1

Outline

1. Built-in variables and functions
2. Built-in modules: fs, os, path, url, querystring, net, http
3. Events and Streams in node.js
4. Sending a web request using the http module
5. Building a basic web server

2

Node Package Manager

3

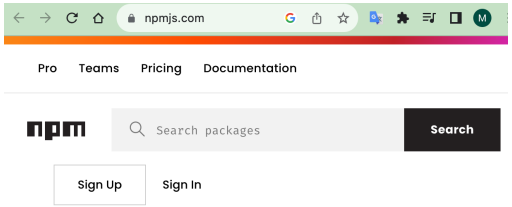
NPM

- npm is a package manager for Node.js that allows you to install and manage JavaScript modules.
- npm has a large repository of modules, including modules for web development, data science, machine learning, and more.
- npm is a command-line tool that you can use to install, update, and uninstall modules.
- npm is also a registry where you can publish your own modules.
- npm is included with Node.js, so there is no need for a separate installation.

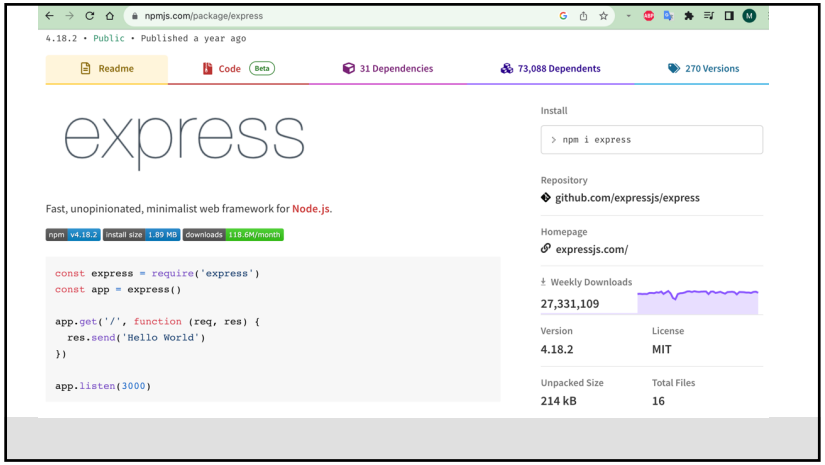
4

NPM

- npmjs.com: The official website for the Node Package Manager (npm).
- Largest Package Registry: Hosts over a million open-source JavaScript packages.



5



6

Using NPM to install a package

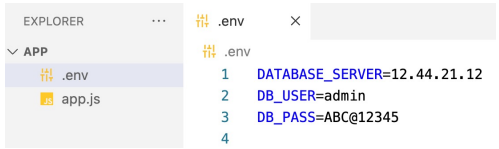
1. **Create a Node.js Project:** `mkdir node-app && cd node-app`
2. **Initialize Your Project:** `npm init`
3. **Install a package:**
   - In this example, we will install the dotenv package which helps you load environment variables from a **.env** file into your Node.js

     `npm install dotenv`
   - This command installs the dotenv package and adds it to your project's node_modules directory.
   - It also updates your package.json file to include dotenv as a dependency.

7

Using NPM to install a package

4. **Create a .env File:**
   - Create a new file in your project's root directory and name it **.env**



