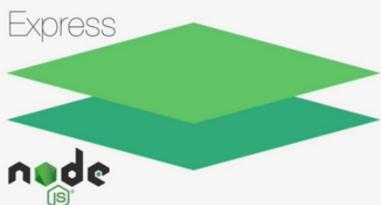


Section 6: Express: Let's Start Building the Natours API!

23 May 2025 21:21



WHAT IS EXPRESS, AND WHY USE IT?



- 👉 Express is a minimal node.js framework, a higher level of abstraction;
- 👉 Express contains a very robust set of features: complex routing, easier handling of requests and responses, middleware, server-side rendering, etc.;
- 👉 Express allows for rapid development of node.js applications: *we don't have to re-invent the wheel*;
- 👉 Express makes it easier to organize our application into the MVC architecture.

From now on, we'll be using **Express** in our application.

So, what is Express, and why use it?

Express is a minimal and flexible Node.js framework. It's built on top of Node.js, offering a higher level of abstraction while still being 100% Node.js under the hood. It's also the most popular Node.js framework out there—kind of the standard.

Express provides a powerful set of features out of the box:

- Complex routing
- Simplified request/response handling
- Middleware support
- Server-side rendering
- ...and more.

This makes building Node.js applications way faster—you won't have to rewrite the same logic or reinvent the wheel for things like routing or templating.

Remember how basic routing felt clunky earlier in the course? With Express, it's much simpler—like 10x easier. It also helps organize your app using the **MVC architecture**, a common and efficient pattern we'll explore throughout the course.

In short: Express makes working with Node.js easier, faster, and way more enjoyable.



A screenshot of the Postman homepage. The header includes the Postman logo, navigation links for "Product", "Plans & Pricing", "Learning Center", "Support", and a "SIGN IN" button. The main content features the headline "Postman Simplifies API Development." with a subtext "Get easy, API-First solutions with the industry's only complete API Development Environment." A "Get Started" button is visible. To the right, there's a large illustration of a rocket launching into space.

A screenshot of the Postman homepage with a red arrow pointing from the text "The Only Complete API Platform" down to three circular icons at the bottom. These icons represent different features: a database icon, a window icon, and a key icon. Above the icons, a "Download the App" button is shown.

The screenshot shows the homepage of the Dog API. At the top, there's a navigation bar with links for Chrome, File, Edit, View, History, Bookmarks, People, Window, and Help. Below the navigation is a header with the text "The internet's biggest collection of open source dog pictures." A red arrow points from the URL bar at the top left towards the header. Another red arrow points down from the "Fetch!" button to the JSON response example.

Documentation

Breeds list

About

Submit your dog

STAY INFORMED OF FUTURE API DEVELOPMENTS

Email [Join](#)

(Highly recommended if you intend to use the API for any major projects)

Buy me a dog treat

JSON

```
{ "status": "success", "message": "https://images.dog.ceo/breeds/dalmatian/cooper1.jpg" }
```

IMAGE

The screenshot shows the Dog API homepage within the Postman application interface. The URL in the address bar is https://dog.ceo/api/breeds/image/random. The main content area displays the same "open source dog pictures" message and the "Buy me a dog treat" button. On the right side, the Postman interface shows a request configuration for a GET request to https://dog.ceo/api/breeds/image/random. A red arrow points from the "Send" button to the JSON response preview area.

Postman

File Edit View Window Help

Now Import Runner Invite

POSTMAN

GET https://dog.ceo/api/breeds/image/random

No Environment

Send Save

Params Authorization Headers Body Pre-request Script Tests Cookies Code Comments

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies (1) Headers (1) Test Results

Pretty Raw Preview JSON

```
1: { "status": "success", "message": "https://images.dog.ceo/breeds/elkhound-norwegian/n02091467_2968.jpg" }
```



Let's now finally setup Express, create a simple server and do some basic routing just to get an initial feeling of how we actually work in Express.

A screenshot of the Visual Studio Code interface. The title bar shows 'jonas.io' and the workspace name '4-natours'. The Explorer sidebar on the left lists '4-NATOURS' with sub-folders 'dev-data', 'public', '.eslintrc.json', and '.prettierrc'. The bottom right corner of the code editor has a small 'DRAFT' watermark. The main area is a terminal window titled 'node' with the command 'npm init' entered. A red arrow points from the text 'This utility will walk you through creating a package.json file.' to the word 'init'.

```
jonas.io > 4-natours > npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ⌘-C at any time to quit.
package name: (4-natours) |
```

A screenshot of the Visual Studio Code interface. The title bar shows 'jonas.io' and the workspace name '4-natours'. The Explorer sidebar on the left lists '4-NATOURS' with sub-folders 'dev-data', 'public', '.eslintrc.json', and '.prettierrc'. The bottom right corner of the code editor has a small 'DRAFT' watermark. The main area is a terminal window titled 'node' with the command 'npm i express@4' entered. A red box highlights the entire JSON configuration object in the code editor.

```
"version": "1.0.0",
"description": "Learning node, express and mongoDB",
"main": "app.js",
"scripts": {
  "test": "echo \\"Error: no test specified\\\" && exit 1"
},
"author": "Jonas Schmedtmann",
"license": "ISC"
```

A screenshot of the Visual Studio Code interface. The title bar shows 'jonas.io' and the workspace name '4-natours'. The Explorer sidebar on the left lists '4-NATOURS' with sub-folders 'dev-data', 'public', '.eslintrc.json', and '.prettierrc'. The code editor shows the 'package.json' file with the following content. A red arrow points from the text 'name: "natours"' to the line number 1.

```
1  {
2    "name": "natours",
3    "version": "1.0.0",
4    "description": "Learning node, express and mongoDB",
5    "main": "app.js",
6    "scripts": {
7      "test": "echo \\"Error: no test specified\\\" && exit 1"
8    },
9    "author": "Jonas Schmedtmann",
10   "license": "ISC"
11 }
```

It's convention to have all kinds of configuration in the 'app.js' file.

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.status(200).send('Hello from the server side!');
});
const port = 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

```
jonas.io 4-natours nodemon app.js
[nodemon] 1.18.11
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching: ***!
[nodemon] starting 'node app.js'
App running on port 3000...
```

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.status(200).send('Hello from the server side!');
});
const port = 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

```
jonas.io 4-natours nodemon app.js
[nodemon] 1.18.11
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching: ***!
[nodemon] starting 'node app.js'
App running on port 3000...
```

Postman

GET 127.0.0.1:3000

127.0.0.1:3000

Send Save

Params Authorization Headers Body Pre-request Script Tests

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results

Pretty Raw Preview HTML

Status: 200 OK Time: 20 ms Size: 27 B Download

Hello from the server side!

We have used get method. So won't work. Express automatically sends this response if you try to send.

The screenshot shows the Postman interface. A red arrow points from the URL field 'POST 127.0.0.1:3000' to the 'Send' button. The response pane displays the following XML error message:

```

<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<h1>Cannot POST /</h1>
</body>
</html>

```

The screenshot shows the VS Code editor with the 'app.js' file open. The code defines a POST endpoint at the root path ('/') that returns a JSON response with the message 'You can post to this endpoint...'. Two lines of code are circled in red:

```

const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.json({ message: 'Hello from the server side!', app: 'Natours' });
});
app.post('/', (req, res) => {
  res.send('You can post to this endpoint...');
});
const port = 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}`);
});

```

Now you can post

The screenshot shows the Postman interface. A red arrow points from the URL field 'POST 127.0.0.1:3000' to the 'Send' button. The response pane displays the message: 'You can post to this endpoint...'.

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

EXPRESS: LET'S START BUILDING THE
NATOURS API

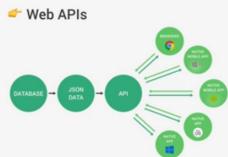
LECTURE

APIS AND RESTFUL API DESIGN

WHAT IS AN API ANYWAY?

API

Application Programming Interface: a piece of software that can be used by another piece of software, in order to allow applications to talk to each other.



👉 Web APIs

- 👉 Node.js' fs or http APIs ("node APIs");
- 👉 Browser's DOM JavaScript API;
- 👉 With object-oriented programming, when exposing methods to the public, we're creating an API;
- 👉 ...

👉 But, "Application" can be other things:

Now that we know what **Express** is, we're almost ready to build our API. But before diving in, let's take a step back and understand **APIs** at a higher level—especially the **REST architecture**, which is the most commonly used API design today.

Why This Matters:

Understanding REST help you know exactly what you're building throughout the course.

So, What Is an API?

API stands for *Application Programming Interface*. At a high level, it's a piece of software that allows different applications to communicate with each other.

We've already touched on **web APIs**—where a server responds with data when a client makes a request. In this setup, you have two apps (server + client) talking to each other. That's the kind of API we'll build here, and it's the most common kind.

But APIs go beyond the web. They're not just about sending data or using JavaScript.

APIs Exist Everywhere:

- **Node.js modules** like fs or http expose APIs. When you use fs.readFile(), you're using the **File System API**.
- In the browser, when we manipulate the DOM, we're using the **DOM API**, not raw JavaScript.
- If you create a class in Java and expose public methods, those methods form the class's API—allowing other code to interact with it.

So, **API** is a broad term—it simply means a way for software to interact with other software.

But for us...

We're focused on **web APIs**, and more specifically, building them using the **REST architecture**.

Let's dive into REST and see how it guides the way we structure our APIs.

THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless

REST, which stands for **Representational State Transfer**, is a way to build web APIs that are logical, structured, and easy to use. Remember—APIs are built for developers to consume, so making them simple and intuitive is key. When we build **RESTful APIs** (APIs that follow the REST architecture), we just need to stick to a few core principles:

◆ 1. Use Logical Resources

Break your API into **resources**—logical pieces of data like users, posts, or products.

◆ 2. Resource-Based URLs

Expose these resources using **clean, structured URLs** like:

GET /users, POST /products, DELETE /orders/123

Avoid putting actions in the URL—use the right HTTP method instead.

◆ 3. Use Correct HTTP Methods

Let the **HTTP method** define the action:

- GET – Read data
- POST – Create data
- PUT/PATCH – Update data
- DELETE – Delete data

Don't misuse the URL to indicate operations.

◆ 4. Use JSON Format

Data sent and received should typically be in **JSON**, with a consistent and clean structure.

◆ 5. Be Stateless

REST APIs should be **stateless**, meaning each request from the client must contain all the information needed to process it. The server doesn't store any state between requests.

That's the high-level overview. Now let's break down each of these principles and apply them as we build our API.

THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless

👉 **Resource:** Object or representation of something, which has data associated to it. Any information that can be **named** can be a resource.

tours users reviews



◆ What Is a Resource?

In REST, the **key abstraction** of information is a **resource**. That means all the data we want to expose through our API should be broken down into **logical resources**.

But what exactly is a resource?

In the context of REST, a **resource** is simply an object or representation of something that holds data. Think of examples like:

- tours
- users
- reviews

In short: **any piece of information that can be named can be a resource**.

👉 The key rule? It should be a **noun**, not a **verb**.

So rather than thinking in terms of actions (like `getTours`), we think in terms of **entities** (tours) and then perform actions using the appropriate **HTTP method**.

THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless

👉 **Resource:** Object or representation of something, which has data associated to it. Any information that can be **named** can be a resource.

tours users reviews



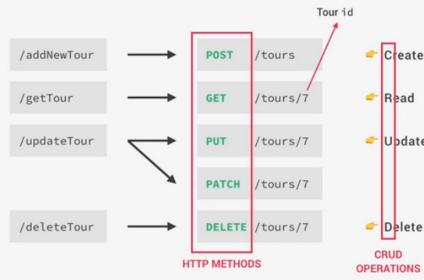
Verb is now the HTTP method.

PUT => client is supposed to send the entire updated object

PATCH => it is supposed to send only the part of the object that has been changed.

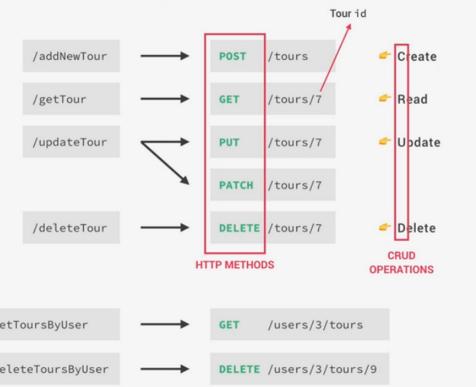
THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless



THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless



JSON (JavaScript Object Notation) is a **lightweight data interchange format** that's widely used in web APIs—regardless of the programming language. It's not just tied to JavaScript! JSON is super popular because it's:

- Easy for humans to read and write
- Easy for machines to parse and generate

You've probably noticed that JSON looks similar to a **JavaScript object**—with its key-value pairs. But there's one key difference:

👉 All keys must be strings.

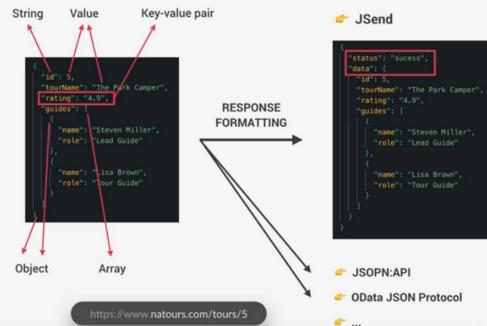
Values can be:

- Strings
- Numbers
- Booleans (true/false)
- Other objects
- Arrays

It's super straightforward! And once you've seen a few examples, you'll quickly get the hang of it.

THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless



Before sending data back to the client, we usually apply **simple response formatting**. One common and easy-to-use standard is called **Jsend**.

With **Jsend**, we wrap our response in a new object that includes:

- A status field (success, fail, or error)
- A data field that holds the actual response data

Example:

```
{  
  "status": "success",  
  "data": {  
    "yourKey": "yourValue"  
  }  
}
```

This wrapping technique is called **enveloping**, and it's a common practice to improve consistency and help prevent certain security issues.

There are also other response formatting standards like **Jsend:API** and the **OData JSON Protocol**, if you ever want to explore more. But for this course, we'll keep it simple with plain Jsend.

THE REST ARCHITECTURE

- 1 Separate API into logical resources
- 2 Expose structured, resource-based URLs
- 3 Use HTTP methods (verbs)
- 4 Send data as JSON (usually)
- 5 Be stateless



A **RESTful API should always be stateless**—but what does that actually mean?

In a **stateless API**, all state (like user login status or current pagination page) is handled **on the client**, *not* on the server.

That means **every request must contain all the necessary information** for the server to process it. The server should never rely on previous requests to understand or respond to the current one.

👉 For example, imagine you're on page 5 of a tour list and want to go to page 6.

If your endpoint is /tours/nextPage, the server would need to remember you were on page 5.

That's *stateful*—and **not allowed in REST**.

Instead, the client should directly request /tours?page=6, making it clear and independent.

No memory needed on the server = stateless = RESTful.

THE REST ARCHITECTURE

1 Separate API into logical resources

2 Expose structured, resource-based URLs

3 Use HTTP methods (verbs)

4 Send data as JSON (usually)

5 Be stateless

👉 Stateless RESTful API: All state is handled **on the client**. This means that each request must contain **all** the information necessary to process a certain request. The server should **not** have to remember previous requests.

👉 Examples of state: `loggedIn` `currentPage`

`currentPage = 5`

`GET /tours/nextPage` **BAD** WEB SERVER `nextPage = currentPage + 1` `send(nextPage)` STATE ON SERVER

`GET /tours/page/6` STATE COMING FROM CLIENT WEB SERVER `send(6)`

JONAS.IO SCHMIDTMANN

SECTION

EXPRESS: LET'S START BUILDING THE NATOURS API!

LECTURE

STARTING OUR API: HANDLING GET REQUESTS

ALL TOURS

LOG IN SIGN UP

THE SEA EXPLORER

MEDIUM 7-DAY TOUR
Exploring the jaw-dropping US east coast by foot and by boat

📍 Miami, USA | June 2021 | 4 stops | 15 people

\$497 per person | 4.8 rating (5) | DETAILS

THE FOREST HIKER

EASY 5-DAY TOUR
Breathtaking hike through the Canadian Banff National Park

📍 Banff, CAN | April 2021 | 3 stops | 25 people

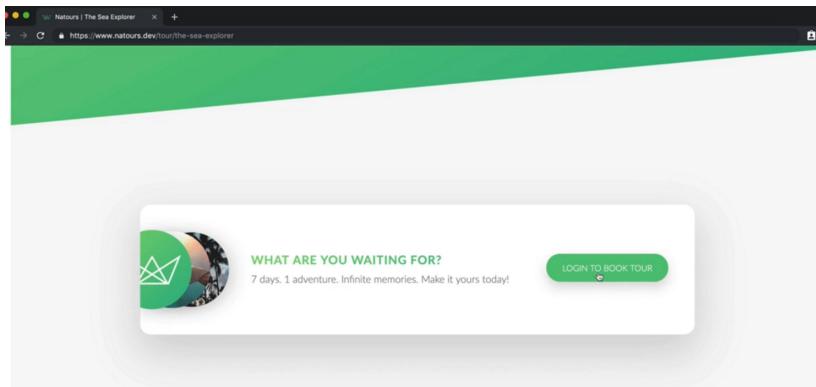
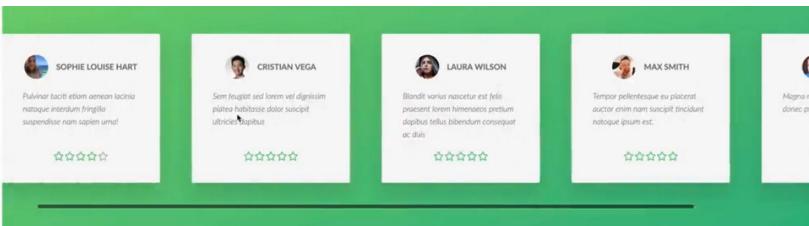
\$397 per person | 5 rating (9) | DETAILS

THE SNOW ADVENTURER

DIFFICULT 4-DAY TOUR
Exciting adventure in the snow with snowboarding and skiing

📍 Aspen, USA | January 2022 | 2 stops | 10 people

\$997 per person | 4.6 rating (5) | DETAILS



Project Overview – Natours

This app is like a travel booking platform:

- 🌎 Browse different **tours**
- 📊 See **details, maps, and user reviews**
- 🧑 Sign up, log in, and book tours

BUT — that fancy UI stuff you just saw? We're **not** doing that first.

Phase 1: Build the API

Before any frontend magic happens, we're going full backend dev mode:

- Build out the **RESTful API** with Express.js
- Connect it to **MongoDB** using Mongoose
- Handle resources like:
 - /api/v1/tours
 - /api/v1/users
 - /api/v1/reviews

Basically: we're building the **engine** before putting the body on the car 🚗

💡 Why API First?

“Why not build the UI first? It looks cooler!”

Totally fair, but here's the vibe:

- Learning **backend** (Express + MongoDB) is easier when we just focus on **data**, not visuals.
- It keeps things clean: one thing at a time.
- And when the UI part comes later, boom 💥 — all the data handling is already solid.

Let's look at the api.

```

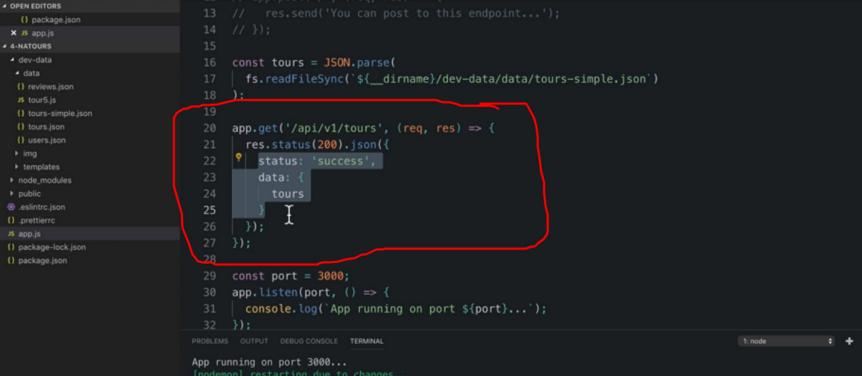
[{"id": "5cb01f4e2f8fb81b56fa185", "name": "The Sea Explorer", "description": "Exploring the jaw-dropping US east coast by foot and by boat.", "imageCover": "Tour-a-cover.jpg", "maxGroupSize": 15, "difficulty": "medium", "role": "guide", "email": "jenny@natours.example.com", "createdAt": "2020-02-20T12:00:00.000Z"}, {"id": "5cb01f4e2f8fb81b56fa185", "name": "The Sea Explorer", "description": "Exploring the jaw-dropping US east coast by foot and by boat.", "imageCover": "Tour-a-cover.jpg", "maxGroupSize": 15, "difficulty": "medium", "role": "guide", "email": "jenny@natours.example.com", "createdAt": "2020-02-20T12:00:00.000Z"}]
  
```

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'New', 'Import', 'Runner', and 'Invite'. The title 'Node.js Course' is displayed above a dropdown menu. On the far right, there are icons for user profile, notifications, and help, followed by 'Upgr...'.

The main workspace has a left sidebar titled 'Collections' with a sub-section 'History'. Below it, a message says 'You don't have any collections' with a 'Create a collection' button. The main area is titled 'Tours' and contains a single API endpoint:

- URL:** https://www.natours.dev/api/v1/tours/5c88fa8cf4afda39709c2955
- Method:** GET
- Preview:** Shows a JSON response with tour details, including coordinates, address, ratings, images, start date, end date, and creation date.

At the bottom, there are tabs for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected, showing the raw JSON response. The status bar at the bottom indicates 'Status: 200 OK', 'Time: 456 ms', 'Size: 4.03 kB', and 'Downloads: 1'.



The screenshot shows the VS Code interface with the following details:

- Left Sidebar (Explorer):** Shows the project structure with files like package.json, app.js, reviews.json, tourSimple.json, users.json, img, templates, node_modules, public, eslintrc.json, prettier.json, and package-lock.json.
- Current File:** app.js
- Code Content (app.js):**

```
13 // res.send('You can post to this endpoint...');  
14 // });  
15  
16 const tours = JSON.parse(  
17 | fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)  
18 );  
19  
20 app.get('/api/v1/tours', (req, res) => {  
21 | res.status(200).json({  
22 | | status: 'success',  
23 | | data: {  
24 | | | tours  
25 | | }  
26 | |});  
27 });  
28  
29 const port = 3000;  
30 app.listen(port, () => {  
31 | console.log(`App running on port ${port}`);  
32 });
```
- Terminal Output:** Shows the command "node app.js" being run and the output "App running on port 3000... [nodemon] restarting due to changes..." repeated three times.
- Bottom Status Bar:** Shows icons for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and a node icon.



Express does not put body data to the request. In order to have that data available, we have to use something called middleware.

```

Code File Edit Selection View Go Debug Terminal Window Help
OPEN EDITORS 1 UNSAVED
app.js
4-NATOURS
dev-data
data
reviews.json
tour5.json
tours-simple.json
tours.json
users.json
img
templates
node_modules
public
.eslintrc.json
.prettierrc
app.js
package-lock.json
package.json
app.js -- app.js
19
20 app.get('/api/v1/tours', (req, res) => {
21   res.status(200).json({
22     status: 'success',
23     results: tours.length,
24     data: [
25       | tours
26     ]
27   });
28 });
29
30 app.post('/api/v1/tours', (req, res) => {
31   |
32 });
33
34 const port = 3000;
35 app.listen(port, () => {
36   | console.log(`App running on port ${port}...`);
37 });
38

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

//*express.json()* is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on *body-parser*.
//*Middleware* is basically just a function that can modify the incoming request data.

```

JS app.js
1 const fs = require('fs');
2 const express = require('express');
3
4 const app = express();
5
6 app.use(express.json()); E
7
8 // app.get('/', (req, res) => {
9 //   res
10 //     .status(200)
11 //     .json({ message: 'Hello from the server side!', app: 'Natours' });
12 // });
13
14 // app.post('/', (req, res) => {
15 //   res.send('You can post to this endpoint...');
16 // });
17
18 const tours = JSON.parse(

```

```

app.post('/api/v1/tours', (req, res) => {
  console.log(req.body); // body is the available now because we used that middleware.
  res.send('Done'); //we always need to send back something in order to finish request-response cycle.
});
```

The screenshot shows the Postman application interface. In the top navigation bar, there are tabs for 'New', 'Import', 'Runner', and 'Postman'. The current tab is 'Postman'. Below the tabs, there's a search bar and a 'History' section with a single entry for 'Natours'. The main workspace has a 'Create New Tour' collection. A POST request is selected with the URL '127.0.0.1:3000/api/v1/tours'. The 'Body' tab is active, showing a JSON payload:

```

1- {
2-   "name": "Test Tour",
3-   "duration": 10,
4-   "difficulty": "easy"
5- }

```

Below the body, the response status is 'Status: 200 OK' with a time of '34 ms' and a size of '207 B'. The response content is a JSON object:

```

{
  "name": "Test Tour",
  "duration": 10,
  "difficulty": "easy"
}

```

The screenshot shows the Visual Studio Code (VS Code) interface. The top menu bar includes 'Code', 'File', 'Edit', 'Selection', 'View', 'Go', 'Debug', 'Terminal', 'Window', and 'Help'. The status bar at the bottom right shows 'jonas.io'. The left sidebar is the 'EXPLORER' view, showing project files like 'app.js', 'tours-simple.json', 'dev-data', 'data', 'reviews.json', 'tour5.js', 'tours.json', 'users.json', 'img', 'templates', 'node_modules', 'public', '.eslintrc.json', '.prettierrc', 'app.js', 'package-lock.json', and 'package.json'. The main editor area contains the 'app.js' file with code for creating tours. The terminal at the bottom shows logs from nodemon:

```

[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
App running on port 3000...
{ name: 'Test Tour', duration: 10, difficulty: 'easy' }

```

```

39
40
41 app.post('/api/v1/tours', (req, res) => {
42   // console.log(req.body); // body is the available now because we used that middleware.
43   const newId = tours[tours.length - 1].id + 1;
44   const newTour = Object.assign({ id: newId }, req.body); // create a new object with the newId and the body of the
45   // request.
46   tours.push(newTour); // push the new tour to the array.
47   fs.writeFileSync(`.${__dirname}/dev-data/data/tours-simple.json`, JSON.stringify(tours), err => {
48     res.status(201).json({
49       status: 'success',
50       data: {
51         tour: newTour,
52       }
53     });
54   }); // why async function? why not just use fs.writeFileSync? because we want to make sure that the server is not
55   // blocked while writing the file. We want to be able to handle other requests while this one is being processed. We
56   // are inside a callback function that is gonna run in the event loop.
57
58   res.send('Done'); //we always need to send back something in order to finish request-response cycle.
59

```

The screenshot shows the Postman interface with a successful API call. The URL is `127.0.0.1:3000/api/v1/tours`. The Body tab contains the following JSON payload:

