



TASK

Express - Middleware

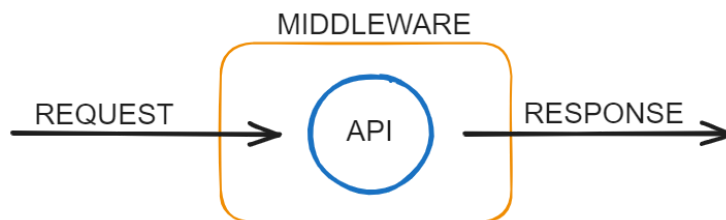
[Visit our website](#)

Introduction

WELCOME TO THE EXPRESS MIDDLEWARE TASK!

Think of middleware as being the middleman between the request object and the response object with any route or endpoint implemented in an Express application. Simply put, middleware acts as a bridge between an incoming request and a final response sent back to the client. It allows you to execute customer logic or perform certain actions on the request and response objects before they reach the final destination.

Middleware refers to functions that are executed in the “middle” of the request-response cycle.



You can implement a lot of custom middleware in a specific endpoint to check any particular portions of data in the request object and modify the response object to suit the requirements of your application. You can look at middleware as a pipeline where you have an input and an output.

WHAT IS CUSTOM MIDDLEWARE?

With the ExpressJS library, there is a multitude of middleware ready for any application, which a developer can use to their advantage. For example, the **`express.json()`** function/middleware can be used to parse any incoming requests to a JSON object. This has the benefit of adding validation of your own to any requests from a particular user to a specific endpoint.

Why Would You Want To Implement Custom Middleware?

You'd implement custom middleware for any functionality you want to add to your application, such as:

- To validate data received from a specific user and manipulate the data according to the application's needs.
- To ensure the data sent to the application is secure and in a format that the database can use to ensure data integrity.

Installation

Using VS Code, open up the example folder located in your task directory. You'll find an existing Node application that also makes use of middleware. We will take a look at this example and see how middleware is used on an Express server.

To install the application run the following command:

```
npm install
```

After the installation is complete we will need to start the application with the following command:

```
npm start
```

Notice that the application already has **nodemon** installed. This allows the server to automatically restart as you make changes to the code.

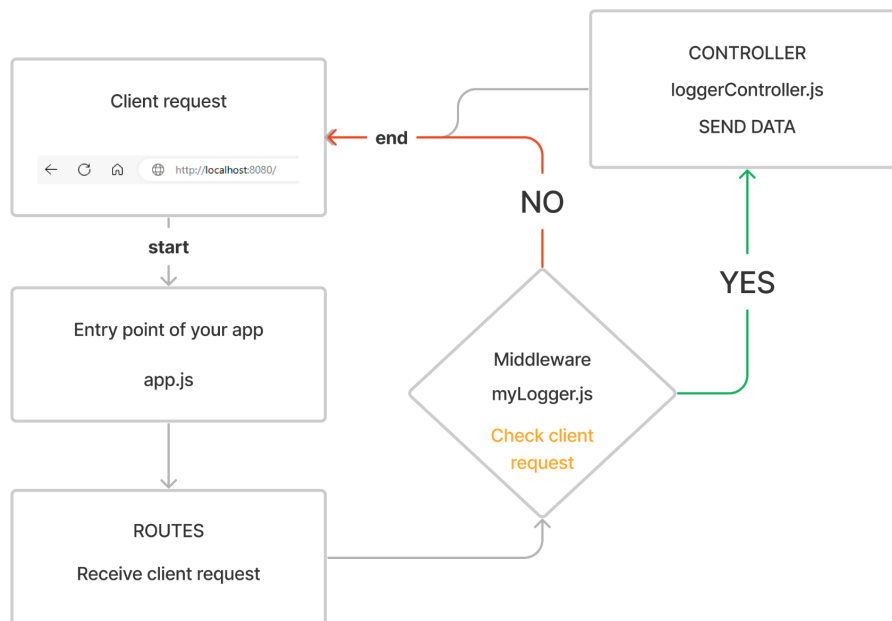
```
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Application up and running on port: 8080  
□
```

Separation of Concerns and Organisation

In the example project, you will also notice that there are several folders. When defining the file structure for a Node and Express application, it is common practice to separate your code into different files. This is a great way to improve a program's maintainability and makes your code easier to work with.

HOW THE SERVER CODE FLOWS

Before we start, we need to understand how the code actually flows when a client makes a request to our server. This will help us get a better understanding of how middleware can be used in our server application. Take a moment to examine the diagram below. Later, we will go through each of these files to examine their code further.



