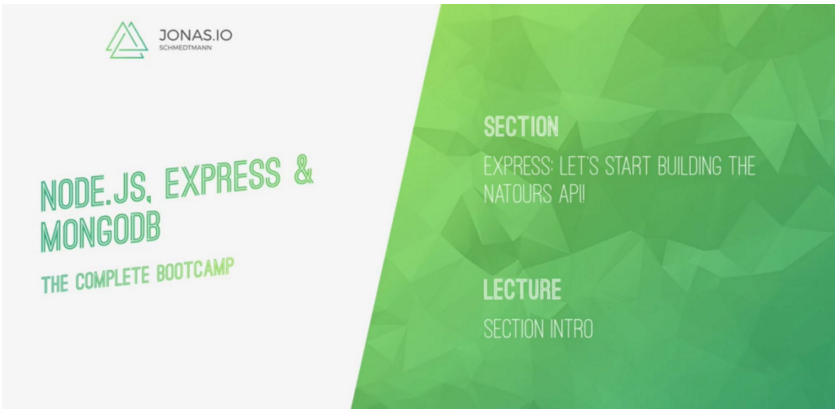
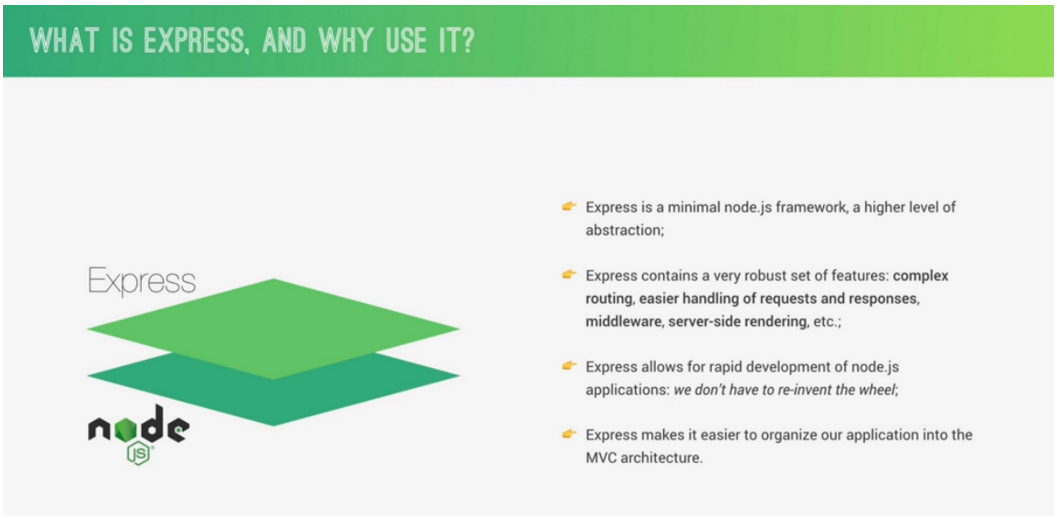
Section 6: Express: Let's Start Building the Natours API!