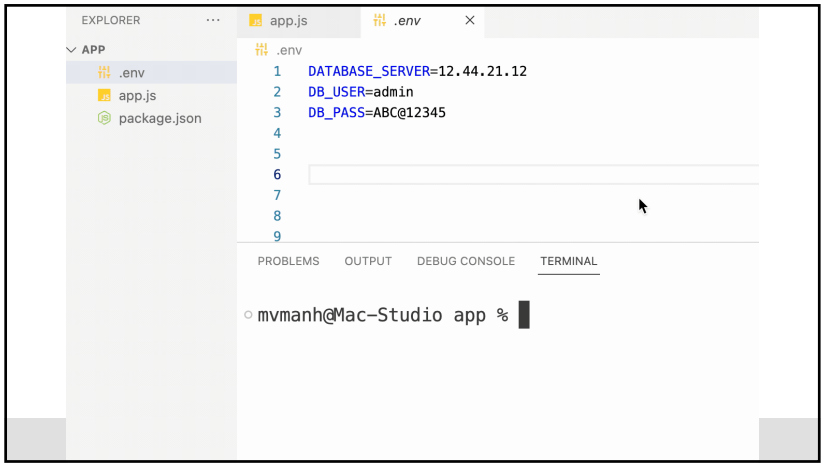
8

Using NPM to install a package

5. **Configure Your Node.js Application:**
   - Import the dotenv module at call the .config() method: `require('dotenv').config();`
   - You can then access the environment variables you defined in the .env file anywhere

```
require('dotenv').config();

const {DATABASE_SERVER} = process.env
const {DB_USER, DB_PASS} = process.env

console.log(`Server IP: ${DATABASE_SERVER}`)
console.log(`Username: ${DB_USER}`)
console.log(`Password: ${DB_PASS}`)
```


9



10

Express Framework

11

Express Framework

- **Express** is a popular Node.js web application framework.
- It simplifies the creation of web applications and APIs.
- Known for its minimalistic and flexible design.
- Provides robust routing, middleware, and templating features.
- Widely used in building modern web applications.
- A vibrant ecosystem of plugins and extensions available.

12

Express Framework

- To create a web application using Express, you will need to:
  1. Create a new project.
  2. Use npm to install Express
  3. Write the code for your application.
  4. Run the application.

13

Simple Express App

- The following is an example of a simple Express application

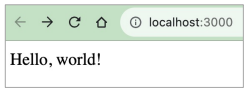
```
const express = require('express')
const app = express()

const serverLogic = (request, response) => {
    response.send('Hello, world!')
}
app.get('/', serverLogic)

app.listen(3000, () => console.log('http://localhost:3000'))
```

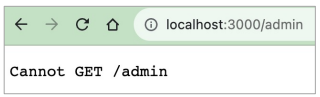

14

Simple Express App

- The following is an example of a simple Express application

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

const app = express()

app.get('/', (req, res) => {
    res.send('Hello, world!')
})

app.listen(3000, () => console.log('http://localhost:3000'))
```





15

Simple Express App

- Multiple routes should be handled in different code block

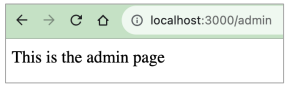
```
app.get('/', (request, response) => {
    response.send('Hello, world!')
})

app.get('/admin', (request, response) => {
    response.send('This is the admin page')
})
```

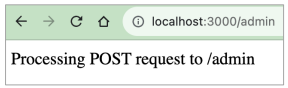
16

Simple Express App

- Express provides multiple convenient methods for handling different request types

```
app.get('/admin', (req, res) => {
    res.send('This is the admin page')
})

app.post('/admin', (req, res) => {
    res.send('Processing POST request to /admin')
})
```

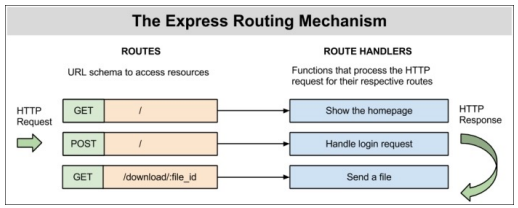




17

Routing in Express

- Routing is the process of determining how an application responds to a client request to a particular endpoint. An endpoint is a URI (or path) and a specific HTTP request method (GET, POST, and so on).



18

Routing in Express

- Routes are defined using the app.METHOD() method, where METHOD is the HTTP method and PATH is the path to the endpoint.

```
app.get('/students', (req, res) => {
    res.send('A list of students')
})
app.post('/students', (req, res) => {
    res.send('Adding a new student')
})
app.get('/students/521H1234', (req, res) => {
    res.send('detail info of student 521H1234')
})
```

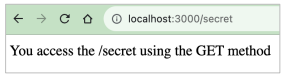
```
app.put('/students/521H1234', (req, res) => {
    res.send('update student 521H1234')
})

app.delete('/students/521H1234', (req, res) => {
    res.send('delete student 521H1234')
})
```

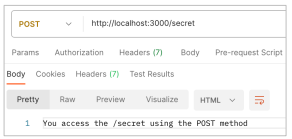

19

Routing in Express

- Express supports methods that correspond to all HTTP request methods: get, post, put, delete, and so on.
- There is a special routing method, app.all(), used to load middleware functions at a path for all HTTP request methods.

