WF1 Week 10 and 11 – Practical: Identity Management

Todays Lesson:

- Creating a user system
    - Client side:
        - Create a sign in page
        - Create a login page
        - Create a NavBar to move through our bookstore
        - Showcase when a user is logged in
    - Server side
        - Create routes to handle the new user information
        - Create a model to help transfer data into the DB
            - Grab a Username, Email, Password
                - Do we store the Password as 'plain-text'?
- The point:
    - Have you built out a user system for an application
    - Restrict webpages and access to logged in users
    - Saving information onto our database and how we know/verify that a user stay logged in

Using your app from week 9, or this reorg file, we will add on a user system first.

HarmanSMann/WF1-TEMP-W9-Mongoose at refactor-book_router (github.com)

You will need to make your own .env file and place your MongoDB string into it

Index:

Part 1: Adding a user system

Look back to how you create an account and login to an application. You provide information into a form, submit it to a server and have the server handle the requests

- Signup, or login -> we would need to make routes for them
- We want to store user information inside a db, and since we are using MongoDB and mongoose, we can make a model with what information we need to store.

Let's decide what information we want to store:

- Name -> as full name? first name and last name?
- Username, email address, phone number, password

Here is a model file for user: user.js

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema(
  {
    first_name: {
      type: String,
      required: true,
    },
    last_name: {
      type: String,
      required: true,
    },
    username: {
      type: String,
      required: true,
      unique: true,
      trim: true,
      minlength: 3,
    },
    email: {
      type: String,
      required: true,
      unique: true,
      trim: true,
      lowercase: true, // Convert email to lowercase
      match: [/.+\@.+\..+/, "Please fill a valid email address"],
    },
    password: {
      type: String,
      required: true,
      minlength: 6,
    },
  },
);
const User = mongoose.model("User", userSchema);
module.exports = User;
```

Following the process, we want to integrate this into our express app. Since we also want to keep the files organized, we can make a new router file for the user operations.

Inside the index.js,

```
const user_router = require("./router/user_router");

app.use("/user", user_router)
```

Next, we must make our router to handle requests. We want to at least setup a signup request and a login request.

Process is something like this:

User inputs their information -> server handles processing the information, validating the fields are filled in -> save the user information into the DB.

We have done something like this with the bookstore, where we add a new book, so we can follow that process. Receive the request to our /signup route, validate all fields are filled in (middleware), assemble an object for the User using the user model and save the information.

```
const express = require("express");
const router = express.Router();
const path = require("path");
const User = require("../models/user");
const { validateUser } = require("../middleware/validator");

router
  .route("/sign-up")
  .get((req, res) => {
    res.sendFile(path.join(__dirname, "../views", "sign-up.html"));
  })
  .post(validateUser, (req, res) => {
    const { first_name, last_name, username, email, password } = req.body;
    const newUser = new User({first_name, last_name, username, email, password,
    });

    newUser
      .save()
      .then(() => {
        res.status(201).json({ message: "Successfully Added" });
      })
      .catch((err) => {
        console.error(err);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });
```

```
  router
  .route("/login")
  .get((req, res) => {
    res.sendFile(path.join(__dirname, "../views", "login.html"));
  })
  .post((req, res) => {
    const { email, password } = req.body;

    User.findOne({ email })
      .then(user => {
        if (!user) {
          console.log(`User with email ${email} not found`);
          return res.status(400).json({ error: `User with email ${email} not
found` });
        }
        // Compare passwords
        if (user.password !== password) {
          console.log(`Incorrect password for user ${email}`);
          return res.status(400).json({ error: `Incorrect password for user
${email}` });
        }
        // Passwords match, login successful
        console.log(`Hello ${user.first_name}, with username ${user.username}
(${email}) logged in successfully`);
        res.json({ message: `Hello ${user.first_name}, with username
${user.username} (${email}) logged in successfully` });
      })
      .catch(error => {
        console.error("Error during login:", error);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });
module.exports = router;
```

I have also modified the bookValidator file to do multiple validations, so I have renamed the file to validator, and added functions for each validation I want to do. This is stored inside /middleware/validator.js

```
const { body, validationResult } = require("express-validator");

// Validator for validating book data
const validateBook = [
  body("title", "Title is required").notEmpty(),
  body("author", "Author is required").notEmpty(),
  body("pages", "Pages is required").notEmpty(),
  body("rating", "Rating is required").notEmpty(),
  body("genres", "Genre is required").notEmpty(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