EXAMPLE OF CUSTOM MIDDLEWARE

NOTE: As we go through these examples, you can always refer to the code example documentation for further explanation.

In **app.js** you can see how we've implemented a port variable.

```
//Listening on port 8080
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

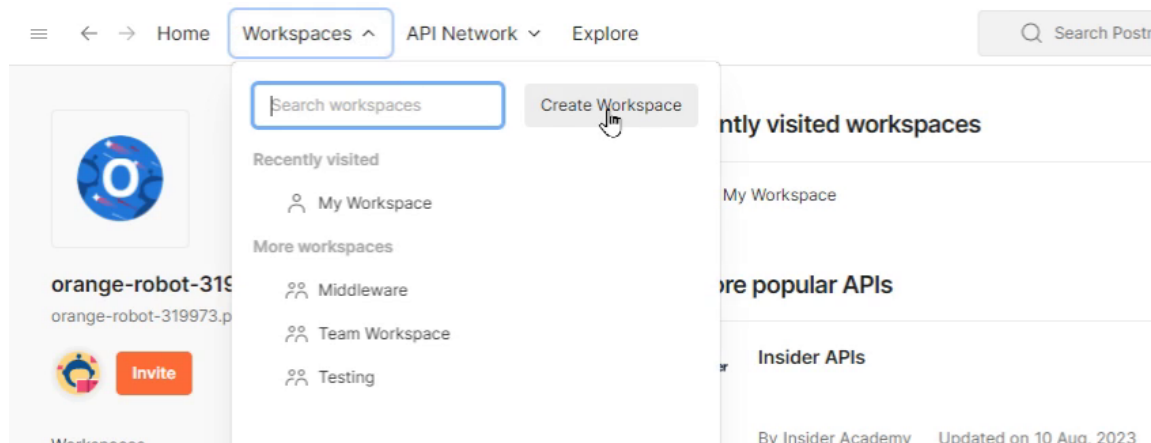
process.env.PORT: This part of the code checks whether there is an environment variable named **PORT** set. This is common practice when deploying applications to production environments (such as cloud services like Heroku or AWS) where the hosting platform dynamically sets the port number through environment variables.

For this example, we will connect to our port using **http://localhost:8080**. You can also open this link in your browser to see the message sent by the controller. We are going to use the popular API application, Postman, to test our API endpoints.

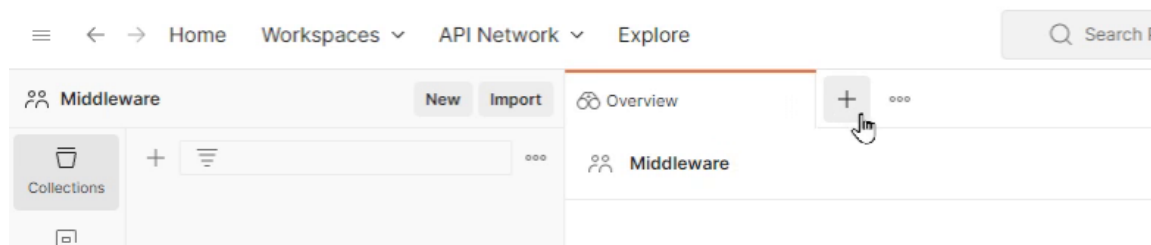
Postman will act as the front-end application, allowing us to interact with our API without the need to create a front-end application or write any code. As such, Postman is a great tool for testing your server endpoints and middleware functions.

USING POSTMAN

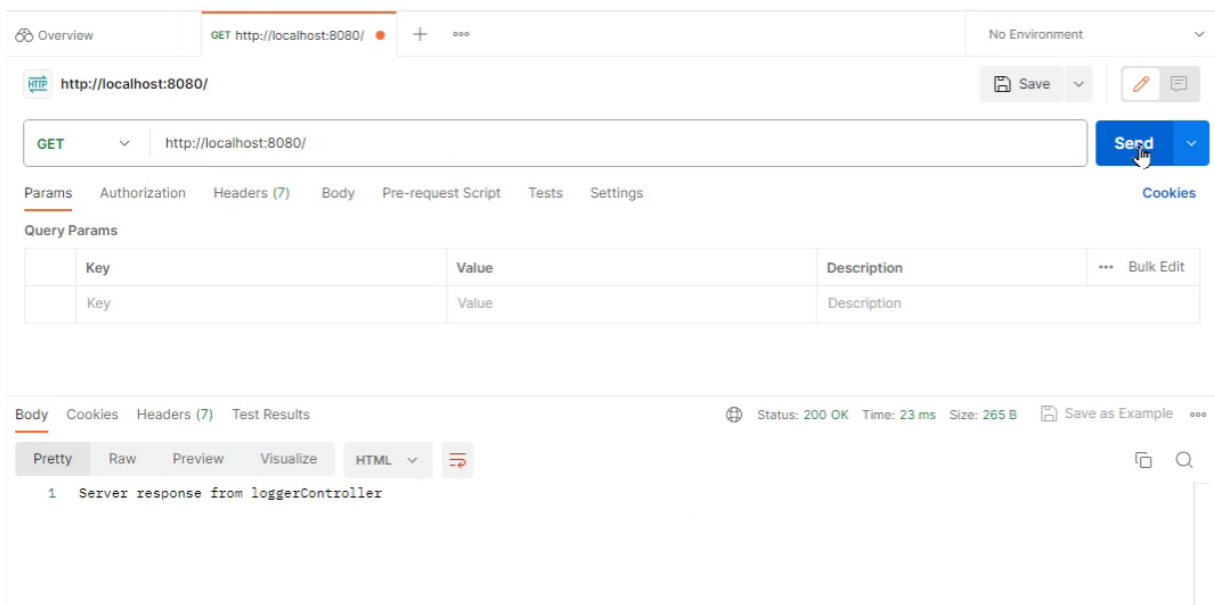
When you open Postman, you will need to create a new Workspace or use an existing one, as shown below.



Next, we will simply open up a new tab where we will connect to our API.



Whenever we create a new tab, the default request will be set to “GET” in the address bar at the URL provided in your **app.js** file. After you click the send button, notice how a message will appear at the bottom. This is the same message we defined inside of our controller.



You might not notice it right away, but our middleware has already executed before the message in Postman appeared.

Let's take a look at the console in VS Code.

