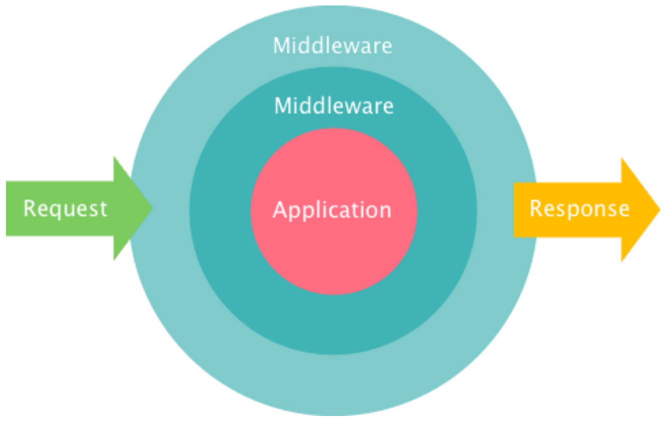
skill academy
by testbook

Express Middleware, Forms and Saving Data in FS

Express Middleware

Middleware in Express is a powerful feature that allows you to modify incoming requests and outgoing responses in your web application. It acts as a bridge between the client and the server, helping to add extra functionality to your web application.
Think of middleware as a set of traffic lights on a road. Just like traffic lights, middleware controls the flow of incoming requests and outgoing responses. When a request reaches the server, it stops at the first middleware. If the middleware approves it, the request is allowed to pass through and reach the next middleware. This process continues until the request reaches its destination.

Middleware functions have access to the request object (req), the response object (res), and the next middleware function to be invoked in the application's request-response cycle.



Middleware functions can perform the following tasks

1. Execute a piece of code

2. Modify the request and the response objects
3. Terminate the request-response cycle
4. Invoke the next middleware function

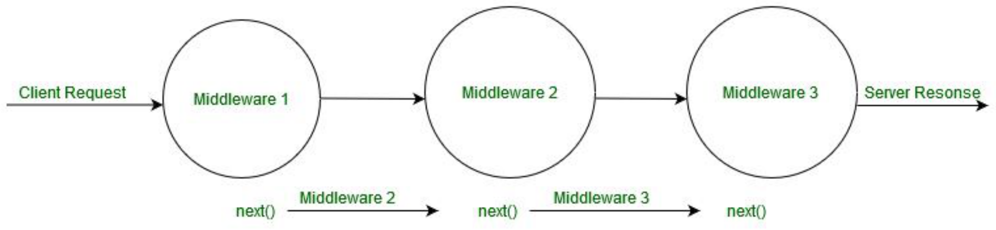
Here's a technical example to help you understand how middleware works:

```
const express = require('express');
const app = express();

// A simple middleware to log the request method and URL
const logMiddleware = (req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  // Note that req.method has information about what http request is made to the server.
  next();
};

// Use the middleware on all incoming requests
app.use(logMiddleware);

app.get('/', (req, res) => {
  res.send('Hello, World!');
});
app.get('/about', (req, res) => {
  res.send('About Us');
});
app.listen(3000, () => {
  console.log('App listening on port 3000!');
});
```

In this example, we create a simple middleware logMiddleware that logs the request method and URL to the console. Then, we use app.use() to apply this middleware to all incoming requests.
So, whenever a client makes a request to the server, the request will first pass through the logMiddleware. This middleware logs the request details and then calls the next() function to allow the request to continue its journey to its destination.

Types of Express Middleware

1. Application-level middleware
2. Router-level middleware
3. Error-handling middleware
4. Built-in middleware
5. Third-party middleware



Application-level middleware

Application-level middleware in Express refers to middleware that is defined and used for an entire Express application. This means that it's executed for every incoming request, regardless of the endpoint or route that the request is intended for.

Like in the previous example we used the middleware logMiddleware as an Application-level middleware. That middleware logs the request details and then calls the next() function to allow the request to continue its journey to its destination, whether it's the root endpoint '/' or the '/about' endpoint.

