



**TASK**

# **Express Web Framework: Middleware**

Visit our website

# Introduction

## WELCOME TO Express Web Framework III TASK!

Think of middleware being the middleman between the request object and the response object with any route or end-point implemented in an Express application. You can implement a multitude of custom middleware in a specific end-point 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.



Get in touch  
**Connect for support**

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at <https://discord.com/invite/hyperdev> where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!



## 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, like the `express.json()` function/middleware to parse any incoming requests to a JSON object. This has the added benefit of adding validation of your own to any requests from a particular user to a specific end-point.

## WHY WOULD YOU WANT TO IMPLEMENT CUSTOM MIDDLEWARE?

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

- To validate data received from a specific user to manipulate data accordingly to suit 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.

## EXAMPLE OF CUSTOM MIDDLEWARE

### Next function from expressJS.

When writing custom middleware for expressJS applications, you will always use the **next** function, so you can expect to see it multiple times in our examples. The **next** function is a function in the Express router that executes the middleware, succeeding the current middleware when invoked.

### Installation:

We will install a normal express application using the `express-generator` node module for this tutorial, using the following command.

```
npm install -g express-generator
```

**OR**

```
npx express-generator
```

Now, with the generator installed, we will create our express application using this command.

```
express --no-view --git middleware-app
```

We are adding the `--git` in our command to ensure we can add it to a GitHub repository for future use and to ensure future employers can view our code.

The following prompts will be displayed in your terminal.

```
create : middleware-app\  
create : middleware-app\public\  
create : middleware-app\public\javascripts\  
create : middleware-app\public\images\  
create : middleware-app\public\stylesheets\  
create : middleware-app\public\stylesheets\style.css  
create : middleware-app\routes\  
create : middleware-app\routes\index.js  
create : middleware-app\routes\users.js  
create : middleware-app\public\index.html  
create : middleware-app\.gitignore  
create : middleware-app\app.js  
create : middleware-app\package.json  
create : middleware-app\bin\  
create : middleware-app\bin\www  
  
change directory:  
  > cd middleware-app  
  
install dependencies:  
  > npm install  
  
run the app:  
  > SET DEBUG=middleware-app:* & npm start
```

Following the instructions as stipulated above, we will go to the directory indicated and install the necessary node modules required. Before we start our application, we will modify some of the files to suit our tutorial.

First, in our `app.js` file, we will implement a port variable.

```
let PORT = 8080 || process.env.PORT;
```

And then, implement the `listen` function from `expressJS` to our `app` object.

```
app.listen(PORT, () => {  
  console.log("Application up and running on port: " + PORT);  
});
```

The next step in the process we need to follow is to update our `package.json` file. By doing the following.

Before:

```
"scripts": {  
  "start": "node ./bin/www"  
},
```

After:

```
"scripts": {  
  "start": "node app.js"  
},
```

To start, type **npm install** to download all the node modules and then **npm start** to run the program.

With the modifications done, you will have the following prompt:

```
> middleware-app@0.0.0 start  
> node app.js  
  
Application up and running on port: 8080  
█
```

Now to ensure we don't have to restart our application each time we make any modifications to it, we will install the nodemon library and modify our package.json file accordingly. Run the following command in your terminal:

**npm install nodemon:**

Adjust the package.json file accordingly:

```
"scripts": {  
  "start": "nodemon app.js"  
},
```

With all the modifications done, you will have the following prompt in your terminal.

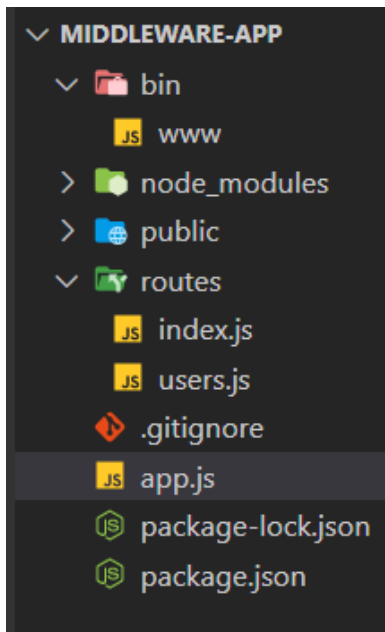
```
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Application up and running on port: 8080  
█
```

The fun will begin momentarily. For the purposes of this tutorial, we will use jsonwebtoken or ("JWT") to authenticate a user and implement other custom middleware for our application.