// Validator for validating user data
const validateUser = [
  body("first_name", "First Name Field is required").notEmpty(),
  body("last_name", "Last Name Field is required").notEmpty(),
  body("username", "Username Name Field is required").notEmpty(),
  body("email", "Valid email is required").isEmail().withMessage('Email is not
valid').notEmpty().withMessage('Email Field is required'),
  body("password", "Password is required").notEmpty(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

module.exports = {
  validateBook,
  validateUser,
};
```

To follow that, we also should setup html pages to handle the routing, OR you can use postman to handle submitting the information.

Here is a login and signup page. Sign up needs more information to be provided, and the login page requires only an email and password

Login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Login</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    label {
      display: block;
      margin-bottom: 5px;
    }
    input {
      width: 100%;
      padding: 8px;
      margin-top: 5px;
      margin-bottom: 10px;
      border: 1px solid #ccc;
      border-radius: 4px;
      box-sizing: border-box;
    }
    button {
      background-color: #4CAF50;
      color: white;
      padding: 10px 20px;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    button:hover {
      background-color: #45a049;
    }
    .alert {
      padding: 15px;
```

```
      margin-bottom: 20px;
      border: 1px solid #ddd;
      border-radius: 4px;
    }
    .alert.success {
      background-color: #4CAF50;
      color: white;
    }
    .alert.error {
      background-color: #ad3229;
      color: white;
    }
  </style>
</head>
<body>
  <h1>Login</h1>

  <!-- Alert Messages -->
  <div id="alertMessage" class="alert" style="display: none;"></div>

  <!-- Login Form -->
  <form id="loginForm" action="/user/login" method="POST">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required><br><br>

    <button type="submit">Login</button>
  </form>

  <p>Don't have an account? <a href="/user/sign-up">Sign Up</a></p>

  <script>
    const loginForm = document.getElementById('loginForm');
    const alertMessage = document.getElementById('alertMessage');

    loginForm.addEventListener('submit', function(event) {
      event.preventDefault();

      const formData = new FormData(loginForm);
      const email = formData.get('email');
      const password = formData.get('password');

      fetch('/user/login', {
```

```
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ email, password }),
      })
      .then(response => response.json())
      .then(data => {
        if (data.message) {
          showSuccessMessage(data.message);
          setTimeout(() => {
            window.location.href = '/bookstore'; // Redirect to homepage after
success message
          }, 2000); // Redirect after 2 seconds
        } else {
          showErrorMessage(data.error);
        }
      })
      .catch(error => {
        console.error('Error logging in:', error);
        showErrorMessage('An error occurred. Please try again later.');
      });
    });

    function showSuccessMessage(message) {
      alertMessage.textContent = message;
      alertMessage.className = 'alert success';
      alertMessage.style.display = 'block';
    }

    function showErrorMessage(message) {
      alertMessage.textContent = message;
      alertMessage.className = 'alert error';
      alertMessage.style.display = 'block';
    }
  </script>
</body>
</html>
```

Signup.html

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Sign Up</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    label {
      display: block;
      margin-bottom: 5px;
    }
    input {
      width: 100%;
      padding: 8px;
      margin-top: 5px;
      margin-bottom: 10px;
      border: 1px solid #ccc;
      border-radius: 4px;
      box-sizing: border-box;
    }
    button {
      background-color: #4CAF50;
      color: white;
      padding: 10px 20px;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    button:hover {
      background-color: #45a049;
    }
  </style>
</head>
<body>
  <h1>Sign Up</h1>
  <form id="signUpForm" action="/user/sign-up" method="POST">
    <label for="first_name">First Name:</label>
    <input type="text" id="first_name" name="first_name" required><br><br>

    <label for="last_name">Last Name:</label>
    <input type="text" id="last_name" name="last_name" required><br><br>

    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required><br><br>
```

```
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required><br><br>

    <button type="submit">Sign Up</button>
  </form>
  <p>Already have an account? <a href="/user/login">Login</a></p>

  <script>
    document.getElementById('signUpForm').addEventListener('submit',
function(event) {
      event.preventDefault();

      fetch('/user/sign-up', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          first_name: document.getElementById('first_name').value,
          last_name: document.getElementById('last_name').value,
          username: document.getElementById('username').value,
          email: document.getElementById('email').value,
          password: document.getElementById('password').value
        })
      })
      .then(response => response.json())
      .then(data => {
        alert(data.message); // Display success message
        window.location.href = '/'; // Redirect to homepage
      })
      .catch(error => {
        console.error('Error:', error);
        alert('Error signing up. Please try again.'); // Display error message
      });
    });
  </script>
</body>
</html>
```

This completes the first part of the Practical. For the next part, we will go over encrypting a password. This is a shorter section.

Part 2: Encrypting a password

Here is a new package I want you to install: npm install bcrypt

Bcrypt will allow us to encrypt a password to store it into our database and handle decrypting the password for comparisons. **Here is a quick example down below**:

```
const bcrypt = require("bcrypt");
const password = "password123";

async function run() {
    const p1 = bcrypt.hash(password, 10);
    const p2 = bcrypt.hash(password, 10);
    const p3 = bcrypt.hash(password, 10);

    const [hashed1, hashed2, hashed3] = await Promise.all([p1, p2, p3]);

    console.log(hashed1);
    console.log(hashed2);
    console.log(hashed3);

    const isMatch = await bcrypt.compare("password123", hashed1);
    const isMatch2 = await bcrypt.compare("password1234", hashed1);
    console.log("Password matches:", isMatch);
    console.log("Password matches:", isMatch2);
}

run();
```

Using the same password, you will notice encryptions are all different.

```
const p1 = bcrypt.hash(password, 10);
```

The hash call will take a given string and add salt onto the password, the value 10 that was provided indicates a **work value**. The work value is setup for how much work our computer should put into encrypting and decrypting the key.

So, for our integration, we want to hash the password and store the hashed password. And when someone try's to login, we fetch the user information, decrypt the user.password and compare it the given password from the request.

Here are the updated files:

```
const express = require("express");
const router = express.Router();
const path = require("path");
const User = require("../models/user");
const bcrypt = require("bcrypt");
const {
  validateUserSignup,
  validateUserLogin,
} = require("../middleware/validator");

// Serve static files
router.get("/user/:id", (req, res) => {
  res.sendFile(path.join(__dirname, "../views", "books.html"));
});

router.get("/user-list", (req, res) => {
  res.sendFile(path.join(__dirname, "../views", "books.html"));
});

// User signup
router
  .route("/sign-up")
  .get((req, res) => {
    res.sendFile(path.join(__dirname, "../views", "sign-up.html"));
  })
  .post(validateUserSignup, (req, res) => {
    const { first_name, last_name, username, email } = req.body;

    bcrypt
      .hash(req.body.password, 10)
      .then((hashedPassword) => {
        const newUser = new User({
          first_name,
          last_name,
          username,
          email,
          password: hashedPassword,
```

```
        });

        return newUser.save();
      })
      .then(() => {
        res.status(201).json({ message: "Successfully Added" });
      })
      .catch((err) => {
        console.error(err);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });

// User login
router
  .route("/login")
  .get(validateUserLogin, (req, res) => {
    res.sendFile(path.join(__dirname, "../views", "login.html"));
  })
  .post((req, res) => {
    const { email, password } = req.body;

    User.findOne({ email })
      .then((user) => {
        if (!user) {
          console.log(`User with email ${email} not found`);
          return res
            .status(400)
            .json({ error: `User with email ${email} not found` });
        }

        return bcrypt.compare(password, user.password).then((isMatched) => {
          if (!isMatched) {
            return res
              .status(400)
              .json({ error: `Incorrect password for user ${email}` });
          }

          console.log(
            `Hello ${user.first_name}, with username ${user.username} (${email})
logged in successfully`
          );
          res.json({
            message: `Hello ${user.first_name}, with username ${user.username}
(${email}) logged in successfully`,
```

```
          });
        });
      })
      .catch((error) => {
        console.error("Error during login:", error);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });

module.exports = router;
```

validator file:

```
const { body, validationResult } = require("express-validator");

// Validator for validating book data
const validateBook = [
  body("title", "Title is required").notEmpty(),
  body("author", "Author is required").notEmpty(),
  body("pages", "Pages is required").notEmpty(),
  body("rating", "Rating is required").notEmpty(),
  body("genres", "Genre is required").notEmpty(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

// Validator for validating user data
const validateUserSignup = [
  body("first_name", "First Name Field is required").notEmpty(),
  body("last_name", "Last Name Field is required").notEmpty(),
  body("username", "Username Name Field is required").notEmpty(),
  body("email", "Valid email is required").isEmail().withMessage('Email is not
valid').notEmpty().withMessage('Email Field is required'),
  body("password", "Password is required").notEmpty(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
```