Router-Level middleware

Router-level middleware in Express refers to middleware that is defined and used specifically for a certain route or a group of routes. This means that it's only executed for requests that match the specified route or routes.

ex :

```
const express = require('express');
const app = express();
const router = express.Router();
const logMiddleware = (req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  next();
};

// Use the middleware for the /admin route
router.use('/admin', logMiddleware);

router.get('/', (req, res) => {
  res.send('Hello, World!');
});

router.get('/admin', (req, res) => {
  res.send('Welcome to the Admin Panel');
});

app.use('/', router);

app.listen(3000, () => {
  console.log('App listening on port 3000!');
});
```

We use router.use() to apply this middleware to the '/admin' route.
So, whenever a client makes a request to the '/admin' route, the request will first pass through the logMiddleware. This middleware logs the request details and then calls the next() function to allow the request to reach its destination, which is the '/admin' endpoint in this case.

However, if the client makes a request to the root endpoint '/', the request will not pass through the logMiddleware because it's only defined for the '/admin' route.

This is just a simple example to give you an idea of how router-level middleware works. In a real-world scenario, you can use router-level middleware to add specific functionality to a group of routes, such as validating inputs, logging request data, or checking for specific conditions.

Error-handling middleware

Error-handling middleware in Express.js is a way to handle errors that may occur during the processing of a request. These errors could be anything from a database connection error to a syntax error in your code.

Here's an example to help illustrate how error-handling middleware works:

Let's say you're building a web application that allows users to search for books. When a user submits a search query, your application sends a request to a third-party API that returns a list of books matching the query. However, what happens if the third-party API is down or returns an error?

This is where error-handling middleware comes in. You can create a middleware function in Express that handles any errors that occur during the request processing. For example, you could define a middleware function like this:

```
function errorHandler(err, req, res, next) {
  console.error(err);
  res.status(500).send('Something broke!');
}
```

This function takes four parameters: the error object (err), the request object (req), the response object (res), and the next function (next). The error object contains information about the error that occurred, such as a message or stack trace.

When an error occurs during the request processing, Express will automatically call the error-handling middleware function. In our example, if the third-party API returns an error, it will be passed to the errorHandler function. The console.error statement logs the error to the console, and the res.status(500).send('Something broke!') line sends a 500 status code and a generic error message to the user.

You can then use the app.use method in Express to register your error-handling middleware function:

```
app.use(errorHandler);
```

By registering your error-handling middleware function with app.use, Express will ensure that it's called whenever an error occurs during request processing.

So, in summary, error-handling middleware in Express is a way to handle errors that may occur during the processing of a request. By defining a middleware function that takes an error object as its first parameter, you can log the error and send an appropriate response to the user.

Built-in middleware

Express.js comes with a number of built-in middleware functions that can be used to perform common tasks. Here are some examples of built-in middleware in Express.js and how they can be used:

1. express.json(): This middleware function is used to parse incoming JSON data. It adds a body object to the req object, which contains the parsed JSON data.
   Ex:

   ```
   // Middleware to parse incoming JSON data
   app.use(express.json());

   app.post('/api/users', (req, res) => {
     // Access the parsed JSON data in the body object
     console.log(req.body);
     res.send('User created successfully!');
   });
   ```

2. express.urlencoded(): This middleware function is used to parse incoming URL-encoded data. It adds a body object to the req object, which contains the parsed URL-encoded data.
   Ex:

   ```
   // Middleware to parse incoming URL-encoded data
   app.use(express.urlencoded({ extended: true }));

   app.post('/api/users', (req, res) => {
     // Access the parsed URL-encoded data in the body object
     console.log(req.body);
     res.send('User created successfully!');
   });
   ```

3. express.static(): This middleware function is used to serve static files, such as HTML, CSS, and images. It takes a directory path as an argument and serves any files in that directory.
   Ex:

   ```
   // Middleware to serve static files
   app.use(express.static('public'));
   ```

4. express.Router(): This middleware function is used to create modular route handlers. It returns a router object that can be used to define routes for a specific URL path.
   Ex:

   ```
   // Create a router for the /api path
   const apiRouter = express.Router();

   // Define a route handler for the /api/users path
   apiRouter.get('/users', (req, res) => {
     res.send('List of users');
   });
   ```

```
// Use the router for the /api path
app.use('/api', apiRouter);
```

These are just a few examples of the built-in middleware functions in Express.js. By using these middleware functions, you can quickly and easily perform common tasks in your Express.js applications.

Third-party middleware

Third-party middleware functions are middleware functions that are not built into Express.js but are provided by third-party packages that can be installed using a package manager like npm

Here are some examples of built-in middleware in Express.js and how they can be used:

1. cors: This middleware function is used to enable Cross-Origin Resource Sharing (CORS) in an application. CORS is a security feature implemented in web browsers that prevents web pages from making requests to a different domain than the one that served the original web page. When you're building a web application, you might want to make requests to a different domain (for example, if your client-side code is running on a different domain than your API), and this is where the CORS middleware in Express.js comes in. Here's an example:
   ```
   const express = require('express');
   const cors = require('cors');

   const app = express();

   // Use the cors middleware
   app.use(cors());

   // Respond to requests on the /hello route
   app.get('/hello', (req, res) => {
     res.send('Hello, world!');
   });

   app.listen(3000, () => {
     console.log('Example app listening on port 3000!');
   });
   ```
2. Cookie Parser is a third-party middleware in Express.js that parses cookies attached to the HTTP request headers. The parsed cookies are then added to the request object as a cookies property, which can be accessed by any handler function that handles the request. Ex:
   ```
   const express = require('express');
   const cookieParser = require('cookie-parser');
   ```

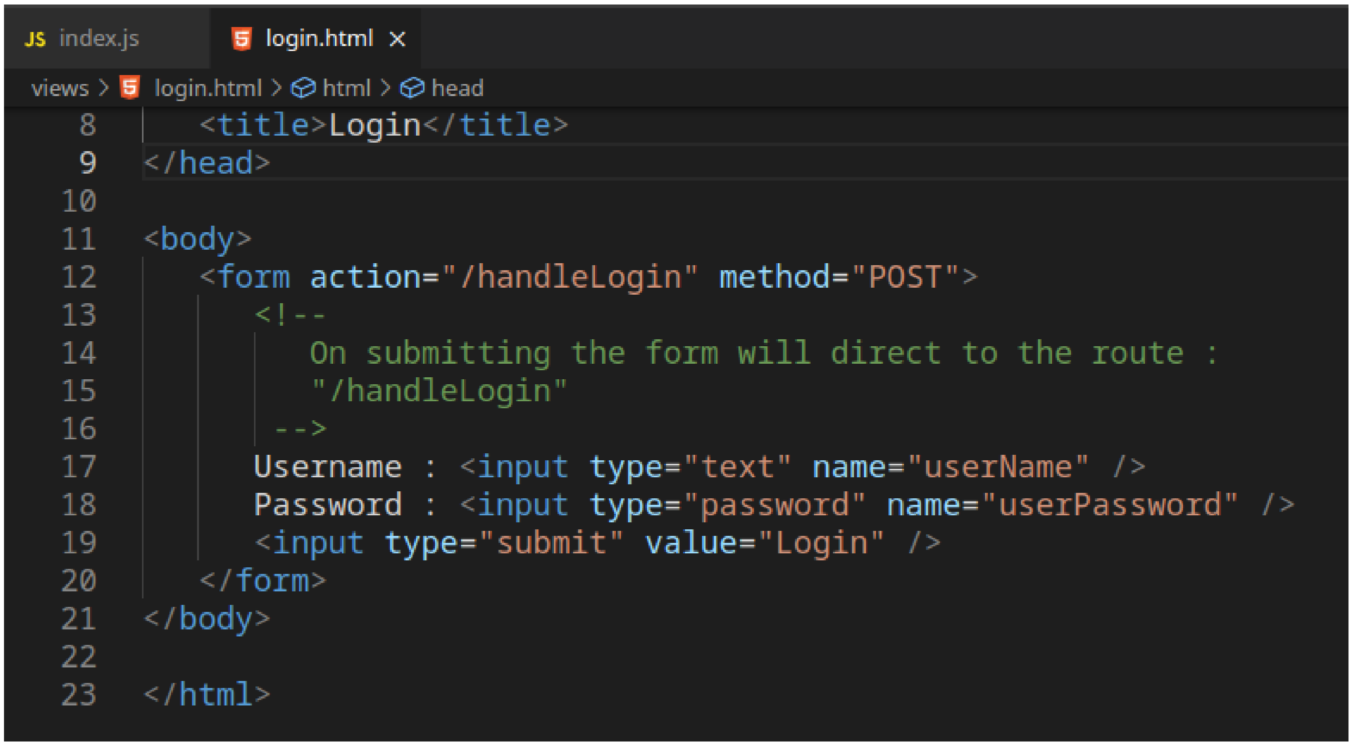
```
const app = express();

// Use the cookie-parser middleware
app.use(cookieParser());

// Set a cookie on the response object
app.get('/', (req, res) => {
  res.cookie('name', 'John Doe');
  res.send('Cookie set!');
});

// Retrieve the cookie from the request object
app.get('/greet', (req, res) => {
  const name = req.cookies.name;
  res.send(`Hello, ${name}!`);
});
```