```

1 - {
2   "name": "Test Tour",
3   "duration": 10,
4   "difficulty": "easy"
5 }

```

The response status is 201 Created, and the response body is:

```

1 - {
2   "status": "success",
3   "data": {
4     "tour": {
5       "id": 1,
6       "name": "Test Tour",
7       "duration": 10,
8       "difficulty": "easy"
9     }
10   }
11 }

```



If we hit

=> `api/v1/tours` ; we would get all the tour

The screenshot shows the Postman interface with a successful API call. The URL is `127.0.0.1:3000/api/v1/tours`. The Headers tab shows a "Content-Type" header set to "application/json". The response status is 200 OK, and the response body is:

```

1 - {
2   "status": "success",
3   "data": [
4     {
5       "tour": [
6         {
7           "id": 1,
8           "name": "The Forest Hiker",
9           "duration": 10,
10          "maxGroupSize": 25,
11          "difficulty": "easy",
12          "ratingAverage": 4.5,
13          "ratingCount": 100,
14          "price": 300,
15          "summary": "Breath-taking hike through the Canadian Banff National Park",
16          "description": "A 10 km hike with some minor elevation, great views, and a waterfall. The trail is well-maintained and suitable for all skill levels. The weather can be unpredictable, so come prepared with rain gear and a map.",
17          "image": "https://www.jonas.io/tour-1-cover.jpg",
18          "images": [
19            ...
20          ]
21        }
22      ]
23    }
24  }
25 }

```

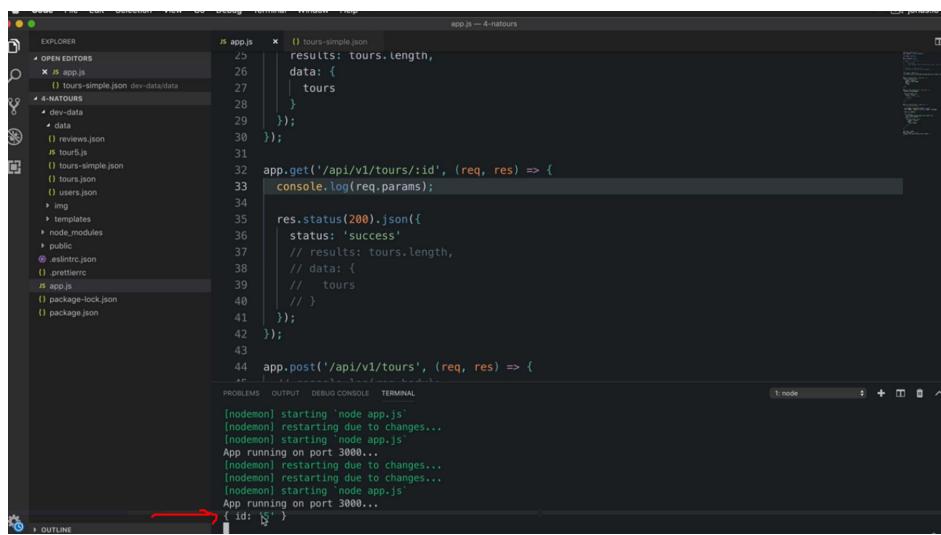
The screenshot shows the Postman interface. In the top navigation bar, there are tabs for 'Import', 'Runner', and '+'. Below the tabs, there's a search bar and a user profile icon. The main area has a title 'Node.js Course' and a sub-section 'Collections'. Under 'Collections', there are two items: 'Get All Tours' and 'Create New Tour'. The 'Get All Tours' item is selected. It shows a 'GET' request to '127.0.0.1:3000/api/v1/tours/5'. A red circle highlights the URL field. Below the request, there are sections for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', 'Tests', 'Cookies', 'Code', and 'Comments'. A message at the bottom says 'Hit the Send button to get a response.'

```

32 app.get('/api/v1/tours/:id', (req, res) => {
33   console.log(req.params);
34
35   res.status(200).json({
36     status: 'success',
37     // results: tours.length,
38     // data: {
39     //   tours
40     // }
41   });
42 });

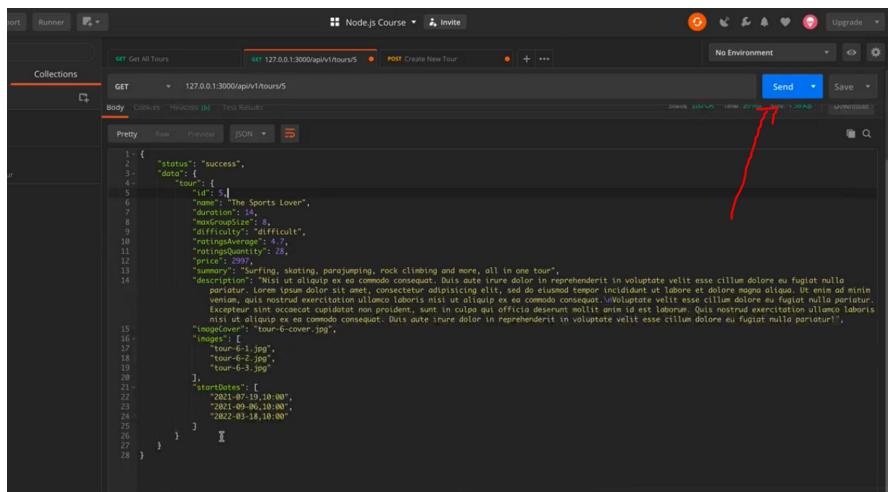
```

Hit from the postman and we have got id to the console.



```
app.get('/api/v1/tours/:id', (req, res) => {
  const id = req.params.id * 1;
  const tour = tours.find(el => el.id === req.params.id);
  res.status(200).json({
    status: 'success',
    data: {
      tour
    }
  });
}

app.post('/api/v1/tours', (req, res) => {
  // console.log(req.body);
});
```



```
app.get('/api/v1/tours/:id', (req, res) => {
  const id = req.params.id * 1;

  if (id > tours.length) {
    return res.status(404).json({
      status: 'fail',
      message: 'Invalid ID'
    });
  }

  const tour = tours.find(el => el.id === id);

  res.status(200).json({
    status: 'success',
    data: {
      tour
    }
  });
});
```

A screenshot of the Postman application interface. At the top, there's a header with 'Postman' and various navigation buttons. Below the header, a search bar and a 'Collections' dropdown are visible. The main workspace shows a list of requests under the 'Natours' collection. A specific request is selected: 'GET 127.0.0.1:3000/api/v1/tours/23'. The 'Params' tab is active, showing a single parameter 'key' with value 'Value'. The 'Body' tab is also visible. On the right side, a red arrow points from the status bar to the response body, which displays a JSON object: { "status": "fail", "message": "Invalid ID" }. The status bar at the bottom indicates 'Status: 404 Not Found'.

A screenshot of the Visual Studio Code (VS Code) code editor. The file 'app.js' is open, showing a Node.js application. A red circle highlights a line of code: 'const tour = tours.find(el => el.id === id);'. A red arrow points from this line to the line above it: 'const const tour: any.d * 1;'. The code editor has a dark theme. The bottom status bar shows 't: node'.



```

M 79 // res.send('Done'); //we always need to send back something in order to finish request-response cycle,
80 });
81
82 app.patch('/api/v1/tours/:id', (req, res) => {
83   const id = req.params.id * 1; // convert the string to a number
84   if(id > tours.length) {
85     return res.status(404).json({
86       status: 'fail',
87       message: 'Invalid ID'
88     });
89   }
90
91   const tour = tours.find(tour => tour.id === id); // find the tour with the given id
92   const updatedTour = Object.assign(tour, req.body); // update the tour with the body of the request
93
94   fs.writeFile(`.${__dirname}/dev-data/data/tours-simple.json`, JSON.stringify(tours), err => {
95     res.status(200).json({
96       status: 'success',
97       data: {
98         tour: updatedTour,
99       }
100     });
101   });

```

You, 2 minutes ago • Uncommitted changes

New Import POST http://localhost:3000/api/ GET http://localhost:3000/api/ GET http://localhost:3000/api/ PATCH http://localhost:3000/api/ + ***

HTTP http://localhost:3000/api/v1/tours/1

PATCH http://localhost:3000/api/v1/tours/1

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Beautify

Body

none form-data x-www-form-urlencoded raw binary JSON

```

1
2   "name": "Israr"
3

```

Send

Status: 200 OK Time: 37 ms Size: 1.26 KB Save Response

Pretty Raw Preview Visualize JSON

```

1
2   "status": "success",
3   "data": {
4     "tour": {
5       "id": 1,
6       "name": "Israr",
7       "duration": 7,
8       "maxGroupSize": 15,
9       "difficulty": "medium",
10      "ratingsAverage": 4.8,
11      "ratingsQuantity": 23,
12      "price": 497,

```



```

104
105 app.delete('/api/v1/tours/:id', (req, res) => {
106   const id = req.params.id * 1; // convert the string to a number
107   if (id > tours.length) {
108     return res.status(404).json({
109       status: 'fail',
110       message: 'Invalid ID'
111     });
112   }
113
114   const tourIndex = tours.findIndex(tour => tour.id === id); // find the index of the tour with the given id
115   tours.splice(tourIndex, 1); // remove the tour from the array
116
117   fs.writeFile(`.${__dirname}/dev-data/data/tours-simple.json`, JSON.stringify(tours), err => {
118     res.status(204).json({
119       status: 'success',
120       data: null,
121     });
122   });
123 });
124 
```

You, 1 second ago • Uncommitted changes

The screenshot shows the Postman interface with a DELETE request to `http://localhost:3000/api/v1/tours/1`. The response details show a status of 204 No Content, time 21 ms, and size 176 B.



Let's now refactor our code a little bit. So basically reorganize some of our route to make the code a lot better.

```

};

> const getTour = (req, res) => { ... };
};

> const updateTour = (req, res) => { ... };
};

> const createTour = (req, res) => { ... };
};

> const deleteTour = (req, res) => { ... };
};

app.get('/api/v1/tours', getAllTours);

app.get('/api/v1/tours/:id', getTour);

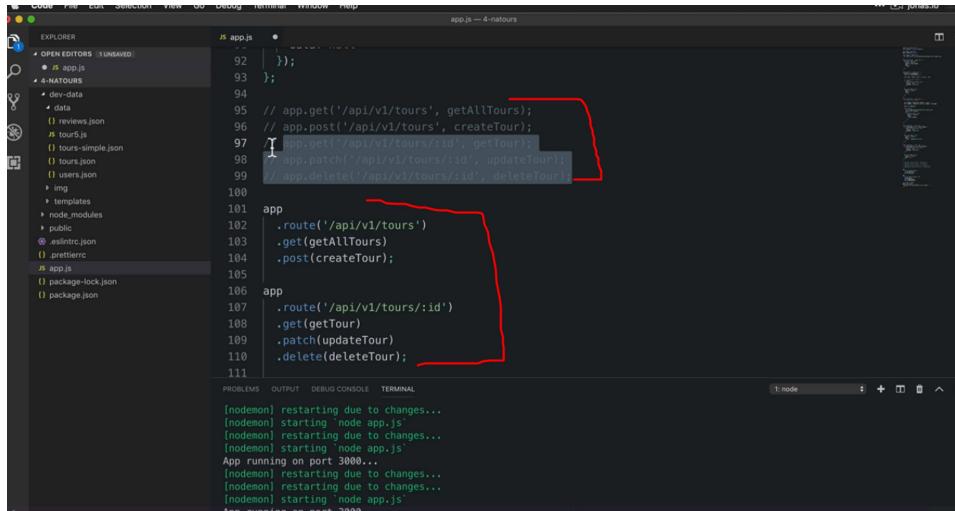
app.post('/api/v1/tours', createTour);
  You, now + Uncommitted changes
app.patch('/api/v1/tours/:id', updateTour);

app.delete('/api/v1/tours/:id', deleteTour);

```

So far it looks good but it is not perfect. For example if we want to change the api version or resource name. we would then have to change it in all of these five places (red marked).

We can do something better.



