

---

# **WT1 Web Applications**

## **Projekt: WeekMe**

---

HTW BERLIN

INTERNATIONALE MEDIENINFORMATIK

JAKUB MUELLER / DAVID SCHMOECKER / JONAS HEINIG  
544832 / 544655 / 550169  
WS18/19

---

## Contents

<b>1</b>	<b>Concept</b>	<b>1</b>
1.1	First Idea . . . . .	1
1.2	First Draft . . . . .	2
<b>2</b>	<b>Technical Documentation</b>	<b>3</b>
2.1	Architecture . . . . .	3
2.2	Backend . . . . .	3
2.3	Frontend . . . . .	10
<b>3</b>	<b>User Documentation</b>	<b>12</b>
3.1	Account . . . . .	12
3.2	Working with tasks . . . . .	12
	<b>List of Figures</b>	<b>13</b>
	<b>List of Tables</b>	<b>13</b>
	<b>References</b>	<b>14</b>

---

# 1 Concept

## 1.1 First Idea

The first idea about this application came to us because our team member Jakub is using an online note taking tool (in this case apple notes) to keep track of his weekly schedule. Using this note tool to organize the schedule requires more work than necessary. Furthermore it can become a little bit unclear to navigate around the different days and tasks for the week. As we conducted research we were not able to find any application that supports the functionality we would like, meaning an application that has functionality that could be described as some kind of mix between a calendar app and a simple todo list. Our goal is to create an application that lets you plan your task for the next 7 days but not any further. In order to have the schedule available at all times and all devices we decided to create a responsive web application that can run on any device with a modern browser, including desktop computers, smartphones and tablets. User registration and authentication permits for anyone to use this app for his personal schedule planning. The image below shows an example of how our team member Jakub used his note taking app to create his weekly schedule. There is a section for each day of the week as well as another section “stack” for unassigned tasks.

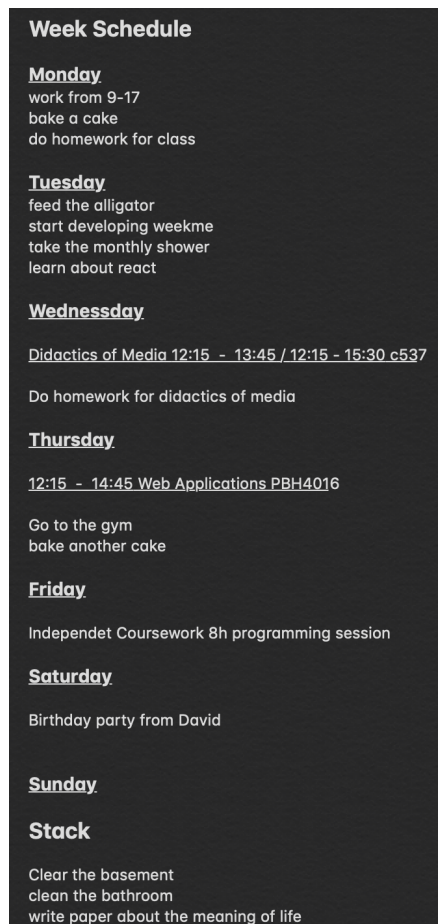


Figure 1: Schedule inside Note Taking App

## 1.2 First Draft

The image below demonstrates our first draft of our WeekMe application. The days Monday to Friday each get their own separate space as well as the stack where unassigned tasks are collected. Each task is represented by a rectangle. Rectangles can be edited, moved around to another day or to the stack or deleted. For our first prototype we decided to use a framework called Gridstack.js. This framework provides functionality for creating flexible and automatically resizable grids that adapt to its contents. The different task within the grid can be moved around by using drag&drop. To make sure user always are aware which day of the week we currently have we decided to highlight the current day in a different color. In order to not lose any unfinished tasks we decided to move them back to the stack once their due day has passed. Our top priority was to create a simple and clear user interface and include only the very least functionality that is needed to plan ahead one week and not more.

