Lesson 5: Learn Express Authentication

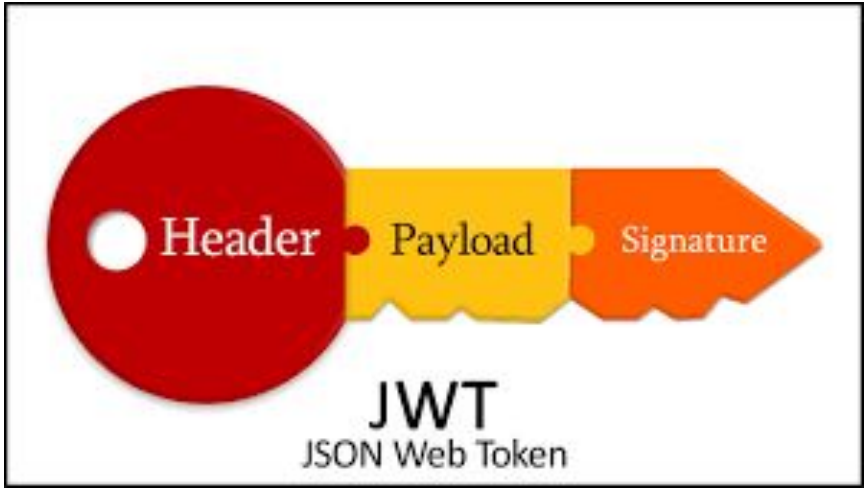
Learn to authenticate users with JSON Web Tokens (JWT) and create custom authentication middleware

What is authentication on the backend?

- Authentication is the process of verifying the identity or pseudo-identity of a user/system.
- In the context of app development, authentication is typically used to control access to protected resources on the backend.
- Our API is the gateway to all of our “backend resources” (our Database / Lightning node) so this is where we need to implement controls to determine who has access, and what they have access to.
- Without authentication, anyone could potentially access your sensitive information or perform actions on your behalf without your permission.
- One popular way to implement authentication is through the use of JSON Web Tokens (JWT).

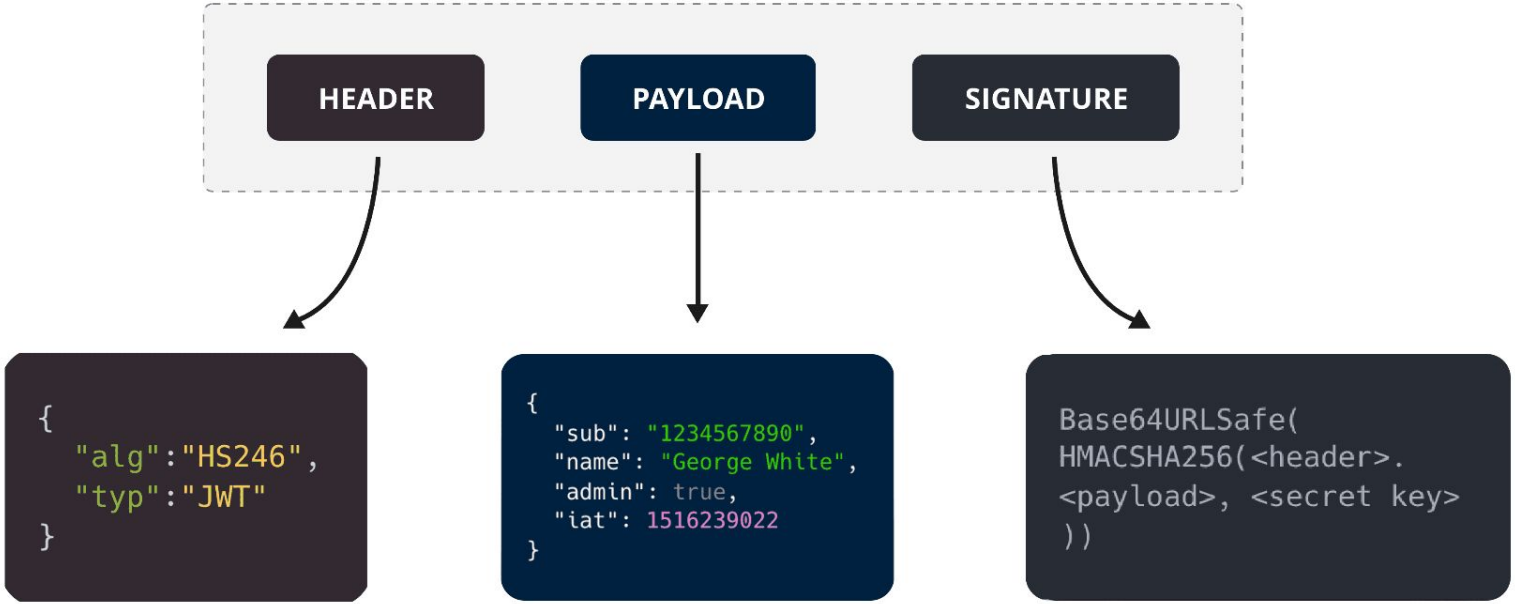
What are JSON Web Tokens? (JWT)

