Smart Mirror

Project Team

- Patricia Pichler 11850893
- Jakob Zethofer 01612980
- Lukas Wais 11816105
- Omar Luis Dueñas Hidalgo 51849509

Description

We implemented a really smart mirror, with two different AIs. He can interact with the user and can tell him jokes. He recognizes if someone is standing in front and activates himself. We implemented only the software (GUI) of the mirror. For "real" a mirror, you would need additional hardware, like a display and for example a Raspberry Pi.

This **Documentation** is splits up into three different sections:

1. Manual, the user's handbook.
2. Smart Mirror Journey, how we made it.
3. Last but not least: Additional Reading, there are some resources to get you started.

Moreover, you can find the JavaDoc of all the public methods right here.

The Link to **Github repository** is here.

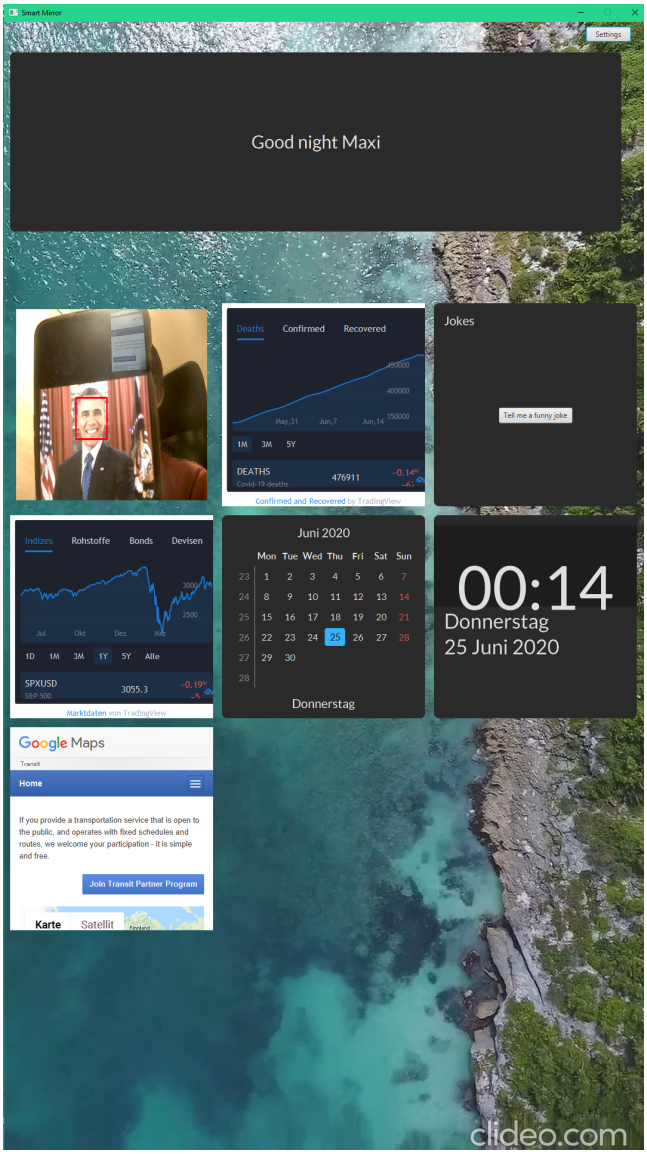
Manual

Key Features

- Personalisation with different user profiles
- Interactive live data
- Public transport
- Two different AIs
- Tells the best jokes 😉
- Relaxing background video

User interface

You will this screen if you boot up the mirror. The default user Max is loaded. He has every widget enabled by default. The user interface consists of a greeting, facial recognition, and other customizable widgets. Greetings and facial recognition are always enabled.



Greetings

The greeting consists of the user's nick name and, depending on the time of day, 'Good morning', 'Good day', 'Good afternoon', 'Good evening' and 'Good night'. The user is greeted by the voice of Amazon Polly.

