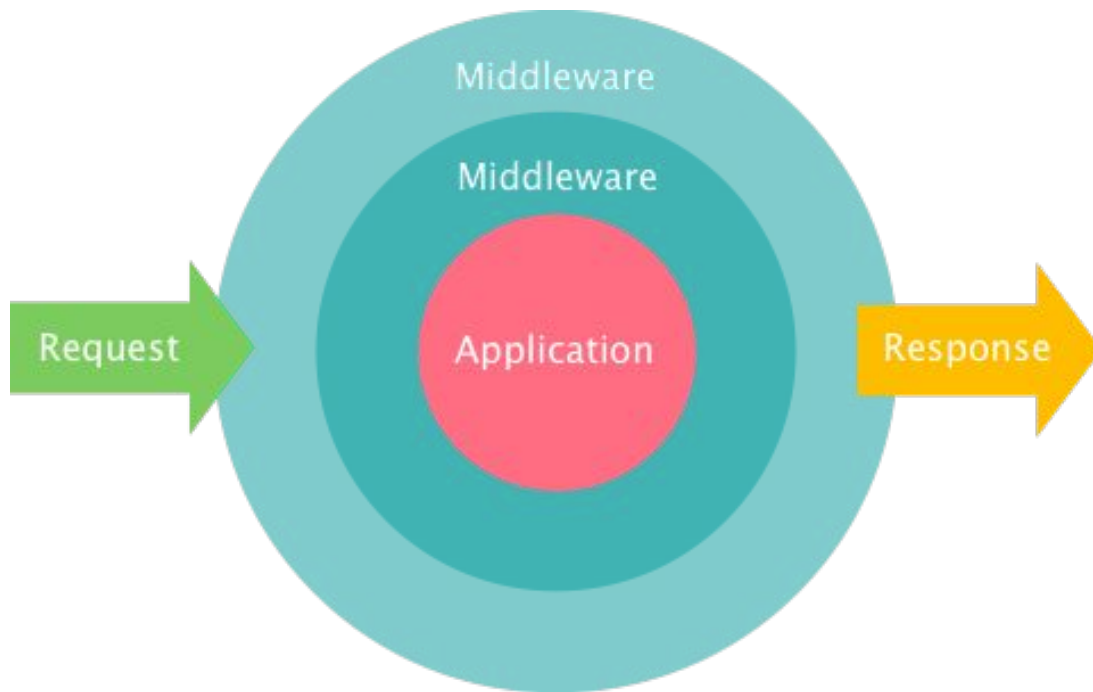


Lesson 4: Learn Express Middleware

- What is middleware
- Using middleware packages
- Securing our server with middleware

What is middleware in Express

- Middleware is a function that sits between the request and response objects and can modify or intercept either one before they reach their destination.
- Middleware can be used to handle common tasks, such as logging, parsing, and authentication, among others.



What do we use middleware for?

1. **Authentication:** Middleware can be used to verify the user's identity and ensure they have the proper permissions to access certain routes or resources.
2. **Logging:** Middleware can be used to log incoming requests and outgoing responses, providing insight into application usage and error tracking.
3. **Error handling:** Middleware can be used to handle errors that occur during the application's runtime, such as server errors or client-side validation errors.
4. **Compression:** Middleware can be used to compress the response body, reducing the size of the data sent back to the client and improving application performance.
5. **CORS:** Middleware can be used to handle Cross-Origin Resource Sharing (CORS) requests, allowing clients from different domains to access your application's resources.
6. **Rate limiting:** Middleware can be used to limit the rate at which clients can access certain resources, preventing excessive traffic and potential server overload.
7. **Caching:** Middleware can be used to cache frequently requested data, reducing the number of database queries and improving application performance.
8. **CSRF protection:** Middleware can be used to protect against Cross-Site Request Forgery (CSRF) attacks, where a malicious client attempts to execute unauthorized requests on behalf of the user.
9. **Custom functionality:** Middleware can be used to add custom functionality to your application, such as processing form data or performing database updates before a response is sent back to the client.

How Express middleware works

- Middleware functions in Express take three arguments: req (the request object), res (the response object), and next (a function that passes control to the next middleware function in the stack).