```
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Server is listening on port 8080
MIDDLEWARE LOGGED
CONTROLLER LOGGED
```

The **MIDDLEWARE LOGGED** was executed first, followed by the **CONTROLLER LOGGED**. You will find these two core components inside their respective folders, **myLogger.js** and **loggerController.js**.

This means that the middleware must execute before the controller's code is executed. This effectively enables us to modify and inspect user requests before they are passed to the controller.

If you look at the flow chart provided earlier, you will observe that middleware serves as an excellent approach to inspecting user-request data before executing the code within our controllers, which typically interacts with databases or makes requests to external sources and immediately sends them to the client. Middleware acts as a valuable tool to ensure the authenticity of the current user making the request, or to verify if the ongoing request aligns with all specified criteria. Consider it as a **security checkpoint**. This approach relieves the controller from handling these evaluations and enables it to focus solely on execution, thereby providing the requested data in response to the client. However, if the

user's request fails to meet any of the expected requirements, the middleware can respond appropriately to the client with an error or alternative message. In the example below, the middleware function only logs a string to the console and then proceeds to call the next function on the round, the controller function.

```
// middleware/myLogger.js
const loggerMiddleware = function (req, res, next) {
  console.log('MIDDLEWARE LOGGED')

  //Calls the next function
  next()
}

// routes/myLoggerRoute.js
app.get('/', loggerMiddleware, loggerController)
```

The Next Function

When writing custom middleware, you will always use the **next** function. The **next** function is a callback function that is passed as a parameter to the middleware function. It is used to pass control from the current middleware to the next middleware or route handler in the sequence.

As demonstrated in the example above, this essentially means that the middleware **next** callback function tells Node that the middleware function has completed its task and the subsequent function (the **loggerController**) on the route should be executed.

USING JWT AS MIDDLEWARE

The fun will begin momentarily! For the purposes of this lesson, we will use a **JSON-Web-Token** (JWT, pronounced “jot”) to authenticate a user before giving them access to our API data. (Although the T in JWT stands for token, it is nonetheless more common to talk about “JWT tokens”, rather than “JWTs” or “JW tokens”.)

What is JWT?

Think of a JWT token as a digital *passport* or *ID*. It's a way to make sure data is real and safe when it moves between different places on the Internet.

This passport/token has three parts: a front cover (header) that says what kind of passport it is, a page inside (payload) where you can write important information, and a stamp (signature) to prove it's real.

When someone gets this digital passport/token, they can use it to show who they are without constantly asking for authority or access. This is great for logging into websites or using apps, because the user can prove their identity without always checking back with the main server.

One great benefit of using a JWT token is that it is tamper-proof. If anyone tries to modify the contents inside the passport, it won't work any more.

Installing JWT

Use the temple example project for this.

First, we are going to install the **JSON-Web-Token** using the following command:

```
npm install express jsonwebtoken
```

JWT Authentication process

Before we begin building secure routes using JWT, we need to understand the process to successfully incorporate JWT into our server. This means we should know when a token is created during a user's interaction with the server, as well as when it is used.

- 1. Sign-up:** The user signs up by providing their username and password. During this process, their credentials are stored securely in the database. At this point, no token is generated.
- 2. Log-in:** When the user logs in using their credentials, the server checks whether the username and password are correct. If they are, the server generates a JWT token for the user. This token contains the user's unique identifier (usually the user ID) and possibly some additional information, like roles or permissions.
- 3. Sending the token:** The server sends this JWT token back to the client (browser) as a response to the successful login. The client stores the token securely, typically in browser cookies or local storage.
- 4. Accessing protected routes:** When the user tries to access a protected route on the server (for example, a user dashboard), **the client includes the**

token in the request's header. The protected routes are guarded by the `jwtMiddleware` middleware that verifies the token's authenticity and extracts the user's information from it.

5. Verification and authorization: The server checks whether the token is valid (i.e., hasn't been tampered with and hasn't expired) and extracts the user's ID from it. Using this ID, the server can perform authorisation checks (e.g., whether the user has the right permissions to access a resource).

6. Response: If the token is valid and the user is authorised, the server responds with the requested resource or data.

Now we'll create the files in their respective folders, as shown below. Some of them will already be there. The files coloured green in the code sample are the ones you will need to create.

```
Root ---
> controllers
  > userDB.js
  > userController.js
  > loggerController
> routes
  > loginRoute.js
  > myLoggerRoute.js
> middleware
  > myLogger.js
app.js
```

For dummy data, the **userDB.js** will simulate user data stored in a database like MongoDB.

```
// controllers/userDB.js
const userInformation = [
  { id: 1, username: 'user1', password: 'password1' },
  { id: 2, username: 'user2', password: 'password2' },
  // ... other user data
];

// Export the userInformation array to be used in userController.js
module.exports = userInformation;
```

Next, we will set up the controller to:

- Find the user in the simulated database (userDB).
- If the user exists, create a JWT token.
- Send the token to the client.

```

// userController.js
// Require the user data from simulated database
const userInformation = require('./userDB');
const jwt = require('jsonwebtoken');

// Define the login controller functions
const userController = (req, res) => {

  //Get the username and password from the request query
  const { username, password } = req.query;

  //Find the user in the database - returns a boolean
  const user = userInformation.find(user =>
    user.username === username && user.password === password
  );

  //If the user is not found, return an error message - end the request
  if (!user) {
    return res.send('Incorrect user credentials');
  }

  // Create a JWT token - payload
  payload = {
    'name': username,
    'admin': false
  }
  // sign(payload, secretOrPrivateKey, [options, callback])
  const token = jwt.sign(JSON.stringify(payload), "HyperionDev",{
    algorithm: 'HS256'
  });

  //The res.send() function sends a string to the client
  console.log(`User ${username} logged in`)
  res.send({ message: `Welcome back ${username}`, token: token })
}

//export controller functions to be used on the myLoggerRoute.js/routes
module.exports = {
  userController,
};

```

The purpose of this controller is to handle only the login process. When a user attempts to log in, the controller will check whether the username and password match the data in the **userInformation** variable.

If there's a match, the code will continue to run, and will generate a JWT token using the payload and **sign()** method:

```

// sign(payload, secretOrPrivateKey, [options, callback])
jwt.sign(JSON.stringify(payload), "HyperionDev",{ algorithm: 'HS256'});

```

A JWT token is fundamentally just a string with three parts: header, payload, and signature. Each part is separated by a dot and is base64-encoded. Base64-encoding is a way of representing any kind of data in text format. In the context of JWT, it's simply used to represent a lot of data in a small text space. The gist of a JWT token is something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNCwibmFtZSI6IkpvaG4gRG91IiwiaWYw
RtaW4iOiOnRydWV9.q_SJ2dKnaN8sjVtwi6yCJ6CpU-OSiiFR-ZBNm0G64
```

Putting it all together, this line of code creates a JWT token by taking a JSON payload, signing it with the "HyperionDev" secret key using the HS256 algorithm (**payload.secretkey.algorithm**). Don't worry about how the algorithm works behind the scenes — these kinds of encoding and encryption functions are written by experts in their fields, and we can benefit by just using their libraries instead of reinventing the wheel. The end result produces a string that represents the JWT token.