- JWT is an open standard for securely transmitting data between parties as a JSON object.
- JWTs can be used to authenticate users and authorize access to protected resources in a stateless and scalable way.
- JWTs can be easily decoded and verified by backend servers which is why it's a popular method for authentication
- JWTs consist of three parts: a header, a payload, and a signature.



The structure of a JWT

- The header specifies the algorithm used to generate the signature
- the payload contains user data
- the signature verifies the authenticity of the token.



JWT debugger

https://jwt.io/

Let's setup jsonwebtoken in our server

- Stop your server if it's running and execute `npm install jsonwebtoken bcryptjs` in your terminal

- Import `const jwt = require("jsonwebtoken");` at the top of your usersRouter.js file
- Import `const bcrypt = require("bcryptjs");` at the top of your usersRouter.js file (we'll get to this module a little later)

Now we will build a generateToken function for creating new JWT's that we can pass back to the user on login for authentication at the bottom of usersRouter.js

Build a generateToken function for JWT's

```
// Function to generate a JSON Web Token (JWT) for a given user
function generateToken(user) {
 // Define the payload to be included in the token, containing user data
 const payload = {
   id: user.id,
   username: user.username,
   admin: user.admin,
 };
  // Get the JWT secret from an environment variable, or use a default value
 const secret = process.env.JWT_SECRET || "Satoshi Nakamoto";

 // Define the options for the JWT, including the token expiration time
 const options = {
   expiresIn: "1d",
 };

 // Generate and return the JWT using the payload, secret, and options
 return jwt.sign(payload, secret, options);
}
```

Storing secrets on our server with environment variables

1. **Purpose of Environment Variables**: Store sensitive data and app configurations separate from code to enhance security and enable easy adjustments without altering the source code.

2. **.env File**: A local file that stores key-value pairs for environment variables, kept out of version control to prevent unauthorized access to sensitive information.

3. **dotenv Package**: A popular npm package that loads environment variables from the .env file into the process.env object, enabling access to these variables throughout the Express application.

4. **Accessing Environment Variables**: Use process.env.VARIABLE_NAME to retrieve the value of an environment variable (e.g., process.env.PORT for the server port number).

5. **Best Practices**: Always include .env files in your .gitignore to prevent unintentional exposure, and use separate environment variables for different environments (development, production, staging) to ensure proper configurations and security.

Create a .env file at the root of our project

- Inside of your project open up the terminal and create a new file called .env by either running touch .env or using your code editor to add the file
- Open up .env and add the two environment variables we currently have in our project (PORT and JWT_SECRET)

```
# secret variable for the port of our server (used in index.js)
PORT=5501
# secret variable for our JWT secret (used in usersRouter.js)
JWT_SECRET=keepitsecretkeepitsafu
```

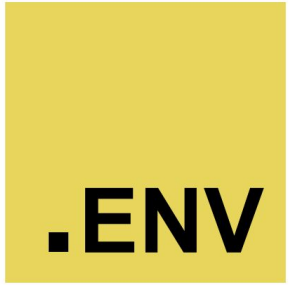
Accessing environment variables with dotenv

dotenv is a package that allows us to load environment variables from a .env file.

dotenv

Dotenv is a zero-dependency module that loads environment variables from a `.env` file into `process.env`. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

build inaccessible | build failing | npm v16.0.3 | code style standard
coverage 100% | license BSD-2-Clause | Conventional Commits 1.0.0
Featured on Openbase | Limited Edition Tee Original
Limited Edition Tee Redacted



- To use dotenv, we need to install the package and require it in our code.
  - Run `npm i dotenv`

- Once we have loaded our environment variables, we can access them in our code using process.env.KEY.