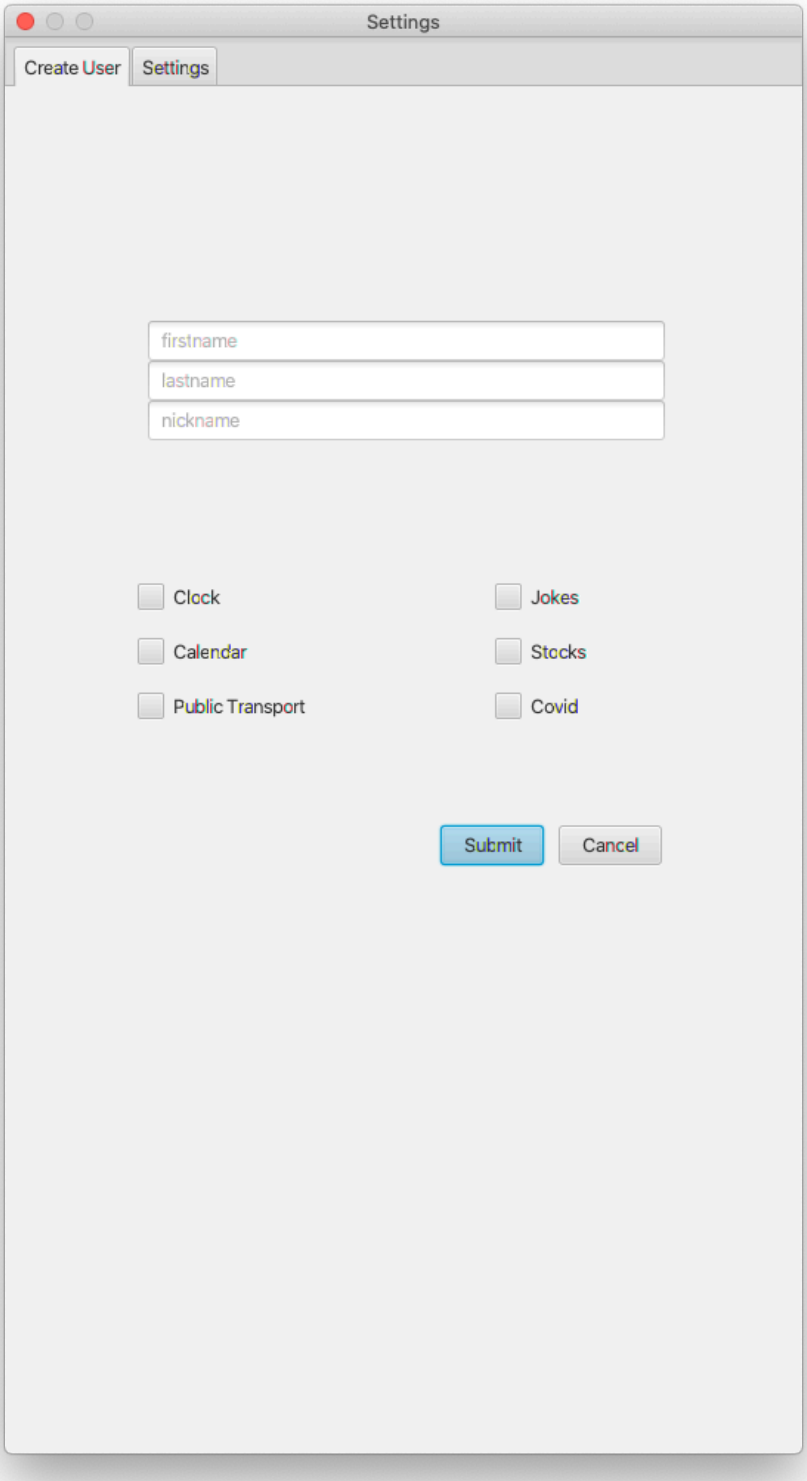
Face recognition

As soon as a person is recognized in front of the mirror all the configured widgets are loaded again. The widgets are hidden again 5 seconds after no one is recognized in front of the mirror.

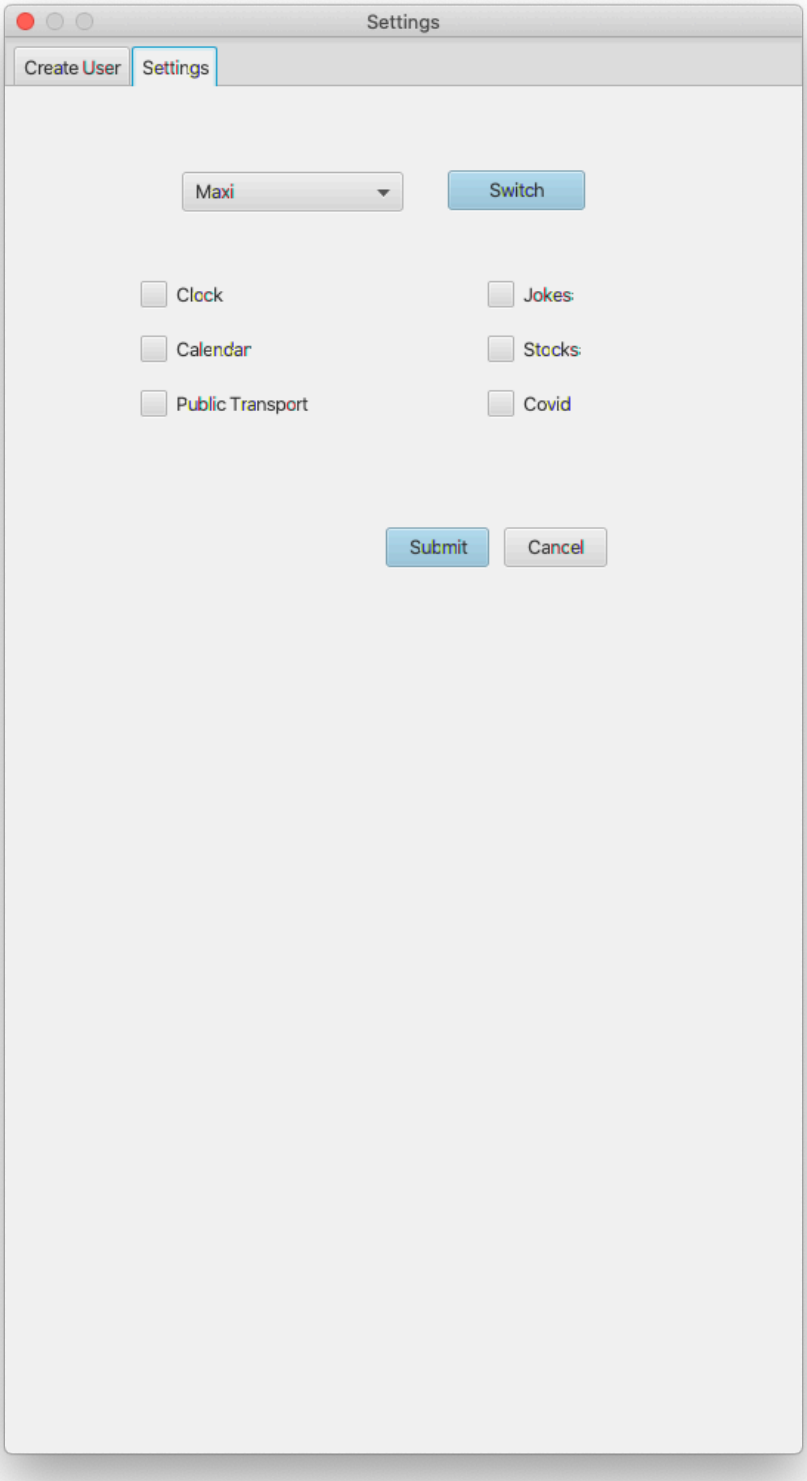
Settings

You will get there by pressing on the `Settings` button in the right-hand corner. Here you can create new users or switch between existing users.