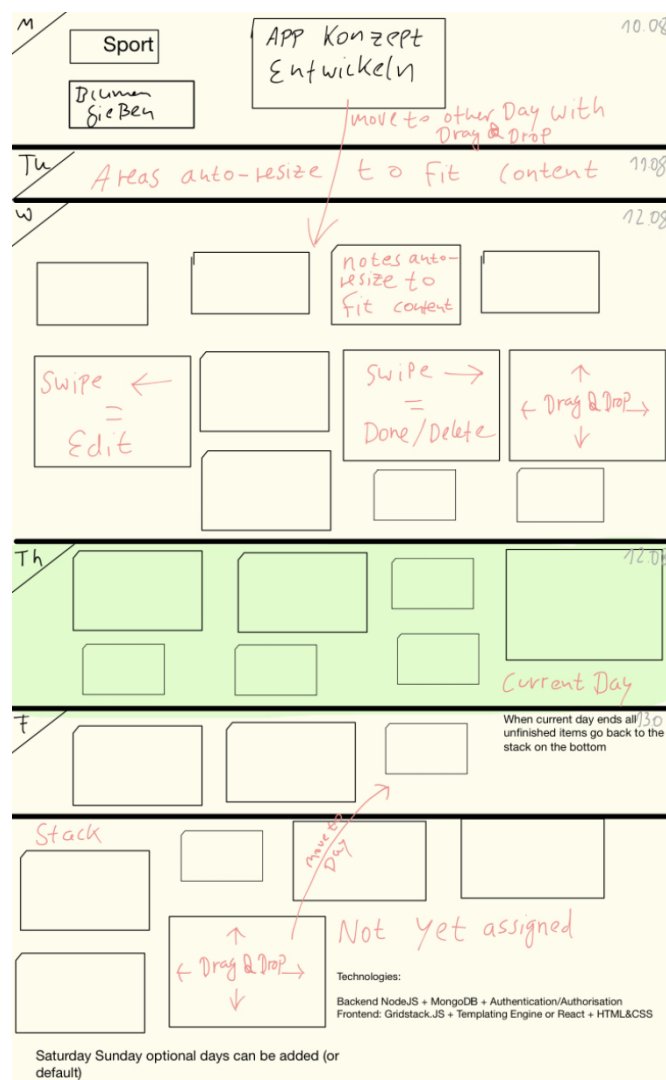


Figure 2: First Draft

---

## 2 Technical Documentation

### 2.1 Architecture

The image below gives an overview about the technologies that we have used in the development of the WeekMe application and where these are used within our overall system.

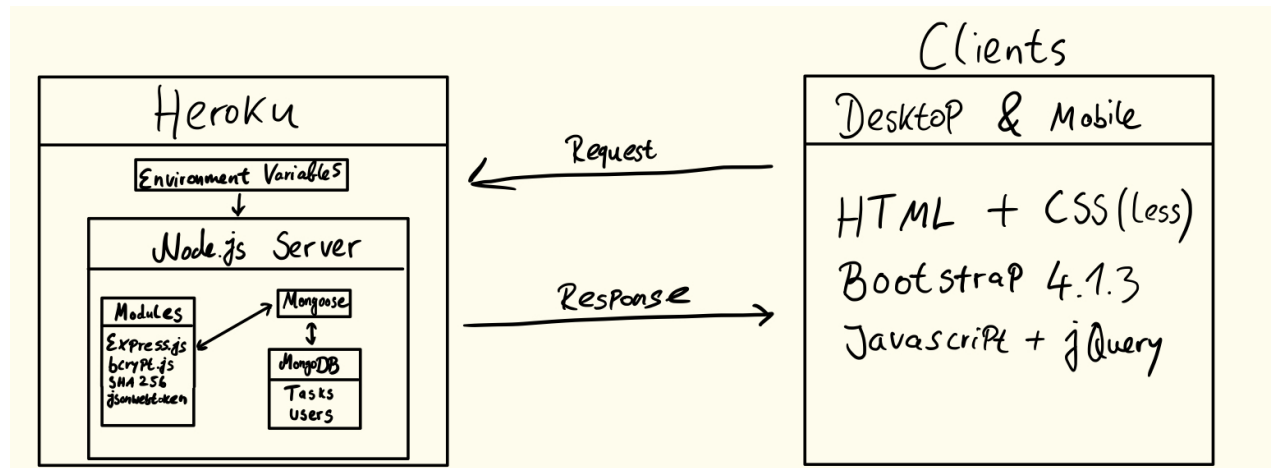


Figure 3: System Architecture

### 2.2 Backend

#### 2.2.1 Node.js

The backbone of our backend is a node.js server. As node natively supports all the functionality we really need for our server we could use it by itself. Instead we chose to extend nodes functionality using express.js, as it generally makes development slightly faster.

