

Bachelor's thesis

Degree programme of Information Technology, International

PINFOS15

2019

DUY LE

DEVELOPING A USER MANAGEMENT DASHBOARD WITH FULLSTACK JAVASCRIPT



BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme of Information Technology

2019 | 35

Author: Duy Le

DEVELOPING A USER MANAGEMENT DASHBOARD WITH FULLSTACK JAVASCRIPT

The goal of the thesis was to develop a full stack web application using only JavaScript frameworks and libraries, which are known as MERN stack (MongoDB, Express.js, React.js and Node.js). This thesis assesses each technology stack, presents the application implementation, and provides suggestions for future improvement and development. The final result was a functional web-based MVP that allows authentication with JWT and contains simple UI interfaces for optimizing workforce management in small and medium-sized business.

KEYWORDS:

React, Redux, MongoDB, Node.js, Express.js

CONTENTS

LIST OF ABBREVIATIONS (OR) SYMBOLS	4
FIGURES	5
1 INTRODUCTION	6
1.1 Background	6
1.2 Objectives	6
2 APPLICATION TECHNOLOGIES AND CONCEPTS	7
2.1 Frontend	7
2.1.1 React	7
2.1.2 Redux	8
2.2 Backend	10
2.2.1 Node.js	10
2.2.2 NoSQL and MongoDB	11
3 IMPLEMENTATION FUNCTIONALITIES	13
3.1 Functionalities	13
3.2 Getting started	13
3.3 Authentication and Authorization	15
3.3.1 Back-end	15
3.3.2 Front-end	22
3.4 Preference form submission for working days of week	26
3.4.1 Back end	26
3.4.2 Front end	28
4 CONCLUSION	34
REFERENCES	35

LIST OF ABBREVIATIONS (OR) SYMBOLS

CORS	Cross-Origin Resource Sharing
CMS	Content Management System
DOM	Document Object Model
JWT	JSON Web Token
MERN	MongoDB, Express.js, React.js, Node.js
MVC	Model View Controller
React	Synonym to React.js
UI/UX	User Interface/User Experience
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

FIGURES

Figure 1. A simple React component.

Figure 2. Example of state in Redux.

Figure 3. Example of action creator and reducer in Redux.

Figure 4. Example of a simple server in Node.js.

Figure 5. Example of server built in Express.js.

Figure 6. Example of MongoDB URI in mLab.

Figure 7. Server implementation with Node.js, Passport.js and Mongoose.

Figure 8. Main app.js React component with Routing.

Figure 9. Company Model.

Figure 10. Example of a Company document in JSON.

Figure 11. User Model.

Figure 12. Example of User document.

Figure 13. Passport strategies implementation in services/passport.js file.

Figure 14. Action creators and reducer for signing up.

Figure 15. Action creator for signing in.

Figure 16. Action creator for signing out.

Figure 17. Authorization Higher Order Component (HOC).

Figure 18. Preference Form Schema.

Figure 19. Setting up custom calls that triggers middleware.

Figure 20. Basic view for submitting form.

Figure 21. Action creator for submitting preference form.

Figure 22. Admin view for listing all submitted preference forms.

Figure 23. Simple selector to map users' data.

Figure 24. Admin view for evaluating a preference form.

Figure 25. Example of an updated User document after admin's evaluation.

1 INTRODUCTION

1.1 Background

During my previous job in the service industry, the company at the time did not have a good solution to manage shifts and employees could not easily select their desired working timetable at their own paces. All of the work was recorded in papers and this took a considerable amount of time to proceed. Therefore, a solution was needed to facilitate and simplify all of the above nuisances by building a web application and digitalize them. The core idea was to let employees submit their preference form online which the supervisor can later check and validate.

There have not been any thesis works that builds a dashboard from scratch as nearly all of them, which are written in English, are built with CMS like Wordpress or Drupal. Therefore, it was decided to implement such an application.

1.2 Objectives

This thesis will cover the development process of building a full-stack web application which is written solely with Javascript. The tech stacks includes React.js Front-end library, Node.js runtime environment for backend, MongoDB as database and Express web framework for Node.js. Together, they are known as MERN stack. The thesis starts with a brief introduction of each tech stack and then goes to the implementation process in greater details.

The final product will be a workforce optimization application, including dashboard systems for different roles of users, aiming to automates trivial daily tasks of small or medium-sized enterprises. Since the web application is written fully in JavaScript and this thesis will not cover the language's basic concepts, a fundamental understanding of the Javascript and minor experience with full stack web development are recommended.

2 APPLICATION TECHNOLOGIES AND CONCEPTS

The application described in this thesis is developed with React.js in the front-end, with the help of Redux, a state container library, to provide better state management for maintainability and scalability. In the back-end, Node.js plays the main role, along with Express.js framework to minimise the verbosity of Node.js and MongoDB for database storage. This chapter will introduce briefly through each technology and provide general insights before moving to the practical development in the next chapter.

2.1 Frontend

2.1.1 React

React is a declarative, component-based JavaScript library that helps build simple, interactive UIs which can be reused for multiple projects [1]. Comparing to the familiar MVC web architecture, React can be considered as the view in the stack. However, React introduces a totally different way of thinking when developing web application. In React, components are the fundamental element. Each component can store its own state and can “re-render”(or update) itself whenever the state of that component changes [2]. Also, components can be composed together to make a complex component hierarchy. Each follows the single responsibility principle, which results to a more robust and scalable application.[3]



```
function Greeting(props) {  
  return <div>Welcome, {props.name}</div>;  
}  
  
ReactDOM.render(<Greeting name="John Doe" />, document.getElementById("mydiv"));
```

Figure 1. A simple React Component.

