
Embrace Opportunity

Software Engineering Day 11

JSON Web Token (JWT) Authentication

Today's Overview

- JSON Web Token (JWT) Authentication.
- Introduction to middleware.
- Authorisation in NodeJS and Express.

- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 11

Exercise

Tell us about your internships

Exercise:

- 5 minutes:
  - Break-out into internship groups.
  - As a group **define** the internship **project goals**.

- 2 minutes (per group):
  - Choose a representative from each group to **present** back **to the class**.

JWT

JSON Web Token.

JWT Authentication - Introduction

JWT Authentication is a URL-safe means of authentication, where a server provides a JWT to an already-authenticated client so that the client does not need to re-provide a password to access protected resources on the server until the JWT expires.

A **JSON Web Token** is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

JWT.IO allows you to decode, verify and generate JWT.

Structure of a JWT

A **JWT** typically consists of three parts: **Header, Payload,** and **Signature**, separated by dots (.):

- Header: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

  Example:

```json
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Structure of a JWT

- Payload: he payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.

  Example:

```json
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Structure of a JWT

- Signature: To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. For example, if you're using the HMAC SHA256 algorithm, the signature will be created in the following way:

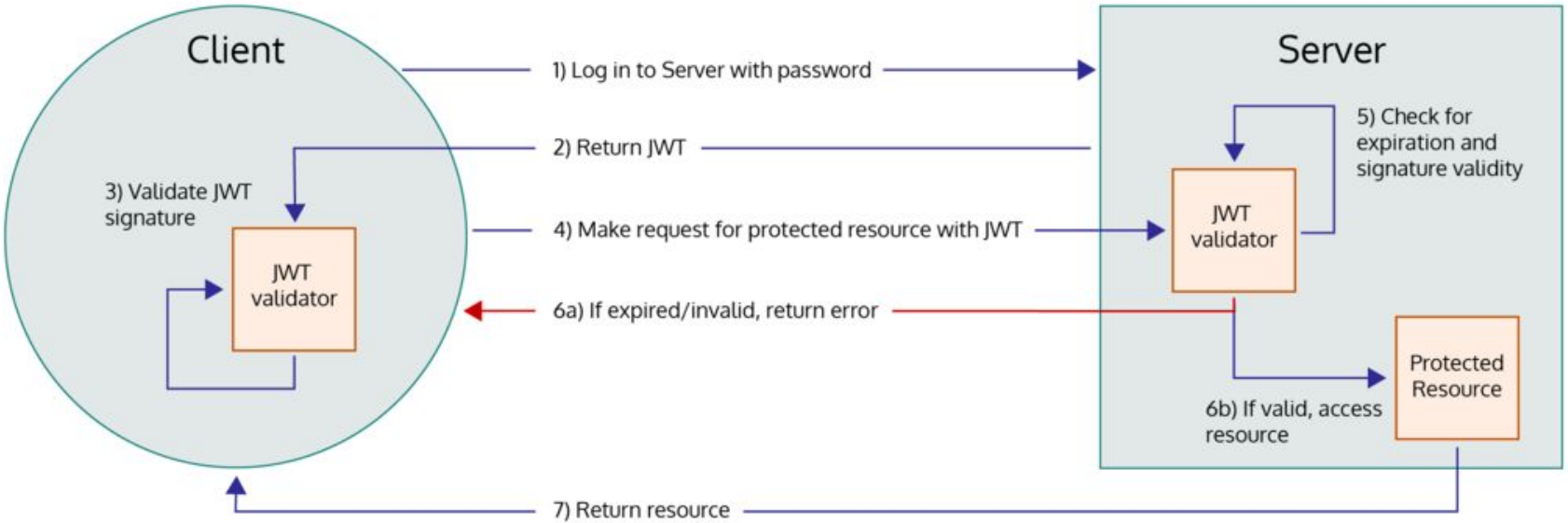
  Example:

```scss
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secret)
```

How JWTs Work

- **Authentication**: After the user logs in using their credentials, a JWT is returned and must be saved locally (typically in local storage, but cookies can also be used). Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema.
- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources permitted with that token.
- **Information Exchange**: JWTs are a good way of securely transmitting information between parties because you can be sure that the senders are who they say they are due to the JWT's signature.