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

HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

```
app.get('/', function(req, res, next) {  
  next();  
})
```

Callback argument to the middleware function, called "next" by convention.

```
app.listen(3000);
```

HTTP **response** argument to the middleware function, called "res" by convention.

HTTP **request** argument to the middleware function, called "req" by convention.

Middleware can handle responses or pass them on

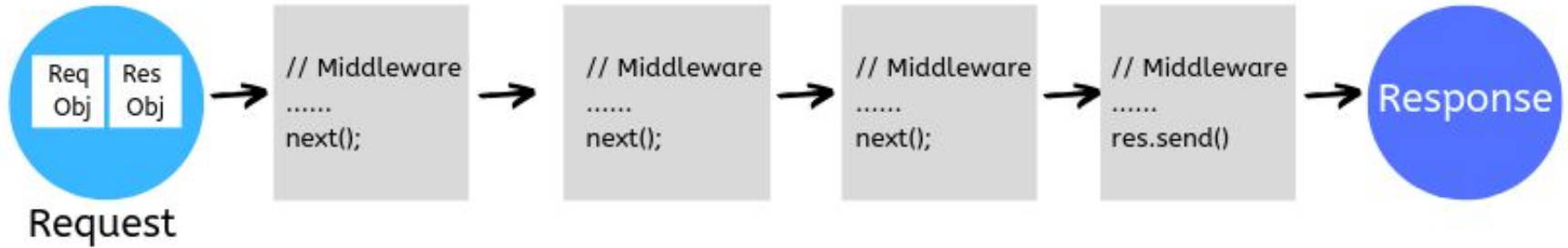
```
// Middleware function 1
app.use((req, res, next) => {
  console.log('Middleware 1');
  // Pass control to the next middleware function
  next();
});

// Middleware function 2
app.use((req, res, next) => {
  console.log('Middleware 2');
  // Terminate the request-response cycle by sending a response to the client
  res.send('Hello World!');
});

// Middleware function 3
app.use((req, res, next) => {
  console.log('Middleware 3');
  // This middleware function won't be executed, as the previous middleware function
  // has already sent a response to the client and terminated the request-response cycle
});
```

The middleware flow

- The order in which middleware functions are declared matters, as they are executed in the order they are defined.



Let's set up some middleware in our backend



Middleware > Middlemen

Lets install the middleware packages we will be using

- In your terminal run the install command for our middleware packages: `npm install helmet morgan cors express-rate-limit`

The Middleware we are adding

- **helmet:** Helps secure your Express app by setting various HTTP headers. For example, it can help protect against cross-site scripting (XSS) attacks by setting the X-XSS-Protection header, or help prevent clickjacking attacks by setting the X-Frame-Options header.
- **morgan:** Logs HTTP request/response information to the console. This can be useful for debugging and monitoring your application.
- **cors:** Enables Cross-Origin Resource Sharing (CORS), which allows web pages from different domains to make requests to your Express app. CORS can help improve security and allow for more flexible access controls.
- **express-rate-limit:** Helps protect against brute-force attacks and other forms of abuse by limiting the number of requests a client can make to your Express app in a given time period. You can configure the rate limit based on factors such as IP address, request method, and more.

Importing our new middleware in index.js

```
// Our updated imports
```

```
const express = require("express");
```

```
const helmet = require("helmet");
```

```
const morgan = require("morgan");
```

```
const cors = require("cors");
```

```
const rateLimit = require("express-rate-limit");
```

```
const usersRouter = require("../routers/usersRouter");
```

```
const lightningRouter = require("../routers/lightningRouter");
```

Initializing helmet morgan and CORS

Be sure to initialize each of these before `server.use(express.json());`

```
// Create a new instance of the Express server
```

```
const server = express();
```

```
// Use helmet middleware for security
```

```
server.use(helmet());
```

```
// Use morgan middleware for logging using the 'common' logging format
```

```
server.use(morgan("common"));
```

```
// Use cors middleware to enable cross-origin requests
```

```
server.use(cors());
```

Add our rate limiting middleware

```
// Use rate limiting middleware to limit the number of requests from a single IP
server.use(
  rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100, // limit each IP to 100 requests per windowMs
  })
);
```


Let's add one more tool before we test!