The snippet above (Fig.1) displays a simple React component and how to have it rendered in the browser. A div tag with content of plain text “Welcome, John Doe” will appear in a DOM-element with id “mydiv”. The HTML-like element in the return statement

is called JSX and it is a syntax extension to JavaScript. JSX may look like a template language but it comes with the full power of JavaScript [4].

A React component is nothing but a normal JavaScript function, which can take input, or props (stands for properties). Components use props to decide what will appear on the browser.[5] The ReactDOM package comes along when developing React applications. Apart from the main React package which is normally used for producing components, ReactDOM exposes different methods to manipulate DOM element.

2.1.2 Redux

“As the requirements for Javascript SPA have become increasingly complicated, our code must manage more state than ever before. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.”[6] Redux provides a solution those problems, in which state mutations are predictable and can only be changed under specific and monitored circumstances.

Redux has 3 core concepts: state to store necessary data, action to dispatch changes made to state, reducer to actually make the changes.

State in redux is basically a plain object. For example,

```
{
  people: [
    {
      name: "John Doe",
      age: 28
    },
    {
      name: "Jane Doe",
      age: 25
    }
  ]
}
```

Figure 2. Example of state in Redux.

Whenever state needs changing, an action is dispatched and reducer will take the role of making the changes to our state.

```
// action
function addNewPerson(personObj) {
  return {
    type: "ADD_PERSON",
    payload: personObj
  }
}

// reducer
function people(state = [], action) {
  switch(action.type) {
    case "ADD_PERSON":
      return [...state, action.payload];
    default:
      return state;
  }
}
```

Figure 3. Example of action creator and reducer in Redux.

“Enforcing that every change is described as an action lets us have a clear understanding of what’s going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened.” [7]

Finally, the reducer makes changes to the state accordingly to which action has been dispatched. Reducers must be a pure function, that contains no side-effect and the same input(arguments) must provide the same output, regardless of the number of executions [8].

It should be taken into consideration that the reducer will not alter the state directly. A copy of the state is created and a new person is added to this state. This is one of the principle of redux: to avoid directly mutating the previous state. [9]

2.2 Backend

2.2.1 Node.js

Node.js is the JavaScript runtime that allows the JavaScript code to be able to run outside the scope of web browsers. Under the hood, Node.js is based on Google Chrome's V8 JavaScript engine, which is written in C++, translating JavaScript code into machine code. Like other JS engines, V8 has to comply with all the standards from the ECMAScript that specify how the language works and what features it should have. Node.js helps use JavaScript to write command line tools and server-side scripting (running scripts server-side) to produce dynamic web page content before the page is sent to the user's web browser.[10] It is possible to claim that JavaScript has come to its prime from the birth of Node.js.



```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello Node.js World!');
}).listen(8080);
```

Figure 4. Example of a simple server in Node.js.

Node.js comes with multiple built-in modules that makes it powerful. From the code above (Fig.4), the built-in http module is leveraged and creates a server listening on port 8080 of our machine. When a browser makes a request, the server will respond with the text 'Hello Node.js World!'.

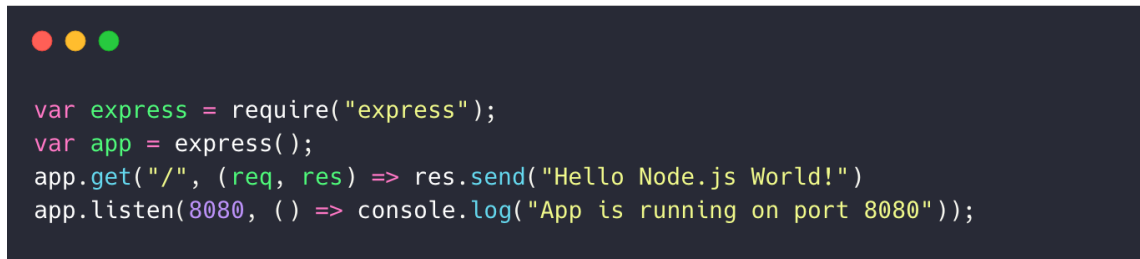
Node package manager (NPM)

Usually in the development process, developers not only make codes on their own but also reuse others' works, which usually comes in packages. Each package is collection of codes managed and maintained by a package management system. "Node Package Manager (npm) is a package manager for Javascript programming language. Npm is included as a recommended feature in Node.js installer [11]." NPM is one of largest JavaScript package manager, which installs and updates JavaScript packages

automatically. There will be many packages, both front-end and back-end wise, to be installed later on during the development of the application.

Express.js

Express.js is a web framework that help simplify the process of building Node.js web server. The same server creation from above code (Fig.4) can also be achieved with Express:



```
var express = require("express");
var app = express();
app.get("/", (req, res) => res.send("Hello Node.js World!"));
app.listen(8080, () => console.log("App is running on port 8080"));
```

Figure 5. Example of server built in Express.js.

Later on, the power of Express can be harnessed when this thesis demonstrates different functionalities of the application such as authentication and authorization, building route sets.