Note: This is NOT the authentication process. We first have to define the controller where the initial token will be created. The token will be used later for authentication when the user attempts to access a secured route. The controller is where you write most of your code. Up next, we'll define a route allowing the user to log in, then add this route to our app. Lastly, we'll test out the route and see the token generated.

Let's create the login **route**. This route is similar to the **myLoggerRoute.js** route. It will allow us to log in when we go to **http://localhost:8080/login** using Postman.

```
// routes/loginRoute.js
// get the userController
const { userController } = require('../controllers/userController');

const loginRoute = (app) => {

  app.get('/login', userController);
  //This route URL will be http://localhost:8080/login
};

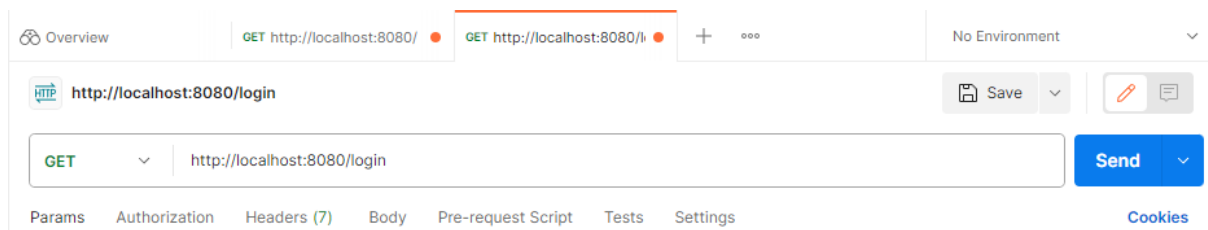
// export the login function to be used in "../app.js"
module.exports = loginRoute;
```

Let's add our route to **app.js** as well, and then we'll test it out.

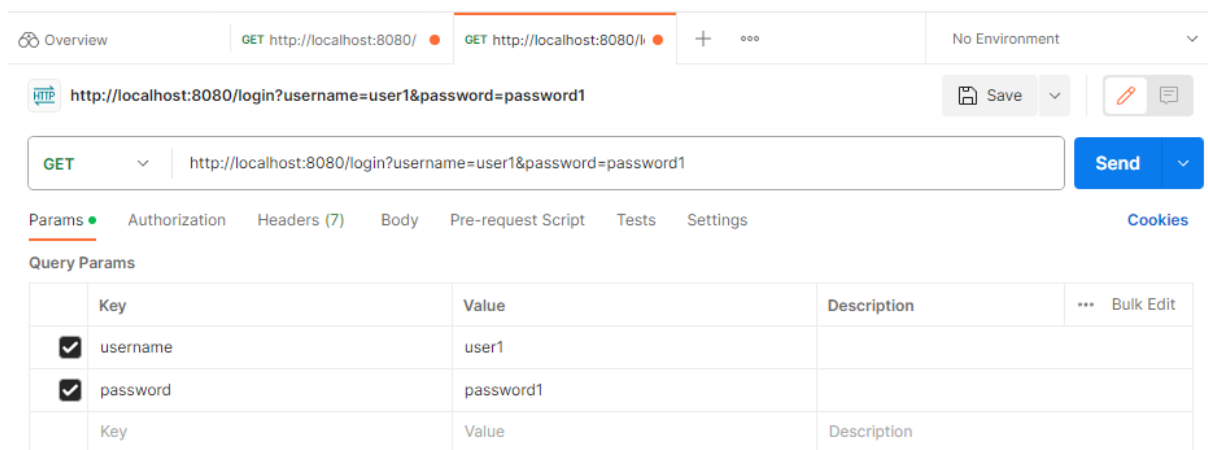
```
// . . .
// import the login route
const loginRoute = require('./routes/loginRoute.js');

// From here go to routes/myLoggerRoute.js
myLoggerRoute(app);
loginRoute(app)
// . . .
```

Before you head on over to Postman to test your login route, make sure that your server is running in the terminal.



- We will connect to the same URL as before but add the **/login** at the end of the URL to connect to our login API.
- Next, we use the query-parameter inputs to make a request to our API.