##### 2.2.1.1 Modules

**Express.js** is one of the most popular modules available on npm. It is a server side web framework that extends the functionality of Node.js with many handy tools making development faster and easier. An express application consists of at least one file usually called app.js. This file registers any middleware and controllers the application is using. Furthermore the web-server is started here. A basic express web-server can be started in the following way:

```
1 const express = require("express");
2 var app = express();
3 app.listen(3000, () => {
4   console.log("Server is up on port: ", 3000);
5 });
```

**Routing** Routing is used to determine how our application responds to client requests at a particular endpoint. Typical client requests HTTP methods are GET, POST, PUT and DELETE. The

---

initial setup of our routing is done using an instance of our express app. We can add routes and their handlers to our app in the following form:

```
1 app.METHOD(PATH, HANDLER);
```

Where app is an instance of express, METHOD is an HTTP request method, PATH is a path on the server and HANDLER is the function executed when the route is matched. The simple examples below demonstrate a basic routing for each HTTP method type. In all of these cases our handler simply returns a string value to the client.

```
1 app.get('/', function (req, res) {
2   res.send('Hello World!')
3 })
4
5 app.post('/', function (req, res) {
6   res.send('Got a POST request')
7 })
8
9 app.put('/user', function (req, res) {
10  res.send('Got a PUT request at /user')
11 })
12
13 app.delete('/user', function (req, res) {
14   res.send('Got a DELETE request at /user')
15 })
```

**Serving static files** In order to be able to serve static files our WeekMe application needs to be configured properly. The most common scenario in which we need to serve public files is when we want to include css, html or other resource files. We use an integrated express middleware function called "express.static" (as explained in the next section) to let our app know where to find our public directory containing the static files. Having done this clients are able to load files from the public directory.

```
1 app.use(express.static('public'));
```

**Express Middleware** lets us run certain code before or after certain events. Express middleware adds on to the express functionality. Middleware functions have access to the request and response object as well as the next middleware function in the execution cycle. Common examples of things express middleware can do are:

- Execute Code
- Change to request or response objects
- Determine whether or not somebody is logged in and may access a page
- respond to a request

---

The following middleware function is used inside the application to determine whether the user that is accessing the route is logged in.

```
1 var authenticate = (req, res, next) => {
2   var token = req.header("x-auth");
3
4   User.findByToken(token).then((user) => {
5     if(!user){
6       return Promise.reject();
7     }
8     req.user = user;
9     req.token = token;
10    next();
11  }).catch((e) => {
12    res.status(401).send();
13  });
14 };
```

After having defined our middleware function, we need to append it to one of our routes so that it will be executed when this route is being accessed.

```
1 //Tell route to use middleware authenticate
2 app.get("/users/me", authenticate, (req, res) => {
3   //because we are using the middleware, we now have all the data we
4   //appended inside that middleware available here inside the req object
5   res.send(req.user);
6 });
```

**Bcrypt** is a module that can be used to hash passwords. Hashing passwords is considered a mandatory operation before storing them to a database in any secure and professional environment, as it prevents attackers that might get access to the dataset from reading and using any passwords that are stored. Hashing is a one way street and always returns the exact same hash for the same string. When hashing a password a salt is used to increase security even further. Salting password hashes is also considered mandatory. Otherwise we will always receive the same exact hash when using the same exact password. This would enable people to use a lookup table that maps thousands of common password phrases or english words to their hashes. Ultimately they would be able to get the password from the hash. That's why an unknown salt is needed. Often a randomly generated value is used for the salting process. This ensures that will never receive the same value twice and prevents users from being able to trick us and change data by generating their own hashes.

```
1 // Hashing a password
2 bcrypt.hash('my password', 'my salt', (err, hash) => {
3   // Store hash password in DB
4 });
5
6 //Checking a password
7 bcrypt.compare('my password', hash, (err, res) => {
8   // res == true or res == false
9 });
```