Under the `Create User` tab you can create a new user. You must set the first name, the last name, and a nickname. Moreover, you can choose which widgets you want to have. If you are done press on the `submit` button.



In the settings pane, you can change between user-profiles and update your widgets. You can see every available user with the dropdown menu. They are displayed by their nicknames.


Settings
Create User
Settings
Maxi
Switch
Clock
Calendar
Public Transport
Jokes
Stocks
Covid
Submit
Cancel

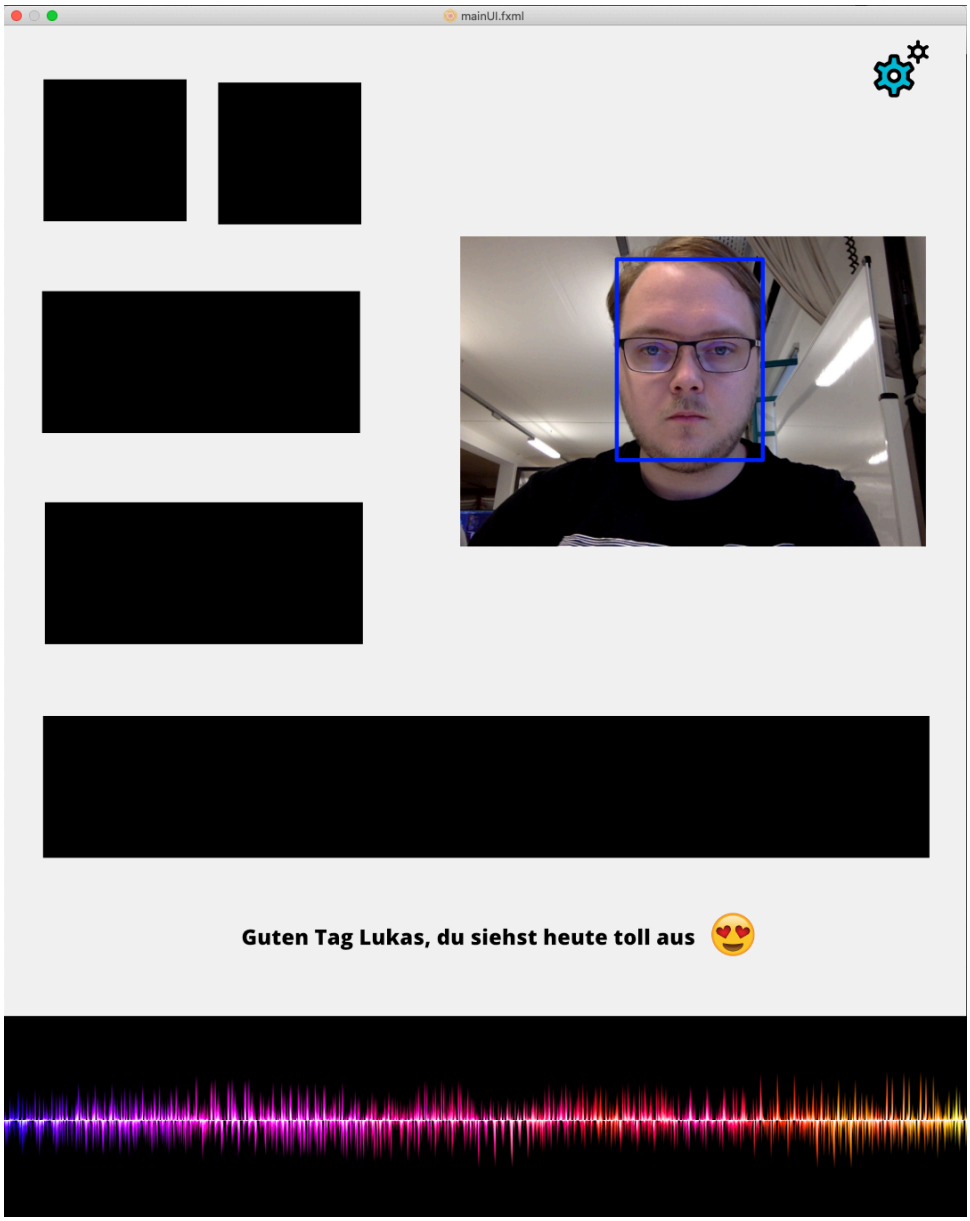
Available widgets

- Clock
- Jokes, tells you funny jokes and tongue twisters.
- Calendar
- Stocks, most important stocks are shown here.
- Public Transport, just type in where you want to go.
- Covid, live statistics of the current Sars-Cov2 pandemic. Shows world wide data.