```

CODE File Edit Selection View Go Debug Terminal Window Help
app.js — 4 natours

EXPLORER OPEN EDITORS UNSAVED
• app.js
• NATOURS
  • day-data
    • data
    • reviews.json
    • tour.js
    • tours.json
    • tours-simple.json
    • users.json
      • img
      • templates
      • node_modules
      • public
      • .eslintrc.json
      • .prettierrc
    • app.js
    • package-lock.json
    • package.json

app.js
92   );
93 };
94
95 // app.get('/api/v1/tours', getAllTours);
96 // app.post('/api/v1/tours', createTour);
97 // app.get('/api/v1/tours/:id', getTour);
98 // app.patch('/api/v1/tours/:id', updateTour);
99 // app.delete('/api/v1/tours/:id', deleteTour);
100
101 app
102   .route('/api/v1/tours')
103     .get(getAllTours)
104     .post(createTour);
105
106 app
107   .route('/api/v1/tours/:id')
108     .get(getTour)
109     .patch(updateTour)
110     .delete(deleteTour);
111

```

The code shows the repetitive nature of the API routes. Red boxes highlight the sections of code where the routes are defined multiple times: once at lines 95-99 and again at lines 101-111. A red bracket also groups the entire set of repeated route definitions.

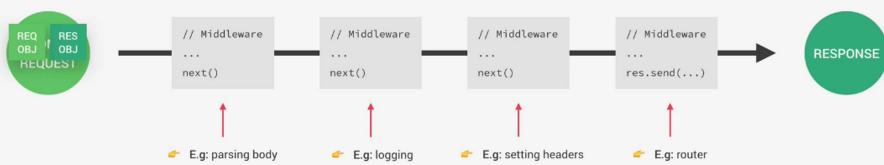


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



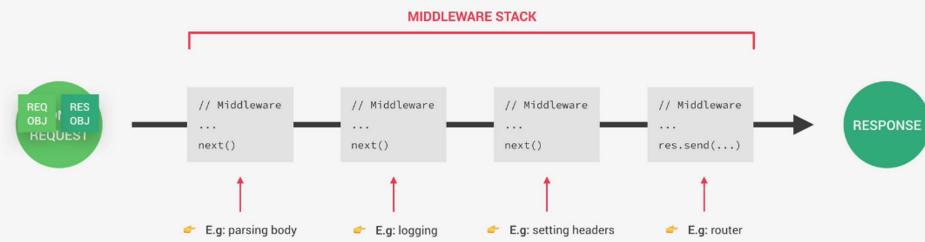
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

👉 "Everything is middleware" (even routers)

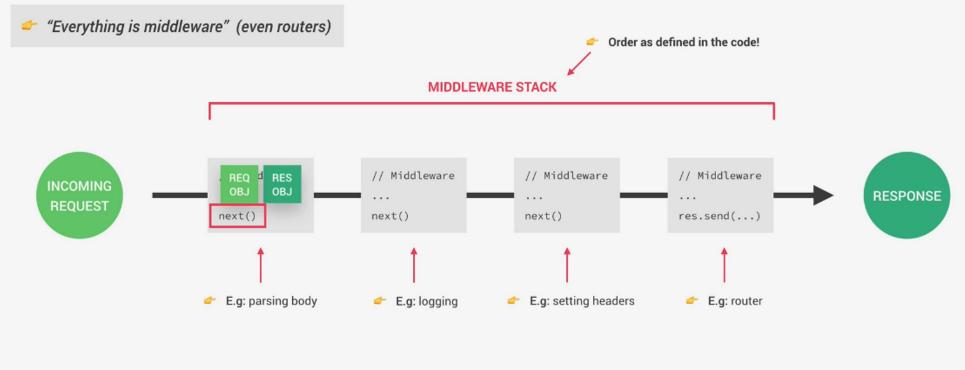


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

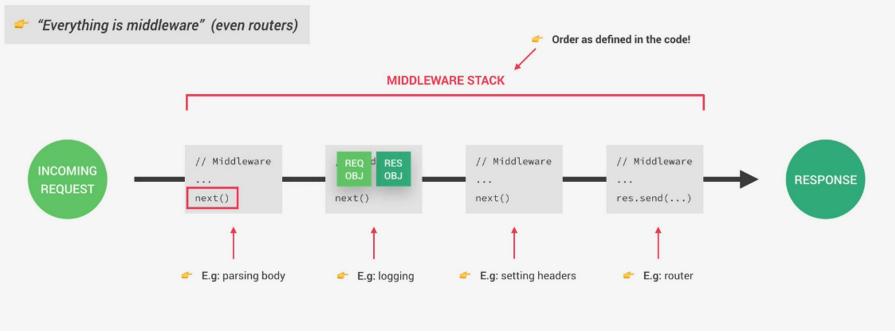
👉 "Everything is middleware" (even routers)



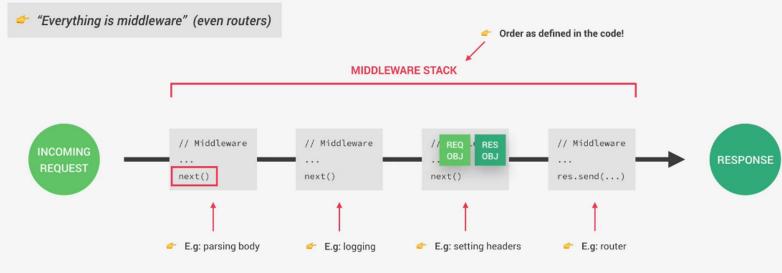
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



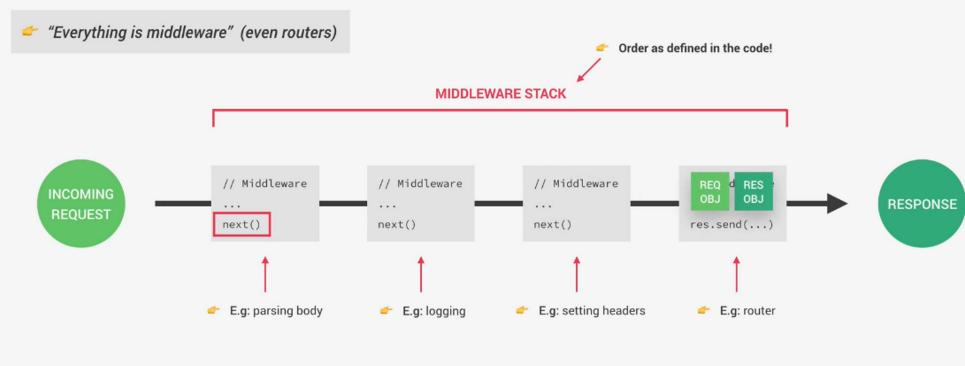
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



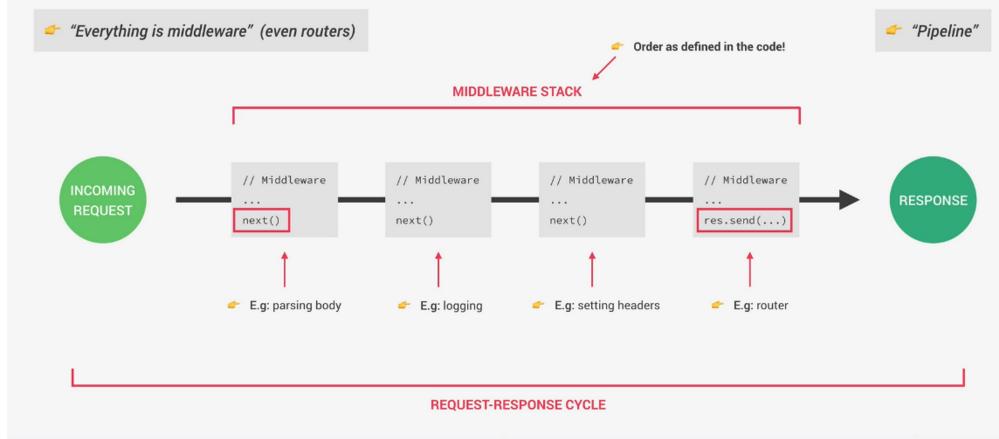
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



In the last few lectures, I hope you've started to grasp the fundamentals of **Express** development. Now it's the perfect time to go a bit deeper—specifically into **middleware** and the **request-response cycle**, which are core to how Express works.

⌚ What is the Request-Response Cycle?

When someone hits your server, Express creates a **request** and **response** object. These are then processed to generate and send back a meaningful response. To handle this process, Express uses **middleware**—functions that run **between** receiving the request and sending the response. They can modify the request/response objects or run any custom logic.

🌐 What is Middleware?

Middleware isn't just about modifying the request or response—it can do all kinds of things. For example:

- express.json() (body parser)
- Logging
- Setting HTTP headers

We've already used express.json() to read the request body—yep, that's middleware too!

Even **route handlers** (like `app.get('/route', ...)`) are middleware—they're just executed for specific routes.

📦 The Middleware Stack

All middleware functions together form the **middleware stack**. And **order matters** here:

- Middleware at the top of your file runs **first**.
- One at the bottom runs **last**.

Each middleware function gets `req`, `res`, and a special `next` function. Calling `next()` passes control to the **next middleware** in line. So, the request and response objects flow through the stack like a **pipeline**—being handled step-by-step.

:green_heart: Route Handler = Final Middleware

Usually, the last middleware is a **route handler**. This is where you send the final response to the client. You typically **don't call `next()` here**—you just `res.send()` or `res.json()` to complete the cycle.

✅ Wrap-up

This **request-response cycle** starts with the incoming request, flows through the middleware stack, and ends with sending the response. It's a **linear process**—like a pipeline.

Understanding this makes working with Express way easier. I wish someone had explained it this clearly when I started—it would've saved a lot of confusion!

Now that you've got this down, you're totally ready to dive deeper into Express 🚀

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

EXPRESS: LET'S START BUILDING THE
NATOURS API!

LECTURE

CREATING OUR OWN MIDDLEWARE

Alright, let's now go ahead and **create our own middleware functions!**

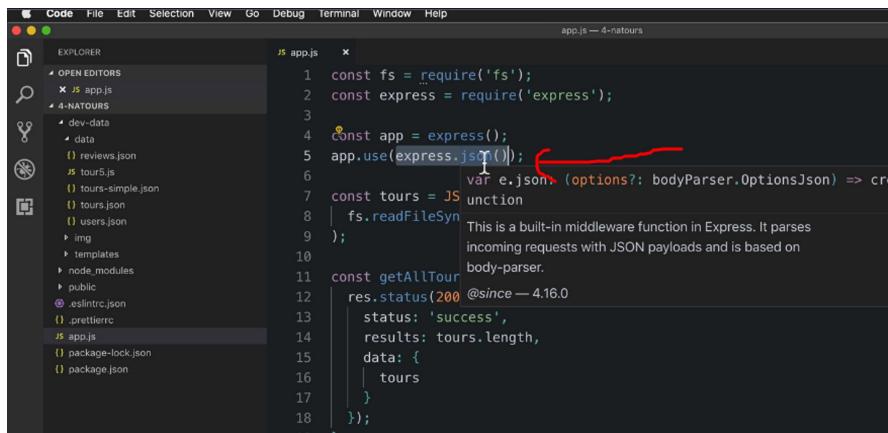
But first, quick reminder—we've actually already used middleware before. Let's take a look. Somewhere earlier in our code, we used:

```
app.use(express.json());
```

This line is key!

- `app.use()` is the method we use to **add middleware** to our **middleware stack**.
- `express.json()` returns a **middleware function**.
- That function is then registered and executed for every incoming request.

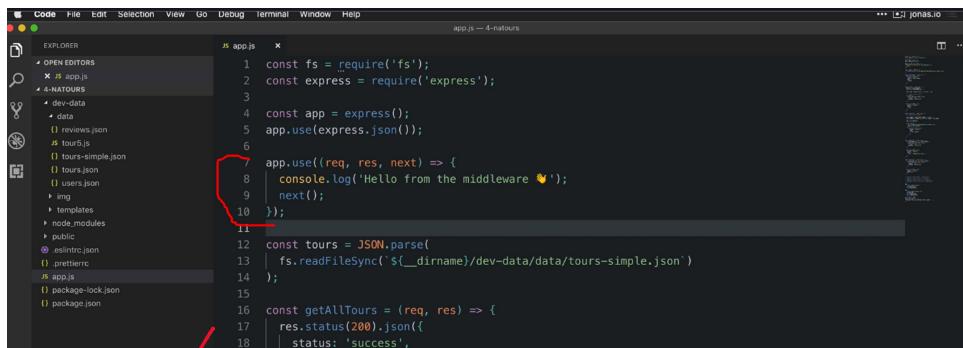
So yeah, we've already been working with middleware—now it's time to build our own! 🎉



```
app.js ━ 4-natours
Code File Edit Selection View Go Debug Terminal Window Help
OPEN EDITORS
  JS app.js
  4-NATOURS
    dev-data
      data
        reviews.json
        tours-simple.json
        users.json
    img
    templates
    node_modules
    public
      .eslintrc.json
      .prettierrc
    app.js
    package-lock.json
    package.json
1 const fs = require('fs');
2 const express = require('express');
3
4 const app = express();
5 app.use(express.json()); // Red box highlights this line
6   var e, json; (options?: bodyParser.OptionsJSON) => create
7 const tours = JSON
8   fs.readFileSync('data/tours-simple.json')
9   This is a built-in middleware function in Express. It parses
10  incoming requests with JSON payloads and is based on
11  body-parser.
12  const getAllTours = (req, res) => {
13    res.status(200).json({
14      status: 'success',
15      results: tours.length,
16      data: {
17        tours
18      }
19    });
20  };
21
```

In each middleware function, we have access to the request and response. We also have the 'next' function.

Never forget to call 'next()' in your middleware function otherwise request would get stuck.