2.2.2 NoSQL and MongoDB

NoSQL

NoSQL databases have a different mechanism to store and retrieve data than the traditional relational databases. Due to the increase of web usage and its complexity, relational databases sometimes cannot catch up to the need due to manual sharding and distributed caching and lack of scalability. NoSQL provides many features to mitigate these drawbacks. NoSQL is built to allow the insertion of data without a predefined schema. That makes it seamless to make significant application changes in real-time, without worrying about service interruptions, meaning that development is faster, code integration is more reliable, and less database administrator time is needed. [12] The application in this thesis will make use of one of the most popular NoSQL database, in particular, MongoDB to help with data storage.

MongoDB

MongoDB is a NoSQL open-source database with dynamic schema. It uses a document storage format call BSON (JSON-like documents in binary style). MongoDB is based on collections and documents. From relational database point of view, they can be considered them as tables and records. The dynamic schema allows each document to have flexible fields and different from one another. Data structures can be changed at any time. MongoDB has simple APIs with support for over 10 programming languages and is still increasing. [13]

mLab

mLab is a fully managed cloud database service featuring automated provisioning and scaling of MongoDB databases, backup and recovery, 24/7 monitoring and alerting, web-based management tools, and expert support [14]. To avoid the steps of installing and running MongoDB in our own operating systems, which sometimes causes some problems, using mLab has been the choice for hosting the database for the application. After signing in to mLab, the next actions are creating a new deployment and adding a new database user with username and password.

```
To connect using the mongo shell:  
% mongo ds125683.mlab.com:25683/thesis-project-development -u <dbuser> -p <dbpassword>  
  
To connect using a driver via the standard MongoDB URI (what's this?):  
mongodb://<dbuser>:<dbpassword>@ds125683.mlab.com:25683/thesis-project-development
```

Figure 6. Example of MongoDB URI in mLab.

Lastly, MongoDB URI is finished and ready to be connected by filling the “<dbuser>” and “<dbpassword>” with the newly-created user’s credentials.

3 IMPLEMENTATION FUNCTIONALITIES

3.1 Functionalities

This thesis guides through the process of making a management dashboard for small and middle-sized companies. Employees(basic) and employers(admin) can register and represent the company they are from. The sole functionality for the app is to help employee to submit their shift preference to choose which days of the week they are willing to work.

There are two different user dashboards for both admin and basic. Users(both roles) are able to register and login to the system. Admin has to do the sign-up first. Then they are given a company code that can be passed on to the basic to register. Basic can select which days they want to go to work, while admin can assess them by choosing to accept or reject it. By default, the application will assume that there would be two shifts a day – morning and evening, no matter which day of the week is that. This is the only feature for now and other features or improvement will be considered if the application continues to be developed.

3.2 Getting started

Back-end

After registering in mLab, mongoose package is used to make connection to Mongo URI. Combining with passport.js and express.js, the following code snippet is a basic starting point:

```
1 const express = require("express");
2 const mongoose = require("mongoose");
3 const bodyParser = require("body-parser");
4 const cors = require("cors");
5 const passport = require("passport");
6
7 const app = express();
8 const port = process.env.PORT || 5000;
9
10 mongoose.Promise = global.Promise;
11 mongoose.set("useCreateIndex", true);
12 mongoose.connect(
13   process.env.MONGO_URI,
14   { useNewUrlParser: true },
15   err => {
16     console.error(err);
17   }
18 );
19
20 app.use(bodyParser.json());
21 app.use(cors());
22
23 app.listen(port, () => {
24   console.log("Listening on port " + port);
25 });
26
```

Figure 7. Server implementation with Node.js, Passport.js and Mongoose.

Stored in .env file as environment variable, MongoDB URI is connected by mongoose and allows us full permissions to all collections in the database.

Express.js is a great framework due to its ability to have middlewares. These intercepts each request, just before it gets to our route handlers, and do some “intermediate” works. In the above figure (Fig.7), body-parser and cors package are used. Body-parser turns incoming request details available under req.body property.

By default, any front-end requests will be stopped by CORS due to difference between front-end and back-end origins. For simplicity, cors is enabled for all requests to the server.

Front-end

In term of client-side, there are create-react-app package for quickly scaffolding a React application and ant-design package for ready-made React UI components. Since the

thesis's plan is to make a SPA, it is necessary to have a method to navigate between different pages.

React-router supports routing provided there are a path (relative URL) and a React component to render.

```
1 import React, { Component } from 'react'
2
3 class App extends Component {
4   render() {
5     return (
6       <BrowserRouter>
7         <Router history={history}>
8           <Layout>
9             <Route
10              exact
11              path="/"
12              component={Landing}
13            />
14            <Route
15              path="/signin"
16              component={SignIn}
17            />
18            <Route
19              path="/signup"
20              component={Signup}
21            />
22            <Route path="/dashboard" component={Dashboard} />
23          </Layout>
24        </Router>
25      </BrowserRouter>
26    )
27  }
28 }
```

Figure 8. Main app.js React component with Routing.

3.3 Authentication and Authorization

3.3.1 Back-end

Below (Table.1) is the list of all the routes created for authentication process:

Table 1. Main URL endpoints for authentication

Route	Purpose
/api/admin/signup	POST save new admin user and company in the database.
/api/basic/signup	POST save a new basic user into the database.
/api/signin	POST receive the credentials and check if it is correct. If yes, send back a JWT to the client.
/api/current_user	GET require authorization by checking JWT in the header request. If the token is valid, the information of the user embedded in the token will be sent back.
/api/company	GET require authorization by checking the JWT in the header request. If the token is valid, the information of the company that user belongs to will be sent back.