23 May 2025 21:21





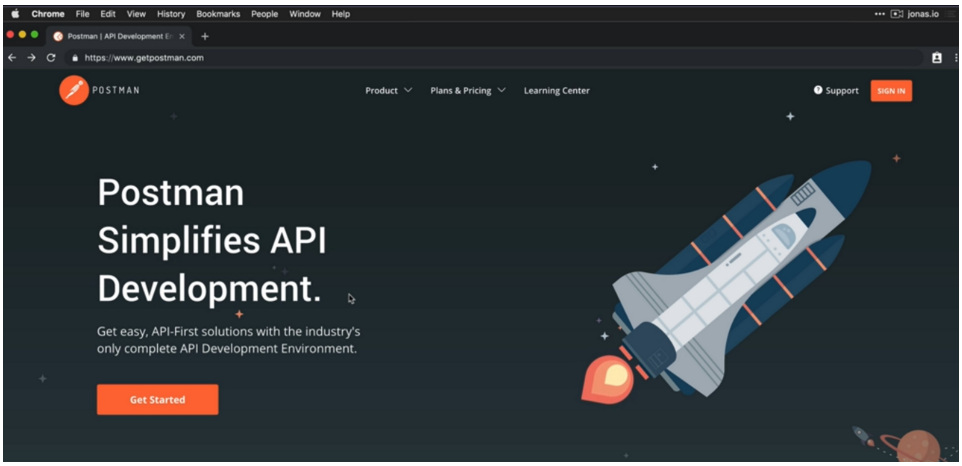
From now on, we'll be using **Express** in our application.
So, what is Express, and why use it?
Express is a minimal and flexible Node.js framework. It's built on top of Node.js, offering a higher level of abstraction while still being 100% Node.js under the hood. It's also the most popular Node.js framework out there—kind of the standard.

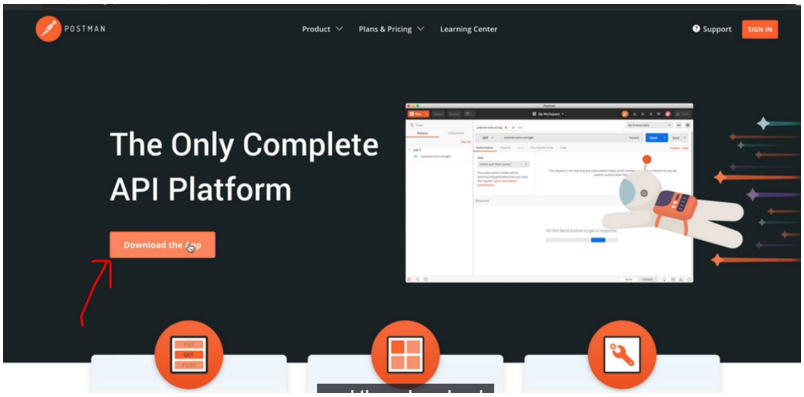
Express provides a powerful set of features out of the box:

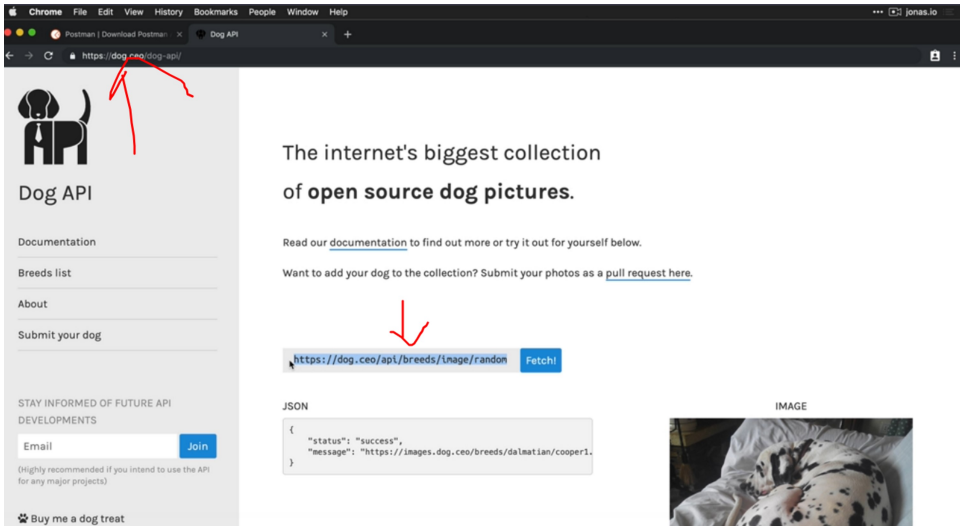
- Complex routing
- Simplified request/response handling
- Middleware support
- Server-side rendering
  ...and more.