Update index.js to load "PORT" env variable

```
const dotenv = require("dotenv")
```

Import dotenv at the top of your index.js

```
dotenv.config()
```

Initialize your environment variables with dotenv.config() right below your imports in index.js

```
// Set the server to listen on the provided port, or 5500 if no port is specified
const PORT = process.env.PORT || 5500;

server.listen(PORT, () => {
 // Log a message to the console when the server starts listening
 console.log(`Server listening on port ${PORT}`);
});
```

Now your port should be 5501 when you restart the server

Using bcryptjs to finish our JWT setup

What is bcryptjs and why are we using it?

- bcryptjs is a library for hashing passwords using the bcrypt algorithm.
- The bcrypt algorithm is a one-way hash function that converts a password into a fixed-length string of characters that is unique to that password.
- By hashing passwords using bcrypt, we can store them securely in a database without exposing the plaintext password.
- When a user registers, we hash their password using bcrypt and store the hashed password in the database.
- When a user logs in, we retrieve their hashed password from the database and compare it to the plaintext password they provided using bcrypt.compareSync() method. If the hashed and plaintext passwords match, then the user is authenticated.
- Using bcrypt adds an additional layer of security to our application by protecting user passwords in case of a data breach.

Now lets update our /login endpoint to use bcryptjs and our new generateToken function

Time to do some hashing!

`*Brrrrr*`

```
router.post("/login", (req, res) => {
 // Extract the username and password from the request body
 const { username, password } = req.body;
 // Placeholder user object - later we will fetch the real user from the database
 const DBuser = {
   username: "test",
   password: "pass1",
 };

 // Hash the password from the request body using bcrypt
 // Later we will compare this hash to the hash stored in the database
 // But for now, we will just do it manually
 const hashedPassword = bcrypt.hashSync(DBuser.password, 14);

 // Check if the user exists and the password matches using bcrypt
 if (DBuser && bcrypt.compareSync(password, hashedPassword)) {
   // Generate a JSON Web Token (JWT) for the user
   const token = generateToken(DBuser);
   // Send a success response with the JWT and user data
   res
     .status(200)
     .json({ message: `Welcome ${DBuser.username}!`, token, DBuser });
 } else {
   // Send an error response if the credentials are invalid
   res.status(401).json({ message: "Invalid credentials" });
 }
});
```

Considering auth / access for the pleb wallet backend

For any backend we build it's important to ask ourselves some basic questions around security / access to our server:

- what resources is our backend is handling?
- which of those resources do we need to keep secure?
- who can request from our server?
- what can they get by requesting our server?
- What happens if the server goes down?
- What happens if the data gets corrupted?

Who will have access to our server and what resource are we hosting

What resources are we hosting:

- User data
- Transaction data
- Lightning node access

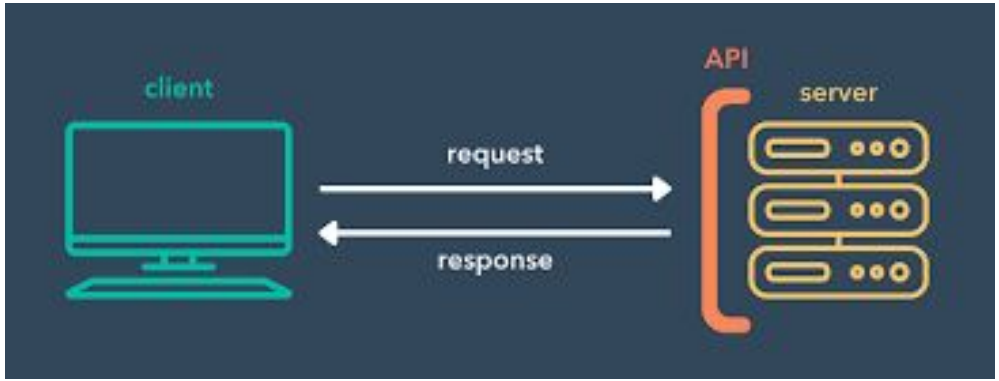
Who will have access and what access will they have:

- Anyone: can look at our wallet balance / transaction list
- Logged in users: Can create an invoice
- Only me: Can pay an invoice

How do we implement this authentication?