Models and Collections

As a dashboard, the app must allow users to sign up and sign in. Since each user will belong to a company, let's start with creating a Company model.


```

1 const mongoose = require("mongoose");
2 const { Schema } = mongoose;
3
4 const companySchema = new Schema({
5   name: {
6     type: String,
7     required: true
8   },
9   company_code: {
10    type: String,
11    required: true
12  }
13 });
14
15 const Company = mongoose.model("Company", companySchema);
16
17 module.exports = Company;

```

Figure 9. Company model.

An instance of mongoose's Schema is formed and named `companySchema`. Schema represents the structure of a single document in a collection. Thus, the company has 2 main keys: `name` and `company_code`. Each is specified as "required", which has to be filled whenever a new company is formed.

Finally, `Company` model is compiled and exported. Newly-created company will be an instance of `Company` model. This model allows to create and modify any documents in the `Company` collections.

```

1 {
2   "_id": {
3     "$oid": "5ccdc94d16b57b6f18375f25"
4   },
5   "name": "Testi Oy",
6   "company_code": "sz0jc3xq",
7   "__v": 0
8 }

```

Figure 10. Example of a `Company` document in JSON.

The key "company_code" is a unique auto-generated 8-character-length random alphanumeric letters. Basic users will fill in this code during registering, along with their

credentials, to eventually let the application know which company user belong before saving them into database.

```
1 const mongoose = require("mongoose");
2 const bcrypt = require("bcryptjs");
3 const validator = require("validator");
4 const { Schema } = mongoose;
5
6 const userSchema = new Schema({
7   role: {
8     type: String,
9     required: true
10  },
11  _company: {
12    type: Schema.Types.ObjectId,
13    ref: "Company"
14  },
15  email: {
16    type: String,
17    required: true,
18    unique: true,
19  },
20  password: {
21    type: String,
22    required: true,
23    minlength: 6
24  },
25  full_name: {
26    type: String
27  },
28 });
29
30 userSchema.pre("save", function(next) {
31   var user = this;
32
33   bcrypt.genSalt(10, (err, salt) => {
34     bcrypt.hash(user.password, salt, (err, hash) => {
35       user.password = hash;
36       next();
37     });
38   });
39 });
40
41 userSchema.methods.comparePassword = function(pw, callback) {
42   bcrypt.compare(pw, this.password, (err, isMatch) => {
43     if (err) {
44       return callback(err);
45     }
46     callback(null, isMatch);
47   });
48 };
49
50 const User = mongoose.model("User", userSchema);
51
52 module.exports = User;
53
```

Figure 11. User Model.

The User model file has the same pattern as Company model but more complex. There are 5 keys in each User document: role, email, password, full_name and _company. The key _company is the most unique. This must have the value to be the id of a company in the Company collection. This is how collections connect to each other. By specifying the id of a company in this field, this makes it simple to check which company a user is from and how to group users in the same company.

In line 30, a hook is attached to the Schema. Whenever a document is about to be saved in the collection, this hook will be called. A general rule of thumb is that user's password is never stored in plain text without encryption. This is where bcryptjs comes in. The package generates random bytes (salt) and combines it with the password sent from front-end before hashing to create unique hashes across each user's password.[15]

Due to the fact that encrypting passwords is a one way process, bcryptjs provide a method to decrypt, which compares password when users request to sign in. A "compare" method is used on line 42, receiving the password that user submits when signing in and compare it with the encrypted password in the database, provided there has been the same email has been registered in the system.

Let's take an example of a basic user with email "test2@mail.com" with password of "123456" that belongs to the above company, the final User document would be:

```
1 {
2   "_id": {
3     "$oid": "5ccdc98d16b57b6f18375f27"
4   },
5   "email": "test2@mail.com",
6   "password": "$2a$10$ls80HVkuzApBK/ts8BL4C.0oITQtk.W7ujXCRzfZS5Is2hh5R87WG",
7   "_company": {
8     "$oid": "5ccdc94d16b57b6f18375f25"
9   },
10  "role": "basic",
11  "full_name": "Basic 1",
12  "__v": 0
13 }
```

Figure 12. Example of User document.

Handling authentication

The key concept of handling authentication in this app is using JWT. Each time a user signs in, a JWT will be sent back and stored in the local storage of the browser. Each subsequent API call from the client-side requesting for authorized route access must have header attached with a JWT, which tell the identity of the person that makes the request and if he/she has the necessary permission. Passport.js facilitates this. It is a middleware authentication package, when specified, that intercepts the request and check if it passes the conditions to receive the response from server. It is extremely flexible and modular, with a comprehensive set of strategies that assists multiple methods of authentication using a username and password, Facebook, Twitter,...[16]. Our server will use two strategies which are passport-jwt and passport-local.