```
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

const validateUserLogin = [
  body("email", "Valid email is required").isEmail().withMessage('Email is not
valid').notEmpty().withMessage('Email Field is required'),
  body("password", "Password is required").notEmpty(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

module.exports = {
  validateBook,
  validateUserSignup,
  validateUserLogin,
};
```

That is the end of Part 2

- incorporated bcrypt, modified the user router and modified the validator for both signup and login

!!

Part 3: Sessions and JWT

Link to the code used for this section: [HarmanSMann/WF1-TEMP-W9-Mongoose at w10-adding-sessions (github.com)]

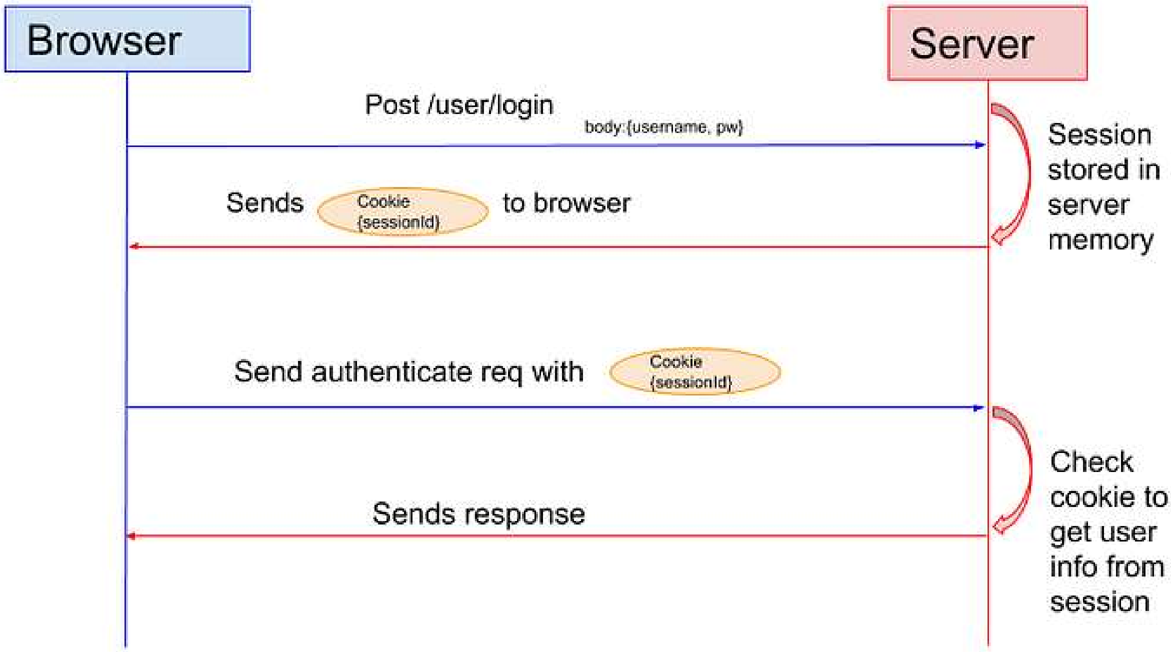
You may notice by now that everything we have completed processes our request, but how to we make sure our app still knows we are logged in? There is a mixture of ways to do this. The concepts are called sessions, cookies, and JWT.

[Session vs Token Based Authentication | by Sherry Hsu | Medium]

[Difference between Session Cookies vs. JWT (JSON Web Tokens), for session management. | by Prashant Ram | Medium]

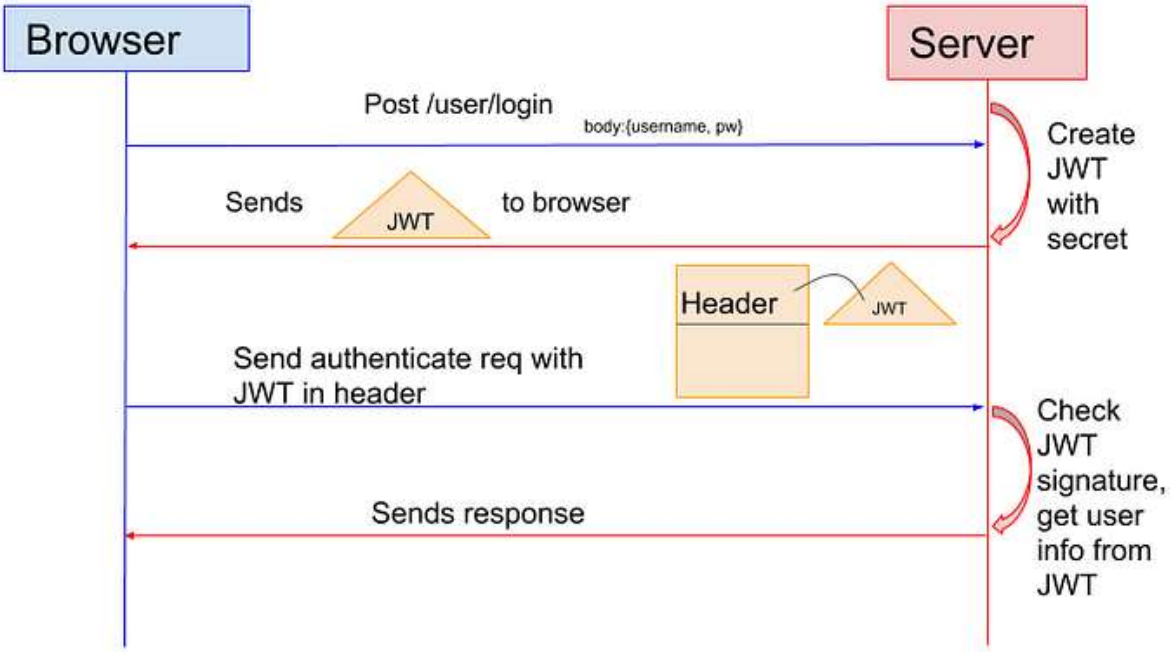
Sessions: Stored on the server, keep track of your login attempt, providing you information on being logged in. A similar system to Humbers's login system. We login through our device, pass through a 2FA and login. It persists until we logout, or until some timer is setoff from the server to log us out.

Cookies: typically, stored inside our web browsers. Used to store information about our session. We would need to also pass along our cookie back and forth to the server, so the server knows we are the right person.



JWT: JSON Web Token, stored similarly to a cookie on your browser. We will be storing it into the localstorage of the web browser, and as we send requests to the server, we will also send them our token

[JWT authentication: Best practices and when to use it - LogRocket Blog]



Part 3 – Version 1 – Sessions

To implement our sessions, we will use express-sessions. We will also consider saving the session into MongoDB, which will require another package to save the session. When a session is saved, it sends a cookie inside the response called **connect_sid**. We can see it in the inspect tool.

express-session - npm (npmjs.com)

Here is an example of how-to setup sessions from the npm site above:

```
var app = express()
var session = require('express-session')
app.set('trust proxy', 1) // trust first proxy
app.use(session({
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: true }
}))
```

For our purpose, we will save our session into mongodb. We can declare where we want to store the session.

First, installing the new packages: **npm install express-session connect-mongo**

```
const session = require("express-session"); //new
const MongoStore = require("connect-mongo"); //new

app.use(
  session({
    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: true,
    store: MongoStore.create({
      mongoUrl: process.env.DB_HOST,
      collectionName: "sessions",
      ttl: 5 // this is in seconds
    }),
```

```
    cookie: { maxAge: 1000 * 5 }, // maxAge: 1000 * 60 * 60 * 24 1 day
  })
);

app.get("/test-session", (req, res) => {
  if (req.session.views) {
    req.session.views++;
    res.setHeader("Content-Type", "text/html");
    res.write("<p>Views: " + req.session.views + "</p>");
    res.end();
  } else {
    req.session.views = 1;
    res.end("Welcome to the session demo. Refresh!");
  }
});
```

Here are a few things we need to do.

- Make a secret key for our session. This can be stored inside our environment file.
    - SESSION_SECRET = WHATEVER_YOU_WANT_HERE

Here is a completed index.js file with the information:

```
require("dotenv").config();
const path = require("path");
const express = require("express");
const session = require("express-session"); //new
const MongoStore = require("connect-mongo"); //new
const mongoose = require("mongoose");

const app = express();

const PORT = process.env.PORT || 8000;
// Configure session middleware
app.use(
  session({
    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: true,
    store: MongoStore.create({
      mongoUrl: process.env.DB_HOST,
      collectionName: "sessions",
    }),
    cookie: { maxAge: 1000 * 60 * 60 * 24 }, // 1 day
```

```
  })
);
const book_router = require("./router/book_router");
const user_router = require("./router/user_router");

// Connect to MongoDB
mongoose.connect(process.env.DB_HOST);
let db = mongoose.connection;

db.once("open", () => {
  console.log("Connected to MongoDB");
});
db.on("error", (err) => {
  console.log("DB Error:" + err);
});