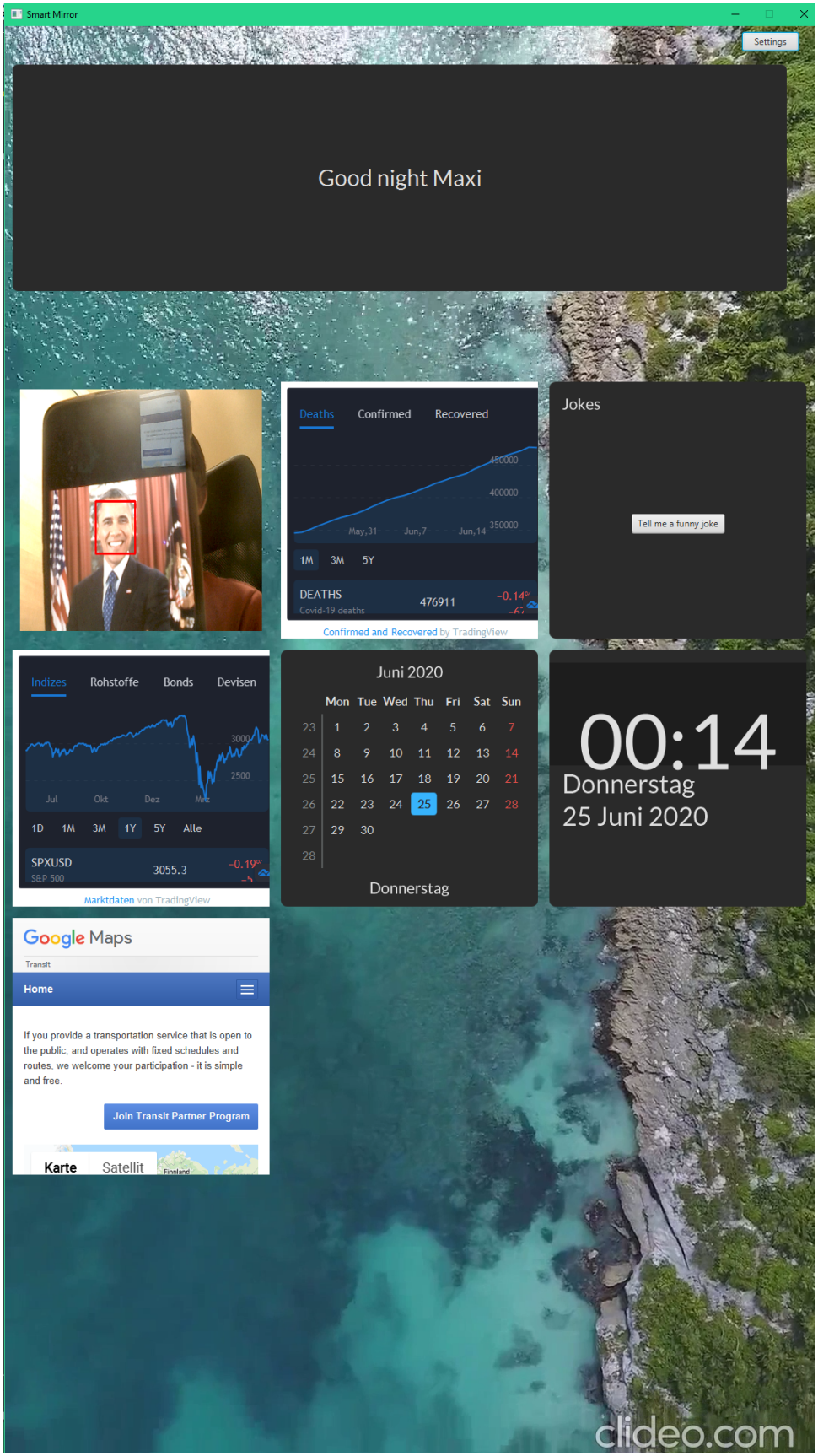
Smart Mirror Journey

This section is about the actual programming of the mirror.

How we came from this:



to this:


Smart Mirror
Settings
Good night Maxi
Deaths
Confirmed
Recovered
480000
400000
350000
May,31
Jun,7
Jun,14
1M
3M
5Y
DEATHS
476911
-0.14%
Covid-19 deaths
Confirmed and Recovered by TradingView
Jokes
Tell me a funny joke
Indizes
Rohstoffe
Bonds
Devisen
3000
2500
Jul
Okt
Dez
1D
1M
3M
1Y
5Y
Alle
SPXUSD
S&P 500
3055.3
-0.19%
Marktdaten von TradingView
Juni 2020
Mon Tue Wed Thu Fri Sat Sun
23 1 2 3 4 5 6 7
24 8 9 10 11 12 13 14
25 15 16 17 18 19 20 21
26 22 23 24 25 26 27 28
27 29 30
28
Donnerstag
00:14
Donnerstag
25 Juni 2020
Google Maps
Transit
Home
If you provide a transportation service that is open to the public, and operates with fixed schedules and routes, we welcome your participation - it is simple and free.
Join Transit Partner Program
Karte
Satellit
Finnland
clideo.com

Technology Stack

Since you are browsing Github and have found our project you must be at least a little bit nerdy. We want to get off with the most interesting first: Which technologies did we use to develop this application?

Languages

- **Java 14** Nearly everything is written in pure Java.
- **HTML** This part does come from some widgets. Those are basically small websites since they can easily get updated and look very well.
- **CSS** Those very small part does come from JavaFX. We did some minor adjustments for user errors at the input.