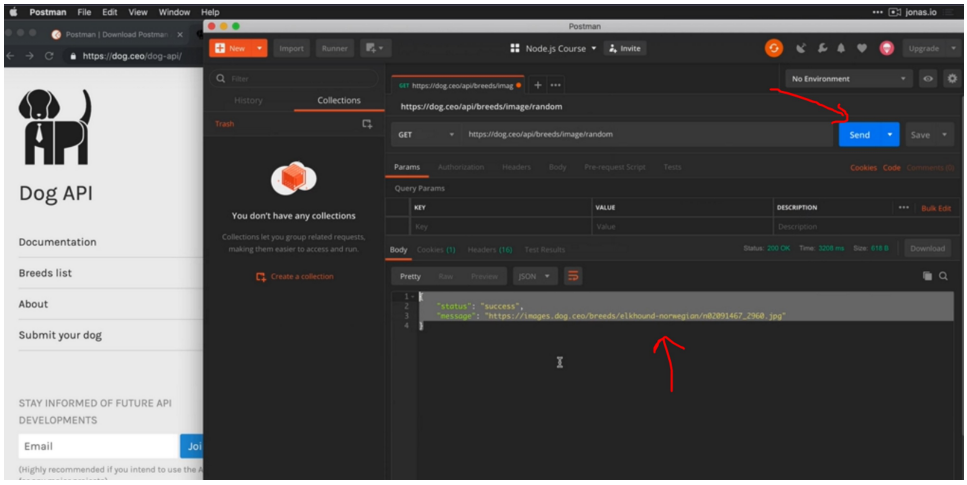
This makes building Node.js applications way faster—you won't have to rewrite the same logic or reinvent the wheel for things like routing or templating.

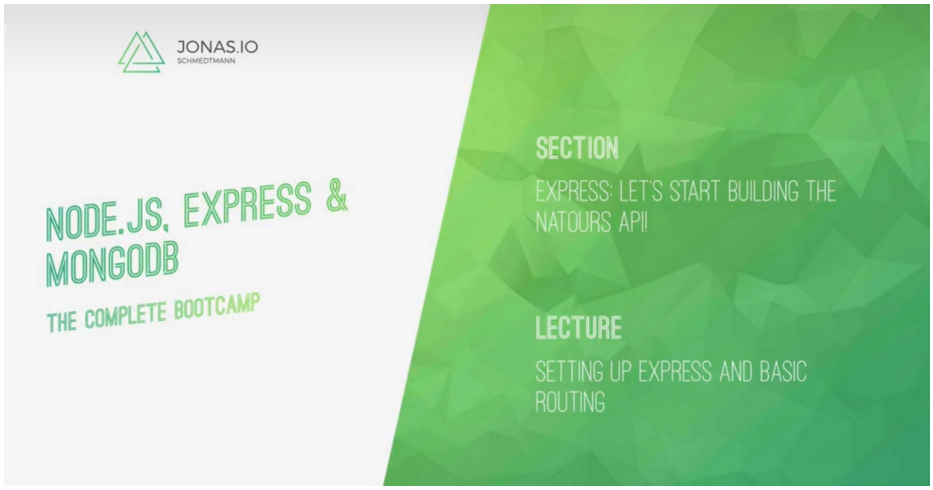
Remember how basic routing felt clunky earlier in the course? With Express, it's much simpler—like 10x easier.
It also helps organize your app using the **MVC architecture**, a common and efficient pattern we'll explore throughout the course.
In short: Express makes working with Node.js easier, faster, and way more enjoyable.