First, we are going to install jsonwebtoken using the following command:

```
npm install jsonwebtoken
```

Now, with the expressJS generator we used, you will see a folder already created named routes. With everything up and running, we will concentrate on users.js inside the routes folder.



By default, you will see the following.

```
let express = require('express');
let router = express.Router();

router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});

module.exports = router;
```

For dummy data, we will create an object called userInfo with the following key pair values.

```
let userInformation = {
  username: "admin@test.co.za",
  password: "P@ssw0rd1"
};
```

Next we will create a post request, where a user needs to login, with our application having only one user. In normal applications, you would query a database to retrieve the data the application requires, but for this example, we only have one user.

The post request will look as follows, as we are only checking one user's information.

```
let express = require("express");
let router = express.Router();
let jwt = require("jsonwebtoken");

let userInformation = {
  username: "admin@test.co.za",
  password: "P@ssw0rd1"
};

router.post("/login", function (req, res) {
  if (
    req.body.username == userInformation.username &&
    req.body.password == userInformation.password
  ) {
    let jwtToken = jwt.sign(
      {
        username: userInformation.username,
        password: userInformation.password,
      },
      "secretKey",
      { expiresIn: "1h" }
    );

    res.send(jwtToken);
  } else {
    res.send({ message: "user not Authenticated" });
  }
});
```





### JWT-Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluQHRlc3QuY28uemEiLCJwYXNzd29yZCI6IiBhY283MHJkMSIsImhhdCI6MTY2MjMjAyOTg2NiwiZXhwIjoxNjYyMDMzNDY2fQ.p0-2b\_O4ld80RZZeJG6rCd-eIMNzFQzJLmYKxSCPCP4

### Note:

Your second part of your JWT will look different than the above because if the expiry date. If there was no expiry times on the tokens they would look identical.

### Implementation:

For all intents and purposes, we simulated a user logging in and received a valid jwt-token to view any and all information on the database.

Now in the users.js file, we have created an array of to-do list items that can only be viewed if a user is logged in.

```
let todos = [
  {
    username: "admin@test.co.za",
    id: 1,
    title: "Implement post route for logging in.",
    completed: true,
  },
  {
    username: "admin@test.co.za",
    id: 2,
    title: "Implement custom middleware to authenticate user..",
    completed: true,
  },
];
```

How will we implement his task?

We will first create another file called **middleware.js** in our routes folder. Please remember this is a tutorial only and is not subject to traditional naming conventions or file structure principles laid out in a [Google Style Guide](#) for this example.

Our middleware.js file will look like the following code snippet.

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

```
function checkJWTToken(req, res, next) {
  if (req.headers.token) {
    let token = req.headers.token;
    jwt.verify(token, "secretKey", function (error, data) {
      if (error) {
        res.send({ message: "Invalid Token" });
        next();
      } else {
        req.username = data.username;
        req.password = data.password;
        next();
      }
    });
  } else {
    res.send({ message: "No token attached to the request" });
  }
}

module.exports = {
  checkJWTToken,
};
```

With us using jwt-tokens, for any request sent to a specific end-point that requires authentication, you will need to attach the token you received from the login endpoint to the headers of your request. For example, using Thunder-Client:

The screenshot shows the Thunder Client interface. At the top, a GET request is configured to `http://localhost:8080/users/` with a 'Send' button. Below this, the 'Headers' tab is selected, showing a list of HTTP headers. The 'token' header is checked and contains a JWT token. The 'header' row is currently empty.