JWT - Introduction



Implementing JWT Authentication

JWT Authentication.

User Model

- **Authentication**: First define a user mongoDB schema (user model)
- Define all fields and field types for user model.
- Export an instance of mongoose model named “Blog” and schema definition.

```js
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema(
  {
    firstName: {
      type: String,
      required: [true, "First name is required"],
    },
    lastName: {
      type: String,
      required: [true, "Last name is required"],
    },
    email: {
      type: String,
      required: [true, "Email is required"],
      unique: [true, "Email is already registered"],
    },
    password: {
      type: String,
      required: [true, "Password is required"],
    },
    image: {
      type: String,
      default:
        "
    },
  },
  {
    timestamps: true,
  }
);

module.exports = mongoose.model("User", userSchema);
```

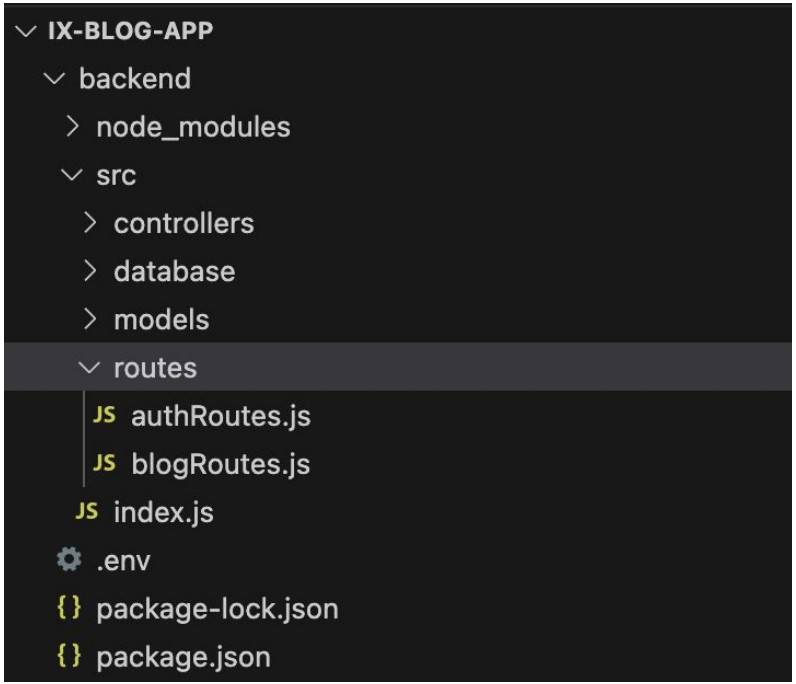
Update Blog Model

- Add the User ref to the blog schema to define which table the authorId is linked to.

```js
const mongoose = require("mongoose");

const blogSchema = new mongoose.Schema(
  {
    authorId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: "User",
    },
    categoryId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: "Category",
    },
    title: {
      type: String,
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    image: {
      type: String,
      required: true,
    },
    content: {
      type: Array,
      required: true,
    },
  },
  { timeStamp: true }
);
```