Nodemon

Nodemon is a utility tool for Node.js that automatically restarts the server whenever a change is detected in the code. This saves us the hassle of having to manually restart the server after each change we make

Run `npm install nodemon` in your terminal

Update scripts in package.json to use nodemon

Update the scripts property to use nodemon whenever we run `npm run start`

```
{
  "name": "backend-course-walkthrough",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

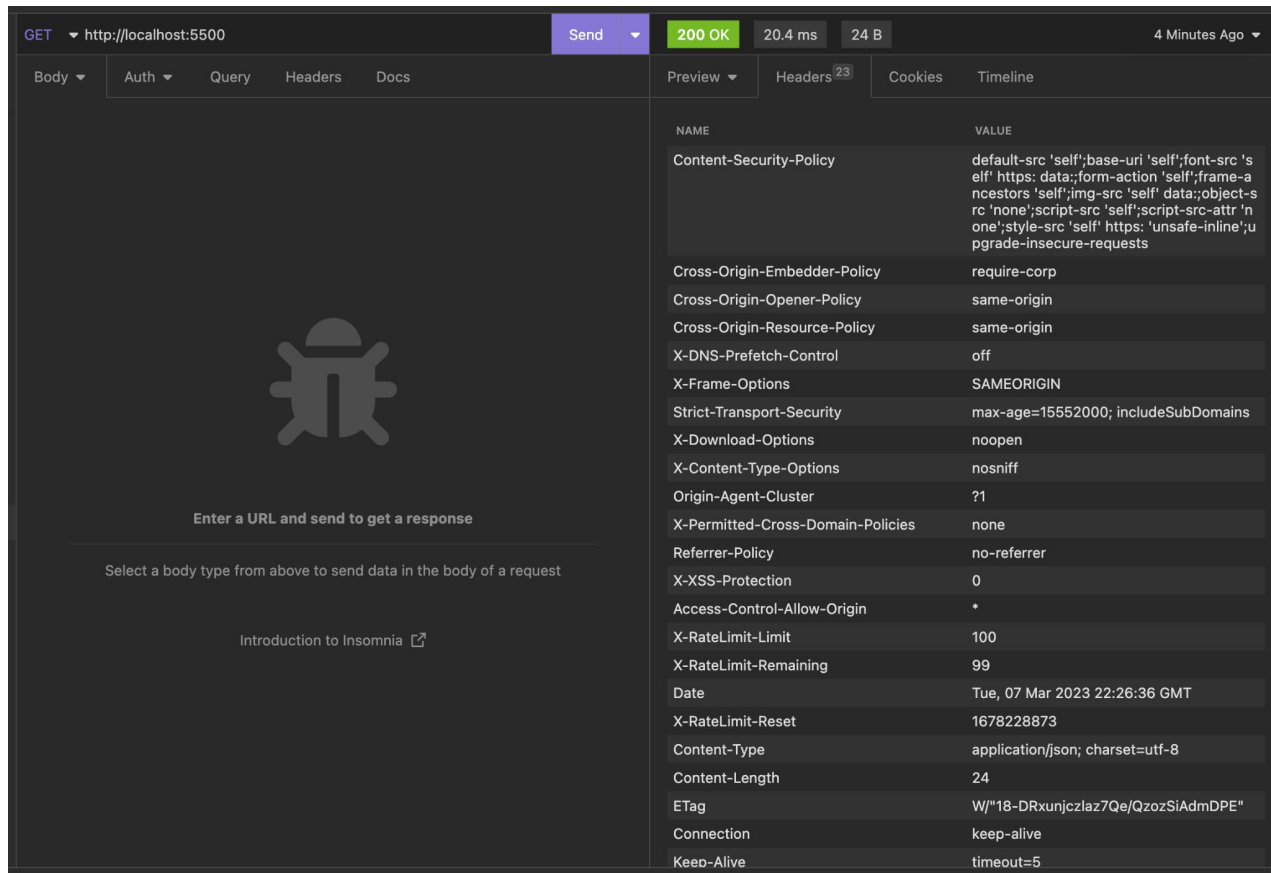
Run the server and test our new middleware

- Now after running ``npm run start`` we should see the server starting up with nodemon
- In the screenshot below I made a request which was logged by morgan on the bottom line of the console