```
app.all('/secret', (req, res) => {
    const {method} = req
    res.send(`You access the /secret using the ${method} method`)
})
```





20

Routing in Express

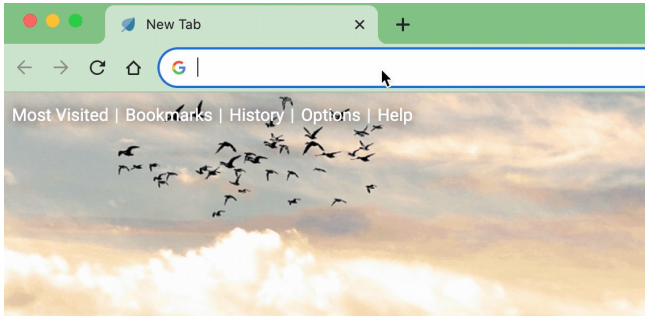
- In Express.js, app.all('*') is a catch-all route handler that matches any HTTP method (GET, POST, PUT, DELETE, etc.) and any path that hasn't been explicitly defined by other route handlers.

```
app.get('/', (req, res) => {
    res.send('Welcome to home page')
})

app.get('/about', (req, res) => {
    res.send('About page')
})

app.post('/login', (req, res) => {
    res.send('Handle user login')
})
```

```
app.all('*', (req, res) => {
    res.status(404).send('Page not found.');
});
```


21

Routing in Express



22

Routing Paths

- Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

```
app.get('/', (req, res) => {
    res.send('Home Page')
})

app.get('/about', (req, res) => {
    res.send('About Page')
})
```

```
app.get('/students?', (req, res) => {
    res.send('/students or /student')
})

app.get(/^\/img(\d{1,3})?$/, (req, res) => {
    res.send('/img or /img1 or /img12 or /img123')
})
```


23

Routing Paths

- **/^\/img(\d{1,3})?$/** is a regular expression that matches a URL that starts with `/img` and has an optional digit at the end. The digit at the end can be from 1 to 3 digits long.
- Here is a breakdown of the regex:
  - **`^`** Matches the beginning of the string.
  - **`\/img`** Matches the literal string `/img`.
  - **`(\d{1,3})`** Matches a digit between 1 and 3 digits long. The `{1,3}` part is called a quantifier. It tells the regex engine to match the preceding character (in this case, a digit) between 1 and 3 times.
  - **`?`** Matches the preceding character zero or one time. This means that the digit at the end of the URL is optional.
  - **`$`** Matches the end of the string.

24

Routing Parameters

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL.

Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: {"userId": "34", "bookId": "8989"}

- To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', (req, res) => {
    res.send(req.params)
})
```


25

26

Response Methods

27

Response Methods

- The methods on the response object (res) can send a response to the client, and terminate the request-response cycle.

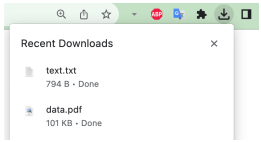
Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

28

Response Methods

- The res.download() method is used to send a file as an attachment in the HTTP response. It allows you to prompt the user to download a file from your server, rather than displaying it in the browser.

```
app.get('/text', (req, res) => {
    res.download('./files/text.txt')
})

app.get('/pdf', (req, res) => {
    const file = path.join(__dirname, 'files', 'data.pdf')
    res.download(file)
})
```



29

Response Methods

- The res.json() method is used to send JSON responses to the client.
- It serializes JavaScript objects into JSON format and sets the appropriate Content-Type header in the HTTP response to indicate that the response is in JSON format.

```
app.get('/user', (req, res) => {
    const john = {name : 'John Smith', age: 30, from: 'Vietnam'}
    res.json(john)
})
```

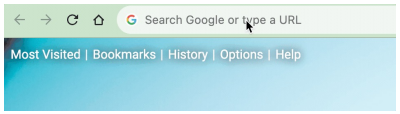
30

Response Methods

- The res.redirect() method is used to redirect the client to a different URL.
- It sends an HTTP response with a redirection status code (usually 302 Found) and a Location header indicating the URL to which the client should be redirected.

```
app.get('/admin', (req, res) => {
    res.redirect('/maintenance')
})
```

```
app.get('/maintenance', (req, res) => {
    res.send('The site is under maintenance')
})
```