We have two different kinds of technologies. On one hand, we have software systems, those are actual libraries we have used for this project. On the other hand, we have the way of working. Sounds fancy right? But what does this actually mean? Those tools and technologies made our development pipeline easier and more comfortable.

Software Systems

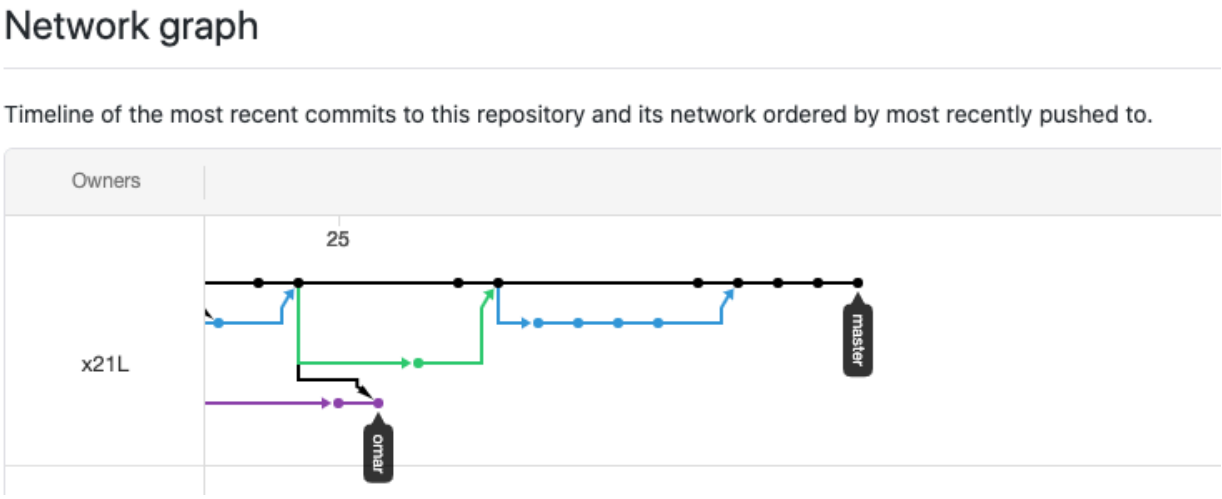
The is to just take a look at out our Maven dependencies. The in my opinion coolest ones are listed below, with additional links do get you started. If you are more interested:

- TilesFX
- H2 Databse
- Amazon Polly

Way of working

To support our development we have to use some professional tools. As you might have guessed we are using Git and Github as a version controls system. It did cost us a lot of nerves at the beginning but in the end, the struggles were worthwhile. We used four branches, one for the GUI, one was the database, one the face recognition, and of course one master.

Here is an excerpt of our network of branches.

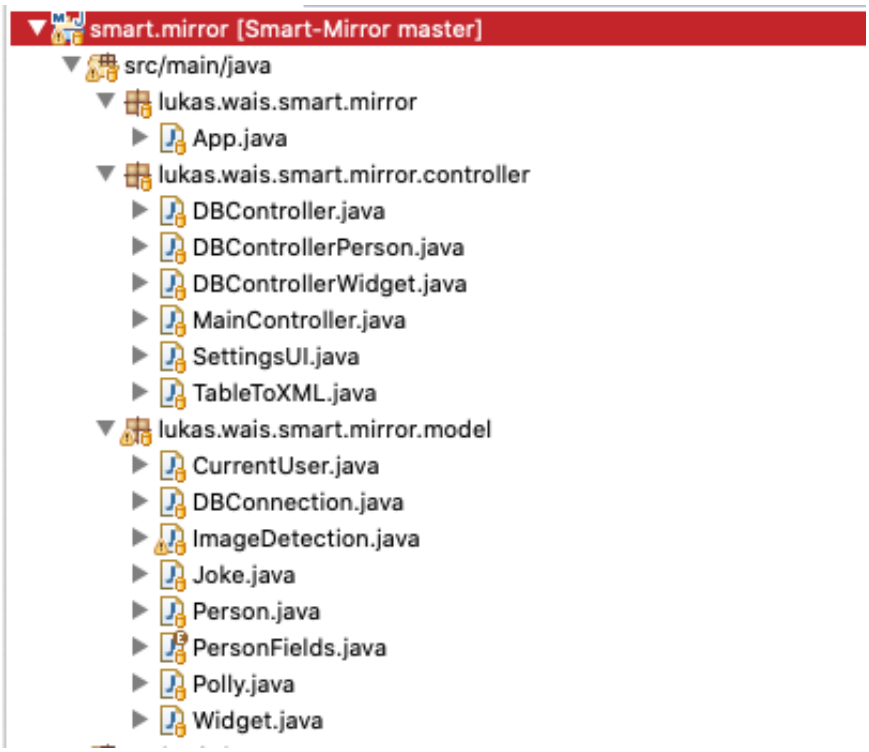


As our integrated development environment, we have used Eclipse. As a build system and for dependency management we went with Apache Maven.

Last but not least some project management. It is a must for bigger projects with multiple people. We made a Kanban board on Github to keep track of the current tasks and errors.

Since we do not have any clients paying or setting up very special requirements which need a lot of planning we used Extreme Programming as software methodology.

We did not forget about the architecture of our project. We have opted on the Model-view-controller MVC pattern. Just take a look at our project structure:



TL;DR

- Github
- Eclipse
- Kanban
- Extreme Programming
- MVC

How we made it

This part will be split up like our branches :wink:

GUI

JavaFX

We are going to start with the user interface. As already mentioned we used openJavaFX 14 to have a good looking GUI. Since we have also used the Scene Builder we are working with FXML files. Which defines our view components, like buttons, and so on. For every FXML file, we have our own controller. The controller takes care of the changes and also initializes the different components.