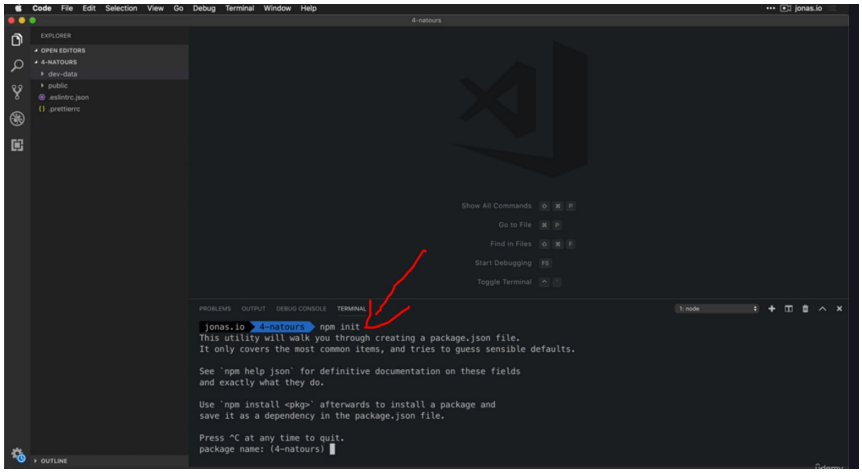


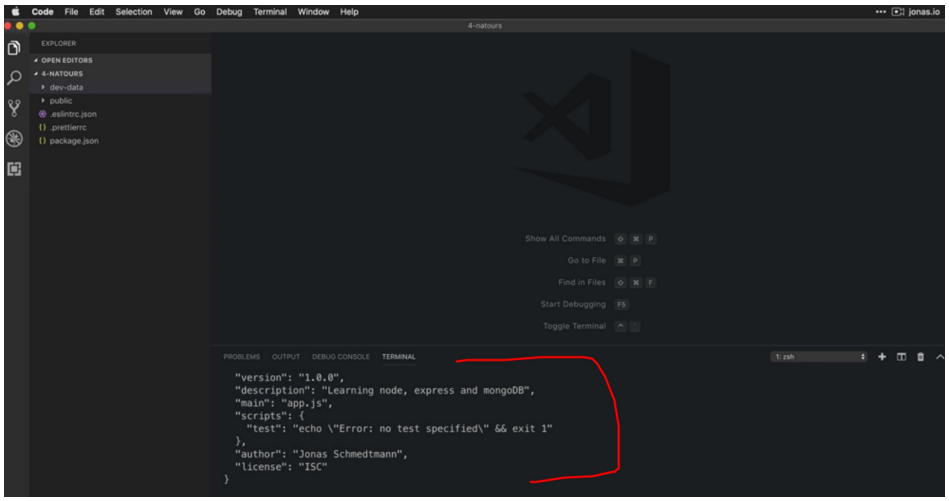


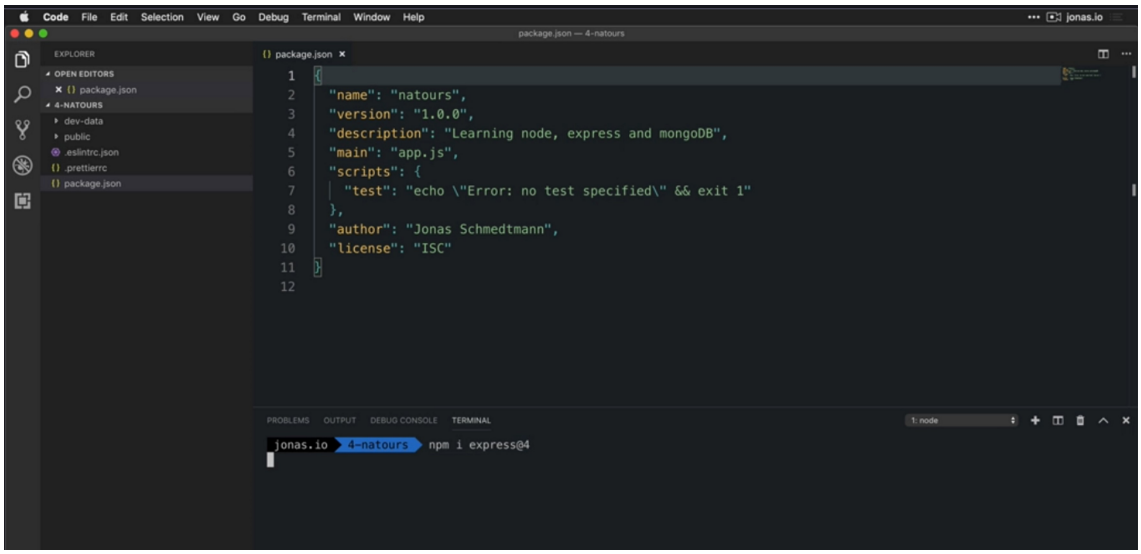




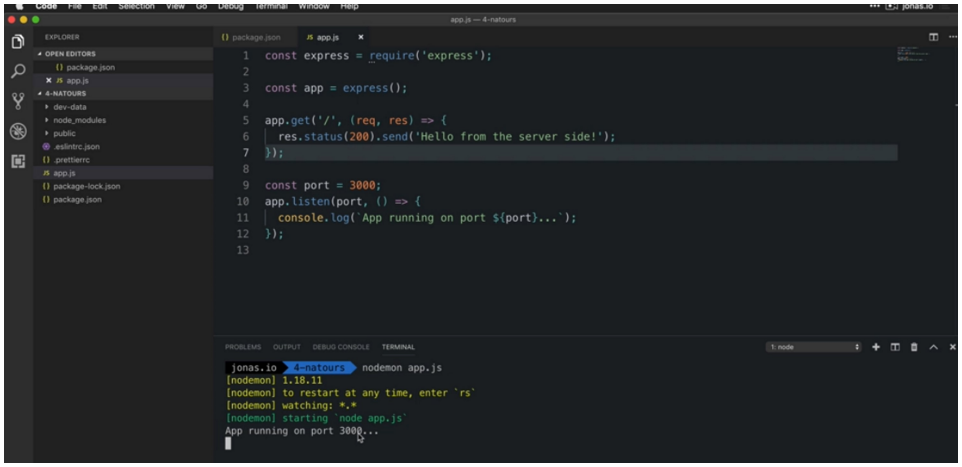