```
app.js ━ 4-natours
Code File Edit Selection View Go Debug Terminal Window Help
jonas.io
OPEN EDITORS
  JS app.js
  4-NATOURS
    dev-data
      data
        reviews.json
        tours-simple.json
        users.json
    img
    templates
    node_modules
    public
      .eslintrc.json
      .prettierrc
    app.js
    package-lock.json
    package.json
1 const fs = require('fs');
2 const express = require('express');
3
4 const app = express();
5 app.use(express.json());
6
7 app.use((req, res, next) => {
8   console.log('Hello from the middleware 🎉');
9   next();
10 });
11
12 const tours = JSON.parse(
13   fs.readFileSync(`${__dirname}/data/tours-simple.json`)
14 );
15
16 const getAllTours = (req, res) => {
17   res.status(200).json({
18     status: 'success',
19   });
20 };
21
```

```
13     fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
14   );
15
16   const getAllTours = (req, res) => {
17     res.status(200).json({
18       status: 'success',
19       results: tours.length,
20       data: {
21
22     }
23   );
24
25   const getTour = (req, res) => {
26     const { id } = req.params;
27     const tour = tours.find((tour) => tour.id === id);
28
29     if (!tour) {
30       return res.status(404).json({ error: 'Tour not found' });
31     }
32
33     res.status(200).json(tour);
34   );
35
36   const updateTour = (req, res) => {
37     const { id } = req.params;
38     const tour = tours.find((tour) => tour.id === id);
39
40     if (!tour) {
41       return res.status(404).json({ error: 'Tour not found' });
42     }
43
44     const { name, image, duration } = req.body;
45
46     tour.name = name;
47     tour.image = image;
48     tour.duration = duration;
49
50     res.status(200).json(tour);
51   );
52
53   const deleteTour = (req, res) => {
54     const { id } = req.params;
55
56     const tour = tours.find((tour) => tour.id === id);
57
58     if (!tour) {
59       return res.status(404).json({ error: 'Tour not found' });
60     }
61
62     const index = tours.indexOf(tour);
63
64     tours.splice(index, 1);
65
66     res.status(200).json({ message: 'Tour deleted successfully' });
67   );
68
69   app.get('/api/v1/tours', getAllTours);
70   app.post('/api/v1/tours', createTour);
71
72   app.route('/api/v1/tours/:id')
73     .get(getTour)
74     .patch(updateTour)
75     .delete(deleteTour);
76
77   app.use((req, res, next) => {
78     console.log('Hello from the middleware 🌟');
79     next(); // call next() to pass control to the next middleware function in the stack
80   });
81
82   app.listen(3000, () => {
83     console.log(`App running on port 3000.`);
84     console.log(`Available routes: ${Object.keys(app.routes()).join(', ')}`);
85   });
86
87   module.exports = app;
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
```

The terminal output shows the application starting on port 3000 and the middleware logging message:

```
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
App running on port 3000...
Hello from the middleware 🌟
```

This middleware is applied to each and every request. That's because we didn't specify any route.

I said before, all route handlers are actually kind of middleware themselves.

```
120
121
122 // app.post('/api/v1/tours', createTour);
123
124 // app.patch('/api/v1/tours/:id', updateTour);
125
126
127 // app.delete('/api/v1/tours/:id', deleteTour);
128
129 app.route('/api/v1/tours')
130   .get(getAllTours)
131   .post(createTour);
132
133 You, 1 second ago • Uncommitted changes
134 app.use((req, res, next) => {
135   console.log('Hello from the middleware 🌟');
136   next(); // call next() to pass control to the next middleware function in the stack
137 });
138
139 app.route('/api/v1/tours/:id')
140   .get(getTour)
141   .patch(updateTour)
142   .delete(deleteTour);
143
```

Let's change the order of the middleware.

From postman, hit this endpoint.

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:3000/api/v1/tours`. The response status is `200 OK`, time is `4 ms`, and size is `7.68 KB`. The response body is a JSON object:

```

{
  "id": "1",
  "name": "Meteora Monasteries Tour",
  "duration": 9,
  "maxGroupSize": 25,
  "difficulty": "easy",
  "ratingAverage": 4.7,
  "ratingQuantity": 37,
  "price": 1200,
  "summary": "Gorgeous hike through the Canadian Banff National Park",
  "description": "Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.\nlorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.",
  "imageCover": "tour-1-cover.jpg",
  "images": [
    "tour-1-1.jpg",
    "tour-1-2.jpg"
  ]
}

```

Now the middleware code does not get executed.

The terminal logs show the application restarting due to changes and running on port 3000:

```

[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...

```

Because we are already sending a response to the client. So, middleware order is very important.

The code editor shows the `getAllTours` function:

```

23
24 const getAllTours = (req, res) => {
25   res.status(200).json({
26     status: 'success',
27     results: tours.length,
28     data: {
29       tours: tours,
30     }
31   });
32 };
33

```

```
Code File Edit Selection View Go Debug Terminal Window Help ... 4-natours

EXPLORER
OPEN EDITORS
JS app.js * 101 app
102   .route('/api/v1/tours')
103   .get(getAllTours)
104   .post(createTour);
105
106 app.use((req, res, next) => {
107   console.log('Hello from the middleware 🌟');
108   next();
109 })
110
111 app
112   .route('/api/v1/tours/:id')
113   .get(getTour)
114   .patch(updateTour)
115   .delete(deleteTour);
116
117 const port = 3000;
118 app.listen(port, () => {
119   | console.log(`App running on port ${port}...`);
120 });

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 🌟
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 🌟
{ id: '2' }
```

```
Code File Edit Selection View Go Debug Terminal Window Help ... 4-natours

EXPLORER
OPEN EDITORS JS app.js * 1 1 const fs = require('fs');
2 const express = require('express');
3
4 const app = express();
5 app.use(express.json());
6
7 app.use((req, res, next) => {
8   | console.log('Hello from the middleware 🌟');
9   | next();
10 });
11
12 app.use((req, res, next) => {
13   | req.requestTime = new Date().toISOString();
14   | next();
15 });
16
17 const tours = JSON.parse(
18   | fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
19 );
20

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 🌟
[nodemon] restarting due to changes...
```

```
Code File Edit Selection View Go Debug Terminal Window Help ... 4-natours

EXPLORER
OPEN EDITORS JS app.js * 13 req.requestTime = new Date().toISOString();
14 | next();
15 })
16
17 const tours = JSON.parse(
18   | fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
19 );
20
21 const getAllTours = (req, res) => {
22   | console.log(req.requestTime);
23
24   res.status(200).json({
25     status: 'success',
26     requestedAt: req.requestTime,
27     results: tours.length,
28     data: {
29       | tours
30     }
31   });
32 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 🌟
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
```



Let's now user a third-party middleware function from npm called Morgan in order to make our development life a bit easier.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[Oh My Zsh] Would you like to update? [Y/n]: n
jonas.io ➜ 4-natours ➔ npm i morgan
npm WARN natours@1.0.0 No repository field.

+ morgan@1.9.1
added 3 packages from 2 contributors and audited 130 packages in 0.905s
found 0 vulnerabilities
  
```

This logging middleware is gonna make our life easier. (Regular Dependency)

```

js app.js
1 const fs = require('fs');
2 const express = require('express');
3 const morgan = require('morgan');
4
5 const app = express();
  
```

```

Code File Edit Selection View Go Debug Terminal Window Help
EXPLORER OPEN EDITORS 1 UNSAVED
app.js
1 const fs = require('fs');
2 const express = require('express');
3 const morgan = require('morgan');
4
5 const app = express();
6
7 // 1) MIDDLEWARES
8 app.use(morgan('dev'));
9
10 app.use(express.json());
11
12 app.use((req, res, next) => {
13   console.log('Hello from the middleware 🎉');
14   next();
15 });
16
17 app.use((req, res, next) => {
18   req.requestTime = new Date().toISOString();
19   next();
20 });
  
```

`morgan()` it returns a middleware function.

Let's pass a invalid id.

The screenshot shows the Postman interface with a collection named 'dev-data'. A request titled 'Get Tour' is selected, showing a GET method to '127.0.0.1:3000/api/v1/tours/233'. The response status is '404 Not Found' with a time of '6 ms' and a size of '208 B'. The response body is a JSON object with 'status': 'fail' and 'message': 'Invalid ID'.

We got 404 in different color.

The screenshot shows a terminal window with Node.js code for a middleware. It includes imports for 'express' and 'morgan', and defines two middlewares: one for general errors and one for request timing. The output shows a log message 'Hello from the middleware' followed by a 404 error for a tour ID. A red arrow points to the '404' in the error message.

```

dev-data
├── data
│   ├── reviews.json
│   └── tour5.js
├── tour-simple.json
└── users.json

app.js
└── package-lock.json
└── package.json

```

```

6 // 1) MIDDLEWARES
7 app.use(morgan('dev'));
8
9 app.use(express.json());
10
11 app.use((req, res, next) => {
12     console.log('Hello from the middleware 🌟');
13     next();
14 });
15
16 app.use((req, res, next) => {
17     req.requestTime = new Date().toISOString();
18     next();
19 });
20
21
Hello from the middleware 🌟
{id: '233'}
GET /api/v1/tours/233 404 0.532 ms - 40

```



The screenshot shows a terminal window with the file 'app.js' open. The code defines three routes for tours and then adds two new routes for users. A red curly brace on the right side of the screen groups the user-related code as 'new'. A red arrow points to a syntax error in the 'getAllUsers' line: 'ReferenceError: getAllUsers is not defined'.

```

DIR: .
EDITOR: app.js
128 app
129     .route('/api/v1/tours/:id')
130     .get(getTour)
131     .patch(updateTour)
132     .delete(deleteTour);
133
134 app
135     .route('/api/v1/users')
136     .get(getAllUsers)
137     .post(createUser);
138
139 app
140     .route('/api/v1/users/:id')
141     .get(getUser)
142     .patch(updateUser)
143     .delete(deleteUser);
144
145 // 4) START SERVER

```

ReferenceError: getAllUsers is not defined
at Object. (/Users/jonas/Desktop/4-natours/app.js:136:8)

```

135 |
136 |   const createUser = (req, res) => {
137 |     res.status(500).json({
138 |       status: 'error',
139 |       message: 'This route is not yet defined!'
140 |     })
141   }
142 |   const updateUser = (req, res) => {
143 |     res.status(500).json({
144 |       status: 'error',
145 |       message: 'This route is not yet defined!'
146 |     })
147   }
148 |
149 |   const getUser = (req, res) => {
150 |     res.status(500).json({
151 |       status: 'error',
152 |       message: 'This route is not yet defined!'
153 |     })
154   }
155 |
156 |   const deleteUser = (req, res) => {
157 |     res.status(500).json({
158 |       status: 'error',
159 |       message: 'This route is not yet defined!'
160 |     })
161   }

```



In this lecture we will now create multiple routers and use a process called 'mounting'. The ultimate goal will be to separate all the code that we have in this file into multiple files.

We have created router and save into this variable.

This 'tourRouter' is actually a middleware.

```

144 |   };
145 |   const tourRouter: Router
146 |   // 3) 'tourRouter' is declared but its value is never read. ts(6133)
147 |   const tourRouter = express.Router();
148 |   app
149 |     .route('/api/v1/tours')
150 |       .get(getAllTours)
151 |       .post(createTour);
152 |
153 |   app
154 |     .route('/api/v1/tours/:id')
155 |       .get(getTour)
156 |       .patch(updateTour);
157 |       .delete(deleteTour);
158 |
159 |   app
160 |     .route('/api/v1/users')
161 |       .get(getAllUsers)

```

[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
App running on port 3000...

We want this 'tourRouter' middleware for this specific route.

```
145
146 // 3) ROUTES
147 app.use('/api/v1/tours', tourRouter);
148
149 const tourRouter = express.Router();
150 tourRouter
151   .route('/api/v1/tours')
152   .get(getAllTours)
153   .post(createTour);
154
155 tourRouter
156   .route('/api/v1/tours/:id')
157   .get(getTour)
158   .patch(updateTour)
159   .delete(deleteTour);
160
161 app
```



Create a new folder named 'routes'

A screenshot of the Visual Studio Code interface. The Explorer sidebar shows a project structure with files like 'app.js', 'tourRoutes.js', 'userRoutes.js', 'reviews.json', 'tours.json', 'users.json', and various configuration files. A red bracket highlights the 'routes' folder in the 'routes' section of the sidebar. The main editor window shows code for 'userRoutes.js'. The terminal at the bottom shows the server starting and receiving requests:

```
userRoutes.js — 4-natours
159
160   .delete(deleteTour);
161
162 userRouter
163   .route('/')
164   .get(getAllUsers)
165   .post(createUser);
166
167 userRouter
168   .route('/:id')
169   .get(getUser)
170   .patch(updateUser)
171   .delete(deleteUser);
172
173 app.use('/api/v1/tours', tourRouter);
174 app.use('/api/v1/users', userRouter);
175
176 // 4) START SERVER
177 const port = 3000;
178
179 app.listen(port, () => {
  console.log(`Hello from the middleware 🌟
2019-04-17T18:24:17.942Z
GET /api/v1/tours 200 4.806 ms - 8745
Hello from the middleware 🌟
GET /api/v1/users 500 0.512 ms - 61
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
`)
```

