function sets the source type of the image source and calls the image picker. Through the image picker users are able to choose an image from their library. The chosen picture will be uploaded to the server by using the API call with NetworkHandler.shared and will be set as their profile picture.

```
@IBAction func uploadImage(_ sender: Any) {

        imagePicker.allowsEditing = false
        imagePicker.sourceType = .photoLibrary

        present(imagePicker, animated: true, completion: nil)

}

//open the gallery
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

        if let pickedImage = info[UIImagePickerController.InfoKey.originalImage] as?
UIImage {

            imageView.contentMode = .scaleAspectFit
            imageView.image = pickedImage

           //API call
            NetworkHandler.shared.uploadProfilePicture(image: pickedImage) { (success,
profilePictureLink) in

                if(success == true){

                    print("profilePictureUpload --SUCCESSFUL--")

                } else {

                    print("profilePictureUpload --FAILED--")

                }

            }

        }

        dismiss(animated: true, completion: nil)

}



//close the gallery
func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {

        dismiss(animated: true, completion: nil)

}
```

Table of figures

References

Douglas, W. (2014). BodyParser NPM
(NPM) Retrieved 07.11.2019 from
https://www.npmjs.com/package/body-parser

Express (2017). ExpressJS
(StrongLoop, IBM) Retrieved 07.11.2019 from
http://expressjs.com/

Oracle (1993). Java Documentation
(Oracle) Retrieved 03.11.2019 from
https://docs.oracle.com/en/java/javase/13/docs/api/index.html

SEUME (2019). JSON Wikipedia
(Wikipedia) Retrieved 06.03.2020 from
https://de.wikipedia.org/wiki/Extensible_Markup_Language

MongoDB (2020). MongoDB
(mongoDB) Retrieved 02.02.2020 from
https://www.mongodb.com/de

Jahoda, P. (2020). MPAndroidChart
(GitHub) Retrieved 07.01.2020 from
https://github.com/PhilJay/MPAndroidChart

Sharp, R. (2020) Nodemon
(nodemon) Retrieved 07.01.2020 from
https://nodemon.io/

Pmatulla (2020). XML
(Wikipedia) Retrieved 01.12.2019 from
https://de.wikipedia.org/wiki/JavaScript_Object_Notation