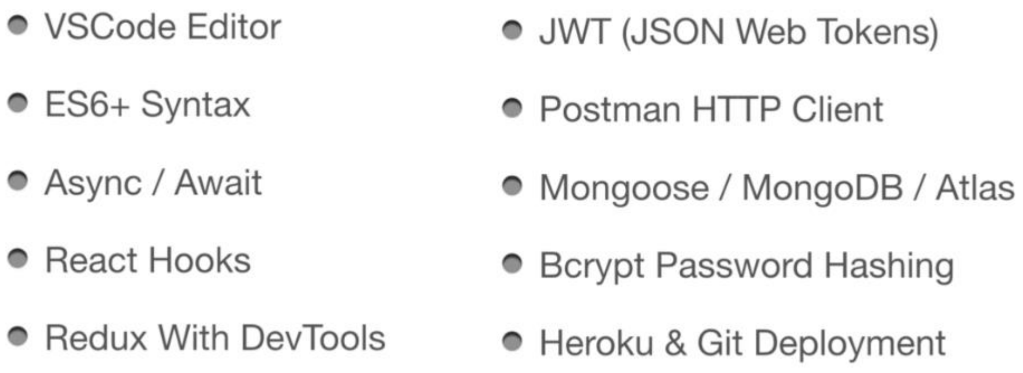
Day 5 Trainee Tasks

Definitions & Understanding MERN Stack Environment



- VSCode Editor
- ES6+ Syntax
- Async / Await
- React Hooks
- Redux With DevTools
- JWT (JSON Web Tokens)
- Postman HTTP Client
- Mongoose / MongoDB / Atlas
- Bcrypt Password Hashing
- Heroku & Git Deployment

MERN Stack

MERN is a popular web development stack that stands for **MongoDB, Express.js, React, and Node.js**. These four technologies work together to build full-stack JavaScript applications, from the backend to the frontend. Here's a brief explanation of each:

- **MongoDB**: A NoSQL database that stores data in a flexible, JSON-like format.
- **Express.js**: A web application framework for Node.js that simplifies server-side development.
- **React**: A front-end JavaScript library for building user interfaces, mainly for single-page applications (SPAs).
- **Node.js**: A runtime environment that allows you to run JavaScript on the server.

Express.js

Express.js is a minimal and flexible web application framework for **Node.js**. It provides a simple interface to:

- Handle routes (URLs).
- Manage HTTP requests (GET, POST, PUT, DELETE).
- Middleware (processes HTTP requests between client and server).
- Handle form submissions, authentication, cookies, and other backend functionality.

Express makes it easier to create REST APIs by providing a simple yet powerful way to structure your web application's backend.

React

React is a JavaScript library developed by Facebook, used for building interactive user interfaces. It is based on components, which are reusable pieces of UI. React allows developers to:

- Create dynamic UIs with state and props.
- Build fast, scalable, and maintainable single-page applications (SPAs).
- Update the DOM efficiently using a virtual DOM.

Node.js

Node.js is a runtime environment that allows JavaScript to run on the server side. Built on Chrome's V8 JavaScript engine, it allows developers to use JavaScript for backend development, making it possible to build full-stack JavaScript applications.

NVM (Node Version Manager)

NVM is a tool that allows you to manage and switch between different versions of **Node.js** on your machine. It is especially useful when working with projects that require different versions of Node. With NVM, you can easily install and switch Node versions without affecting other projects.

Mongoose

Mongoose is an Object Data Modeling (ODM) library for **MongoDB**. It provides a schema-based solution to model your application data, allowing you to define data structure, validation, and relationships between data. Mongoose simplifies interactions with MongoDB, making it easier to work with MongoDB in **Node.js** applications.

MongoDB

MongoDB is a NoSQL database that stores data in a flexible, JSON-like format. It is scalable, fast, and efficient for handling large datasets. MongoDB is document-oriented, meaning data is stored in documents (which are similar to JSON objects), making it suitable for modern web applications that deal with unstructured or semi-structured data.