In general our API endpoints are the gateways into our backend, basically any functionality or logic that can be executed from the users side will pass through endpoints we open up.

And how do we intercept logic in between the request/response of our server? Middleware!



Authentication middleware

We need to build an auth middleware for each unique permission set that our server will handle

This will include:

- Middleware for regular logged in users they will get access to create invoices
- Middleware for admins (me) will get access to pay invoices
- No auth/middleware necessary for rendering the main wallet page and seeing balance / transactions list

Example of custom middleware in Express:

```
// Define the middleware function
const myMiddleware = (req, res, next) => {
   // Perform some checks or modifications to the request or response objects
   // For example, you could add a custom header to the response:
   res.setHeader('X-Custom-Header', 'Hello, world!');
   // Call the next middleware function in the chain
   next();
 };


// Use the middleware function on a specific endpoint
app.get('/my-endpoint', myMiddleware, (req, res) => {
   // Handle the request as normal
   res.send('Hello, world!');
 });
```

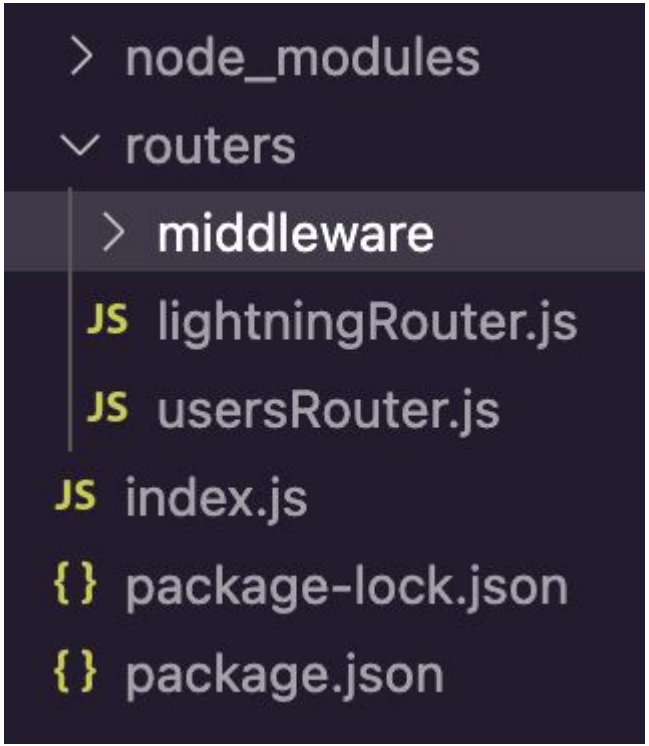
Define a middleware function with the correct parameters (req, res, next)

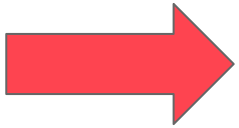
Place our new middleware function in between the route and the (req, res) parameters

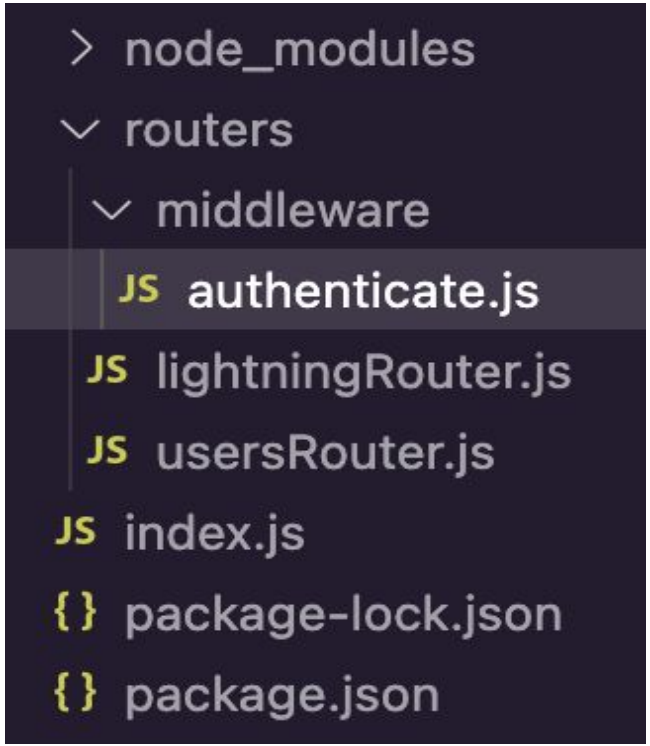
What will our two auth middlewares do?