Note: When we click send, our values defined in `UserController.js` will be passed into the query object, as shown below. Instead of using the syntax `req.query.username` and `req.query.password` each time, we will destructure each value as variables.

```
const { username, password } = req.query;
```

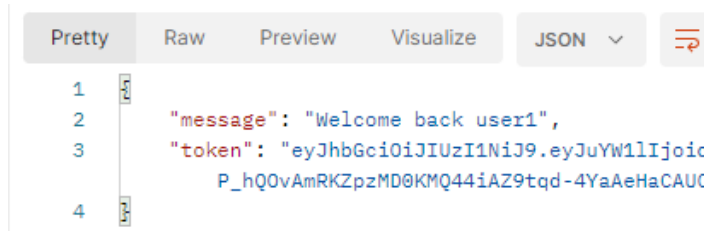
If we ran a **GET** request on the following endpoint with the correct credentials, we would get a JWT token in response.

Note: Every time this endpoint is requested by any user, a new JWT token will be created.

Endpoint:

<http://localhost:8080/login>

Response:



```
1 {
2   "message": "Welcome back user1",
3   "token": "eyJhbGciOiJIUzI1NiJ9.eyJ1bm11IjoidXNlcjEiLCJhZG1pbiI6ZmFsc2V9.
4     P_hQOvAmRKZpzMD0KM044iAZ9tqd-4YaAeHaCAUCr9o"
```

Implementation

In essence, we simulated a user logging in and obtained a valid JWT token. This token grants them access to view a variety of information in the database. It's important to note that upon logging in, we only received a message along with the token. When the user requests additional data, such as their personal information, we'll use JWT authentication to ensure that we securely send them the relevant data.

Now in the **userDB.js** file, let's add another object called **todos** and create a secured endpoint that allows a user to receive their to-do list.

```
// controllers/userDB.js
const userInformation = [
  {
    id: 1,
    username: 'user1',
    password: 'password1',
    todos: ['eat', 'sleep', 'code']
  },
  {
    id: 2,
    username: 'user2',
    password: 'password2',
    todos: ['By groceries', 'Food the dog', 'Make dinner']
  },
  // ... other user data
];

// export the userInformation array to be used in userController.js
module.exports = userInformation;
```

How will we implement this task? We are now only going to create a new route to secure our user's data and the JWT middleware to authenticate users who attempt to access the secured route.

Your file structure should now look like this:

```
Root ---
> controllers
  > userDB.js
  > userController.js
> routes
  > secure > userDataRoute.js
  > protected.js
  > login.js
> middleware
  > jwtMiddleware.js
app.js
```

The **jwtMiddleware.js** file will look like this:

```
// middleware/jwtMiddleware.js
const jwt = require('jsonwebtoken');

// Define a secured middleware function
function jwtMiddleware(req, res, next) {
  // Get the token from the request headers
  const jwtToken = req.headers['authorization']

  // Split the token from the Bearer
  const tokenExtract = jwtToken.split(' ')[1]

  try {
    // Verify the token using the secret key
    const payload = jwt.verify(tokenExtract, 'HyperionDev');

    // Attach the payload to the request object
    req.payload = payload;

    // Proceed to the protected route
    next();
  } catch (error) {
    // If token verification fails, return a forbidden response
    res.status(403).json({ message: 'Invalid token' });
  }
}

// Export the middleware to be used in userDataRoute.js
module.exports = {
```

```
    jwtMiddleware
  };
```

- The Middleware function retrieves the JWT token from the request headers using `req.headers['authorization']`.
- It then splits the token to extract the actual token part by removing the "Bearer" prefix. This is done using the `split(' ')[1]` operation.
- It tries to verify the token using the `jwt.verify()` function. If the verification is successful, the decoded payload is stored in the `payload` variable. The `next()` method is triggered, executing the controller function.
- If the verification fails (e.g., due to an invalid or expired token), an error is caught, and a response with a status code of 403 (Forbidden) is sent with a JSON message indicating "Invalid token."
- After successfully verifying the token and obtaining the decoded payload, we attach the payload to the `req` (request) object. This is done by adding a new property named `payload` to the `req` object. This ensures the payload will be available to subsequent middleware functions or route controllers in the request. We will later use this payload to extract some user data. This approach allows us to only connect to the 'login/data' route, without passing in any parameters/user data, as the token already contains user data in its payload.