MongoDB Atlas

MongoDB Atlas is a cloud-based database service that provides an easy way to deploy, manage, and scale MongoDB. It offers features like automated backups, real-time performance monitoring, and global distribution, allowing developers to focus on building applications rather than managing database infrastructure.

Redux

Redux is a predictable state management library for JavaScript applications, often used with React. It helps manage the application state in a central store, which can be accessed by any component. The core concepts of Redux are:

- **Store**: The central state of the application.
- **Actions**: Events that describe something happening in the application (e.g., user login).
- **Reducers**: Functions that specify how the state changes in response to actions.
- **Dispatch**: A method to send actions to the store.

Redux is used to manage complex state logic, especially when an application has multiple components that need to share and update state.

React Hooks

React Hooks are functions introduced in React 16.8 that allow you to use state and other React features without writing a class. The most common hooks are:

- **useState**: Allows you to add state to functional components.
- **useEffect**: Manages side effects (e.g., fetching data, setting up subscriptions).
- **useContext**: Provides access to React's context API to share data between components without passing props manually.

Hooks simplify component logic and make it easier to reuse stateful logic across components.

JWT (JSON Web Token)

JWT is a compact, URL-safe token format used for securely transmitting information between parties. It is commonly used for:

- **Authentication**: After a user logs in, the server generates a JWT and sends it to the client. The client includes this token in subsequent requests to prove its identity.
- **Authorization**: JWT tokens are often used to determine access levels (roles, permissions).

A JWT consists of three parts: a header, a payload (containing claims like user info), and a signature. It is widely used in modern web applications for securing APIs and handling user authentication in a stateless manner.

These components together enable the creation of powerful and scalable web applications using only JavaScript.

NPM (Node Package Manager)

NPM is the default package manager for **Node.js**, which comes bundled with Node.js when you install it. It is used for:

- **Managing Dependencies**: NPM allows developers to install, update, and remove JavaScript libraries (packages) required for their project.
- **Version Control**: It tracks the versions of the libraries, ensuring compatibility and enabling easy updates.
- **Running Scripts**: NPM can also run custom scripts such as starting a development server or building a project.

Why We Use NPM

- To install and manage third-party libraries and tools that help in building applications (e.g., Express, React, Mongoose).
- To automate tasks like running tests, starting a server, and bundling the application.

Postman

Postman is a tool used to test APIs by sending HTTP requests and receiving responses. It simplifies API development and debugging.

Why We Use Postman

- **Testing APIs**: You can send GET, POST, PUT, DELETE, and other types of HTTP requests to endpoints.
- **Debugging**: Postman helps visualize responses and detect issues with API requests.
- **Automation**: You can save API requests and organize them in collections, which can be shared or used in automated tests.

Bcrypt

Bcrypt is a library used for hashing passwords before storing them in a database. It's designed to be slow to brute-force attacks and adds a salt to the hash, making each password unique even if multiple users have the same password.

Why We Use Bcrypt

- **Password Security**: It ensures that user passwords are not stored in plain text in the database. Bcrypt hashes passwords in a way that makes it computationally expensive to crack them.
- **Salting**: It adds random data (salt) to the password before hashing, protecting against rainbow table attacks.

Why We Use MongoDB Atlas

MongoDB Atlas is a cloud-based service that makes it easier to deploy, manage, and scale MongoDB clusters. Some benefits include:

- **Managed Service**: MongoDB Atlas handles database management tasks like backups, monitoring, and scaling.
- **Global Distribution**: You can deploy clusters in various regions globally, allowing for low-latency applications.
- **Automatic Backups**: It provides built-in backup options, ensuring data integrity.

Key MongoDB Terms:

- **Cluster**: A cluster in MongoDB Atlas is a collection of servers (nodes) that host your MongoDB database. These servers replicate data for redundancy and high availability.
- **Database (db)**: A database is a container for collections in MongoDB. You can have multiple databases within a cluster.
- **Collections**: A collection is equivalent to a table in a SQL database. It stores multiple documents (which are JSON-like objects).
- **Documents**: The data itself, stored in a collection, is a document. It's similar to a row in a SQL table but in a JSON-like format.