- **authenticateMiddleware:**
  - Will check the request for an “authentication” header
  - Will pull the value out of that header (the value should be a JWT given to the user from their login)
  - Will attempt to parse and validate the JWT using the ‘jsonwebtoken’ npm module
  - If parsed and validated; let the logic in the endpoint continue executing ie: **next()**
  - If the JWT is not parsed or validated; return a status code 401 (unauthorized) in the middleware which will cancel the execution / response from the endpoint that middleware was placed on
- **authenticateAdminMiddleware:**
  - Same flow as authenticateMiddleware ~
  - Except while parsing the JWT for adminMiddleware we will also check the “payload” for a secret “adminKey” key/value pair that we will create in secret to allow us to have exclusive access to the ‘payInvoice’ endpoint
  - If the JWT is parsed, validated, and the adminKey is present and valid; let the logic in the endpoint continue executing ie: **next()**
  - If the JWT is not parsed or validated, or the adminKey is not present or not valid; return a status code 401 (unauthorized) in the middleware which will cancel the execution / response from the endpoint this middleware is placed on

Create middleware directory / authenticate.js







Create authenticate.js middleware

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

// Exporting a middleware function that takes the arguments req, res, and next
module.exports = (req, res, next) => {
 // Extracting the token from the Authorization header of the request
 const token = req.headers.authorization;
 // Extracting the secret used to sign the JWT from an environment variable or using a default value
 const secret = process.env.JWT_SECRET || "Satoshi Nakamoto";
 // Checking if a token was provided in the request header
 if (token) {
   // Verifying the token using the provided secret
   jwt.verify(token, secret, (err, decodedToken) => {
     if (err) {
       // If the token is not verified, return a status code of 401 and an error message
       res.status(401).json({ message: "Not Allowed", Error: err });
     } else {
       // If the token is verified, execute the next middleware or endpoint logic
       next();
     }
   });
 } else {
   // If no token was provided, return a status code of 401 and a message
   res.status(401).json({ message: "No token!" });
 }
};
```

Add authenticate middleware to createInvoice

1. import your new authenticate middleware into lightningRouter.js
2. Add it to the createInvoice endpoint in between the endpoint url and the req/res callback

```
const authenticate = require("../routers/middleware/authenticate");

...

// POST required info to create an invoice
router.post("/createInvoice", authenticate, (req, res) => {
 const { value, memo } = req.body;

 console.log(value, memo);

 res.status(200).json({ message: "I'm alive!" });
});
```

Testing the authenticate middleware

We can now call our createInvoice endpoint with no authentication

We should be denied with a 401

“No token!”

POST http://localhost:5500/lightning/createInvoice Send

401 Unauthorized 6.07 ms 23 B

JSON Auth Query Headers 1 Docs

Preview Headers 23 Cookies Timeline

```
1 ...
```

```
1 {
2   "message": "No token!"
3 }
```

Getting authentication from the API

Now let's make a post request to /login with our hardcoded DBuser credentials to get a JWT authentication token

POST http://localhost:5500/users/login Send

JSON Auth Query Headers 1 Docs

```
1 {
2     "username": "test",
3     "password": "pass1"
4 }
```

200 OK 3.57 s 238 B 17 Minutes Ago

Preview Headers 23 Cookies Timeline


```
1 {
2   "message": "Welcome test!",
3   "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3QiLCJpYXQiOjE2Nzk
0Mzg1MjEsImV4cCI6MTY3OTUyNDkyMX0.PV5kWMstWFbjb_EHw98kEdZ5GfprxNHMC_sDLF-
zAMM",
4   "DBuser": {
5     "username": "test",
6     "password": "pass1"
7   }
8 }
```

Testing an authenticated request to /createInvoice

Take the token from the successful request to /login and add it as a header with the key “authorization”

POST http://localhost:5500/lightning/createInvoice

Send

200 OK 64.3 ms 24 B

JSON Auth Query Headers 2

Add Delete All Toggle Description

Content-Type	application/json
authorization	w98kEdZ5GfprxNHMC_sDLF-zAMM

Preview Headers 23 Cookies Timeline