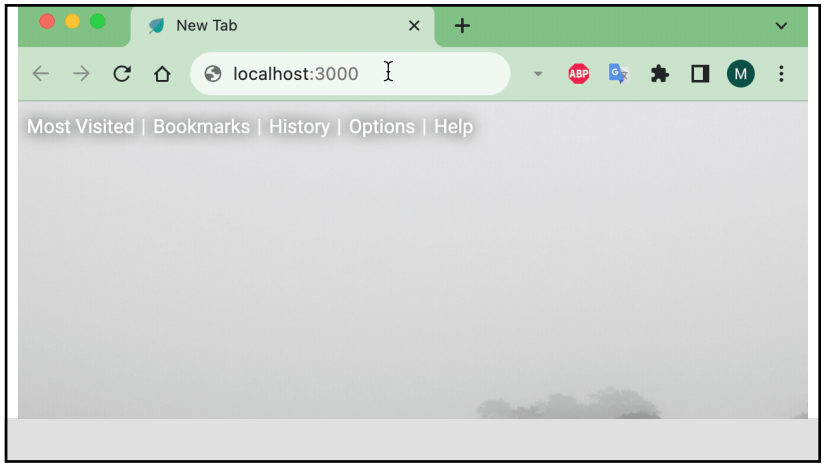
31

Response Methods

- The res.sendFile() method is used to send files as responses to HTTP requests.
- It allows you to serve static files, such as HTML, images, CSS, JavaScript, and more, to clients.
- This method sends the specified file with the appropriate headers, including the Content-Type header, based on the file's extension.

```
app.get('/', (req, res) => {
    const file = path.join(__dirname, 'files', 'index.html')
    res.sendFile(file) // must be absolute path
})

app.get('/html', (req, res) => {
    res.download('./files/index.html') // can be relative path
})
```


32



33

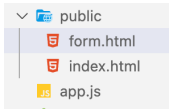
A Simple Web Example

34

Express Examples

- In this example we have two web pages: / (home page) and /form displaying the calculation interface.

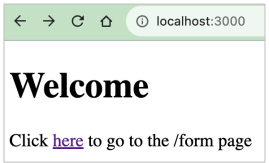
```
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'index.html'))
})
app.get('/form', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'form.html'))
})
```



35

Home Page

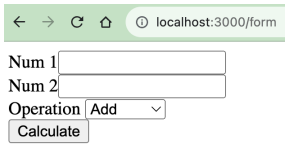
```
<body>
    <h1>Welcome</h1>
    Click <a href="/form">here</a> to go to the /form page
</body>
</html>
```



36

Form Page

```
<form action="/submit" method="get">
    Num 1<input type="text" name="a" >
    Num 2<input type="text" name="b" >

    Operation <select name="op">
        <option value="+">Add</option>
        <option value="-">Subtract</option>
        <option value="*">Multiply</option>
        <option value="/">Divide</option>
    </select>

    <button>Calculate</button>
</form>
```



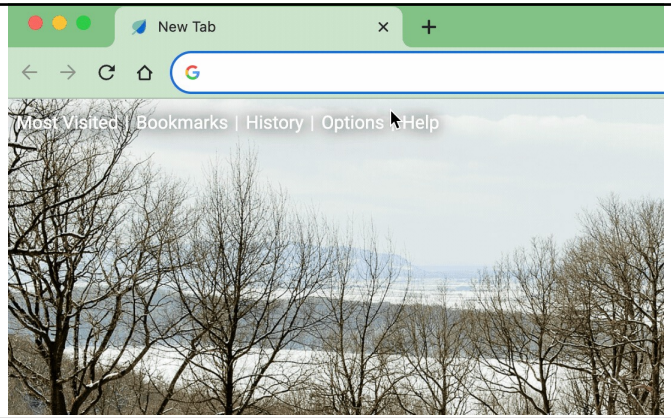
37

Handle Form Submission (GET)