> **NOTE:** The FXML files are stored in the resources directory. In case you have wondered earlier that we do not have a view package since we said we were using MVC. The view directory is in the resources directory since they are not Java files and Maven looks for them in this directory.

We have two different views and therefore two controllers and two FXML files.

1. `MainUI.fxml` and `MainController.java`
2. `CreateUserUI.fxml` and `SettingsUI.java`

We defined everything possible in pure fxml. For example the `onAction` of the buttons. So you just have simple methods in your class like this one:

```
@FXML
void cancel() {
    Stage stage = (Stage) pane.getScene().getWindow();
    stage.close();
    openMain();
}
```

The @FXML annotation is very important, it basically toggles the JavaFX functionality of the method.

Widgets

The widgets are represented in tile panes, they adjust dynamically with a fixed `width = 340` and `height = 340`. We have an own widget factory class which is generating the nodes for the tile pane.

> You add nodes to the tilePane. This works as easy as this:
> `tilePane.getChildren().add(node);`

As mentioned earlier we have two different kinds of widgets HTML and tilesFX. Two examples of both kinds are the following.

```
public Node getCalendar() {
    return TileBuilder.create().skinType(SkinType.CALENDAR).build();
}
```

TilesFX

```
public Node getMarkets() {
    return htmlToNode("../html/markets.html", 340, 340);
}
```

HTML

As you can see we have a small helper function which transforms local stored HTML files into Nodes with the help of the JavaFX Webview. Just take a look:

```
private Node htmlToNode(String path, double width, double height) {
    URL url = this.getClass().getResource(path);
    WebView webView = new WebView();
    webView.getEngine().load(url.toString());
    webView.setPrefSize(width, height);
    return webView;
}
```

Polly

As you may have recognized since you have already read the manual :wink:, the mirror tells you only the best jokes and tongue twisters. We implemented an AI called Polly to speak with us. Unfortunately, we did not have enough time to develop and train our own AI we took an existing one. Polly is made Amazon. You can check it out here.

You may be asking how does this work? Inside our program, we just call the `static` method `speak` from the self-implemented Polly class.

Just like this: `Polly.speak("Hello I am Polly");` .

We take a closer look at the Polly class now. This is the constructor, it loads AWS credentials, Amazon Webservice is of course not free to use. Secondly, it sets the region of the service (this is the server location of Polly)

```
private final AmazonPollyClient pollyClient;
private static Polly instance;

public Polly(Region region) {
    // create an Amazon Polly client in a specific region
    pollyClient =
    new AmazonPollyClient(new DefaultAWSCredentialsProviderChain(),
     new ClientConfiguration());

    pollyClient.setRegion(region);
}
```

Polly is implemented as Singleton to ensure that only one instance is active. This is used for multithreading. The image you press the button twice or the MainUI is running with the video. If you press the button twice Polly will tell you a random joke or tongue twister. If Polly greets you the background video is still running.

To ensure Polly only tells one Joke at the time we used a `ThreadPoolExecutor`, take a look:
It is `SingleThreadExecutor` since we said only one Joke at a time.

```
public static void speak(String text) {
    Executor executor = Executors.newSingleThreadExecutor();
    Thread speakerThread = new Thread (() -> {
        Polly.getInstance().play(text);
    });
    executor.execute(speakerThread);
}
```

User Profiles

You can change between different user profiles, so user can change which widgets they want to use. You will see how those profiles are stored in the database section. Here we are talking about the GUI perspective of this. Supporting different profiles is not so easy, because the whole `MainUI` can change. Therefore we close it and open it again with a new user and his preferred widgets. Polly also greets you with a welcome message, depending on the day time.

For the current user, we used again the very handy Singleton pattern. In the `MainController` we check if the current user is null, this means the mirror was freshly botted up or botted up for the first time. In this case, we load the default user called "Max" from the database. He is inserted by default. All widgets are active by him. He is the only user with just a number as ID, all others have a generated `UUID` from Java, like this one

`295ff6e2-b025-4bd6-bd3e-47b3de9ea4d4` it is 128 Bit long.

```
Person user = CurrentUser.getInstance().getUser();

if (user == null) {
    user = DBControllerPerson.selectPerson("1");
    widgetsUser.addAll(DBControllerWidget.selectWidget("1"));
} else {
    widgetsUser.addAll(DBControllerWidget.selectWidget(user.getID()));
}
```

But how we set a new user? This works with the `choiceBox` in the settings. It toggles the following method setter method in the `CurrentUser` class:

```
public void setUser(Person person) {
    this.person = person;
}
```

By default the `Person` object is null.