Firstly, Passport-local leverages the traditional sign-in with username, or email in our app, and password. Below, in line 16, a familiar comparePassword function is implemented, almost the same as the identically-named function that the User model declares. The second strategy is the passport-jwt. The jwtOptions object created in line 28 instructs the passport-jwt how to look for the JWT in the request. Literally, the package is told to search a key in the request's header with the name "authorization". As JWT consists of a header, payload and a secret, the secret stored in .env file in the root level folder is provided so that passport knows how to decode the payload underlying within the token. The last step is to prompt passport.js to use the strategies by calling "passport.use".

```

1 const passport = require("passport");
2 const JwtStrategy = require("passport-jwt").Strategy;
3 const ExtractJwt = require("passport-jwt").ExtractJwt;
4 const LocalStrategy = require("passport-local");
5 const User = require("../models/User");
6
7 const localOptions = { usernameField: "email" };
8 const localLogin = new LocalStrategy(localOptions, (email, password, done) => {
9   User.findOne({ email }, (err, user) => {
10    if (err) {
11      return done(err);
12    }
13    if (!user) {
14      return done(null, false);
15    }
16    user.comparePassword(password, (err, isMatch) => {
17      if (err) {
18        return done(err);
19      }
20      if (!isMatch) {
21        return done(null, false);
22      }
23      return done(null, { email: user.email, id: user._id, role: user.role, _company: user._company });
24    });
25  });
26 });
27
28 const jwtOptions = {
29   jwtFromRequest: ExtractJwt.fromHeader("authorization"),
30   secretOrKey: process.env.JWT_SECRET
31 };
32
33 const jwtLogin = new JwtStrategy(jwtOptions, (payload, done) => {
34   User.findById(payload.sub, (err, user) => {
35     if (!user) {
36       return done(err, false);
37     }
38     if (user) {
39       return done(null, { email: user.email, id: user._id, role: user.role, _company: user._company });
40     } else {
41       return done(null, false);
42     }
43   });
44 });
45 passport.use(localLogin);
46 passport.use(jwtLogin);

```

Figure 13. Passport strategies implementation in services/passport.js file.

3.3.2 Front-end

To minimize the work of doing CSS styling from scratch, this app will utilize 2 packages: ant-design and styled-component. Ant-design (antd for short) is a React UI-framework that provides ready-to-use components containing basic styling and functionalities. Obviously, not every UI framework can fit our need. That's when styled-component comes in. This supports styling to any React components given to it. Due to the fact that React components with rich UI designs are verbose and lengthy, only reducers and actions and some screenshots of the app on browser will be shown to demonstrate the logics and final results.

Sign up

```

1 import axios from 'axios'
2 import history from '../services/history'
3
4 // Action Types
5 const SIGNUP_REQUEST = 'auth/SIGNUP_REQUEST'
6 const SIGNUP_SUCCESS = 'auth/SIGNUP_SUCCESS'
7 const SIGNUP_FAILURE = 'auth/SIGNUP_FAILURE'
8
9 // Action Creators
10 const requestSignup = () => ({
11   type: SIGNUP_REQUEST,
12   payload: {
13     isFetching: true,
14     isAuthenticated: false,
15     errorMessage: '',
16   },
17 })
18
19 const signupSuccess = () => ({
20   type: SIGNUP_SUCCESS,
21   payload: {
22     isFetching: false,
23     isAuthenticated: false,
24     errorMessage: '',
25   },
26 })
27
28 const signupFailure = message => ({
29   type: SIGNUP_FAILURE,
30   payload: {
31     isFetching: false,
32     isAuthenticated: false,
33     errorMessage: message,
34   },
35 })
36
37 export const signupUser = (data, m) => async dispatch => {
38   dispatch(requestSignup())
39   let response
40   try {
41     response = await axios.post(
42       `${process.env.REACT_APP_BACK_END_URL}/api/${
43         data.user.role
44       }/signup`,
45       data
46     )
47     dispatch(signupSuccess())
48     m(response.data.message)
49     history.replace('/')
50   } catch (e) {
51     dispatch(signupFailure(e.response.data.error))
52   }
53 }
54 }
55
56 // Reducer
57 const INITIAL_STATE = {
58   isFetching: false,
59   isAuthenticated: false,
60   errorMessage: '',
61 }
62
63 export default (state = INITIAL_STATE, action) => {
64   switch (action.type) {
65     case SIGNUP_FAILURE:
66       return { ...state, ...action.payload }
67     case SIGNUP_SUCCESS:
68       return { ...state, ...action.payload }
69     case SIGNUP_REQUEST:
70       return { ...state, ...action.payload }
71     default:
72       return state
73   }
74 }
75

```

Figure 14. Action creators and reducer for signing up.

When user requests for signing up, provided all form's validations passed, `signupUser` function is called. The first `requestSignup` action creators is dispatched immediately. This passes a new state to the reducers and implies that the app is doing some asynchronous work which requires waiting time. A loading/spinner indicator is shown in the browser to let users know something is happening.

During this time, a POST request to `/api/signup` is made. It contains the data from the sign up form that user has just submitted. The call is wrapped in a Javascript try catch phrase, which throws any errors gracefully that may happen during the request. If the request did not return any error, the second action creator `signupSuccess` is dispatched. This stopped the fetching process and then redirect user to the main page to start signing in. Else, when errors occur, `signupFailure` is called, together with a user-friendly error message appearing in the browser.

Sign in

Same pattern is applied in signing in.

```

1 // ...
2 export const signinUser = ({ email, password }) => async dispatch => {
3   dispatch(requestLogin())
4   let response
5   try {
6     response = await axios.post(
7       `${process.env.REACT_APP_BACKEND_URL}/api/signin`,
8       { email, password }
9     )
10    await localStorage.setItem('token', response.data.token)
11    // set 'authorization' headers for every requests from now on
12    axios.defaults.headers.common['authorization'] = response.data.token
13
14    // dispatch action to change redux state
15    dispatch(receiveLogin(response.data.user))
16    // redirect user to /dashboard
17    history.replace('/dashboard')
18  } catch (e) {
19    if (e.response.status === 401) {
20      dispatch(loginError('Wrong email or password!'))
21    } else {
22      dispatch(
23        loginError('Something went wrong! Please try again later!')
24      )
25    }
26  }
27 }
28 // ...