// Application level middleware
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use("/bookstore", book_router);
app.use("/user", user_router);

app.get("/test-session", (req, res) => {
  if (req.session.views) {
    req.session.views++;
    res.setHeader("Content-Type", "text/html");
    res.write("<p>Views: " + req.session.views + "</p>");
    res.end();
  } else {
    req.session.views = 1;
    res.end("Welcome to the session demo. Refresh!");
  }
});

app.get("/", (req, res) => {
  res.redirect("/bookstore");
});

app.listen(PORT, () =>
  console.log(`Server started on http://127.0.0.1:${PORT}`)
);
```

Because we are using sessions for the user system, we should assume that's what we will edit.

- We want the session to be made when we login, so we need to modify that
- We need to also handle deleting the session when they logout, so we need a logout button.

Modifying the login route:

```
  .post(validateUserLogin, (req, res, next) => {
    const { email, password } = req.body;

    User.findOne({ email })
      .then((user) => {
        if (!user) {
          console.log(`User with email ${email} not found`);
          return res
            .status(400)
            .json({ error: `User with email ${email} not found` });
        }

        bcrypt.compare(password, user.password).then((isMatched) => {
          if (!isMatched) {
            return res
              .status(400)
              .json({ error: `Incorrect password for user ${email}` });
          }



          // Save user info in session
          req.session.user = {
            id: user._id,
            email: user.email,
            username: user.username,
            first_name: user.first_name,
            // Add any other necessary user data
          };

          // Save the session explicitly
          req.session.save((err) => {
            if (err) {
              console.error("Error saving to session storage: ", err);
              return next(new Error("Error saving session"));
            }
```

```
            console.log(
              `Hello ${user.first_name}, with username ${user.username}
(${email}) logged in successfully`
            );
            res.json({
              message: `Hello ${user.first_name}, with username ${user.username}
(${email}) logged in successfully`,
            });
          });
        });
      })
      .catch((error) => {
        console.error("Error during login:", error);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });
```

```
// User logout
router.post("/logout", (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      console.error("Error during logout:", err);
      return res.status(500).json({ error: "Internal Server Error" });
    }

    res.json({ message: "Logged out successfully" });
  });
});
```

We have established setting up the session once the user logs in and when they logout we destroy the session. How does the HTML/Frontend handle the information? We receive the connectsid, but as a user, there is no direct indication that the user has logged in. To work with that, we can add a message to the user, indicating they have logged in, and replace the login button with the logout button.

We have to setup a route that handles fetching the session, to see if the user is logged in and send the user information from the server to the HTML pages.

Here is the code for it. When we reach this route, we check if the session contains a user inside it

```
// Check login status
router.get("/check-login", (req, res) => {
  if (req.session.user) {
    // If user session exists, return user data
    res.json({
```

```
      isLoggedIn: true, user: req.session.user,
    });
  } else {
    // If user session does not exist, return isLoggedIn false
    res.json({ isLoggedIn: false,});
  }
});
```

By sending the status of the session to the client side, we can have the client side always know if the user is logged in or out during their navigation through the site.

What I have also included, as an example is a middleware operation that will fetch user information from the server. User information is sensitive to us, so we want authorized personal to use the route, for now, we will restrict at level 1. Any user, that has an account, can also see the content. Moving to additional levels means checking for their level of authority within the site. (Something to think about for the future).

Here is the idea

1. I as a logged in user want to access the information of all users on the DB
2. I send a request to the /user/getall route
3. The server takes in the request,
    a. Verify that the user session is correct
        i. On success – send them the information
        ii. On failure – send them an error message

Here is an example that will fetch the user data for 1 user in the DB

```
router.get("/api/user/:id", authMiddleware, (req, res) => {
  User.findById(req.params.id)
    .then((user) => {
      if (!user) {
        return res.status(404).json({ error: "User not found" });
      }
      res.json(user);
    })
    .catch((err) => {
      console.error("Error fetching user data:", err);
      res.status(500).json({ error: "Internal Server Error" });
    });
});
```

Along with this route we have a new authMiddleware: here is the code to verify the session. If we have a session and we have a user for that session, then move them along to getting the information.