Commands to Set Up a MERN Stack Project

Initialize a Git Repository

```
git init
```

This command initializes a Git repository, allowing version control for your project.

Install Express.js

```
npm install express
```

Installs **Express.js**, which is used to create the backend server.

Install Nodemon (for auto-restarting server during development)

```
npm install -D nodemon
```

Installs **Nodemon** as a dev dependency (`-D` flag), which automatically restarts the server when files are modified.

Create a Basic File Structure

Create the following structure for your project:

```
/project-root
  /backend
    - app.js
  /frontend
    - src/
```

Set Up Node.js Project

```
npm init -y
```

This command creates a `package.json` file in your project, which will keep track of dependencies and scripts.

Install Other Dependencies

1. **Body-parser** for parsing incoming request bodies:

   ```
   npm install body-parser
   ```

2. **Mongoose** for MongoDB interactions:

   ```
   npm install mongoose
   ```

3. **CORS** to allow cross-origin requests (needed for connecting frontend and backend):

   ```
   npm install cors
   ```

4. **Dotenv** to handle environment variables:

   ```
   npm install dotenv
   ```

Set Up React in Frontend

1. Navigate to the frontend folder and use the following command to set up a React app:

   ```
   npx create-react-app .
   ```

2. Install **Axios** to handle HTTP requests from React:

   ```
   npm install axios
   ```

Sample Workflow for Setting Up a MERN Stack Project

1. **Backend Setup:**

   ```
   git init
   npm init -y
   npm install express mongoose cors dotenv body-parser bcrypt
   jsonwebtoken
   npm install -D nodemon
   ```

2. **Frontend Setup:**

   ```
   cd frontend
   npx create-react-app .
   npm install axios
   ```

3. **Common Development Commands:**
   - Start the backend with Nodemon:

     ```
     nodemon app.js
     ```

   - Start the React frontend:

     ```
     npm start
     ```

Dev Dependencies

Dev dependencies are libraries or tools that are only required during the development process of your application but are not needed in production. These dependencies help with tasks such as testing, debugging, or compiling code but aren't used by the application when it's deployed to production. Examples include **Nodemon** (for auto-restarting the server), **ESLint** (for code linting), and **Jest** (for testing).

How to Install Dev Dependencies

To install dev dependencies, use the `-D` or `--save-dev` flag with `npm`. For example:

```
npm install nodemon --save-dev
```

or shorter:

```
npm install -D nodemon
```

This command adds **Nodemon** to your `package.json` under the `devDependencies` section. It will not be included when you run your project in production.

Middlewares

Middleware functions are functions that execute during the lifecycle of an HTTP request.
In **Express.js**, middleware functions have access to the request (`req`), response (`res`), and the next middleware function in the application's request-response cycle. Middleware functions can:

- Execute code.

- Modify the request and response objects.
- End the request-response cycle.
- Call the next middleware function.

Middleware is used to handle things like logging, authentication, handling errors, parsing request bodies, and more.

Types of Middleware:

1. **Application-Level Middleware**:
   - Bound to the application object using `app.use()`.
   - Runs for every request.
   - Example: Handling request logging.

   ```
   app.use((req, res, next) => {
     console.log('Request URL:', req.originalUrl);
     next();
   });
   ```

2. **Router-Level Middleware**:
   - Works at the router level, used for specific routes.
   - Example: Middleware for a specific route.

   ```
   const router = express.Router();
   router.use('/user/:id', (req, res, next) => {
     console.log('User Route Accessed');
     next();
   });
   ```

3. **Error-Handling Middleware**:
   - Takes four arguments: `err`, `req`, `res`, `next`.
   - Used to catch and handle errors in the application.
   - Example:

   ```
   app.use((err, req, res, next) => {
     console.error(err.stack);
     res.status(500).send('Something broke!');
   });
   ```

4. **Built-in Middleware**:
   - Provided by **Express** to handle common tasks.
   - Examples:
     - `express.json()`: Parses incoming JSON requests.
     - `express.urlencoded()`: Parses incoming requests with URL-encoded payloads.
5. **Third-Party Middleware**:
   - Middleware from external libraries.