```

Figure 15. Action creator for signing in

This is where all the magic for making requests to authorized routes comes from. After user successfully signs in, JWT is set to the local storage of the browser. Local storage is just a plain object containing key-value pairs. A key “token” is stored with value of the JWT string. Then the configuration of axios (the fetching package) is changed. For any future requests made by axios, request’s header will always contain a key of “authorization” with the value of JWT string. By doing this, all requests to server’s protected routes will satisfy the requirement of the server.

Sign out

```
1 // ...
2 export const signoutUser = () => async dispatch => {
3   await localStorage.removeItem('token')
4
5   delete axios.defaults.headers.common['authorization']
6
7   dispatch(receiveSignout())
8
9   history.replace('/')
10 }
11 // ...
```

Figure 16. Action creator for signing out.

This is the reverse version of signing in. Firstly, JWT is removed from local storage, as well as its value in axios configuration. Finally, browser redirects user to the main page to let them know that they are logged out successfully.

Higher-order components (HOC) for authorization

Not all the routes are freely accessible for everyone. An approach has to be made to prevent unauthorized entries. This is also a good chance to introduce an advance concept in React, higher-order components (HOC). They are simply another React components but rather than emitting JSX, they produce another React components, or be more exact, enhanced ones. The aim here is to create an HOC that wraps every component demanding authorization before accessing.

```

1 // HOCs/Authorization.jsx
2 import { connect } from 'react-redux'
3
4 const Authorization = allowedRoles => WrappedComponent => {
5   class WithAuthorization extends React.Component {
6     render() {
7       const role = this.props.user.role
8       if (allowedRoles.includes(role)) {
9         return <WrappedComponent {...this.props} />
10      } else {
11        return <ErrorPage />
12      }
13    }
14  }
15   const mapStateToProps = state => ({ user: state.auth.user })
16
17   return connect(mapStateToProps)(WithAuthorization)
18 }
19
20 // App.js
21 const Admin = Authorization(['admin'])
22 const Basic = Authorization(['basic'])
23 const User = Authorization(['admin', 'basic'])
24
25 class App extends Component {
26   render() {
27     //...
28     <Route path="/dashboard" component={User(Dashboard)} />
29     <Route path="/basic/preference/new" component={Basic(PreferenceForm)} />
30     <Route path="/admin/preference/list" component={Admin(PreferenceList)} />
31     // ...
32   }
33 }

```

Figure 17. Authorization Higher Order Component (HOC).

Simply put, we first make a higher-order function that receives a role as parameter. The returned is a HOC. There are some routes that are only available to either type of users and some must be accessible to both. There are 3 different HOCs made for this purpose: Admin, Basic and User. Then, we decide who can navigate to which route, before wrapping the component with equivalent HOC.

3.4 Preference form submission for working days of week

3.4.1 Back end

Table 2. URL Endpoints for handling preference form

Route	Purpose
/api/admin/preference/all	GET get all basic preference forms
/api/basic/preference/new	POST create a new preference form for basic
/api/admin/preference/edit/:form_id	UPDATE update a preference form. Route parameter "form_id" let the server know which form it has to make changes to

As always, it starts with creating schema for a preference form in MongoDB:

```

1 const preferenceDetailSchema = new Schema({
2   date: Number,
3   morning: {
4     type: Boolean,
5     default: false
6   },
7   evening: {
8     type: Boolean,
9     default: false
10  }
11 });
12
13 const preferenceFormSchema = new Schema({
14   _company: { type: Schema.Types.ObjectId, ref: "Company" },
15   _user: { type: Schema.Types.ObjectId, ref: "User" },
16   status: {
17     type: String,
18     default: "Pending"
19   },
20   admin_comment: {
21     type: String,
22     default: ""
23   },
24   submitted_at: {
25     type: Date,
26     required: true
27   },
28   preference_detail: [preferenceDetailSchema]
29 });
30
31 module.exports = mongoose.model("PreferenceForm", preferenceFormSchema);
32

```

Figure 18. Preference Form Schema.

This schema contains two fields with type ObjectId, which are “_company” and “_user”. Since a form must have a sender, “_user” is an id of any basic user in the database. Each schema can have fields to be another schema. “preference_detail” is an array of instance of preferenceDetailSchema. This new schema contains date which implies a day in week. The “date” field runs from 1 to 7 representing Monday to Sunday

3.4.2 Front end

Redux-axios-middleware

As the principle here is to bring the best possible UX for user, each asynchronous call should be handled gracefully by having 3 corresponding action creators listening to it. This can be seen clearly in the authentication process. However, there would be a lot of repetitive code as there are many requests in this section. This is where redux middleware is utilized to do us the favour of the “boring” work.

```

1 export const apiCall = ({
2   endpoint,
3   type,
4   types,
5   payload,
6   method = 'GET',
7   ...opts
8 }) => dispatch => {
9   const token = getToken()
10  const authHeaders = makeAuthHeaders(token)
11
12  return dispatch({
13    type,
14    types,
15    payload: {
16      request: {
17        url: endpoint,
18        method,
19        headers: {
20          ...authHeaders,
21        },
22        ...opts,
23      },
24      options: {
25        onError: handleApiError,
26        onSuccess: handleApiSuccess,
27      },
28    },
29  })
30 }
31

```

Figure 19. Setting up custom calls that triggers middleware.