Create Auth Routes

- In *backend/src/routes* create a file called *authController.js*



Define Auth Routes

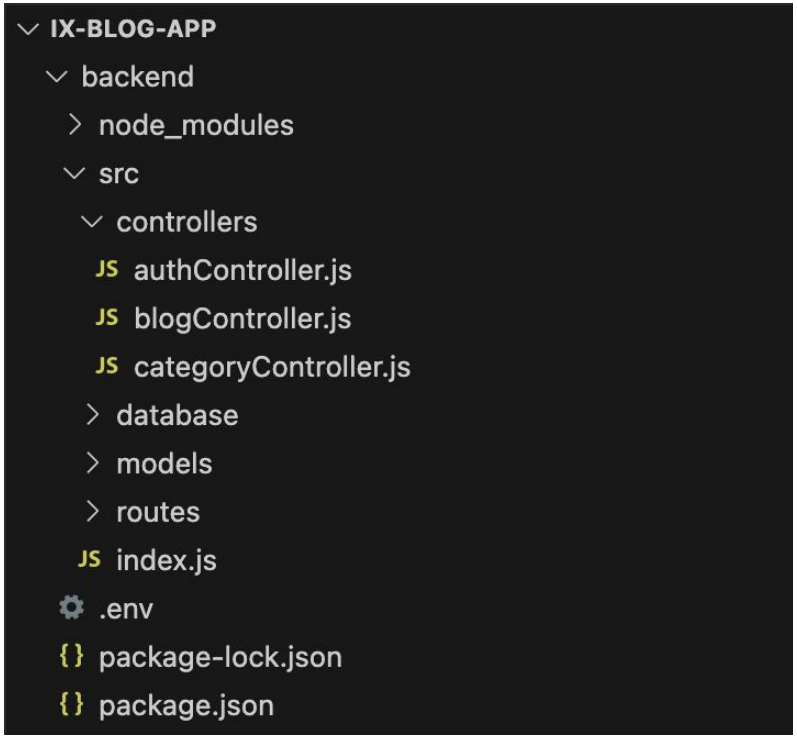
```js
const express = require("express");
const router = express.Router();

const { login, register } = require("../controllers/auth");

router.post("/login", (req, res) => {
  login(req, res);
});

router.post("/register", (req, res) => {
  register(req, res);
});

module.exports = router;
```

Update routes in index.js

```js
const express = require("express");
const cors = require("cors");
require("dotenv").config();

const blogsRoutes = require("./routes/blogs");
const categoryRoutes = require("./routes/categories");
const authRoutes = require("./routes/auth");

const connectDB = require("./database/db");

connectDB();

const port = process.env.PORT || 8000;
const app = express();

app.use(cors());

app.use(express.json());

app.use("/api/blogs", blogsRoutes);
app.use("/api/categories", categoryRoutes);
app.use("/api/auth", authRoutes);

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

Create Auth Controller

- In *backend/src/controllers* create a file called *authController.js*



Define Default Auth Controller

- Define register function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user with this email exists.
  - Create user if one does not already exist.
  - Return user.

```js
const User = require("../models/userModel");

const register = async (req, res) => {
  try {
    const { firstName, lastName, email, bio, password } = req.body;
    //check payload
    if (!firstName || !lastName || !email || !bio || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if email already exists
    const userExists = await User.findOne({ email });
    if (userExists) {
      res.status(400).json({ message: "User already exists", data: [] });
      return;
    }
    // create new user
    const user = new User({
      firstName,
      lastName,
      email,
      bio,
      password,
    });
    const newUser = await user.save();
    res.status(201).json({ message: "New user created!", data: newUser });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};
```

Define Default Auth Controller

- Define login function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user exists with this email exists.
  - Check password if user exists.
  - Return user.

```js
const login = async (req, res) => {
  try {
    const { email, password } = req.body;
    // check payload
    if (!email || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if user exists
    const user = await User.findOne({ email });
    if (!user) {
      res.status(400).json({ message: "User does not exist", data: [] });
      return;
    }
    // check if password is correct
    if (user.password !== password) {
      res.status(400).json({ message: "Invalid credentials", data: [] });
      return;
    }
    res.status(200).json({ message: "Login successful!", data: user });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};

module.exports = {
  register,
  login,
};
```

Install bcrypt

- It is not best practice to save *clear text* passwords in the database. Typically we want to save an encrypted version of the users password to reduce the risk of a cyber security attack.
- **bcrypt** is an npm library use to encrypt and decrypt passwords typically used for authentication.