This is how `getInstance` methode of a singleton pattern does like:

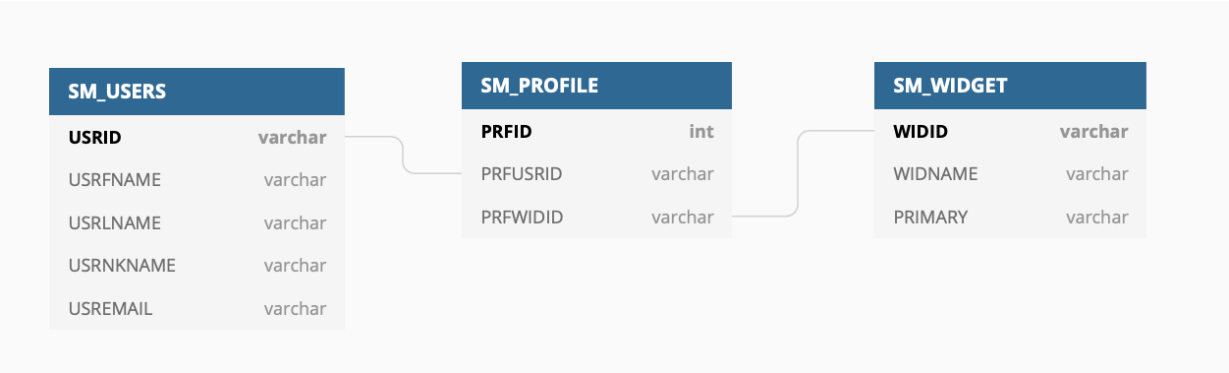
```
public static CurrentUser getInstance() {
    if (instance == null) instance = new CurrentUser();
    return instance;
}
```

It ensures that there is only one instance of an object.

> **NOTE**: Everything in the GUI is as abstract designed and implemented as possible, to be more flexible and scalable. It is a breeze to add new widgets to the factory for example.

Database

Table Structure



The idea with the table structure was to keep it as simple as possible, as dynamic as well. Therefore are the three tables implemented a must. The structure provides big flexibility for the programming part in java as well as for the SQL part managing the database data. Let's do a quick introduction to each table:

```
TABLE SM_USERS (
    USRID VARCHAR(255) NOT NULL,
    USRFNAME VARCHAR(255) NOT NULL,
    USRLNAME VARCHAR(255) NOT NULL,
    USRNKNAME VARCHAR(255) NOT NULL,
    USREMAIL VARCHAR(255),
    PRIMARY KEY (USRID)
);
```

The user table contains all the users needed for the mirror. Nevertheless, it is possible to insert new users at any time with the needed data. As we can see in the code snippet the only field which one is optional is the email, all the other ones have to be filled in order to be able to insert a new user to the database.

```
TABLE SM_WIDGET (
    WIDID VARCHAR(255) NOT NULL AUTO_INCREMENT,
    WIDNAME VARCHAR(255) NOT NULL,
    PRIMARY KEY (WIDID)
);
```

For the widget table, the data have been inserted directly in the H2 console and there is no possibility to manage this table in the GUI at any time now. The reason behind this is, that we actually have a limited widget number in the code and want to avoid users inserted widgets in the database and creating errors because the widget is not actually available in the java code. An important feature to mention is, that the combination "WIDID" and "WIDNAME" has to be unique in order to avoid duplicate insertions.

```
TABLE SM_PROFILE (
    PRFID int NOT NULL AUTO_INCREMENT,
    PRFUSRID VARCHAR(255) NOT NULL,
    PRFWIDID VARCHAR(255) NOT NULL,
    PRIMARY KEY (PRFID),
    FOREIGN KEY (PRFUSRID) REFERENCES SM_USERS(USRID),
    FOREIGN KEY (PRFWIDID) REFERENCES SM_WIDGET(WIDID)
);
```

The link table in our structure is the profile table. With this table, we are able to link the table SM*USERS and SM*WIDGET. There is a 1…n relation between users and widgets, which means that a user could have multiple widgets. For these propose the table profile has been implemented. Here we stored which users contain which widgets. Moreover, there is the possibility to change every profile in the GUI, so a user is able to choose at any time, which widgets should be visible for him.

Relation Model-Controller

For our project, we decided to implement the MVC concept, in order to be able to separate the part of the mirror efficiently. The concept makes it easier, that multiple persons work on the same project, even the same function, but on a different level. On the model side, the connection to the database is established. On the controller side, we use the connection to execute certain statements. For each query, which should be executed in the database, there will be a statement on the controller side. In the controller classes, the statements are stored in constant variables and as prepared statements executed in order to prevent SQL injection.

Database Connection

As mentioned bevor, for this project the H2 emended database is being used. The database connection is implemented as a Singleton pattern. So we make sure that there is no possibility to start a second/parallel connection. For the database connection are the URL, user, and password required. This data is hardcoded in the Model class DBConnection.

SQL queries example

The queries are implemented as constant in java in order to prevent SQL injection. The ? are the place holders for the input parameters.

```
private static final String INSERTPROFILE = "INSERT INTO SM_PROFILE
    (PRFUSRID, PRFWIDID) SELECT (SELECT USRID
    FROM SM_USERS WHERE  USRID = ?) as PRFUSRID,
    (SELECT WIDID FROM SM_WIDGET WHERE WIDNAME  =?)
    as PRFWIDID”;
```

With this query, a new profile can be inserted. The place holders are in this case the user ID, which describes the specific user, and the widget name to be assigned.

```
private static final String SELECTALL =
    "SELECT WIDNAME FROM SM_WIDGET WHERE WIDID
    IN ( SELECT PRFWIDID   FROM SM_PROFILE
    WHERE  PRFUSRID =?)”;
```

Here we select all the widgets names for a specific user in the database.

```
private static final String DELETEPROFILE
    = "DELETE FROM SM_PROFILE WHERE PRFUSRID = ?";
```

When a user wants to make some changes for his profile, the existing profile has to be deleted first due to the given table structure.