    - Examples:
        - `cors`: To enable Cross-Origin Resource Sharing.
        - `body-parser`: For parsing incoming request bodies.
        - `morgan`: HTTP request logger.

HTTP Response Status Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. They are grouped into five categories:

1. 1xx (Informational): The request was received, and the server is continuing to process it.

- **100 Continue**: The initial part of the request was received and the client should continue with the request.

2. 2xx (Success): The request was successfully received, understood, and accepted.

- **200 OK**: The request was successful, and the response contains the requested data.
- **201 Created**: The request was successful, and a resource was created (often used for POST requests).

3. 3xx (Redirection): The client must take additional action to complete the request.

- **301 Moved Permanently**: The requested resource has been permanently moved to a new URL.
- **302 Found**: The requested resource is temporarily located at a different URL.

4. 4xx (Client Errors): The request contains bad syntax or cannot be fulfilled.

- **400 Bad Request**: The server cannot process the request due to malformed syntax.
- **401 Unauthorized**: The request lacks valid authentication credentials.
- **403 Forbidden**: The server understands the request but refuses to authorize it.
- **404 Not Found**: The server could not find the requested resource.

5. 5xx (Server Errors): The server failed to fulfill a valid request.

- **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
- **502 Bad Gateway**: The server, acting as a gateway, received an invalid response from the upstream server.
- **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overloading or maintenance.

Mongoose

Mongoose is an Object Data Modeling (ODM) library for **MongoDB** and **Node.js**. It provides a schema-based solution for modeling your application data. Mongoose helps manage relationships between data, provides schema validation, and makes interacting with MongoDB easier.

Key Features of Mongoose:

- Schema-based data modeling.
- Middleware (hooks) for handling asynchronous pre- and post-processing of data.
- Schema validation to enforce data integrity.
- Query building for efficient database queries.

Mongoose Model & Schema

Mongoose Schema

A **Schema** in Mongoose defines the structure of the documents within a collection in MongoDB. It allows you to define the fields, their types, default values, and validation rules.

Example of a basic schema:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  date: {
    type: Date,
    default: Date.now
  }
});
```

Mongoose Model

A **Model** in Mongoose is a compiled version of the schema, and it provides an interface to interact with the database. Once you define a schema, you compile it into a model and use it to create, read, update, and delete documents in the MongoDB collection.

Example of creating a model:

```
const User = mongoose.model('User', userSchema);
```

You can then use this model to interact with the `User` collection:

```
User.find({}, (err, users) => {
  if (err) console.log(err);
  console.log(users);
});
```

Gravatar

Gravatar stands for "Globally Recognized Avatar." It's a service that allows users to create a globally recognized avatar that will show up across many websites based on their email address. Gravatar is often used in web applications to provide profile pictures for users without requiring them to upload an image.

In an Express.js app, you can generate a Gravatar URL using the user's email address (after hashing it with MD5). Example using the `gravatar` package:

```
const gravatar = require('gravatar');

const avatarUrl = gravatar.url(userEmail, {
  s: '200', // Size
  r: 'pg',  // Rating
  d: 'mm'   // Default image
});
```

Bcrypt, Gensalt & Password Hashing

Bcrypt is a password-hashing library used to securely hash passwords before storing them in a database. It applies a one-way hashing algorithm, meaning the original password cannot be easily retrieved from the hashed value. It also adds a **salt** to the password to protect against certain attacks like rainbow table attacks.

Bcrypt Gensalt

`bcrypt.genSalt()` is used to generate a salt that will be added to the password before hashing. Salting makes sure that even if two users have the same password, their hashed passwords will be different.

Password Hashing

`bcrypt.hash()` is used to create the hashed version of the password by combining the plain password and the generated salt. The result is a hash that can be securely stored in the database.

Example of hashing a password with Bcrypt:

```
const bcrypt = require('bcrypt');

const saltRounds = 10;
bcrypt.genSalt(saltRounds, (err, salt) => {
  bcrypt.hash('user_password', salt, (err, hash) => {
    // Store hash in the database
    console.log(hash);
  });
});
```

For verifying the password during login:

```
bcrypt.compare('user_password', hash, (err, result) => {
  if (result) {
    console.log('Passwords match');
  } else {
    console.log('Passwords do not match');
  }
});
```

Registration, Login, and Validation Logic in Express.js

In a typical Express.js application, the registration and login process works by validating user inputs, securely storing passwords, and authenticating users during login.

1. Registration Process:

The registration process involves:

1. **Input Validation**: Validate user inputs such as email, password, and name to ensure they are valid (e.g., using validation libraries like `express-validator`).
2. **Check for Existing Users**: Before creating a new user, check if the email already exists in the database.

3. **Password Hashing**: Hash the user's password using Bcrypt before storing it in the database.
4. **Save User**: Store the new user's data (including the hashed password) in the database.

Example Registration Route:

```
const bcrypt = require('bcrypt');
const User = require('../models/User');

app.post('/register', async (req, res) => {
  const { name, email, password } = req.body;

  // Validate input here (e.g., using express-validator)

  try {
    // Check if user already exists
    let user = await User.findOne({ email });
    if (user) {
      return res.status(400).json({ msg: 'User already exists' });
    }

    // Create a new user
    user = new User({ name, email, password });

    // Hash the password
    const salt = await bcrypt.genSalt(10);
    user.password = await bcrypt.hash(password, salt);

    // Save user to database
    await user.save();

    res.status(201).json({ msg: 'User registered successfully' });
  } catch (err) {
    res.status(500).json({ msg: 'Server error' });
  }
});
```

2. Login Process:

The login process involves:

1. **Input Validation**: Validate the login credentials (email and password).
2. **Check for User**: Check if the user exists in the database using their email.
3. **Compare Password**: Use `bcrypt.compare()` to check if the entered password matches the hashed password stored in the database.
4. **Generate JWT**: If the password is correct, generate a JSON Web Token (JWT) to authorize the user for future requests.

Example Login Route:

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
```

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  // Validate input here (e.g., using express-validator)

  try {
    // Check if user exists
    let user = await User.findOne({ email });
    if (!user) {
      return res.status(400).json({ msg: 'Invalid credentials' });
    }

    // Compare password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(400).json({ msg: 'Invalid credentials' });
    }

    // Generate JWT (JSON Web Token)
    const payload = {
      user: {
        id: user.id
      }
    };
    jwt.sign(payload, 'secretToken', { expiresIn: 3600 }, (err, token) => {
      if (err) throw err;
      res.json({ token });
    });
  } catch (err) {
    res.status(500).json({ msg: 'Server error' });
  }
});
```

3. JWT Validation Middleware:

Once a user logs in, you want to protect certain routes that require authentication. You can use a middleware to check if a valid JWT token is provided with each request.

Example JWT Middleware:

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

const auth = (req, res, next) => {
  // Get the token from the header
  const token = req.header('x-auth-token');

  // Check if token exists
  if (!token) {
    return res.status(401).json({ msg: 'No token, authorization denied' });
  }

  // Verify the token
  try {
    const decoded = jwt.verify(token, 'secretToken');
    req.user = decoded.user;
    next();
  } catch (err) {
    res.status(401).json({ msg: 'Token is not valid' });
  }
};

module.exports = auth;
```

Flow Summary:

1. **Registration**:
   - User submits details.
   - Server validates input and checks for existing user.
   - Password is hashed using Bcrypt.
   - User data (with hashed password) is saved to the database.
2. **Login**:
   - User submits credentials.
   - Server validates input and checks if the user exists.
   - Password is compared using Bcrypt.
   - If correct, a JWT is generated and sent to the user.
3. **Protected Routes**:
   - For protected routes, the server checks for a valid JWT in the request headers.
   - If the JWT is valid, access is granted; otherwise, access is denied.

This system ensures that user credentials are securely managed, and only authenticated users can access protected resources in the application.