---

**JsonWebToken** is yet another npm module that helps us to setup a secure environment for our application. Its main use case is generating unique tokens that can be used for user authentication. Once generated and assigned to a specific user, the tokens can be used to access private routes that could otherwise not be accessed. This way not anyone can make a HTTP request to our server and we make sure that only verified and authenticated requests are being made. In fact, most routes inside WeekMe will be private and only the login and signup routes will be public. In the code snippet below we use jsonwebtoken to create a token. When the token is decoded using jsonwebtoken's verify method, we need to pass in the exact same salt that we used when creating the token, otherwise we will not be able to access the id and iat (=issuedAt) properties that are stored inside the data object.

```
1  const jwt = require("jsonwebtoken");
2
3  var data = {
4    id: 10
5  };
6
7  var token = jwt.sign(data, "mysaltysalt");
8  console.log(token);
9  var decoded = jwt.verify (token, "mysaltysalt");
10 console.log("Decoded", decoded);  //Decoded --> {id: 10, iat: 1526910769}
```



---

We can visit <https://jwt.io> to get information about the hash we have created using the jsonwebtoken module. Each component of our token stores a different information. The red part is the header. The purple part is the payload storing the actual information (id, iat) and the blue part is the hash that lets us verify that the payload has never been changed.

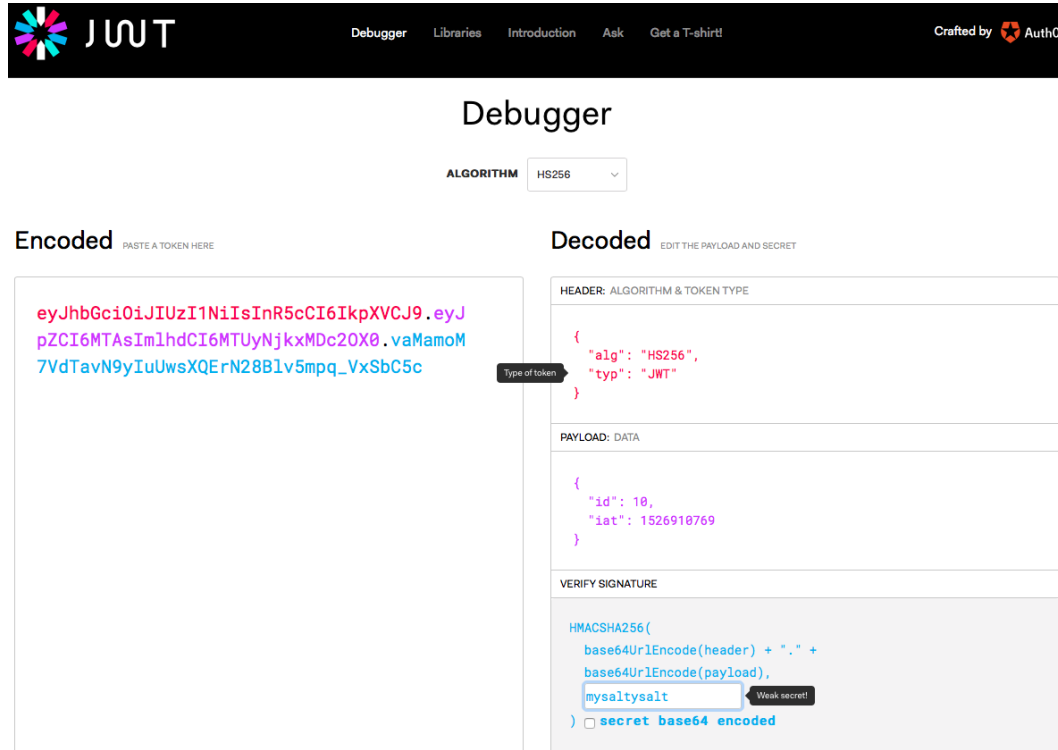


Figure 4: Jsonwebtoken info at: <https://jwt.io>

When a user signs up, a unique token will be created and stored into the database. From now on this token is used for any authentication when accessing private routes instead of the email and password properties that have been provided by the user in the signup process.

```
1 UserSchema.methods.generateAuthToken = function () {
2   var user = this;
3   var access = "auth";
4   var token = jwt.sign({_id: user._id.toHexString(), access}, process.env.
    JWT_SECRET);
5
6   user.tokens = user.tokens.concat([{access, token}]);
7   return user.save().then(() => {
8     return token;
9   });
10 };
```

---

### 2.2.2 Heroku

Heroku is a handy and powerful online platform that enables us to easily deploy and test our applications in the cloud. Once we have setup our account on Heroku.com a series of terminal commands is required in order to get our application up and running on Heroku. The first step is to login to our Heroku account inside of the terminal.

```
$ heroku login
```

Once we are logged in we have to add our public ssh key to Heroku.

```
$ heroku keys:add
```

If we would like to delete a key from Heroku we use the following command while providing the corresponding email address.

```
$ heroku keys:remove example@mail.com
```

We can check for any keys linked to the Heroku account in the following way.

```
$ heroku keys
```

In order to create a new application in Heroku and link it to our Git repository we use the create command.

```