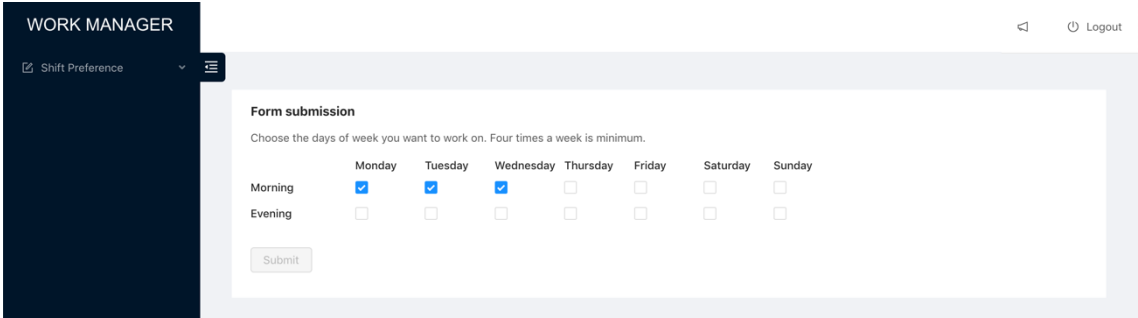
“Redux-axios-middleware” middleware handles any actions that have payload containing a “request” key. Thus, a custom “apiCall” (whose job is to make API calls) is made that contains a payload object with “request” key. It does the normal request at first, but the catch is that after receiving response, depending on status code (successful if 2xx and vice versa), it chains with another action. So a successful or error response will trigger “handleApiSuccess” and “handleApiError” respectively. With this, our codebase clean and readable and reuse it for every future network requests.

Basic user’s view

The main functionality for this app is to let basic users submit their decision to choose which days of the week they can work. The page includes:

- Checkboxes representing each workday from Monday to Sunday and morning/evening shifts.
- Button to submit

The sole form validation is that at least four boxes are checked, until then the button will be disabled. From a business point of view, this guarantees each user has to go to work at least 4 times a day. In the future, this app can be made to be able to adjust this limit in the application settings, which varies between different company’s requirements.



	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Morning	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Evening	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 20. Basic view for submitting form.

After a submission, user will be redirected to the main dashboard and a successful/error message will be shown.

```

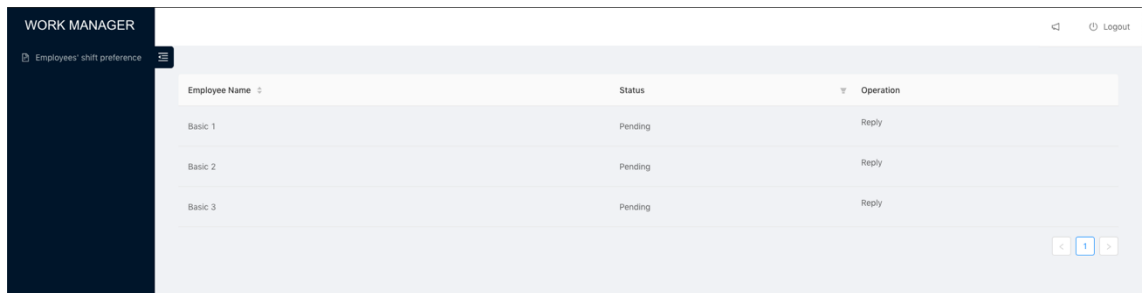
1 export const submitPreferenceForm = preference_detail =>
2   apiCall({
3     type: SUBMIT_PREFERENCE_FORM,
4     url: '/api/basic/preference/new',
5     method: 'POST',
6     data: { preference_detail },
7   })

```

Figure 21. Action creator for submitting preference form.

Admin user's view

Admin's view is more complicated. There will be two smaller views: list of all submitted preference forms and assessment view.



Employee Name	Status	Operation
Basic 1	Pending	Reply
Basic 2	Pending	Reply
Basic 3	Pending	Reply

Figure 22. Admin view for listing all submitted preference forms.

The former is where admin sees all basic preference forms. The main view is a simple table with 3 columns: Employee name, status and Operation. Each row tells user whose the form belongs to, its status and action that admin can take.

```
1 import { createSelector } from 'reselect'
2
3 export const getBasicUsers = state => state.replacement.basicUserList
4
5 const getPreferenceForms = state => state.preference.preferenceForms
6
7 export const computedPreferenceForms = createSelector(
8   getBasicUsers,
9   getPreferenceForms,
10  (users, forms) => {
11    return forms.map(form => {
12      const { full_name } =
13        users.find(user => user._id === form._user) || ''
14      return { ...form, full_name }
15    })
16  }
17 )
```

Figure 23. Simple selector to map users' data.

A commonly-used redux pattern is introduced: selectors. In short, this is where complex business logics are stored. It is recommended that developers separate concerns when doing things (create view components and compute logics) and this is also the way of thinking in React and Redux. We avoid doing expensive data computation in action/reducer or UI component files but in other selector files. This helps our codebase modular and readable and easier to fix if a bug shows up.

In preference form schema, it does not hold user detail but only user's id as “_user”. So how does React show basic users' names? There is the list of basic users kept in redux store and map it with the list of preference forms to create a “computedPreferenceForms”, which is now attached with basic users' “full_name”, without having to make another request to server.