```sh
npm i bcrypt
```

Update Auth Controller

- Update register function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user with this email exists.
  - **Hash password**
  - Create user if one does not already exist.
  - Return user.

```js
const register = async (req, res) => {
  try {
    const { firstName, lastName, email, bio, password } = req.body;
    //check payload
    if (!firstName || !lastName || !email || !bio || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if email already exists
    const userExists = await User.findOne({ email });
    if (userExists) {
      res.status(400).json({ message: "User already exists", data: [] });
      return;
    }
    // hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    // create new user
    const user = new User({
      firstName,
      lastName,
      email,
      bio,
      hashedPassword,
    });
    const newUser = await user.save();
    res.status(201).json({ message: "New user created!", data: newUser });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};
```

Update Auth Controller

- Update login function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user with this email exists.
  - Check **hashed password** if user exists.
  - Return user.
- How do we track that a user is logged in? **(JWT)**

```js
const login = async (req, res) => {
  try {
    const { email, password } = req.body;
    // check payload
    if (!email || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if user exists
    const user = await User.findOne({ email });
    if (!user) {
      res.status(400).json({ message: "User does not exist", data: [] });
      return;
    }
    // check hashed password
    const validPassword = await bcrypt.compare(password, user.password);
    if (!validPassword) {
      res.status(400).json({ message: "Invalid credentials", data: [] });
      return;
    }
    res.status(200).json({ message: "Login successful!", data: user });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};
```

Install jsonwebtoken

- **jsonwebtoken** is an npm package that implements JSON Web Tokens (JWT) in Node.js applications. JSON Web Tokens are an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

Update Auth Controller

- Add a random secret in your .env file

```
PORT=8000
MONGO_URI=mongodb://localhost:27017/blog_app
JWT_SECRET=123abc
```

- Create a **generateToken** function in *src/controllers.authcontroller.js*

```js
const jwt = require("jsonwebtoken");

.
.
.

const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    // expiresIn: "1d",
  });
};
```

Update Auth Controller

- Update register function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user exists.
  - Hash password
  - Create user if one does not already exist.
  - Generate JWT
  - Return user with JWT