$ heroku create
```

When we want to deploy our application to Heroku we simply need to push it to the heroku branch that has been created in the last step.

```
$ git push heroku
```

Using the heroku open command a browser window with our application will be launched.

```
$ heroku open
```

In order to make sure that heroku is actually launching the correct file we need to edit the package.json file and go to the scripts section. In there we edit the value of the start property to launch the server.

```
1  ...
2  "scripts": {
3    "test": "echo \"Error: no test specified\" && exit 1",
4    "start": "node server.js" // Heroku will execute this line on startup
5  }
6  ...
```

---

### 2.2.3 Making the App Secure

In order to have a truly secure application we need make sure that no sensitive information like API keys or passwords are stored inside the code. For that matter we create a config.js file that will be added to the .gitignore file to exclude it from being pushed to our repository. Sensitive data should not be pushed to the repository because anybody that has access to the repository will be able to see it.

```
1 //config.json
2 {
3   "development": {
4     "PORT": 3000,
5     "MONGODB_URI": "mongodb://localhost:27017/WeekMe",
6     "JWT_SECRET": "asdahjsdajsdbhjg1hjdhj1"
7   }
8 }
```

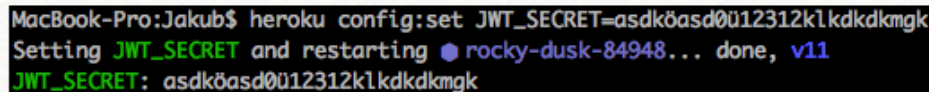
One way to remove sensitive information from our code is to use environment variables. In the code snippet below we set the environment variable depending on whether we are in our development (localhost) or production (Heroku) environment.

```
1 config.js
2 var env = process.env.NODE_ENV || "development";
3 console.log("env ***** ", env);
4 if(env === "development" || env === "test"){
5   var config = require("./config.json"); //When using require on a json file
6   it will automatically be converted to a javascript object.
7   var envConfig = config[env]; //get env property of config
8   Object.keys(envConfig).forEach((key) => {
9     process.env[key] = envConfig[key];
10  });
11 }
```

Once we have assigned all keys to the process.env variable we may access them from anywhere inside our code.

```
1 var token = jwt.sign({_id: user._id.toHexString(), access}, process.env.
  JWT_SECRET);
```

Last but not least we need to make sure that Heroku is also provided with an environment variable storing the JWT\_SECRET value so that the application may function properly in production.



```
MacBook-Pro:Jakub$ heroku config:set JWT_SECRET=asdköasd0ü12312klkdkdkmgk
Setting JWT_SECRET and restarting ● rocky-dusk-84948... done, v11
JWT_SECRET: asdköasd0ü12312klkdkdkmgk
```

Figure 5: Setting an environment variable in Heroku

---

## 2.2.4 MongoDB & Mongoose

All data that is created by WeekMe is stored inside a MongoDB database instance. As MongoDB works very well together with node and due to the fact that our entire business logic is coded in JavaScript we decided to use a JSON style document based database. This allows for easy transformation from the JavaScript objects to JSON and vice versa reducing development time and the amount of code.