These are just a few examples of the third-party middleware functions that can be used in Express.js. By using these middleware functions, you can easily add additional functionality to your Express.js applications.

Handling Forms in ExpressJS

Forms are present in almost every website, ranging from a simple survey form, feedback form to a simple search button.

Create a new login.html file inside views folder in VSCode add the following html :

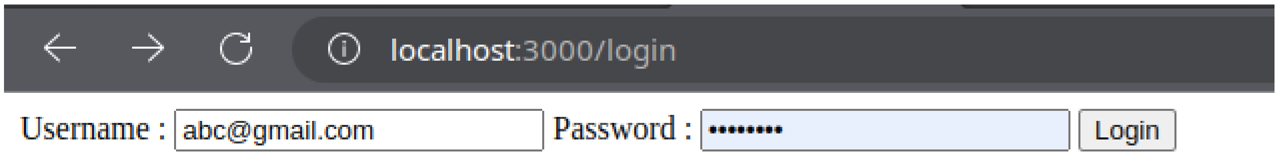


index.js login.html

views > login.html > html > head

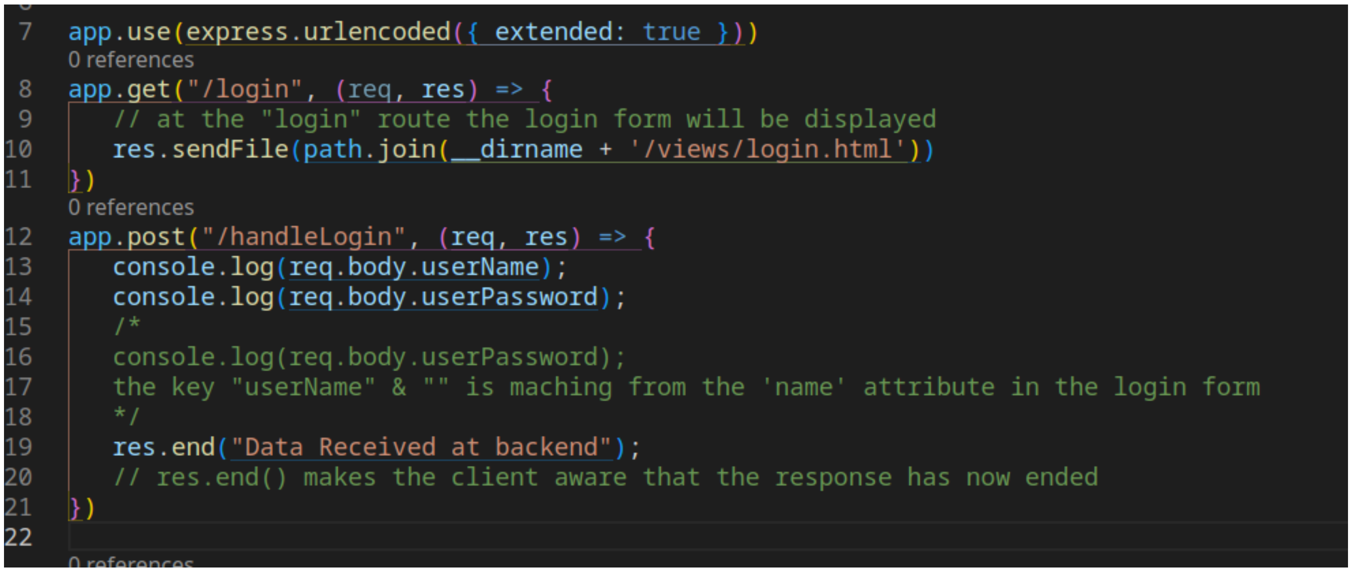
```html
    <title>Login</title>
</head>

<body>
    <form action="/handleLogin" method="POST">
        <!--
            On submitting the form will direct to the route :
            "/handleLogin"
        -->
        Username : <input type="text" name="userName" />
        Password : <input type="password" name="userPassword" />
        <input type="submit" value="Login" />
    </form>
</body>

</html>
```

In order to get the data we will have to use express.urlencoded() which will parse the data and add it to req.body , this is a built-in middleware function in Express. It parses incoming requests with urlencoded payloads, extended:true option allows parsing urlencoded data with query string library when it's false or the qs library when true