```js
const register = async (req, res) => {
  try {
    const { firstName, lastName, email, bio, password } = req.body;
    //check payload
    if (!firstName || !lastName || !email || !bio || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if email already exists
    const userExists = await User.findOne({ email });
    if (userExists) {
      res.status(400).json({ message: "User already exists", data: [] });
      return;
    }
    // hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    // create new user
    const user = new User({
      firstName,
      lastName,
      email,
      bio,
      password: hashedPassword,
    });
    const newUser = await user.save();
    newUser.password = undefined;
    newUser.token = generateToken(newUser._id);
    res.status(201).json({ message: "New user created!", data: newUser });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};
```

Update Auth Controller

- Update login function.
  - Extract payload parameters.
  - Verify payload parameters.
  - Check if user with this email exists.
  - Check hashed password if user exists.
  - Generate JWT
  - Return user with JWT.

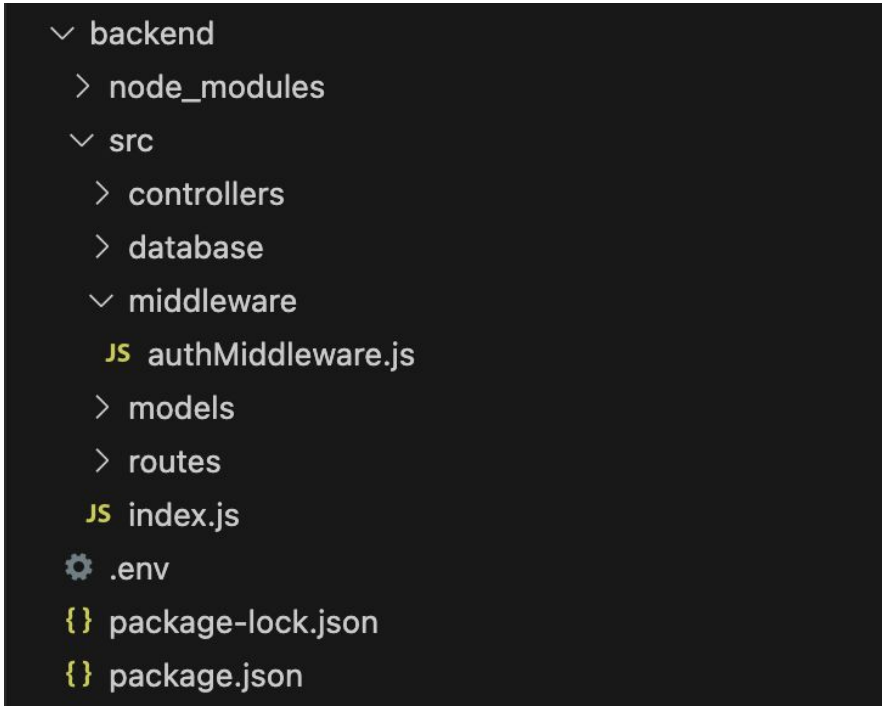
```js
const login = async (req, res) => {
  try {
    const { email, password } = req.body;
    // check payload
    if (!email || !password) {
      res.status(400).json({ message: "All fields are required", data: [] });
      return;
    }
    // check if user exists
    const user = await User.findOne({ email });
    if (!user) {
      res.status(400).json({ message: "User does not exist", data: [] });
      return;
    }
    // check hashed password
    const validPassword = await bcrypt.compare(password, user.password);
    if (!validPassword) {
      res.status(400).json({ message: "Invalid credentials", data: [] });
      return;
    }
    user.password = undefined;
    user.token = generateToken(newUser._id);
    res.status(200).json({ message: "Login successful!", data: user });
  } catch (error) {
    res.status(500).json({ message: error.message, data: [] });
  }
};
```

Protect routes with JWT Authentication

JWT Authentication.

Create Auth Middleware

- In *backend/src/middleware* create a file called *authMiddleware.js*



Protect existing routes

- Update existing blog routes with auth middleware.
  - Middleware is passed in as the second parameter in express router.
  - Only protect create, update and delete blog routes.

```js
const express = require("express");
const router = express.Router();

const blogController = require("../controllers/blogs");

const { protect } = require("../middleware/authMiddleware");

router.post("/", protect, (req, res) => {
  createBlog(req, res);
});

router.get("/", (req, res) => {
  getBlogs(req, res);
});

router.get("/:id", (req, res) => {
  getBlog(req, res);
});

router.put("/:id", protect, (req, res) => {
  updateBlog(req, res);
});

router.delete("/:id", protect, (req, res) => {
  deleteBlog(req, res);
});