## 2.3 Frontend

### 2.3.1 Technologies

#### 2.3.1.1 Gridstack vs. Bootstrap

#### 2.3.1.2 Bootstrap

We To get drag and drop functionality we used bootstrap.js

#### 2.3.1.3 less

The used stylesheets are generated from less code. Leaner Styles Sheets (less) is a backwards-compatible language extension for CSS. It adds features like variables, mixins and functions.

### 2.3.2 Architecture

### 2.3.3 Calling our API

### 2.3.4 Picker Popup

To create a new task or to edit an existing task we decided to create an overlaying popup window with several creating steps. We wanted the creation process to be as easy and fast as possible to reduce the needed afford and enable a quick usage. So we split the process in three steps:

Step one: The popup window opens with a customized headline, a textarea for the task-description, a colored button for the color picker, a button for the day picker if you want to choose another day as pre-selected and a done button to create the save the task immediately. In step one it's required to enter a task-description. The options to choose a color for the task or a day in which the task will be saved are optional. So if the user clicks the done button immediately after he typed a description the task will be saved with the pre-selected color and the pre-selected day.

On the other hand the user can decide to change the color and press the button which is colored in the selected color of the task in the left bottom corner of the popup window. This will trigger step two in which the user can choose the color for the task. The popup window for step one will disappear and the color picker popup will appear. After the user clicks on one of the six color buttons or the back button, step two will disappear and the step one will be shown again with an updated color picker button and the task description you entered before.

To enter step three you have to click on the "another day" button to open the day picker popup. Now you can choose another day or the stack out of a dropdown menu to save the task to. Then you can finish the creation process with a click on done or you can enter step one again if you click back. In this case the day picker will be reset.

---

To realize such a popup window we used the “modal” component from bootstrap. The advantage of using this modal is that it is responsive, so it works and looks good on any device or browser.

At first the html-code of all three steps of the picker-popup is created and will be inserted to the DOM. This is realized in the functions `createModal()` and `insertModal()`. The picker to select the day, the task will be added to, is created and inserted right after the modal creation at the functions `createPicker()` and `insertPicker()`. How the days are arranged at the dropdown menu depends on the current day. The pre-selected day is the day you clicked on to starting the task creating process. The functions `createPicker()` and `insertPicker()` will be executed at other points of the program to reset the picker options and the selected day.

To open the task creation popup you have to call the `showPicker()` function of the `ui_picker.js` document. In order how you make the call the picker will occur in different shapes. The popup style depends on which button you clicked at the surface. If you want to edit a task, create a task at a specific day or creating a task in general. The difference between the shown popups are the headline, color, existing task-description and which buttons are shown. I implemented some shortcut buttons who was planned to be used in the case if the user want to create a task in general with a button not related to a certain day. We didn't implemented such a button for day-independent-task-creation yet. These option was not necessary at this point, but we look forward to implement this as well. Cause these Shortcut buttons are designed to assign the created task with one click to the current day or to the stack, which we expect will be the most used options, so the user don't have to go over the second popup-window providing the day-picker. This will short the time of task creation and improve the user experience. To prove our expectation we look forward to use these already implemented but hidden shortcut buttons to run some A/B-testing.

---

## **3 User Documentation**

### **3.1 Account**

### **3.2 Working with tasks**

#### **3.2.1 Creating tasks**

#### **3.2.2 Editing tasks**

#### **3.2.3 Deleting tasks**

---

## List of Figures

1	Schedule inside Note Taking App . . . . .	1
2	First Draft . . . . .	2
3	System Architecture . . . . .	3
4	Jsonwebtoken info at: <a href="https://jwt.io">https://jwt.io</a> . . . . .	7
5	Setting an environment variable in Heroku . . . . .	9

## List of Tables

---

## References