As admin chooses “Reply”, for instance to the form belonging to “Basic 1”, the next view is:

The screenshot shows a web application interface titled 'WORK MANAGER'. On the left is a dark blue sidebar with the text 'Employees' shift preference'. The main content area is light gray and contains the following information:

- Employee name: Basic 1
- Status: Pending
- Date submit: 05/08/2019

Below this, there are seven rows representing days of the week, each with an 'Available:' label and a corresponding input field:

- Monday: Available: Morning Evening
- Tuesday: Available: Morning Evening
- Wednesday: Available:
- Thursday: Available:
- Friday: Available:
- Saturday: Available:
- Sunday: Available:

At the bottom of the form, there is a question: 'How do you want to reply?:' with two radio buttons labeled 'Accept' and 'Decline'. Below this is a text input field labeled 'Your comment/suggestion:'. At the very bottom are two buttons: 'Submit' (blue) and 'Cancel' (red).

Figure 24. Admin view for evaluating a preference form.

This form contains:

- Details about this shift preference form: name of employee, form status, submitted date, the desired timetable
- Action to take: accept or decline. Comment is optional.

If “decline” is chosen, the submission must be redone by basic user. If admin takes “accept”, the equivalent User document will be changed to the following:


```
1 {
2   "_id": {
3     "$oid": "5ccdc98d16b57b6f18375f27"
4   },
5   "email": "test2@mail.com",
6   "password": "$2a$10$ls80HVkuzApBK/ts8BL4C.0oITQtk.W7ujXCRzfZS5Is2hh5R87WG",
7   "_company": {
8     "$oid": "5ccdc94d16b57b6f18375f25"
9   },
10  "role": "basic",
11  "full_name": "Basic 1",
12  "shift_preference": [
13    {
14      "morning": true,
15      "evening": true,
16      "_id": {
17        "$oid": "5d4884c4fd4cd20a58a6adeb"
18      },
19      "date": 1
20    },
21    {
22      "morning": true,
23      "evening": true,
24      "_id": {
25        "$oid": "5d4884c4fd4cd20a58a6adea"
26      },
27      "date": 2
28    },
29    {
30      "morning": false,
31      "evening": false,
32      "_id": {
33        "$oid": "5d4884c4fd4cd20a58a6ade9"
34      },
35      "date": 3
36    },
37    ...
38  ],
39  "__v": 0
40 }
```

Figure 25. Example of an updated User document after admin's evaluation.

4 CONCLUSION

This thesis aimed to build a full stack web application that utilizes the power of Javascript under MERN technology stacks (MongoDB, Express.js, React.js and Node.js). Each major technology that contributes to the development of the application and their examples was discussed, followed by presenting the implementation process with authentication/authorization, preference form submission from the view's point of front-end and back-end respectively.

Eventually, the end product was a workforce management dashboard for small and mid-sized companies in the service industry. Employees were able to register in the app's system and express the requests to select which days they want to work and their employers would validate them accordingly. Overall, this application fulfilled the feature requirements planned out from the beginning and it can be tested in very small scale businesses.

Nevertheless, this application should be treated with care if deployed live in production since there are still some areas for improvement. For example, testing is still missing. Without tests, it is unsafe to continue developing since it is highly likely for developers to make mistakes during development. Besides, new features could be implemented such as making a scheduler from each basic user's shift preference, custom configuration for admins. If properly nurtured, this application could be a success and widely adopted by multiple companies in Finland.

REFERENCES

- [1] React – A Javascript library for writing interfaces. Available at <https://reactjs.org>, accessed July 5, 2019
- [2] State and Lifecycles – React. Available at <https://reactjs.org/docs/state-and-lifecycle.html>, accessed Sep 1, 2019
- [3] Thinking in React – React. Available at <https://reactjs.org/docs/thinking-in-react.html>, accessed Sep 1, 2019
- [4] Introducing JSX – React. Available at <https://reactjs.org/docs/introducing-jsx.html>, accessed July 5, 2019
- [5] Components and Props – React. Available at <https://reactjs.org/docs/components-and-props.html>, accessed July 5, 2019
- [6] Motivation • Redux. Available at redux.js.org/introduction/motivation, accessed July 8, 2019
- [7] Core concepts • Redux. Available at <https://redux.js.org/introduction/core-concepts>, accessed July 8, 2019
- [8] Side effect (computer sciences) - Wikipedia. Available at [https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science)), accessed July 8, 2019
- [9] Three principles • Redux. Available at <https://redux.js.org/introduction/three-principles>, accessed Sep 1, 2019
- [10] Node.js – Wikipedia. Available at <https://en.wikipedia.org/wiki/Node.js>, accessed July 10, 2019
- [11] npm (software) – Wikipedia. Available at [https://en.wikipedia.org/wiki/Npm_\(software\)](https://en.wikipedia.org/wiki/Npm_(software)), accessed July 15, 2019
- [12] What is NoSQL | MongoDB. Available at mongodb.com/nosql-inline, accessed July 21, 2019
- [13] What is MongoDB | MongoDB. Available at <https://www.mongodb.com/what-is-mongodb>, accessed July 21, 2019
- [14] About | mLab Cloud MongoDB Hosting. Available at <https://mlab.com/company>, accessed July 22, 2019
- [15] How bcryptjs works – Rohan Paul – Medium. Available at <https://medium.com/@paulrohan/how-bcryptjs-works-90ef4cb85bf4>, accessed August 1, 2019
- [16] Passport.js. Available at <http://www.passportjs.org>, accessed August 2, 2019