module.exports = router;
```

Exercise

Testing with postman

Exercise:

- Let’s test our login and register end points:
  - Register

    https://byrondev121.github.io/ix-vue-press/day-11/postman-testing.html#register

  - Login

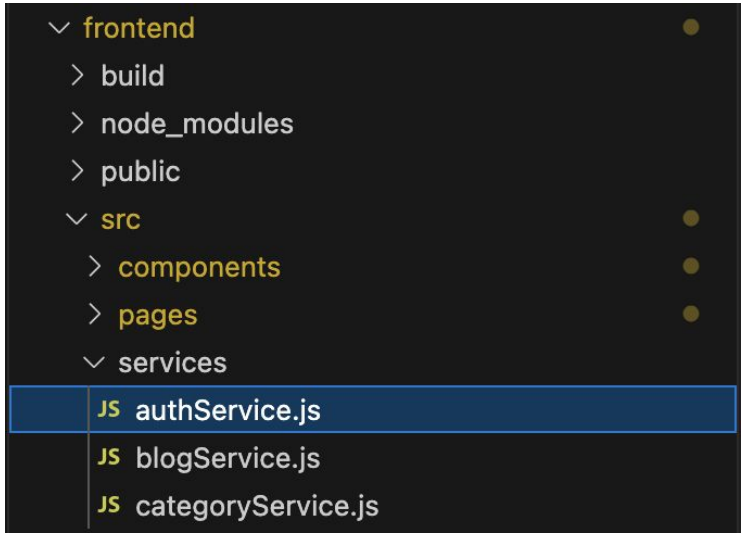
    https://byrondev121.github.io/ix-vue-press/day-11/postman-testing.html#login

Frontend

JWT Authentication.

Create Auth Service

- In *frontend/src/services* create a file called *authService.js*.



Frontend - Service

- POST - /api/auth/register

```js
const register = async (userData) => {
  const response = await fetch("http://localhost:8000/api/auth/register", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(userData),
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  localStorage.setItem("user", JSON.stringify(responseData.data));
  return responseData;
};

const authService = {
  register,
};

export default authService;
```

Frontend - Service

- POST - /api/auth/login

```js
...

const login = async (userData) => {
  const response = await fetch("http://localhost:8000/api/auth/login", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(userData),
  });

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  localStorage.setItem("user", JSON.stringify(responseData.data));
  return responseData;
};

const authService = {
  register,
  login
};

export default authService;
```

Frontend - Service

- POST - /api/auth/user/:id

```js
...

const getUser = async (authorId) => {
  const response = await fetch(
    `http://localhost:8000/api/auth/user/${authorId}`
  );

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};

const authService = {
  register,
  login,
  getUser
};

export default authService;
```

Frontend - Service

- POST - /api/auth/auth/user/update/:id

```js
...

const updateUser = async (userId, userData) => {
  const response = await fetch(
    `http://localhost:8000/api/auth/user/${userId}`,
    {
      method: "PUT",
      headers: {
        // "Content-Type": "application/json",
        Authorization:
          "Bearer " + JSON.parse(localStorage.getItem("user")).token,
      },
      body: userData,
    }
  );

  if (!response.ok) {
    let res = await response.json();
    throw res;
  }

  const responseData = await response.json();
  return responseData;
};

const authService = {
  register,
  login,
  getUser,
  updateUser
};

export default authService;
```

Frontend - Pages

- Register
  - In *frontend/src/page* create Register dir and add files called *index.js and index.css.*

Similar to the register pages we built on Day 3. We can use this register page in react:

https://byrondev121.github.io/ix-vue-press/day-11/frontend.html#register-page

Frontend - Pages

- Login
  - In *frontend/src/page* create Login dir and add files called *index.js and index.css.*

Similar to the login pages we built on Day 3. We can use this register page in react:

https://byrondev121.github.io/ix-vue-press/day-11/frontend.html#login-page

Frontend - Authenticated Routes

- Add the following header to all authenticated routes

```
    headers: {
      "Content-Type": "application/json",\
      Authorization: "Bearer " + JSON.parse(localStorage.getItem("user")).token,
    },
```

Homework

Hide the add and edit buttons

Homework

- Makes sure you are up to date with the blog app.
  - JWT Authentication implementation on backend
    - User registration
    - User login
    - Authenticated routes
  - JWT Authentication implementation on frontend
    - Register and Login Pages
    - Passing user token for authenticated routes

- Only show the Add, Edit and Delete buttons for Blogs and Categories if a user is logged in.

Homework

- In your internship groups Create a Document detailing the requirements of the project in your internship.
- Define frontend designs - (rough sketches are fine).
- Define App functionality and flow.
- Define all DB models you (use UML).
- Present back to the class on Thursday.

Next Class

React Lifecycle Methods, Hooks, Context API and Routes