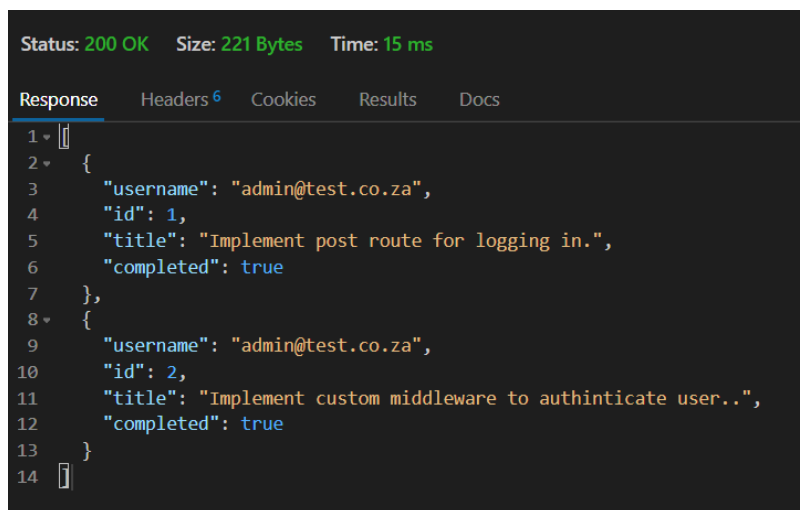
Http Headers	
<input checked="" type="checkbox"/> Accept	*/*
<input checked="" type="checkbox"/> User-Agent	Thunder Client (https://www.thunderclient.com)
<input checked="" type="checkbox"/> token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vyd
<input type="checkbox"/> header	value

Now we have our checkJWTToken middleware implemented. We will implement a normal get request to view our todos array, doing the following and modifying the user.js file.

```
router.get("/", checkJWTToken, function (req, res) {  
  res.send(JSON.stringify(todos));  
});
```

As you can see, your end-point is simple, clean and efficient because your middleware function is doing all the work for you!

Now with the jwt-token passed in our header request, we will get a successful response as follows.

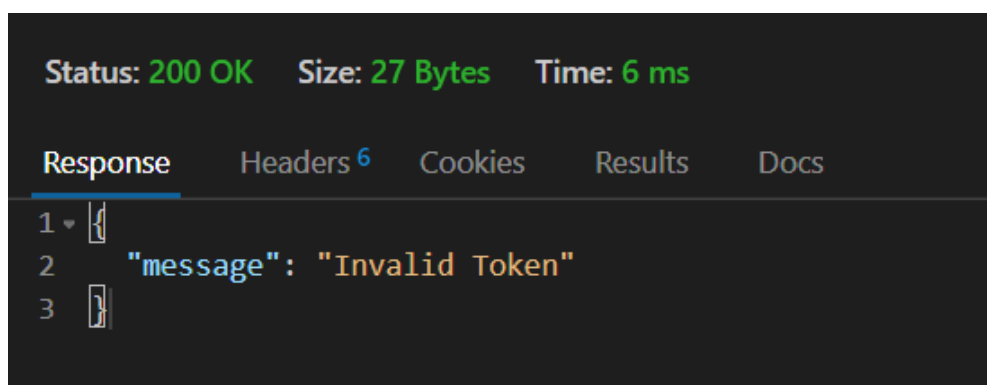


The screenshot shows a REST client interface with the following details:

- Status: 200 OK
- Size: 221 Bytes
- Time: 15 ms
- Response tab is selected, showing a JSON array of two objects.

```
1 {  
2   {  
3     "username": "admin@test.co.za",  
4     "id": 1,  
5     "title": "Implement post route for logging in.",  
6     "completed": true  
7   },  
8   {  
9     "username": "admin@test.co.za",  
10    "id": 2,  
11    "title": "Implement custom middleware to authenticate user..",  
12    "completed": true  
13  }  
14 }
```

Our middleware is working, but let's test all applicable outcomes. Let's generate an incorrect jwt-token by leaving one letter out of our header request, and see what happens.



The screenshot shows a REST client interface with the following details:

- Status: 200 OK
- Size: 27 Bytes
- Time: 6 ms
- Response tab is selected, showing a JSON object with a message.

```
1 {  
2   "message": "Invalid Token"  
3 }
```

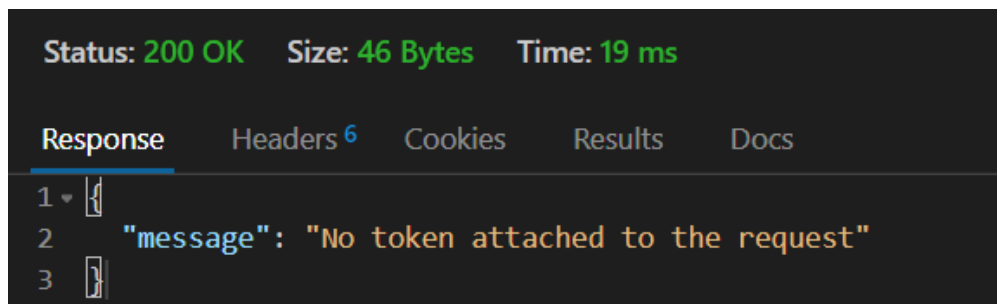
An expected result with how we structured our verify function verifying the jwt-token.

```

jwt.verify(token, "secretKey", function (error, data) {
  if (error) {
    res.send({ message: "Invalid Token" });
    next();
  } else {
    res.send({ message: "No token attached to the request" });
  }
}

```

Or leaving out the whole jwt-token completely in our end-point request.