Cut this code from here

```
File Edit Selection View Go Run Terminal Help
starter
controllers userController.js M x userRoutes.js M x tourController.js M
controllers userController.js M x userRoutes.js M x tourController.js M
routes userRoutes.js ...
You, 55 minutes ago | 1 author (You)
1 const express = require('express');
2 const { getAllUsers, createUser, getUser, updateUser,
3   deleteUser } = require('../controllers/userController');
4
5
6 const router = express.Router(); //We have created router and
7 save into this variable.
8
9 router
10   .route('/')
11     .get(getAllUsers)
12     .post(createUser);
13
14 router
15   .route('/:id')
16     .get(getUser)
17     .patch(updateUser)
18     .delete(deleteUser);
19
20 module.exports = router;
You, 55 minutes ago | 1 author (You)
1 exports.getAllUsers = (req, res) => {
2   res.status(500).json({
3     status: 'error',
4     message: 'This route is not yet defined!'
5   })
6
7
8 exports.createUser = (req, res) => {
9   res.status(500).json({
10     status: 'error',
11     message: 'This route is not yet defined!'
12   })
13
14
15 exports.updateUser = (req, res) => {
16   res.status(500).json({
17     status: 'error',
18     message: 'This route is not yet defined!'
19   })
20
21
22 exports.getUser = (req, res) => {
23   res.status(500).json({
24     status: 'error',
25     message: 'This route is not yet defined!'
26   })
27
28
29 exports.deleteUser = (req, res) => {
30   res.status(500).json({
31     status: 'error',
32     message: 'This route is not yet defined!'
33   })
34 }
```

```

File Edit Selection View Go Run Terminal Help
File tourRoutes.js tourController.js
routes > tourRoutes.js > express
You, 1 hour ago | 1 author (You)
1 const express = require('express');
2 const { getAllTours, createTour, getTour, updateTour, deleteTour } = require
('..controllers/tourController');
3
4
5 //__dirname => current directory of where this file is located
6
7
8 const router = express.Router(); //We have created router and save into this
variable.
9
10 router
11   .route('/')
12     .get(getAllTours)
13     .post(createTour);
14
15
16 router
17   .route('/:id')
18     .get(getTour)
19     .patch(updateTour)
20     .delete(deleteTour);
21
22
23
24 module.exports = router;
25

```

```

File tourRoutes.js tourController.js
controllers > tourController.js > ...
You, 1 hour ago | 1 author (You)
1 const fs = require('fs');
2
3 const tours = JSON.parse(
4   fs.readFileSync(`${__dirname}/../dev-data/data/tours-simple.json`)
5 );
6
7
8 exports.getAllTours = (req, res) => {
9
10   You, 1 hour ago * Uncommitted changes
11   res.json(tours);
12 };
13
14 exports.getTour = (req, res) => {
15
16   You, 1 hour ago * Uncommitted changes
17   const tour = tours.find((tour) => tour.id === req.params.id);
18   if (!tour) return res.status(404).json({ error: 'Tour not found' });
19   res.json(tour);
20 };
21
22 exports.updateTour = (req, res) => {
23
24   You, 1 hour ago * Uncommitted changes
25   const tour = tours.find((tour) => tour.id === req.params.id);
26   if (!tour) return res.status(404).json({ error: 'Tour not found' });
27   tour.name = req.body.name;
28   tour.description = req.body.description;
29   tour.price = req.body.price;
30   tour.image = req.body.image;
31   tour.startDate = req.body.startDate;
32   tour.endDate = req.body.endDate;
33   tour.duration = req.body.duration;
34   tour.rating = req.body.rating;
35   tour.reviewCount = req.body.reviewCount;
36   tour.tourists = req.body.tourists;
37   res.json(tour);
38 };
39
40 exports.createTour = (req, res) => {
41
42   You, 1 hour ago * Uncommitted changes
43   const tour = new Tour({
44     name: req.body.name,
45     description: req.body.description,
46     price: req.body.price,
47     image: req.body.image,
48     startdate: req.body.startDate,
49     enddate: req.body.endDate,
50     duration: req.body.duration,
51     rating: req.body.rating,
52     reviewcount: req.body.reviewCount,
53     touristcount: req.body.touristCount
54   });
55   tour.save();
56   res.json(tour);
57 };
58
59 exports.deleteTour = (req, res) => {
60
61   You, 1 hour ago * Uncommitted changes
62   const tour = tours.find((tour) => tour.id === req.params.id);
63   if (!tour) return res.status(404).json({ error: 'Tour not found' });
64   tours = tours.filter((tour) => tour.id !== req.params.id);
65   res.json({ message: 'Tour deleted' });
66 };
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

Its better to keep everything related to server in one file and express another.

```

Code File Edit Selection View Go Debug Terminal Window Help
File server.js app.js tourRoutes.js tourController.js
OPEN EDITORS
server.js
app.js
tourRoutes.js
tourController.js
4-NATOURS
controllers
tourController.js
userController.js
dev-data
data
reviews.json
tour1.json
tour2.json
tour3.json
tour4.json
tour5.json
tour6.json
users.json
img
templates
node_modules
public
routes
tourRoutes.js

```

```

File Edit Selection View Go Run Terminal Help
File app.js server.js userController.js tourRoutes.js
app.js > ...
28
29 // 2) Route Handlers
30 //3) Routes
31
32 app.use('/api/v1/tours', tourRouter); //connecting
'tourRouter' to the main application.
33 app.use('/api/v1/users', userRouter); //connecting
'userRouter' to the main application.
34
35 module.exports = app; // Exporting the app so that it can
be used in other files, such as server.js

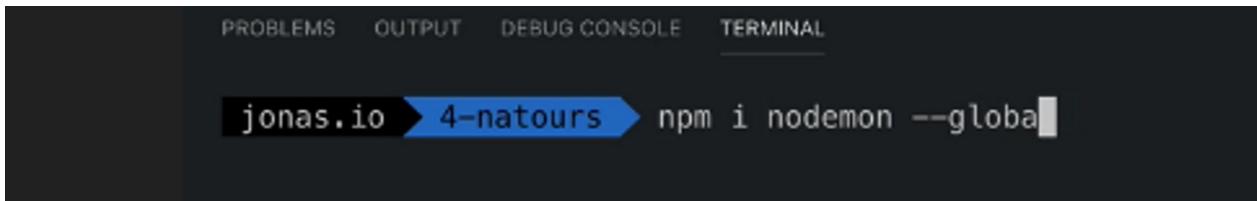
```

```

 8   "version": "1.0.0",
 9   "description": "Learning Express",
10  "main": "yes",
11  "scripts": {
12    "start": "nodemon server.js", ←
13    "test": "echo \"Error: no test specified\" && exit 1"
14  },
15  "author": "Khalid_Mahmud",
16  "license": "ISC"
17 }
18

```

npm start



In this video, we're gonna create a **special type of middleware** called **param middleware**.

[What is Param Middleware?](#)

Param middleware only runs when a **specific route parameter** is present in the URL—like an id. It's perfect when you want to run logic whenever a route uses a certain param.

For example, in a route like /tours/:id, we can create middleware that only runs when the id param exists.

How to Create It

It's super simple. Use the .param() method on your router:

```
router.param('id', (req, res, next, val) => {
  console.log(`Tour ID is: ${val}`);
  next(); // Don't forget to call next!
});
```

Key Points:

- 'id' is the name of the route parameter you want to watch.
- The middleware function has **four** parameters: req, res, next, and val.
 - val is the actual value of the id in the URL.
- **Always call next()** or the request will get stuck and never reach the final route handler.

So with this, anytime a route like /tours/123 is hit, the middleware will run, log the ID, and continue the request-response cycle smoothly. Let's go ahead and try it in code!

```

3
4 const router = express.Router();
5
6 router.param('id', (req, res, next, val) => {
7   console.log(`Tour id is: ${id}`);
8   next();
9 })
10
11 router
12   .route('/')
13   .get(tourController.getAllTours)
14   .post(tourController.createTour);
15
16 router
17   .route('/:id')
18   .get(tourController.getTour)
19   .patch(tourController.updateTour)
20   .delete(tourController.deleteTour);

```

Node.js Course

GET Get All Tours GET Get Tour

Get Tour

GET 127.0.0.1:3000/api/v1/tours/2

Params	Authorization	Headers	Body	Pre-request Script	Tests
Query Params					
KEY			Value		DESCRIPTION
Key			Value		Description
Body Cookies Headers (6) Test Results					
<pre>Pretty Raw Preview JSON</pre> 1 - { 2 - "status": "success", 3 - "data": { 4 - "tour": { 5 - "id": 2, 6 - "name": "Miskito Coast", 7 - "duration": "1 week", 8 - "participants": 2 9 - } 10 } 11 }					

Send Save Download

```

at trim_prefix (/Users/jonas.io/Desktop/4-natours/node_modules/express/lib/router/index.js:317:13)
at /Users/jonas.io/Desktop/4-natours/node_modules/express/lib/router/index.js:284:7
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
App running on port 3000...
Hello from the middleware 🙌
Tour id is: 2
{ id: '2' }
GET /api/v1/tours/2 200 7.546 ms - 886

```

Collections

Get User

GET 127.0.0.1:3000/api/v1/users/2

Params	Authorization	Headers (S)	Body	Pre-request Script	Tests
Query Params					
KEY			Value		DESCRIPTION
Key			Value		Description
Body Cookies Headers (6) Test Results					
<pre>Pretty Raw Preview JSON</pre> 1 - { 2 - "status": "error", 3 - "message": "This route is not yet defined!" 4 - }					

NO log like before. Because this middleware function is only specified in our tour router.

A screenshot of a code editor showing the `tourRoutes.js` file. The file contains the following code:

```
1 const express = require('express');
2 const tourController = require('../controllers/tourController');
3
4 const router = express.Router();
5
6 router.param('id', (req, res, next, val) => {
7   console.log(`Tour id is: ${val}`);
8   next();
9 });
10
11 router
12   .route('/')
13   .get(tourController.getAllTours)
14   .post(tourController.createTour);
15
16 router
```

Till now it is not really useful, but we can make it useful.

In all the handler function, we check if the id is valid. Or not

A screenshot of a code editor showing the `tourController.js` file. The file contains the following code with annotations:

```
20 exports.getTour = (req, res) => {
21   console.log(req.params);
22   const id = req.params.id * 1;
23
24   const tour = tours.find(el => el.id === id);
25
26   // if (id > tours.length) {
27   if (!tour) {
28     return res.status(404).json({
29       status: 'fail',
30       message: 'Invalid ID'
31     });
32   }
33
34   res.status(200).json({
35     status: 'success',
36     data: {
37       tour
38     }
39   });
40 }
```

Annotations: A red arrow points to the first line of the `getTour` function. A red bracket groups the `if (!tour)` block and its associated code.

A screenshot of a code editor showing the `tourController.js` file. The file contains the following code with annotations:

```
62 };
63
64 exports.updateTour = (req, res) => {
65   if (req.params.id * 1 > tours.length) {
66     return res.status(404).json({
67       status: 'fail',
68       message: 'Invalid ID'
69     });
70   }
71
72   res.status(200).json({
73     status: 'success',
74     data: {
75       tour: '<Updated tour here...>'
76     }
77   });
78 }
```

Annotations: A red bracket groups the `if (req.params.id * 1 > tours.length)` block and its associated code.

A screenshot of a code editor showing the `tourController.js` file. The file contains the following code with annotations:

```
76   },
77
78   exports.deleteTour = (req, res) => {
79     if (req.params.id * 1 > tours.length) {
80       return res.status(404).json({
81         status: 'fail',
82         message: 'Invalid ID'
83       });
84     }
85
86     res.status(204).json({
87       status: 'success',
88       data: null
89     });
90   };
91
92 }
```

Annotations: A red bracket groups the `if (req.params.id * 1 > tours.length)` block and its associated code.

We all know it is not good to repeat code.

Now we can apply the concept 'param middleware' and check id.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files `app.js` and `tourController.js`.
- Search Bar:** Contains the text "starter".
- Activity Bar:** On the left, there are icons for file operations like Open, Save, Find, and others.
- Code Editor:** Displays the `tourController.js` file content. The code is as follows:

```
const fs = require('fs');
const tours = JSON.parse(fs.readFileSync(`${__dirname}/../dev-data/data/tours-simple.json`));
exports.checkID = (req, res, next, val) => {
  console.log(`Tour id is: ${val}`); // this will log the id that was passed in the URL
  if (req.params.id * 1 > tours.length) {
    return res.status(404).json({
      status: 'fail',
      message: 'Invalid ID'
    });
  }
  next(); // call the next middleware function in the stack
}
```

```
tourRoutes.js M ×
routes > tourRoutes.js > ...
1 const express = require('express');
2 const { getAllTours, createTour, getTour, updateTour, deleteTour, checkID } = require('../controllers/tourController');
3 const { route } = require('express/lib/application');
4
5
6
7 //__dirname => current directory of where this file is Located
8
9
10 const router = express.Router(); //We have created router and save into this variable.
11
12 router.param('id', checkID); ↑
13
14 router
15   .route('/')
16   .get(getAllTours)
17   .post(createTour);
```

NODE.JS, EXPRESS & MONGODB

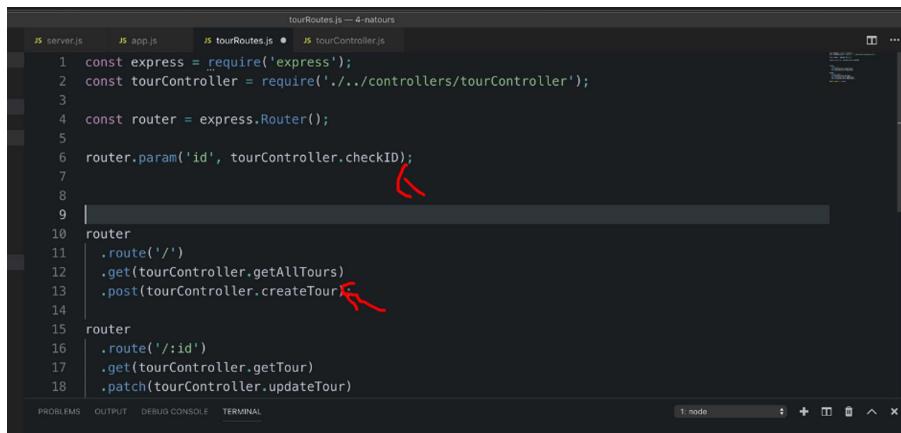
THE COMPLETE BOOTCAMP

SECTION

EXPRESS: LET'S START BUILDING THE
NATOURS API!

LECTURE

CHAINING MULTIPLE MIDDLEWARE
FUNCTIONS



