INF 440a Technologies for Web-based Information Systems

Project A: E-Library

**Sprint 1 Report**

Nikola Lazarov

Irdi Caushi

**Individual Contributions**

**Nikola Lazarov**

This sprint my responsibility were two of the four user stories to be implemented. More specifically:

* "As a user, I want to see a “welcome page” when I visit the web application’s home"
* "As an anonymous user, I want to be able to register in the e-Library system"

Because my stories were the first two, I had to create the general project structure. Our team decided to go with a MERN (MongoDB, Express, React, Node) stack and I created the project repo with an express server and a single-page react application as the view. We use Mongoose as an ORM to interact with the MongoDB database.

**Story 1: As a user, I want to see a “welcome page” when I visit the web application’s home**

The view itself is easy, containing only a single headline. What I had to do is create a new home route which renders the welcome message. In addition to that, as this was the first page, I created the shared header and navigation which will be used later. I used bootstrap for the UI components (navigation, forms, grid etc.) As the react view is a single page application (SPA), it has only a single index.html file to be server on all routes and the routing is handled by react-router on the client. Because of this, I added a viewController module which catches all GET routes and returns the index.html. The other controllers have a more specific api/ route and they override this catch-all controller.

app.get('\*', (req, res) => {

res.sendFile(path.join(\_\_dirname + '../../../dist/index.html'));

});

**Story 2: As an anonymous user, I want to be able to register in the e-Library system**

The first part that had to be created was the user model. As we are using the mongoose ORM, this is straightforward. First of all, the schema is defined as a JavaScript Object, where the data type and validation rules are defined.

const emailValidator = value => {

return emailVal.validate(value);

};

const passValidator = value => {

const passRegEx = /^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{6,}$/;

return passRegEx.test(value);

};

const phoneValidator = value => {

const phoneRegEx = /\d{10}/;

return phoneRegEx.test(value);

};

const userSchema = new mongoose.Schema({

email: {

type: String,

validate: [emailValidator, 'Email address contains invalid characters'],

required: [true, 'Email address is required']

},

password: {

type: String,

validate: [passValidator, 'Password should be at least 6 digits long and contain both letters and digits'],

required: [true, 'Password is required']

},

phone: {

type: String,

validate: [phoneValidator, 'Phone number should be a 10-digit number'],

required: [true, 'Phone is required']

}

});

The validator functions in the model are the server-side validation, which in most cases should not be reached, because of the HTML5 client-side validation. Client-side validation, however, can always be bypassed by malicious users. Client-side validation is also valuable however, for user experience, because it works instantly and the user does not have to wait for a server response.

The next part of the registration is to create a POST endpoint of the users controller under the /api/users route, which would allow the react view layer to create a user using an AJAX HTTP request. The controller code deals primarily with routing the request body to the model and sending back the result.

app.post('/api/users', (req, res) => {

User.findOne({ email: req.body.user.email }, (error, user) => {

// Handle database connection errors

if (error) return res.status(500).send(error.message);

// Check if email already exists in the database

if (user !== null) {

return res.status(500).send('A user already exists with this email.');

}

// Create new user and save to the database

const newUser = new User(req.body.user);

newUser.save((error) => {

if (error) res.status(500).send(error.message);

return res.send('User created successfully.');

});

});

});

The HTTP protocol is utilized for error handling, by sending a 500 (internal server error) status code and an error message to the client in case of a database, server or validation error. In the view this is handled elegantly using the JavaScript Fetch API and an error or success message is shown to the user:

fetch('/api/users', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

user: {

email,

password: pass,

phone

}

})

})

.then(res => {

if (res.ok) return res.text();

throw res;

})

.then(res => {

if (res !== '') setSuccess(res);

})

.catch(res => {

res.text()

.then(err => setError(err));

});

**Irdi Caushi**

<Contribution goes here>

**Acceptance Tests**

### Story 1: As a user, I want to see a “welcome page” when I visit the web application’s home

#### Tests 1 and 2: When users (anonymous and/or registered) visit the main URL (home page) of the app, they should see the welcome page. The welcome page should contain the following message “Welcome to my e-Library”

A screenshot of a social media post

Description automatically generated

Figure 1. Home (Welcome page), showing the welcome message as an H1 HTML element and a header.

### Story 2: As an anonymous user, I want to be able to register in the e-Library system

#### Test 1: Non-logged-in users should see a “Register” menu link on all pages, which leads them to the registration page

#### 

Figure 2. The top navigation showing the Register link, leading to the registration page.

#### Test 2: The registration process requires the user’s email, password and phone number

#### A screenshot of a cell phone Description automatically generated

Figure 3. Registration page with Email, Password and Phone Number fields. Note the Repeat Password field, used to make sure that the user entered their password correctly.

A screenshot of a cell phone

Description automatically generated

Figure 4. Per the requirements, all fields are required. This is achieved by both clent-side and server-side validation.

#### Test 3: The email should be a valid email address and not exist in the system (two registered users cannot have the same email)

#### A screenshot of a cell phone Description automatically generated

Figure 5. Invalid email caught using client-side validation.

A screenshot of a cell phone

Description automatically generated

Figure 6. The client-side validation allowed an invalid email to pass by and the server-side email validation caught it and returned the error to the user.

A screenshot of a cell phone

Description automatically generated

Figure 7. A user is already registered with the input email address and the server returns an error message which indicates to the user he already has an account.

#### Test 4: Passwords should be at least 6 characters long and should contain both letters and numbers

#### A screenshot of a cell phone Description automatically generated

Figure 8. Invalid password is input and a detailed error message is displayed. This is handled by server-side validation.

#### Test 5: Proper validation should be performed by the system, and detailed error messages displayed

As visible by Figures 6 – 8 proper validation is implemented and detailed error messages are displayed to the users for all form fields.