Face Recognition

The face recognition is used to see, if a user is currently standing in front of the mirror. The primary camera of the computer is used, to get an image every 250 milliseconds. Then faces are detected in the current image and surrounded with a rectangle. This image is displayed in an `ImageView` on the smart mirror:

```
if (now-lastUpdate > 250) {
    lastUpdate = now;
    view.setImage(ImageDetection.getCaptureWithFaceDetection());
}
```

For the detection the `opencv` framework is used. The detection is done in the class `ImageDetection`. The class is initialized with a `CascadeClassifier`, that is loaded from an XML-file that is stored as a resource. The file is a pretrained model that can be downloaded from the opencv Github repository.

The method `getCaptureWithFaceDetection()` captures an image from the camera and stores it in a matrix. This matrix is passed to the method `detectFace(Mat inputImage)`, that uses the classifier and gets an array of rectangles, that contain the detected faces. If the array lenght is at least one, this means that a face has been detected and the corresponding variables are updated:

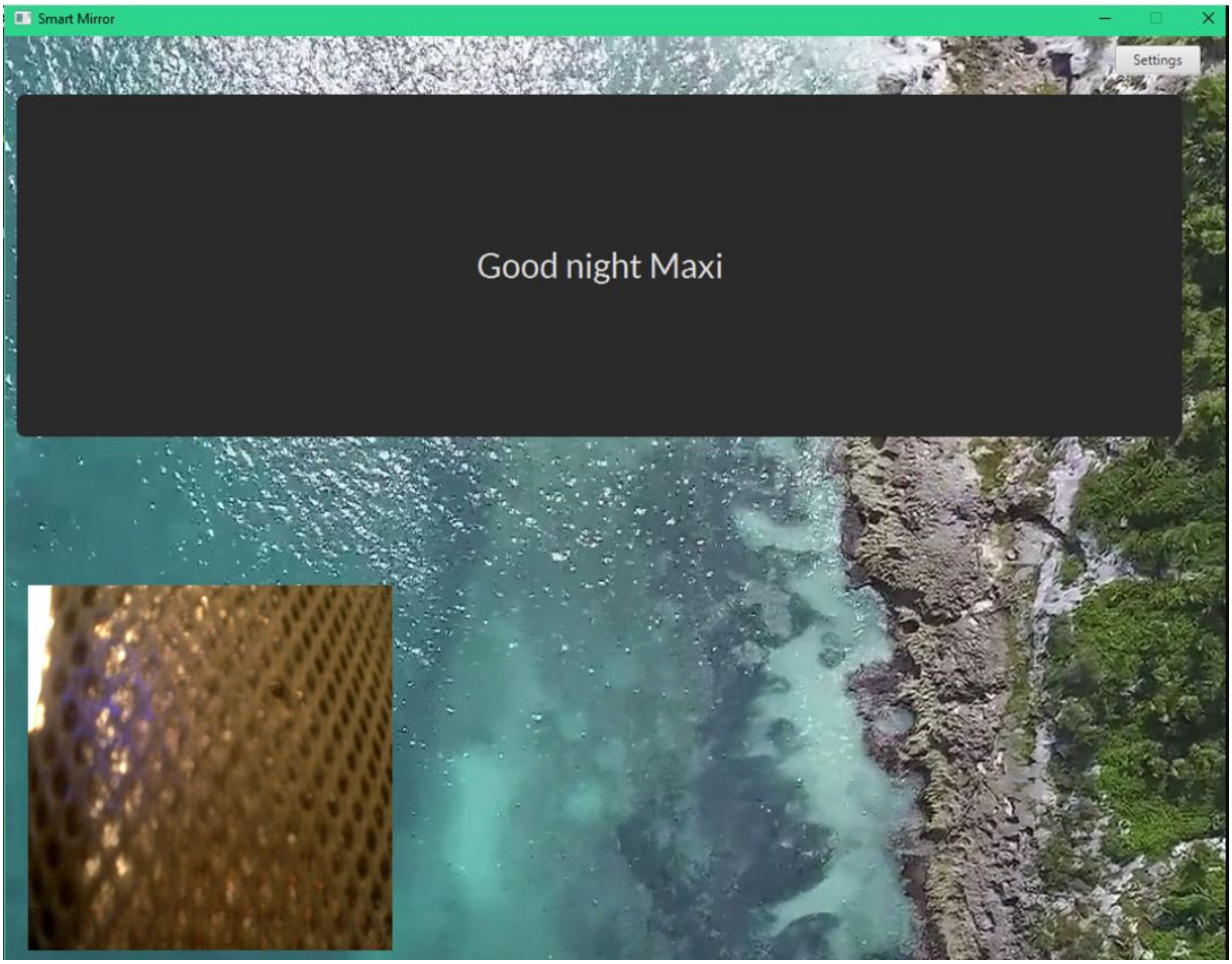
```
if(facesArray.length > 0) {
    detected = true;
    lastDetected = DateTime.now();
}
```

The rectangles are inserted into the image matrix and the matrix is converted into an `Image` that can be displayed by the `ImageView`.

In the `MainController` an `AnimationTimer` handles the updates of the `ImageView`. When the mirror was active (person was detected before) and now there hasn't been a person detected in the last five seconds, all widgets are removed and only the

camera image is shown:

```
else if (!ImageDetection.detected && mirrorActive) {
        tilePane.getChildren().clear();
        tilePane.getChildren().add(view);
        mirrorActive = false;
    }
```



When there is a person detected after the mirror was inactive, the widges are added again:

```
else if (!ImageDetection.detected && mirrorActive) {
    tilePane.getChildren().clear();
    tilePane.getChildren().add(view);
    mirrorActive = false;
}
```

Additional Reading

- TilesFX
- H2 Databse
- Amazon Polly
- The OpenCV Github repository
- Our Github repository
- Our JavaDoc