We also need to update **app.js** to initialise the route as follows:

```
// import the userData route
const userDataRoute = require('./routes/secure/userDataRoute');
userDataRoute(app)
```

Lastly, we define the controller function to send the user's todo list once they have been verified:

```
// Require the user data from simulated database
const userInformation = require('./userDB');
const jwt = require('jsonwebtoken');

// Define the login controller functions
const userController = (req, res) => {...
}

// Define the user data controller function
const getTodos = (req, res) => {
  // extract username for the payload
```

```

const { name, admin } = req.payload;

//Find the user in the database - checking if the username and password
matches
const user = userInformation.find(user => user.username === name);

// If the user is found, return the user's todos
if (user) {
  return res.send(user.todos);
}
}

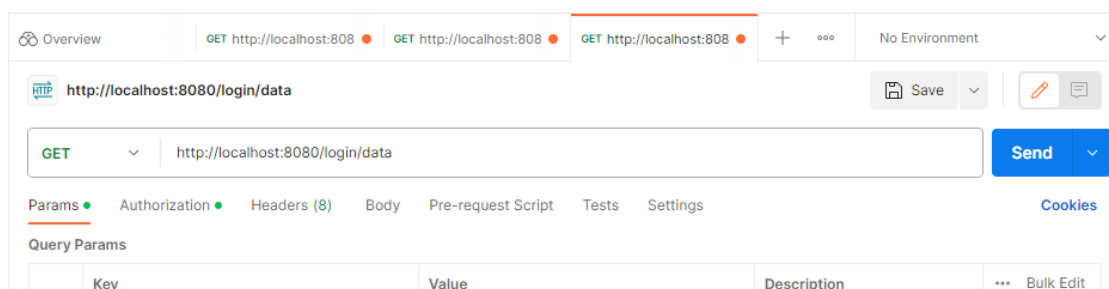
//export controller functions to be used on the myLoggerRoute.js/routes
module.exports = {
  userController,
  getTodos
};

```

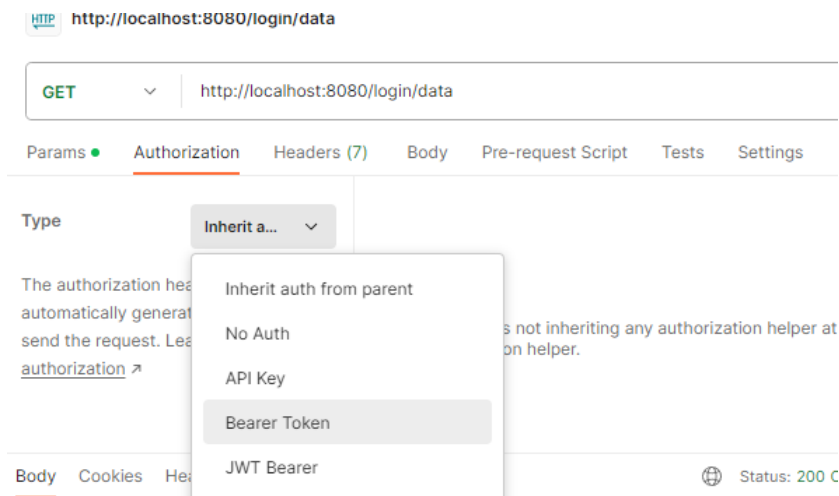
- The **getTodos** function extracts the username from the request query using **req.payload**. This was created and stored in the token when it was created.
- It searches for the user with the specified username in the **userInformation** array using the **Array.find()** method.
- If the user is found, it sends the user's todos array in the response using **res.send(user.todos)**.
- If the user is not found, it sends a 404 status response with the message "User not found."

When using JWT for any request sent to a specific endpoint that requires authentication, you will need to attach the token you received from the login endpoint to the headers of your request. For example, when using Postman, repeat the previous step to generate a JWT. **Copy the token**, and follow these steps to test the secured route.

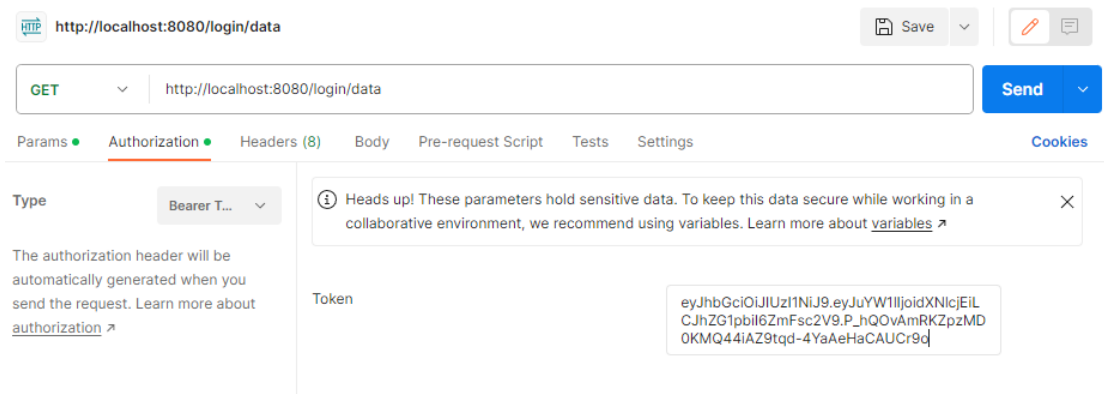
- In Postman, create a new tab and add **/data** to the end of the previous URL. We won't be using any parameters as we have already logged in.