- When the user submits the form, the form data is sent as query parameters in the URL to the /submit route.
- In the /submit route handler, we use req.query to access the form data.

```
app.get('/submit', (req, res) => {
    const {a, b, op} = req.query
    if (op == '+') res.send(`${a} + ${b} = ${a+b}`)
    if (op == '-') res.send(`${a} - ${b} = ${a-b}`)
    if (op == '*') res.send(`${a} * ${b} = ${a*b}`)
    if (op == '/') res.send(`${a} / ${b} = ${a/b}`)
    else res.send('Invalid operation')
})
```


38



39

Using Mathematical Operators on String

- For adding two number-like strings, we need to convert them to number first with the parseInt() function

```
console.log('5' - '2') // 3
console.log('5' * '2') // 10
console.log('5' / '2') // 2.5
console.log('5' + '2') // 52
```

```
console.log(parseInt('5') + parseInt('2'))
```


40

Handling POST Request

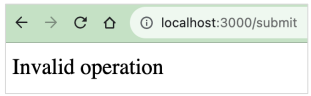
41

Handling POST Request

- Unlike GET requests, where the data is included in the URL's query string and can be easily accessed using req.query, POST requests typically send data in the request body.

```
<form action="/submit" method="post">
    Num 1<input type="text" name="a" > <br/>
    Num 2<input type="text" name="b" > <br/>
    <!-- other elements -->
</form>
```

```
app.post('/submit', (req, res) => {
    const {a, b, op} = req.query
    if (op == '+') res.send(`${a} + ${b} = ${a+b}`)
    if (op == '-') res.send(`${a} - ${b} = ${a-b}`)
    if (op == '*') res.send(`${a} * ${b} = ${a*b}`)
    if (op == '/') res.send(`${a} / ${b} = ${a/b}`)
    else res.send('Invalid operation')
})
```



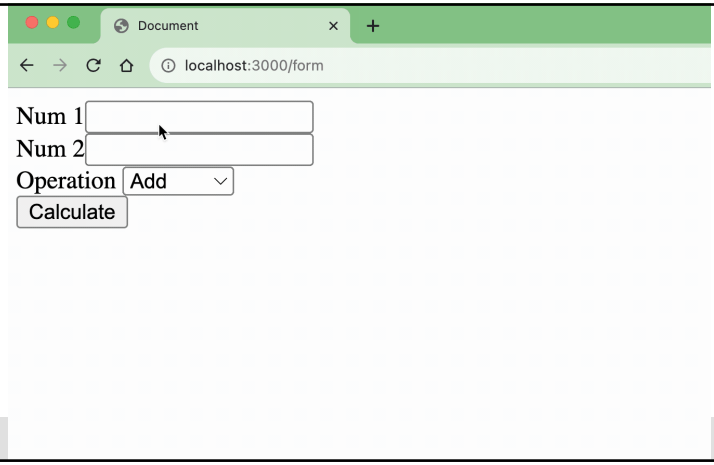
42

Handling POST Request

- To access request body data, you need to use middleware to parse and make it available in the req.body object.
- The most commonly used ones are body-parser and the built-in express.json() and express.urlencoded() middlewares.

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

app.use(express.urlencoded({extended: true})) // handle post form submission
app.use(express.json()) // handle json post request

app.post('/submit', (req, res) => {
    const {a, b, op} = req.body
    // perform processing here
})
```


43



44

Middlewares

45

Middlewares

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

```
app.get('/', (req, res) => {
    res.send('Welcome to Express')
})
```

Registering a route handler in Express

```
(req, res) => {
    res.send('Welcome to Express')
}
```

An example of a middleware

- Middleware functions are chained together, and each function in the chain has access to the request and response objects, as well as the next function in the chain.

46

Middlewares

- In Express.js, when you define multiple route handlers for the same route, the first matching route handler will be executed, and subsequent ones will be ignored.

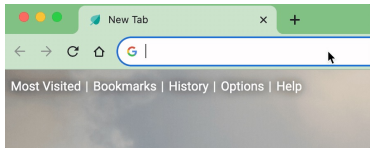
```
app.get('/home', (req, res) => {
    res.send('Home Page 1')
})

app.get('/home', (req, res) => {
    res.send('Home Page 2')
})
```

47

Middlewares

- When a request is made to "/home", the first route handler will be executed. It will print "Home Page 1" to the console but will not send a response to the client browser.