```

tourRoutes.js — 4 natours
1 const express = require('express');
2 const tourController = require('../controllers/tourController');
3
4 const router = express.Router();
5
6 router.param('id', tourController.checkID);
7
8
9 router
10 .route('/')
11 .get(tourController.getAllTours)
12 .post(tourController.createTour) ↖
13
14 router
15 .route('/:id')
16 .get(tourController.getTour)
17 .patch(tourController.updateTour)
18

```

⌚ Chaining Multiple Middleware Functions in Express

So far, we've been using **only one middleware function** per route. For example, in a POST request, we might only use a createTour handler:

app.post('/api/v1/tours', createTour);

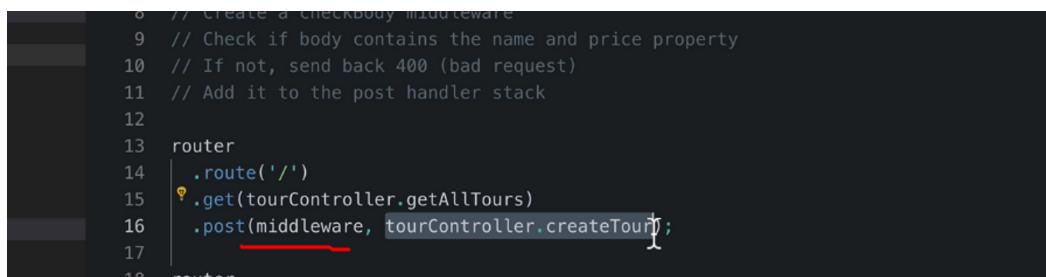
But what if we want to run **multiple middleware functions** before reaching the final route handler?

🤔 Why Chain Middleware?

Sometimes, we want to **pre-process** or **validate** data before running the main logic. For example:

- 🔍 Check if req.body contains valid data
- 🔒 Authenticate the user
- ⌚ Perform input validation

This way, our actual route handler stays clean and focused on its main job (like creating a tour).



```

6 // Create a checkBody middleware
7 // Check if body contains the name and price property
8 // If not, send back 400 (bad request)
9 // Add it to the post handler stack
10
11 router
12 .route('/')
13 .get(tourController.getAllTours)
14 .post(middleware, tourController.createTour) ↖
15
16
17 router
18

```



Chain

The screenshot shows two files in a code editor:

- app.js**: Contains a `checkID` middleware function and a `middleware` function. The `middleware` function checks if req.body.name and req.body.price are present. If not, it returns a 400 response with status: 'fail' and message: 'Missing name or price'. Otherwise, it calls the next middleware in the stack.
- tourRoutes.js**: Contains a `router` object. It has routes for `/`, `/:id`, and `/tours`. The `/tours` route uses the `middleware` function before the `createTour` handler.

Red arrows highlight the flow from the `middleware` function in **app.js** to its call in **tourRoutes.js**.

```
app.js
exports.checkID = (req, res, next, val) => {
  if (val === undefined) {
    return res.status(400).json({
      status: 'fail',
      message: 'Invalid ID'
    });
  }
  next(); // call the next middleware function in the stack
}

//middleware
exports.middleware = (req, res, next) => {
  if (!req.body.name || !req.body.price) {
    return res.status(400).json({
      status: 'fail',
      message: 'Missing name or price'
    });
  }
  next(); // call the next middleware function in the stack
}

tourRoutes.js
router.param('id', checkID);

//Create a checkBody middleware
//Check if body contains the name and price property
// If not, send a 400 response (bad request)
// Add it to the post handler stack

router
  .route('/')
  .get(getAllTours)
  .post(middleware,createTour);
```

The screenshot shows a POST request in Postman to `http://localhost:3000/api/v1/tours`. The request body is JSON with the following data:

```
{
  "name": "Khalid Mahmud Khan",
  "duration": 10,
  "difficulty": "easy"
}
```

The response status is 400 Bad Request, with the following JSON body:

```
{
  "status": "fail",
  "message": "Missing name or price"
}
```



🌐 Serving Static Files in Express

Let's now learn how to **serve static files** using Express.

But first — what *are* static files?

Static files are files like **HTML, CSS, JavaScript, images**, etc., that live in our file system and don't change dynamically. Right now, we have some of these, like `overview.html`, in our `public` folder.

However, at this point, we can't access them from the browser—there's no route to reach them yet.

For example:

- `public/overview.html` ✅ exists
- But <http://localhost:3000/overview.html> ❌ doesn't work yet

Let me show you how to fix that using Express. It's super simple. We'll make these files publicly available with just one line of code.

Stay tuned — this is a core part of building full-stack apps! 🚀



```

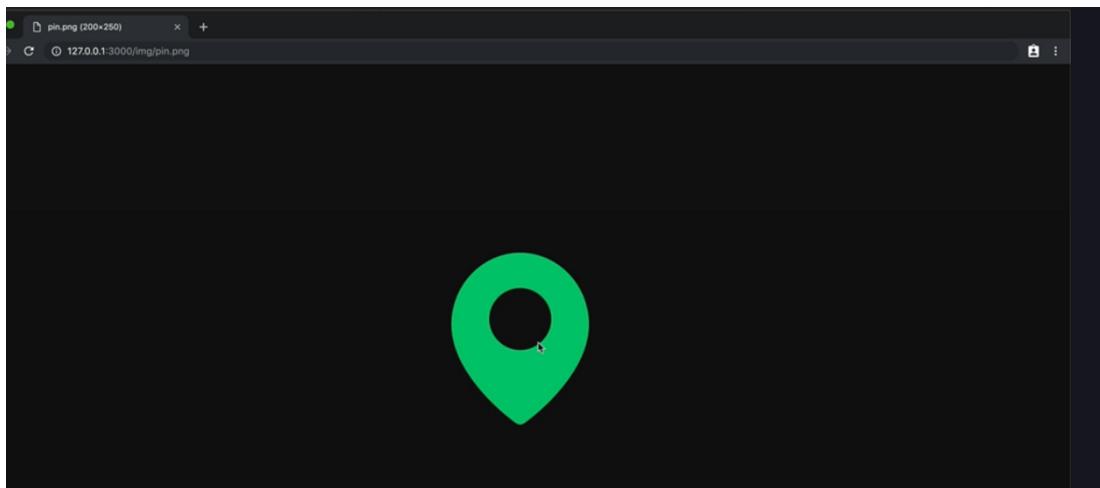
app.js -- 4-natours
1  // Importing required modules
2  const express = require('express');
3  const userRouter = require('./routes/userRoutes');
4
5  const app = express();
6
7  // (1) MIDDLEWARES
8  app.use(express.json());
9  app.use(express.urlencoded({ extended: true }));
10 app.use(morgan('dev'));
11
12 app.use(express.static(`${__dirname}/public`));
13
14 app.use((req, res, next) => {
15   console.log('Hello from the middleware');
16   next();
17 });
18
19 app.use((req, res, next) => {
20   req.requestTime = new Date().toISOString();
21   next();
22 });

```

The tooltip for the `static` keyword in the code is displayed, providing the following description:

Create a new middleware function to serve files from within a given root directory. The file to serve will be determined by combining `req.url` with the provided root directory. When a file is not found, instead of sending a 404 response, this module will instead call `next()` to move on to the next middleware.

A screenshot of a web browser displaying the Natours website. The page shows three tour cards arranged horizontally. The first card is titled 'THE FOREST HIKER' and describes an 'EASY 5-DAY TOUR' in Banff, Canada from April 2021, costing \$297 per person with a 4.9 rating. The second card is titled 'THE SEA EXPLORER' and describes a 'MEDIUM-DIFFICULT 7-DAY TOUR' in Oregon, US from June 2021, costing \$497 per person with a 4.8 rating. The third card is titled 'THE SNOW ADVENTURER' and describes a 'DIFFICULT 3-DAY TOUR' in Aspen, USA from January 2022, costing \$697 per person with a 4.9 rating. Each card includes a 'DETAILS' button.



A screenshot of a presentation slide. On the left, there's a logo for 'JONAS.IO SCHMEDTMANN' and text for 'NODE.JS, EXPRESS & MONGODB THE COMPLETE BOOTCAMP'. On the right, there's a large green polygonal graphic. Overlaid on the graphic are the words 'SECTION', 'EXPRESS: LET'S START BUILDING THE NATOURS API', 'LECTURE', and 'ENVIRONMENT VARIABLES'.

Environment Variables in Node.js / Express

Welcome back! 🙌

In this video, we're diving into **environment variables** — what they are, why we care, and how we use them in our apps.

What Are Environment Variables?

Environment variables are **key-value pairs** that store config values outside your actual code.

Think of them like:

- PORT=3000
- DB_PASSWORD=supersecret123
- NODE_ENV=production

You set them once and access them anywhere in your app.

💡 Why Use Them?

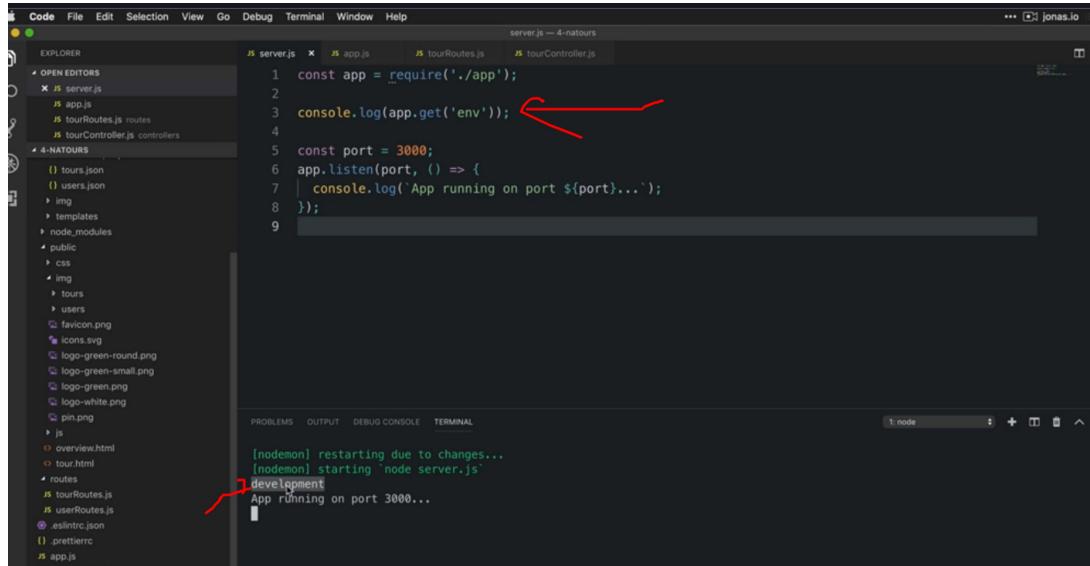
Apps can run in **different environments** like:

- 🖌️ development (what you use while coding)
- 🚀 production (what real users see)
- 🏁 staging, test, etc. (used in big teams or complex apps)

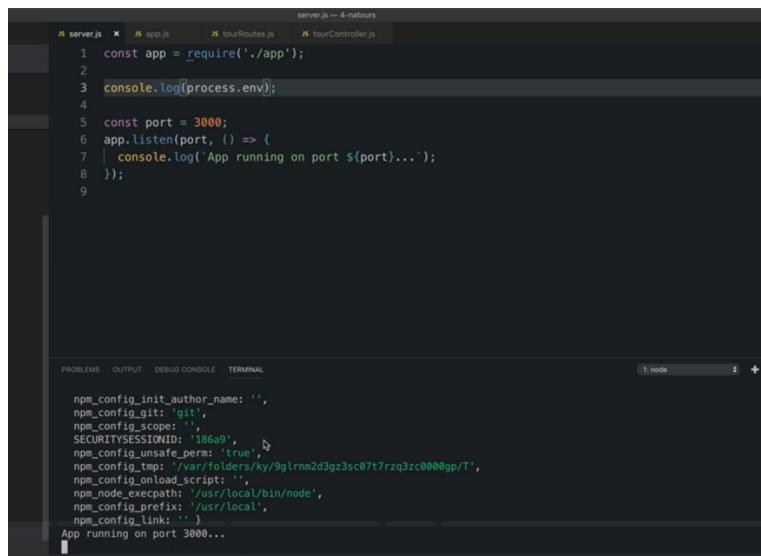
Depending on the environment, you might want:

- Different **databases**
- Debugging turned **on/off**
- Logging behavior to **change**
- APIs to point to **different URLs**

Using environment variables helps you **switch between those easily** — without rewriting your code. Neat, right?



A screenshot of the Visual Studio Code interface. The left sidebar shows a file tree with files like server.js, app.js, tourRoutes.js, tourController.js, and various JSON and image files. The main editor window shows a portion of server.js with the line `console.log(app.get('env'))` highlighted by a red arrow. The terminal at the bottom shows the output of running the application: `[nodemon] restarting due to changes... [nodemon] starting 'node server.js' [development] App running on port 3000...



A screenshot of the Visual Studio Code interface, similar to the one above. The file tree and terminal output are identical. The main editor window shows a portion of server.js with the line `console.log(process.env)` highlighted by a red arrow. The terminal output shows the environment variable values: `npm_config_init_author_name: '', npm_config_git: 'git', npm_config_scope: '', SECURITYSESSIONID: '186a9', npm_config_unsafe_perm: 'true', npm_config_tmp: '/var/folders/ky9glrm2d3gz3sc07t7rzq3zc0000gp/T', npm_config_onload_script: '', npm_node_execpath: '/usr/local/bin/node', npm_config_prefix: '/usr/local', npm_config_link: ''` followed by the message `App running on port 3000...

A screenshot of the Visual Studio Code interface. The left sidebar shows a tree view of files and folders for a project named '4-natours'. The main editor area displays the contents of the 'package.json' file:

```
1 {
2   "name": "natours",
3   "version": "1.0.0",
4   "description": "Learning node, express and mongoDB",
5   "main": "app.js",
6   "scripts": {
7     "start": "nodemon server.js"
8   },
9   "author": "Jonas Schmedtmann",
10  "license": "ISC",
11  "dependencies": {
12    "express": "^4.16.4",
13    "morgan": "^1.9.1"
14  }
15}
16
```

The terminal tab at the bottom shows a command being run: `* jonas.io > 4-natours > NODE_ENV=development nodemon server.js`. A red arrow points from the text 'development' in the command to the 'NODE_ENV' field in the package.json file.

A screenshot of a terminal window showing the following output:

```
PWD: '/Users/jonas.io/Desktop/4-natours',
SHLVL: '1',
ZSH: '/Users/jonas.io/.oh-my-zsh',
TERM_PROGRAM: 'vscode',
TERM_PROGRAM_VERSION: '1.33.0',
LANG: 'en_GB.UTF-8',
TERM: 'xterm-256color',
LC_CTYPE: 'en_GB.UTF-8',
NODE_ENV: 'development', _____
_: '/usr/local/bin/nodemon' }
App running on port 3000...
```

A screenshot of the VS Code terminal showing the same environment variables and application status as the previous terminal window. The output is identical to the one above, with a red arrow pointing to the 'development' value in the `NODE_ENV` variable.

```

Code File Edit Selection View Go Debug Terminal Window Help
config.env — 4-natours
OPEN EDITORS
server.js config.env package.json app.js tourRoutes.js tourController.js
1 NODE_ENV=development
2 PORT=8000
3 USER=jonas
4 PASSWORD=123456
4-NATOURS
controllers tourController.js userController.js dev-data node_modules public routes tourRoutes.js userRoutes.js eslint.json .prettierrc app.js config.env package-lock.json package.json server.js
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
jonas.io 4-natours NODE_ENV=development X=2 nodemon server.js

```

Now how do we actually connect this (.env) file with our node application?

```

Code File Edit Selection View Go Debug Terminal Window Help
config.env — 4-natours
OPEN EDITORS
server.js config.env package.json app.js tourRoutes.js tourController.js
1 NODE_ENV=development
2 PORT=8000
3 USER=jonas
4 PASSWORD=123456
4-NATOURS
controllers tourController.js userController.js dev-data node_modules public routes tourRoutes.js userRoutes.js eslint.json .prettierrc app.js config.env package-lock.json package.json server.js
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
jonas.io 4-natours npm i dotenv : difftree: all install generateActionsToTake

```

```

Code File Edit Selection View Go Debug Terminal Window Help
server.js — 4-natours
OPEN EDITORS
server.js config.env package.json app.js tourRoutes.js tourController.js
1 const dotenv = require('dotenv');
2 const app = require('./app');
3
4 dotenv.config({ path: './config.env' })
5
6 console.log(process.env);
7
8 const port = 3000;
9 app.listen(port, () => {
10   console.log('App running on port', port);
11 });
12

```

config(options?: DotenvConfigOptions): DotenvConfigOutput

- controls behavior

Loads .env file contents into {@link https://nodejs.org/api/process.html#process_process | process.env}. Example: 'KEY=value' becomes { parsed: { KEY: 'value' } }

@returns — an object with a parsed key if

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
jonas.io 4-natours npm start

```

```
10 |     console.log(`App running on port ${port}`);
11 |   });
12 |
13 |   SECURITYSESSIONID: '10680',
14 |   npn_config_unsafe_perm: 'true',
15 |   npn_config_tmp: '/var/folders/kv/9girm2d3gz3sc07t7rzq3c000gp/T',
16 |   npn_config_node_execpath: '/usr/local/bin/node',
17 |   npn_config_prefix: '/usr/local',
18 |   npn_config_link: '',
19 |   NODE_ENV: 'development',
20 |   PORT: '3000',
21 |   PASSWORD: '123456' }
22 | App running on port 3000...
```

```
8 // 1) MIDDLEWARES
9 if (process.env.NODE_ENV === 'development') {
10   app.use(morgan('dev')); // Morgan is a HTTP request logger middleware
11   // process of logging requests to your application, making it easier to analyze
12   // performance.
13 }

You, 4 hours ago • Using 3rd-Party Middleware
```

```
File Edit Selection View Go Run Terminal Help ← → 🔍 starter
server.js M x app.js M config.env U
server.js > [o] port
You, 22 seconds ago | 1 author (You)
1 const dotenv = require("dotenv");
2 const app = require("./app");
3
4 // console.log(app.get('env'));
5 dotenv.config({ path: "./config.env" }); //read environment variables from config.env file
6
7
8 console.log(process.env);
9
10 //4) START SERVER
11 const port = process.env.PORT || 3000; You, 21 seconds ago • Uncommitted changes
12
13 app.listen(port, () => {
14   console.log(`App running on port ${port}...`);
15 });
16
```

```
File The Complete Bootcamp
Edit Selection View Go Debug Terminal Window Help
package.json -- 4-natours
server.js package.json x app.js tourRoutes.js tourController.js
1 {
2   "name": "natours",
3   "version": "1.0.0",
4   "description": "Learning node, express and mongoDB",
5   "main": "app.js",
6   "scripts": {
7     "start:dev": "nodemon server.js",
8     "start:prod": "NODE_ENV=production nodemon server.js"
9   },
10  "author": "Jonas Schmedtmann",
11  "license": "ISC",
12  "dependencies": {
13    "dotenv": "^7.0.0",
14    "express": "^4.16.4",
15    "morgan": "^1.9.1"
16  }
17 }
18
```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
x jonas.io ➜ 4-natours npm run start:prod ↵
> natours@1.0.0 start:prod /Users/jonas.io/Desktop/4-natours
> NODE_ENV=production nodemon server.js

[nodemon] 1.18.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node server.js` ↵
production
App running on port 3000...

```

⌚ What's the Point of This Lecture?

It's all about **environment variables** — those little secret ingredients that help you change app behavior based on whether you're in development, production, or some other setup. They're *not* specific to Express, but they're super important in real-world Node.js apps.

🧠 What's an Environment Variable?

Think of env vars like secret notes your app reads at runtime to decide how to behave. Examples:

- Which database to connect to
- Whether to show debug info
- What port to listen on

🏗 Development vs. Production

- **Development:** You're coding, testing stuff, seeing logs, maybe using a test DB.
- **Production:** Your app is live. You want it clean, fast, and secure.

So yeah, you may use different values (DBs, ports, APIs) based on which environment you're in.

🔗 Where Does Express Fit In?

By default, Express assumes you're in **development** mode. You can check this with:

```
js
CopyEdit
console.log(app.get('env')); // prints 'development'
But the real juicy part is how to control this value yourself — especially for prod.
```

✍ How to Set Env Vars (2 Ways)

◆ 1. Inline in Terminal (temp)

You can set them temporarily like this:

```
bash
CopyEdit
NODE_ENV=development nodemon server.js
Or:
```

```
bash
CopyEdit
NODE_ENV=production PORT=4000 nodemon server.js
These vars live only while the app runs and only in that terminal session.
```

◆ 2. Using a .env File (recommended)

Create a .env file like this:

```
env
CopyEdit
NODE_ENV=development
PORT=3000
USERNAME=jonas
PASSWORD=123456
Install the package to read this:
```

```
bash
CopyEdit
npm install dotenv
Then add this line at the top of your server.js (important: before any other code uses env vars):
```

```
js
CopyEdit
require('dotenv').config({ path: './config.env' });
Now process.env.NODE_ENV, process.env.PORT, etc., will work in any file in your app.
```

💡 Why This Matters

A lot of packages (like Morgan for logging) will only activate when NODE_ENV is 'development'. You can do:

```
js
CopyEdit
if(process.env.NODE_ENV === 'development') {
  app.use(morgan('dev'));
}
```

And it won't run in production — that's 💥 for performance and security.

✖ Common Mistake Highlighted

He had the .env loading **after** the app was required — that's too late! 🤦

Make sure you do this in server.js:

```
js
CopyEdit
require('dotenv').config({ path: './config.env' });
const app = require('./app'); // Only after env vars are ready!
```

🚀 Final Touch: Dynamic Port

This lets you switch the port easily:

```
js
CopyEdit
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

✓ Summary Time:

💡 Concept

🧠 Meaning

process.env Built-in Node object holding all env vars

.env file File to store vars like PORT, NODE_ENV, etc.

dotenv package Loads your .env into process.env

Use case Configs that change per environment (DB, logging, etc.)

NODE_ENV Convention for telling the app if it's in 'development' or 'production'



Before moving to the next section, I want to show you how to set up **ESLint** with **Prettier** in **VS Code** to improve code quality.

If you're not using VS Code or already have a linter in your workflow, feel free to skip this and mark it as complete.

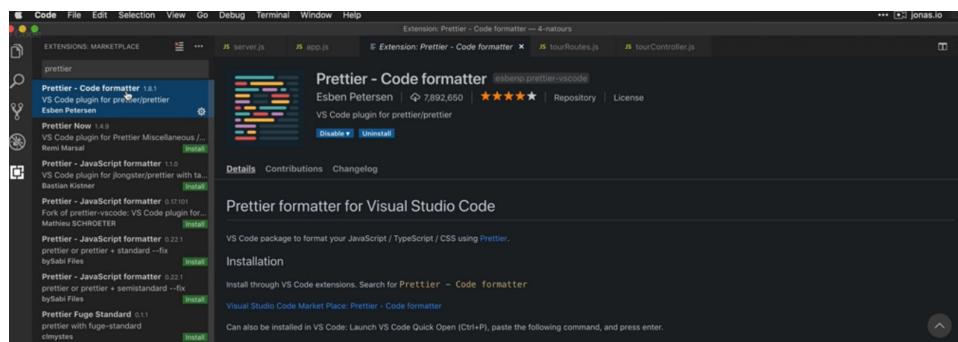
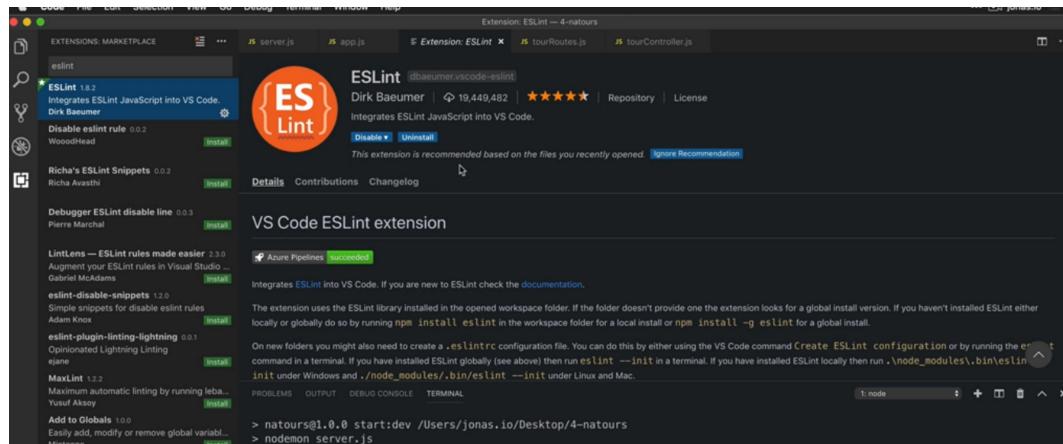
But if you're new to **ESLint** and want cleaner, more consistent code, let's dive in!

ESLint is a tool that scans your code for errors and bad practices. It's super customizable, so you can tailor it to

fit your coding style.

Although ESLint can handle code formatting, we'll continue using **Prettier**, which we set up earlier, as our main formatter. We'll configure ESLint to highlight issues based on the rules we define, while Prettier takes care of formatting.

It might sound a bit confusing now, but don't worry — let's set it up step by step.



```
npm i eslint@5.16.0 prettier@1.18.2 eslint-config-prettier@4.1.0 eslint-plugin-prettier@3.1.0 eslint-config-airbnb@17.1.1 eslint-plugin-node@8.0.1 eslint-plugin-import@2.18.0 eslint-plugin-jsx-a11y@6.2.3 eslint-plugin-react@7.14.2 --save-dev
```

From <<https://www.udemy.com/course/nodejs-express-mongodb-bootcamp/learn/lecture/15064896#questions/7502170>>