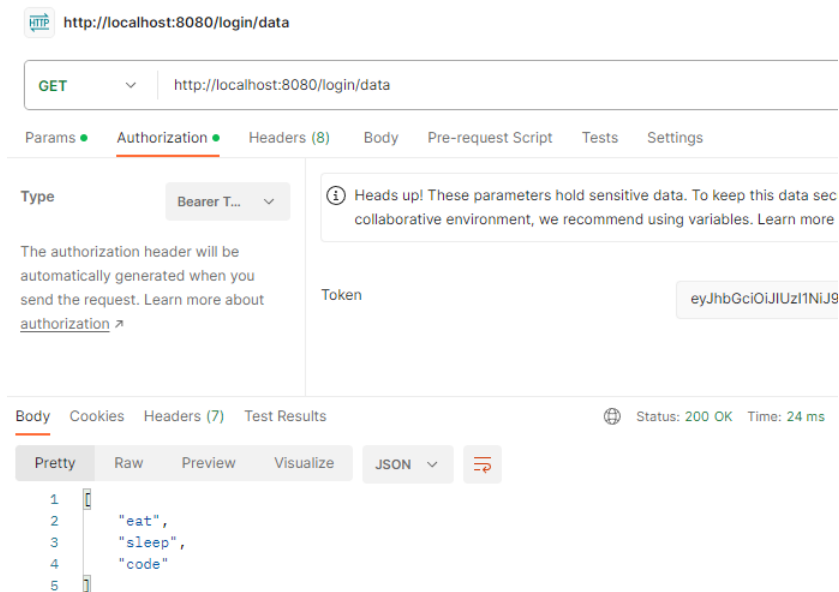
- Next, we need to use the copied JWT token and send it with the header request in Postman. Select the Header tab .



- Next, select **Bearer Token** from the drop-down list and paste your JWT token inside the input on the right.



- After you click the send button, you should see the user's to-do array.



As you can see, the endpoint is simple, clean, and efficient because the middleware function is doing all the work for you. All you had to do was to send the token with the request.

The middleware is working, but let's test all applicable outcomes. Let's modify the JWT token (simulating how someone could possibly try to alter it) by removing one letter in the token, and see what happens:



As expected, this string came directly from the middleware function defined in `jwtMiddleware.js`:

```
res.status(403).json({ message: 'Invalid token' });
```

This also means that the request never reached the controller function, which accesses our database, making this a secure approach for retrieving user data.

USING BODY PARSER

In the examples, we've been using Postman and `req.query` to send user data to the server. In a front-end application, you would use `req.body` to pass data through the request body. To facilitate data exchange between your server and client applications, you'll need to install the body-parser tool, like this:

```
npm install body-parser
```

In your server **app.js**, import the tool like this:

```
const bodyParser = require('body-parser')
app.use(bodyParser.json())
```

After adding this to your **app.js**, you can utilise the `req.body.username` syntax to request user data from the client/front-end app.

In conclusion, JWT offers several advantages for modern applications. Tokens serve as digital passports, enabling users to authenticate once and access resources securely without constant server interaction. This enhances performance and scalability while reducing the reliance on a database, providing a secure measure. JWT tokens are tamper-proof, making them secure against data manipulation. They enhance data privacy by including only essential information and avoiding sensitive data exposure. Their versatility accommodates various scenarios, from authentication to data sharing between services. Overall, JWT tokens streamline authentication, boost security, and promote efficient communication within applications.

Instructions

Read and run the accompanying example files provided before doing the task to become more comfortable with the concepts practiced..

Compulsory Task

- You will be required to develop a full-stack React to-do list application for this task.
- A user will need to register and log in to the application.
- Ensure a user can add/edit/remove/read tasks.
- Write middleware to:
 - Respond with an HTTP 403 to all requests by users whose usernames don't end with the substring '@gmail.com'.
 - Reject the addition of tasks that exceed 140 characters.
 - Reject any requests that are not of the JSON content type. You can test against image content types.
- The user will only have the capabilities mentioned above if logged in.
- Please remember to submit your code files in the relevant task folder.
- **Tip:** Ensure that all relevant endpoints to a to-do list route are **secure**.
- **Optional:** Use MongoDB as the database management system.



Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