```
1 {
2   "message": "I'm alive!"
3 }
```

Create your authenticateAdmin middleware

Almost there!

Create authenticateAdmin.js part #1

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

module.exports = (req, res, next) => {
 // Extracting the token from the request header
 const token = req.headers.authorization;

 // Setting up the JWT secret for token verification
 const secret = process.env.JWT_SECRET || "Satoshi Nakamoto";

 // If token is present, attempt to verify it using the JWT module
 if (token) {
   jwt.verify(token, secret, async (err, decodedToken) => {
     // If token is not verified, return 401 error
     if (err || !decodedToken) {
       res.status(401).json({ message: "Error with your verification"  });
     } else {
```

Create authenticateAdmin.js part #2

```
      // If token is verified, find the user using their username from the database
      // Placeholder user object - later we will fetch the real user from the database
      const user = {
        username: "test",
        password: "pass1",
        adminKey: 1234,
      };
      // Extracting admin key from user object if it exists
      const adminKey = user?.adminKey?.toString() ?? "";
      // Checking if extracted admin key matches with the one in env variables
      if (adminKey !== process.env.ADMIN_KEY) {
        // If admin key does not match, return 401 error
        res.status(401).json({ message: "Must be an admin" });
      } else {
        // If admin key matches, let the endpoint continue executing
        next();
      }
    }
  });
} else {
  // If no token is present, return 401 error
  res.status(401).json({ message: "No token!" });
}
};
```

Add the new ADMIN_KEY env variable to .env

```
# secret variable for the port of our server (used in index.js)
PORT=5501
# secret variable for our JWT secret (used in usersRouter.js)
SECRET=keepitsecretkeepitsafu
# secret variable for admin key (used in authenticateAdmin.js)
ADMIN_KEY=1234
```

Add authenticateAdmin to payInvoice endpoint

1. import your new authenticate middleware into lightningRouter.js
2. Add it to the createInvoice endpoint in between the endpoint url and the req/res callback

```
const authenticateAdmin = require("../routers/middleware/authenticateAdmin");

...

// POST an invoice to pay
router.post("/payInvoice", authenticateAdmin, (req, res) => {
 const { payment_request } = req.body;

 console.log(payment_request);

 res.status(200).json({ message: "I'm alive!" });
});
```

Testing the authenticateAdmin middleware

We can now call our payInvoice endpoint with no authentication

We should be denied with a 401

“No token!”

POST http://localhost:5500/lightning/payInvoice

Send

401 Unauthorized 133 ms 23 B

Body Auth Query Headers Docs

Preview Headers 23 Cookies

Add Delete All Toggle Description

Name Value

```
1 {
2   "message": "No token!"
3 }
```

Testing an authenticated request to /payInvoice

- Take the token from your previous login session
- Add a header named “authorization”
- Add the token as the value of the header

POST http://localhost:5500/lightning/payInvoice Send 200 OK 3.52 ms 24 B

Body Auth Query Headers 2 Docs

Preview Headers 23 Cookies Timeline

```
1 {
2   "message": "I'm alive!"
3 }
```

Review

Our server authentication at a high level

Develop and apply JWT, verify user identities, and regulate permissions using middleware.

- **Authentication using JSON Web Tokens (JWT)**
  - Verify user identity and control access to backend resources
- **Structure of JWT**
  - Header, payload, and signature for secure data transmission
- **Environment Variables and dotenv package**
  - Securely store sensitive data and app configurations
- **Bcryptjs for password hashing**
  - Securely store hashed passwords and compare during login
- **Custom Authentication Middleware**
  - Control access to endpoints based on user roles and permissions

Resources

- JSON Web Token (JWT) - https://jwt.io/ - Official website for JSON Web Tokens, including information on how they work, and a debugger to test and verify JWTs.
- bcryptjs - https://www.npmjs.com/package/bcryptjs - An npm package for hashing passwords using the bcrypt algorithm. This library provides an easy way to hash and compare passwords in Node.js applications.
- dotenv - https://www.npmjs.com/package/dotenv - An npm package for loading environment variables from a .env file into the process.env object, enabling access to these variables throughout the Express application.
- Introduction to Middleware in Express - https://expressjs.com/en/guide/using-middleware.html - The official guide to middleware in Express, including how to create custom middleware functions and how to use them in your application.