
WT1 Web Applications

Projekt: WeekMe

HTW BERLIN

INTERNATIONALE MEDININFORMATIK

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

Contents

1	Technical Documentation	1
1.1	Architecture	1
1.2	Backend	1
1.3	Frontend	8
2	User Documentation	8
	List of Figures	8
	List of Tables	8
	References	9

1 Technical Documentation

1.1 Architecture

Folgendes Bild gibt einen groben Überblick über die in der WeekMe Webapplikation verwendeten Technologien und wo diese im Gesamtsystem zum Einsatz kommen.

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.

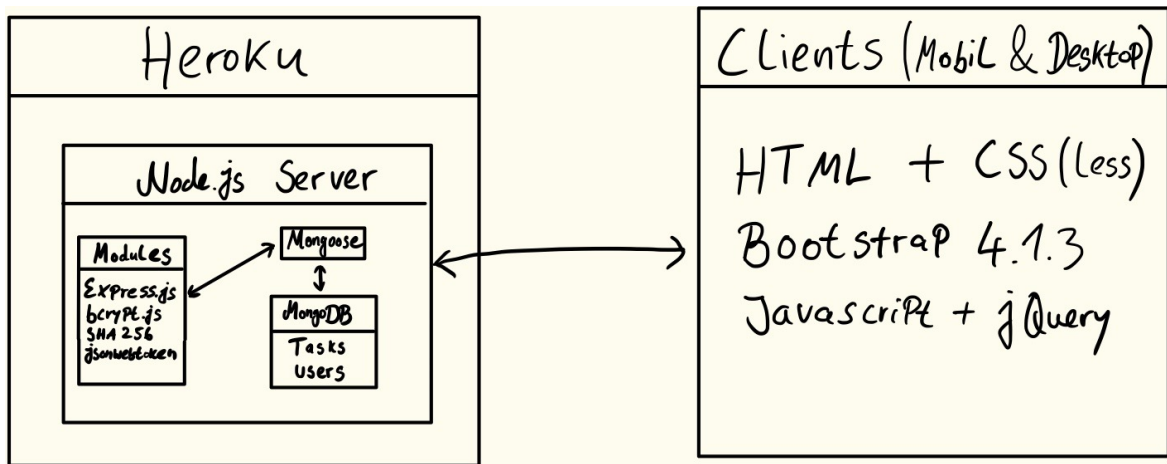


Figure 1: System Architecture

1.2 Backend

1.2.1 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.

1.2.1.1 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.

```
$ 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  ...
```

1.2.3 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": "aÃijaspldapÃij201122309dxkasldk10"
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 }

```

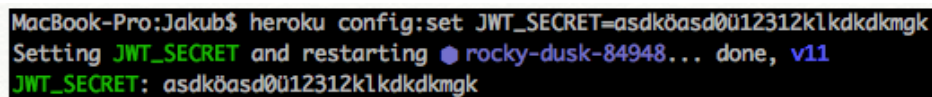
Once we have assigned all keys to the process.env variable we may access them from anywhere inside our code.

```

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=asdköasd0ü12312klkdkmkgk
Setting JWT_SECRET and restarting ● rocky-dusk-84948... done, v11
JWT_SECRET: asdköasd0ü12312klkdkmkgk

```

Figure 3: Setting an environment variable in Heroku

1.2.4 MongoDB & Mongoose

1.3 Frontend

1.3.1 Gridstack vs. Bootstrap

1.3.2 Architecture

1.3.3 Calling our API

2 User Documentation

List of Figures

1	System Architecture	1
2	Jsonwebtoken info at: https://jwt.io	5
3	Setting an environment variable in Heroku	7

List of Tables

References