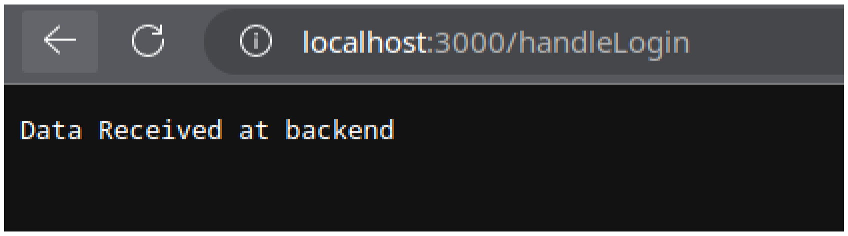


localhost:3000/login

Username : abc@gmail.com Password : •••••••• Login

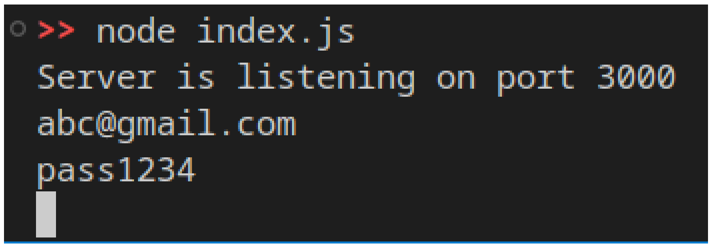


```
app.use(express.urlencoded({ extended: true }))
0 references
app.get("/login", (req, res) => {
    // at the "login" route the login form will be displayed
    res.sendFile(path.join(__dirname + '/views/login.html'))
})
0 references
app.post("/handleLogin", (req, res) => {
    console.log(req.body.userName);
    console.log(req.body.userPassword);
    /*
    console.log(req.body.userPassword);
    the key "userName" & "" is maching from the 'name' attribute in the login form
    */
    res.end("Data Received at backend");
    // res.end() makes the client aware that the response has now ended
})

0 references
```



localhost:3000/handleLogin

```
Data Received at backend
```



```
>> node index.js
Server is listening on port 3000
abc@gmail.com
pass1234
```

Express validator Package

Express-validator is a popular third-party middleware in Express.js that allows you to validate and sanitize data submitted in HTTP requests. It provides a set of validation and sanitization methods that you can use to check that the data in your HTTP requests meets your application's requirements.