```
module.exports = (req, res, next) => {
    if (req.session && req.session.user) {
      next(); // User is logged in, proceed to the next middleware/route handler
    } else {
      res.status(401).json({ message: "Unauthorized: Please log in" });
    }
  };
```

From this point on for the session section, just examine the code from the GitHub link. Check the routes, see which html pages they send out. Then check where it goes into the server and how the route handles it. – That should be the end of class for week 10.

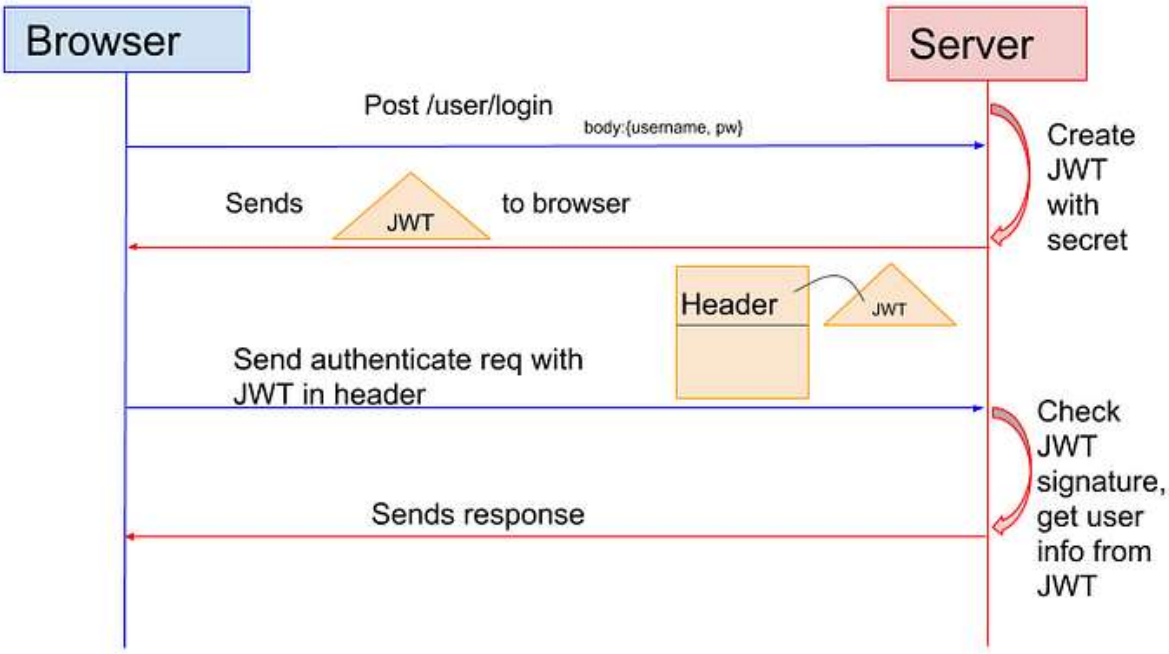
Quick summary:

Sessions are stored on the server, and a cookie is sent to the user. We compare that cookie to our session when we receive the request in sensitive routes. We have now established a user system, authentication and some authorization.

Part 3 – Version 2 – JWT

HarmanSMann/WF1-TEMP-W9-Mongoose at w10-adding-jwt (github.com)

Using JWT is slightly different. You send them a token, essentially a variable, that is stored on their devices. Then they must bundle that token on each subsequent request for access to sensitive routes, like our session version. This is sent in with our requests, through the header of each request we send over.



To class, we will get this package from the npm site:

jsonwebtoken - npm (npmjs.com)

```
npm install jsonwebtoken
```

Look at when we form the token, after we make the login request. We will focus mostly on user route this time around. We need to generate the token during the login request, then send that over for the html to store the token in its storage.

Generating the token:

```
        const token = jwtUtils.generateAccessToken({
```

```
            _id: user._id,
            username: user.username,
            email: user.email,
            first_name: user.first_name,
            last_name: user.last_name,
          });
```

Here is all the jwt code we have added.

1. Generating a token for the user
    a. Jwt.sign
2. Authenticate and verify token
    a. We extract the token from the header,
    b. we need to decode the token
        i. jwt.verify
    c. verify its contents before passing the final judgement

```
const jwt = require('jsonwebtoken');

function generateAccessToken(user) {
  return jwt.sign({
    userId: user._id,
    username: user.username,
    email: user.email,
    firstName: user.first_name,
    lastName: user.last_name
  }, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '30m' });
}

const verifyToken = (req, res, next) => {
  const token = req.header("Authorization")?.replace("Bearer ", ""); // Extract
token from Authorization header
  if (token == "null") {
    return res.status(401).json({ error: "Not Logged in" });
  }
  if (!token) {
    return res.status(401).json({ error: "Unauthorized" });
  }
  try {
    const decoded = jwt.verify(token, process.env.ACCESS_TOKEN_SECRET); // Verify
JWT token
    req.user = decoded; // Attach user information to request object
    next(); // Move to the next middleware
  } catch (err) {
    console.error(err);
    res.status(401).json({ error: "Unauthorized" });
  }
```