```
app.get('/home', (req, res) => {
    //res.send('Home Page 1')
    console.log('Home Page 1')
})

app.get('/home', (req, res) => {
    res.send('Home Page 2')
})
```

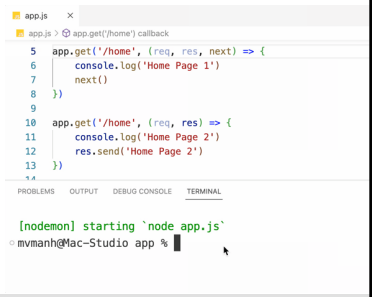


48

Middlewares

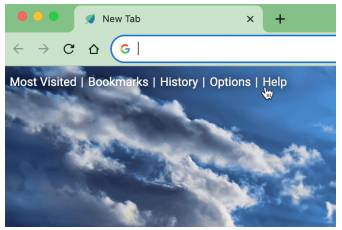
- If you want to have multiple handlers for the same route and execute them all, you can use the next function to pass control to the next middleware function.

```
app.get('/home', (req, res, next) => {
    // res.send('Home Page 1') MUST NOT CALL
    console.log('Home Page 1')
    next()
})

app.get('/home', (req, res) => {
    console.log('Home Page 2')
    res.send('Home Page 2')
})
```



49

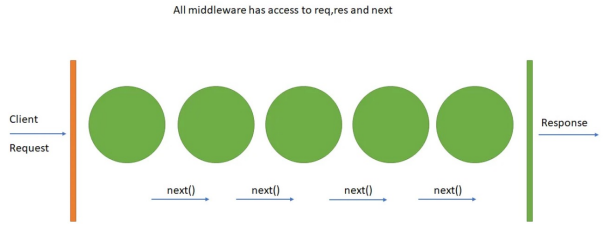
```
app.get('/home', (req, res, next) => {
    req.user = 'admin'
    req.random = Math.random() * 10
    next()
})

app.get('/home', (req, res, next) => {
    const {random} = req
    req.lang = (random > 5 ? 'English' : 'Vietnamese')
    next()
})

app.get('/home', (req, res) => {
    const {random, user, lang} = req
    res.json({random, user, lang})
})
```



50

Middlewares

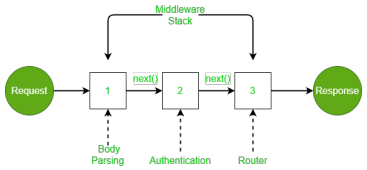
- Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser.

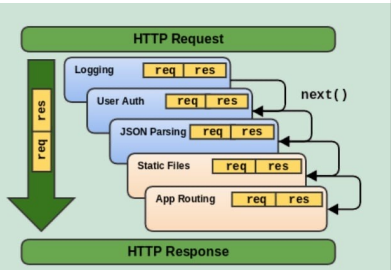


51

Advantages of using middleware

- Middleware can be used to perform common tasks, such as logging, validation, and encryption.
- Middleware can be chained together to create complex functionality. This can make your applications more powerful and flexible.
- Middleware can be used to handle errors.

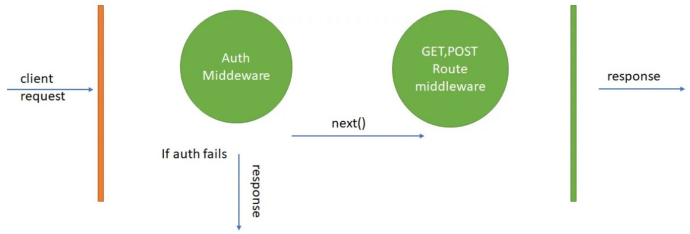




52

An Auth Middleware Example

- Authentication middleware allows you to modularize your authentication logic. You can create a middleware function that handles authentication, and then use it across different routes and endpoints in your application.



53

Creating Custom Middlewares

- To create custom middlewares, you need to define a function that takes three arguments: req, res, and next