```
> nodemon index.js  
  
[nodemon] 2.0.21  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node index.js`  
Server listening on port 5500  
::ffff:127.0.0.1 - - [07/Mar/2023:22:26:36 +0000] "GET / HTTP/1.1" 200 24
```

Testing Helmet

We can now check the response from our last request and verify that helmet is working by checking for all the additional headers that should be added to the response



GET http://localhost:5500 Send 200 OK 20.4 ms 24 B 4 Minutes Ago

Body Auth Query Headers Docs Preview Headers 23 Cookies Timeline

Enter a URL and send to get a response

Select a body type from above to send data in the body of a request

Introduction to Insomnia

NAME	VALUE
Content-Security-Policy	default-src 'self';base-uri 'self';font-src 'self' https: data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Cross-Origin-Embedder-Policy	require-corp
Cross-Origin-Opener-Policy	same-origin
Cross-Origin-Resource-Policy	same-origin
X-DNS-Prefetch-Control	off
X-Frame-Options	SAMEORIGIN
Strict-Transport-Security	max-age=15552000; includeSubDomains
X-Download-Options	noopen
X-Content-Type-Options	nosniff
Origin-Agent-Cluster	?1
X-Permitted-Cross-Domain-Policies	none
Referrer-Policy	no-referrer
X-XSS-Protection	0
Access-Control-Allow-Origin	*
X-RateLimit-Limit	100
X-RateLimit-Remaining	99
Date	Tue, 07 Mar 2023 22:26:36 GMT
X-RateLimit-Reset	1678228873
Content-Type	application/json; charset=utf-8
Content-Length	24
ETag	W/"18-DRxunjczlazz7Qe/QzozSiAdmDPE"
Connection	keep-alive
Keep-Alive	timeout=5

A bit more on rate limiting and Denial of Service attacks

Denial of Service (DoS) attacks are a common type of cyber attack where the attacker tries to overwhelm a server or network with traffic, making it unavailable to legitimate users. One common method of DoS attacks is to flood the server with a high volume of requests, causing the server to crash or slow down to the point where it is unusable.

In fact, many types of cyber attacks boil down to some form of repetitive or looping behavior. Attackers often use automated tools or scripts that repeatedly perform certain actions or send a high volume of requests to a server or network, making it difficult for the system to keep up with the incoming traffic. Understanding this basic mechanism of attacks can help in developing countermeasures to mitigate or prevent such attacks.

Now let's attack our own server to test our rate limiting middleware!

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

function sendRequest() {
  const options = {
    host: 'localhost',
    port: 5500,
    path: '/',
    method: 'GET',
  };

  const req = http.request(options, (res) => {
    console.log(`Response status code: ${res.statusCode}`);
  });

  req.on('error', (e) => {
    console.error(`Request error: ${e.message}`);
  });

  req.end();
}

setInterval(sendRequest, 100);
```

We can run this code in the terminal

- Open up a new terminal session
- Run 'node'
- Paste in the code from the previous slide
- Now you should see requests flying by and the status code will change from 200 to 429 after 100 requests

```
[austinkelsay@Austins-MacBook-Pro ~ % node
Welcome to Node.js v16.13.1.
Type ".help" for more information.
> 
```

Review

- Middleware is any process or function that runs in between the request and response in our API
- Securing our server with middleware: using helmet for security, morgan for logging, cors for enabling cross-origin requests, and express-rate-limit for limiting requests per IP
- The order in which middleware functions are declared matters
- Middleware functions can terminate the request-response cycle by sending a response to the client or passing control to the next middleware function

By using middleware, we can modularize our server logic and add additional functionality to our server. We learned how to use several popular middleware packages to enhance our server's security and performance. Remember to always consider security when building an Express server and to use middleware packages that can help protect against common attacks like DoS attacks.

Resources

- Official Express Middleware documentation:
<https://expressjs.com/en/guide/using-middleware.html>
- A tutorial on using middleware in Express:
https://www.tutorialspoint.com/expressjs/expressjs_middleware.htm
- Express.js Fundamentals - 6 - Middleware Explained:
<https://www.youtube.com/watch?v=9HOem0amlyg>
- A guide to using helmet middleware for securing Express apps:
<https://github.com/helmetjs/helmet#how-it-works>
- A tutorial on using morgan middleware for logging in Express:
<https://expressjs.com/en/resources/middleware/cors.html>