Here's an example of how to use the express-validator middleware in an Express.js application:

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

const app = express();

// Use the express-validator middleware
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 6 })
);

// Handle form submission
app.post('/login', (req, res) => {
  // Check for validation errors
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  // Retrieve and sanitize form data
  const { email, password } = req.body;

  // Authenticate user
  // ...
});

app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});
```

In this example, we import the express-validator middleware and add it to our application using app.use(). We use the body() method to create validation and sanitization rules for the email and password fields. We then define a route handler for the /login route that retrieves and sanitizes the form data and checks for validation errors using the validationResult() method. If there are validation errors, we return a 400 response with the errors as a JSON object.

The express-validator middleware makes it easy to validate and sanitize data submitted in HTTP requests in your Express.js applications. It provides a set of powerful and flexible validation and sanitization methods that can help you ensure that your application only accepts data that meets your requirements. By using express-validator, you can build more secure and reliable web applications that are less vulnerable to data input errors and security breaches.

Try experimenting with this package and we will be using it and learning more about it later on.

NodeJS File System

After receiving data from the client we would like to store it in the server, we will be looking at the fs module of NodeJS and how it handles data and stores it.
The file system (or "fs" for short) is a module in Node.js that allows you to interact with the file system on your computer.

These are the major actions we can perform on files :

1. Read data from a file
2. Create a new file
3. Update an existing file
4. Delete an existing file
5. Rename an existing file

We will look at one by one how to do these operations using the fs module.
By the way do note that these are the operations which will be performed in our system, there will be many times when we will have to host the file on a remote machine and provide the link to that. When we will be uploading the file on our own server then we can use the fs module to access the files.

Here's an example of how to use the fs module in Node.js:

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

// Read data from a file
fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});

/**
 * Creating a file
*/
fs.appendFile('myfile.txt', 'This is some text', function (err) {
  if (err) throw err;
  console.log('Saved!');
});

fs.open('myfile.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});

fs.writeFile('myfile.txt', 'This is some text', function (err) {
  if (err) throw err;
  console.log('Saved!');
});


/**
 * Write to file, i.e., update an existing file
 **/
fs.appendFile('myfile.txt', 'This is some text', function (err) {
// if some file content already exists then it writes the content after the last char of the file.
  if (err) throw err;
  console.log('Saved!');
});

fs.writeFile('newfile.txt', 'Hello, world!', (err) => {
// If some file content already exists then it writes the content after replacing the entire content of the file.
  if (err) throw err;
  console.log('File written successfully');
});

// Create a directory
fs.mkdir('mydir', (err) => {
  if (err) throw err;
  console.log('Directory created successfully');
});

// Delete an existing file
fs.unlink('file.txt', (err) => {
  if (err) throw err;
  console.log('File deleted successfully');
});

// Rename a directory
fs.rename('myfile.txt', 'mynewfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

In this example, we import the fs module and use it to read and write files, create directories, and delete files. The fs.readFile() method reads the contents of a file and prints it to the console. The fs.writeFile()

method creates a new file and writes the string "Hello, world!" to it. The fs.mkdir() method creates a new directory, and the fs.unlink() method deletes a file.

The fs module in Node.js is very powerful and provides a lot of functionality for working with the file system. It's important to be careful when using the fs module because you can accidentally overwrite or delete important files. However, by using the fs module, you can create, read, write, and delete files and directories, which is essential for many kinds of applications.