```
};

module.exports = {
  generateAccessToken, authenticateToken, verifyToken
};
```

Inside the login.html, we need to take the response from the server and save the token. Here is a sample:

```
fetch('/user/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ email, password }),
      })
      .then(response => response.json())
      .then(data => {
        if (data.token) { // Assuming the response includes a 'token' property
          // Store token in local storage

          localStorage.setItem('token', data.token);// RIGHT HERE
          showSuccessMessage(data.message);
          setTimeout(() => {
            window.location.href = '/bookstore'; // Redirect to homepage after
success message
          }, 2000); // Redirect after 2 seconds
        } else {
          showErrorMessage(data.error);
        }
      })
      .catch(error => {
        console.error('Error logging in:', error);
        showErrorMessage('An error occurred. Please try again later.');
      });
```

To add the token into our use case process:

All html pages that require submission for sensitive information or saving data into our DB, we should pass the token. Here is an example using the book_add.html

```
          const token = localStorage.getItem("token");
          if (!token) {
            alert("You are not logged in. Please log in to add a book.");
            window.location.href = "/user/login"; // Redirect to login page
            return;
```

```
        }

fetch("/bookstore/add", {
            method: "POST",
            headers: {
              "Content-Type": "application/json",
              Authorization: `Bearer ${token}`,
            },
            body: JSON.stringify({
              title: document.getElementById("title").value,
              author: document.getElementById("author").value,
              pages: document.getElementById("pages").value,
              rating: document.getElementById("rating").value,
              genres: document.getElementById("genres").value,
              // Add other fields as needed
            }),
          })
            .then((response) => {
              if (response.status === 201) {
                alert(response.message);
                window.location.href = "/bookstore"; // Redirect to homepage
              } else {
                throw new Error("Failed to add book");
                console.error("Error adding book:", response.error);
              }
            })
            .catch((error) => {
              console.error("Error adding book:", response.error);
              // Handle error scenario
            });
```

Here is the /bookstore/add route after making the changes and implementing jwt

```
router
  .route("/add")
  .get((req, res) => {
    res.sendFile(path.join(__dirname, "../views/book_add.html"));
  })
  .post(verifyToken, validateBook, (req, res) => {
    let book = new Book();
    book.title = req.body.title;
    book.author = req.body.author;
    book.pages = req.body.pages;
    book.genres = req.body.genres;
    book.rating = req.body.rating;
```

```
    book
      .save()
      .then(() => {
        res.status(201).json({ message: "Successfully Added" });
      })
      .catch((err) => {
        console.error(err);
        res.status(500).json({ error: "Internal Server Error" });
      });
  });
```

To add a book, we now have 2 pieces of middleware. Verify the Token and Verify the Contents in the Book form. Another thing we can add now, since we have the token is the user that added the token. I will allow you to do that for the class activity.

You have the header information still, so we can decode the token again and fetch the _id of the user stored inside the token. Reference: the jwtUtil file and the user_router. We also need to modify the book model to include the person who added the book.

```
          const token = jwtUtils.generateAccessToken({
            _id: user._id,
            username: user.username,
            email: user.email,
            first_name: user.first_name,
            last_name: user.last_name,
          });
```

For the final part, how do we log out? We can remove the token from the clients localstorage

You can add an event listenter in the html to remove the token from the localstorage

```
              authButton.addEventListener("click", function () {
                // Implement logout logic here (clear token)
                localStorage.removeItem("token");
                authButton.textContent = "Login / Signup";
                userWelcome.style.display = "none";
                window.location.href = "/"; // Redirect to homepage after logout
              });
```

Final Notes:

I have a few additional branches going through the additions in order:

- W10-UserSystem
- W10 adding bcrypt
- W10 adding sessions
- W10 adding jwt

References to material used and the research:

Difference between Session Cookies vs. JWT (JSON Web Tokens), for session management. | by Prashant Ram | Medium
json - JWT vs cookies for token-based authentication - Stack Overflow
JWT authentication: Best practices and when to use it - LogRocket Blog
jaredhanson/passport-google-oauth2: Google authentication strategy for Passport and Node.js. (github.com)
JWT Decoder (jstoolset.com)
Should you use Express-session for your production app? - DEV Community
Session vs Token Based Authentication | by Sherry Hsu | Medium
connect-mongo - npm (npmjs.com)
express-session - npm (npmjs.com)
jsonwebtoken - npm (npmjs.com)