```
const loggingMiddleware = (req, res, next) => {
    const {method, url} = req
    console.log(`New ${method} request to ${url}`)
    next() // or res.send()
}
```

- If the current middleware function does not end the request-response cycle, it must call **next()** to pass control to the next middleware function. Otherwise, the request will be left **hanging**.

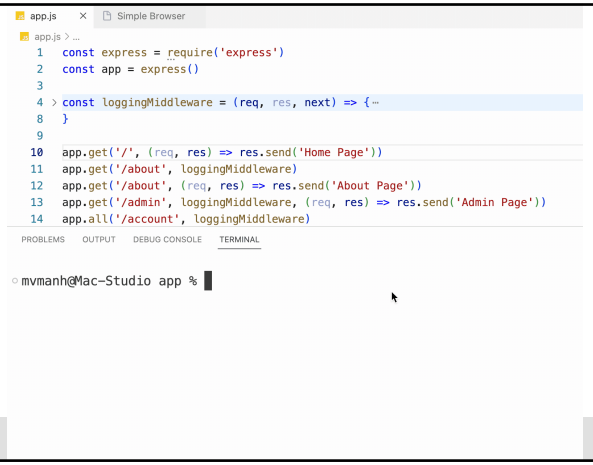
54

Using Middleware in Routes

- Middlewares must be added to a chain before being used:

```
app.get('/', (req, res) => res.send('Home Page'))

app.get('/about', loggingMiddleware)
app.get('/about', (req, res) => res.send('About Page'))

app.get('/admin', loggingMiddleware, (req, res) => res.send('Admin Page'))

app.all('/account', loggingMiddleware)
```


55



56

Using Middleware in Routes

- To use the a defined middleware in all Express.js routes, you can add it using the app.use() method:

```
app.use(loggingMiddleware)
```

- Middleware in Express.js is executed in the order they are added to the application. This order can significantly impact the behavior of your application

```
app.use(loggingMiddleware)
app.use(express.urlencoded())
app.use(express.json())
```


57

Middleware Best Practices

- Keep your middlewares focused on a single task. This makes your code more modular and maintainable.
- Use built-in middlewares whenever possible to handle common tasks like parsing request bodies and serving static files.
- Define error-handling middlewares to centralize error handling and provide consistent error responses.
- Pay attention to the order of middlewares. The order can affect the behavior of your application.

58

Middleware Types

- An Express application can use the following types of middleware:
  - Application-level middleware
  - Router-level middleware
  - Error-handling middleware
  - Built-in middleware
  - Third-party middleware
- You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

59

Built-in Middlewares

- Express.js comes with several built-in middlewares that can be easily integrated into your application. Some of the most common built-in middlewares include:
  - express.json(): Parses incoming JSON data and populates the req.body object.
  - express.urlencoded(): Parses incoming URL-encoded data and populates the req.body object.
  - express.static(): Serves static files like images, stylesheets, and JavaScript files.
  - morgan: A logging middleware for logging HTTP requests.

60

Third-party middleware

- Express has a rich ecosystem of third-party middleware packages that can significantly enhance your application's functionality and simplify development:
  - Logging Middleware
  - CORS Middleware
  - Compression Middleware
  - Error Handling Middleware

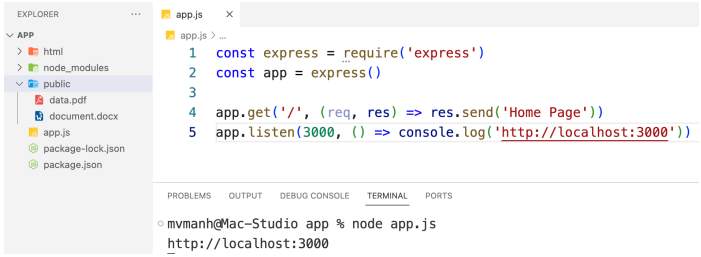
61

Static middleware

62

The static middleware

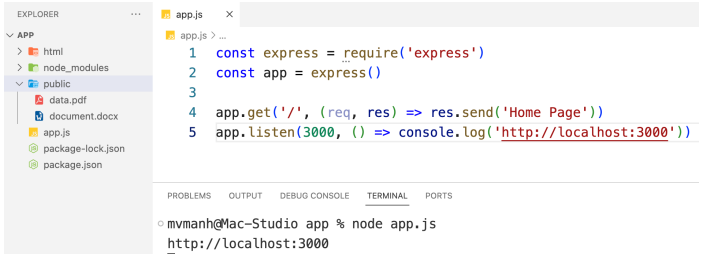
- Consider the following source code and folder structure:



63

The static middleware

- What happen when we access http://localhost:3000/public/data.pdf ?



64

The static middleware

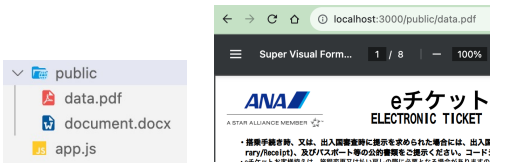
- What happen when we access http://localhost:3000/public/data.pdf ?

65

The static middleware

- Want to download the data.pdf file when accessing http://localhost:3000/public/data.pdf ?
- The traditional way to solve the problem:

```
app.get('/public/data.pdf', (req, res) => {
    const file = path.join(__dirname, 'public', 'data.pdf')
    res.sendFile(file)
})
```



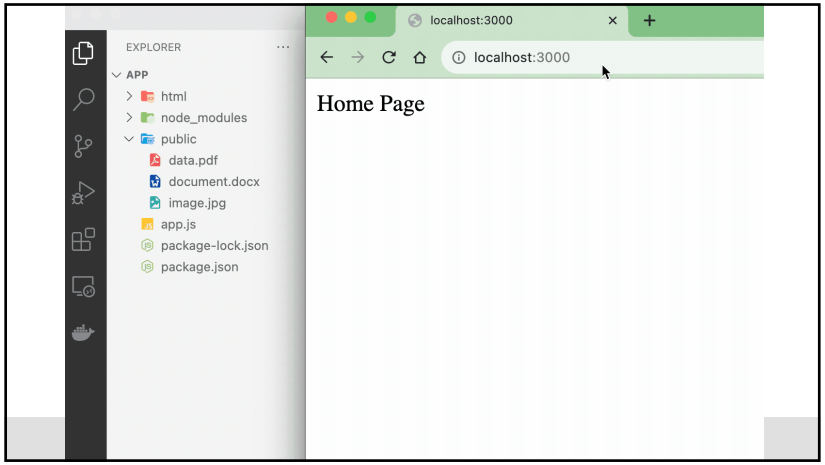
66

The static middleware

- The `express.static()` in Express.js is used to serve static files such as images, CSS files, and JavaScript files. It is a built-in middlewamiddlewarere function that is based on the serve-static module.
- The `express.static()` middleware takes two arguments:
  - The `root` argument specifies the root directory from which to serve static assets.
  - The `options` argument is an optional object that can be used to configure the middleware.
- The following is an example of how to use the `express.static()` middleware:

```
app.use(express.static('public'))
app.use('/files', express.static('public'))
```


67



68

Morgan middleware

69

The Morgan middleware

- Morgan is a popular Node.js middleware for logging HTTP requests.
- It can be used to log requests, errors, and more to the console or to a file.
- To use Morgan in your Express.js application, you will first need to install it using npm or yarn:

```
npm install morgan
```

- Once Morgan is installed, you can add it to your Express.js application using the .use() method:

```
const morgan = require('morgan');
const app = express();
app.use(morgan('dev'));
```


70

71

Morgan Predefined Formats

- There are various pre-defined formats provided:
  - combined: Standard Apache combined log output.
    `:remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version" :status :res[content-length] ":refe`
  - Common: standard Apache common log output.
    `:remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version" :status :res[content-length]`
  - dev: Concise output colored by response status for development use.
    `:method :url :status :response-time ms - :res[content-length]`
  - tiny: The minimal output.
    `:method :url :status :res[content-length] - :response-time ms`

72

Multer middleware

73

The Morgan middleware

- Morgan is a popular Node.js middleware for logging HTTP requests.
- It can be used to log requests, errors, and more to the console or to a file.
- To use Morgan in your Express.js application, you will first need to install it using npm or yarn:

```
npm install morgan
```

- Once Morgan is installed, you can add it to your Express.js application using the .use() method:

```
const morgan = require('morgan');
const app = express();
app.use(morgan('dev'));
```


74