Now for a future demonstration, we will implement another route where a user can change their password. We will use our `checkJWTToken` function middleware and another function to check the parameters of a password specification. We will only check the length of the string to ensure it's greater than or equal to six letters, and to confirm the new password and confirmation password match. You can add more constraints using third-party libraries, like special characters or upper- and lower-case letters. For our demonstration, our verification middleware will look something like this.

```

function changePasswordVerification(req, res, next) {

  if (

    req.body.newPassword == req.body.confirmPassword &&

    req.body.newPassword.length >= 6

  ) {

    req.newUserpassword = req.body.newPassword;

    next();

  } else if (req.body.newPassword.length < 6) {

    res.send({

```

```

        message: "The new password needs to be longer than six characters.",
    });

    next();
} else {
    res.send({
        message: "Conformation Password and New Password does not match.",
    });

    next();
}
}

```

Ensuring we can use our newly implemented middleware we will modify the `module.exports` object.

```

module.exports = {
    checkJWTToken,
    changePasswordVerification
};

```

Refactoring our `user.js` file with the applicable imports.

```

let { checkJWTToken, changePasswordVerification } = require("../middleware");

```

Implementing our new `/changePassword` end-point.

```

router.put(
    "/changePassword",
    checkJWTToken,
    changePasswordVerification,

```

```
function (req, res) {

  userInformation.password = req.newUserpassword;

  res.send({

    message: "Password Successfully changed",

    newPassword: userInformation.password,

  });

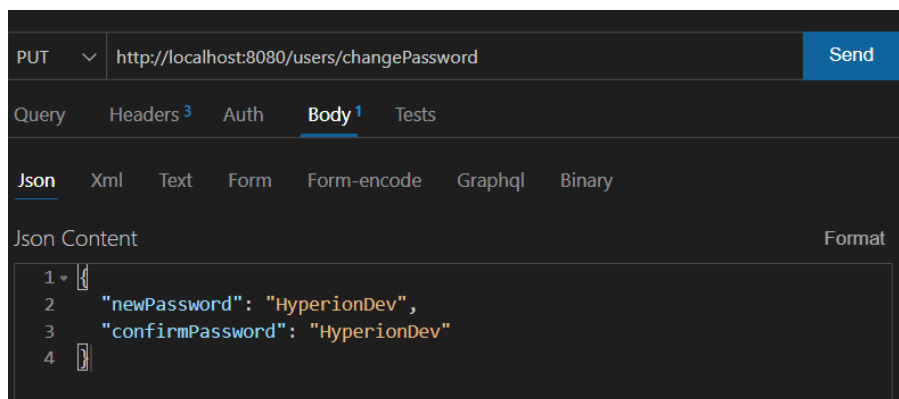
}

);
```

Execute the request to the newly created end-point, with the applicable req.body.

End-Point:

<http://localhost:8080/users/changePassword>



By now, you should understand how to implement custom middleware to suit your application. But there is more than one way to implement middleware on a specific route or multiple endpoints associated with a specific route. For instance, say I want all the applicable endpoints in my user's route to check for a valid jwt-token. How would I do this? By implementing the middleware function into our app.js file.

```
let { checkJWTToken } = require("../routes/middleware");
```

And then, modifying the route which requires the applicable middleware, as below:

```
app.use("/users", checkJWTToken, usersRouter);
```

With this implementation above, each endpoint applicable to the user's route must have a valid jwt-token to proceed with any data manipulation in the database.

## 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.
- **TIP:** Ensure all relevant endpoints to a to-do list route are secure.
- **Optional:** Using MongoDB as the database management system.

If you are having any difficulties, please feel free to contact our specialist team [on Discord](#) for support.



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.