Let's now finally setup Express, create a simple server and do some basic routing just to get an initial feeling of how we actually work in Express.

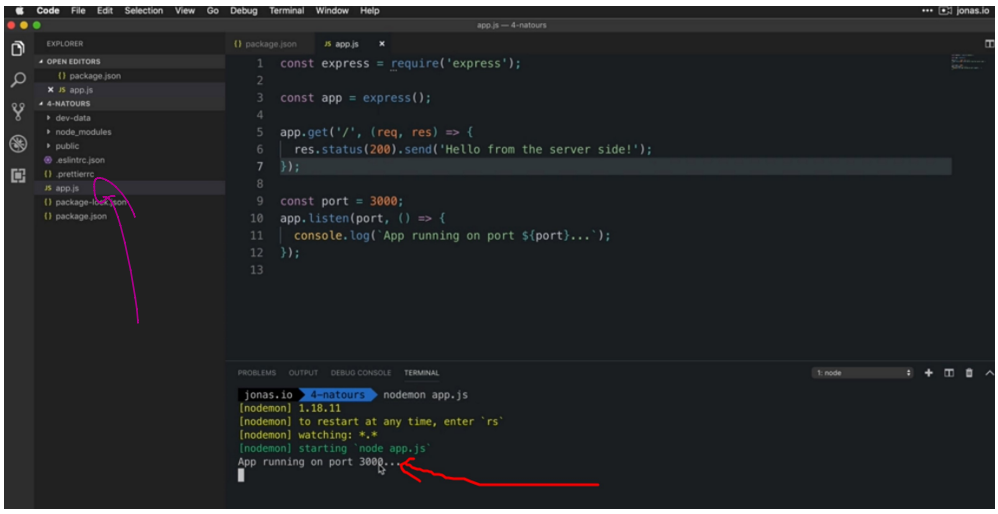


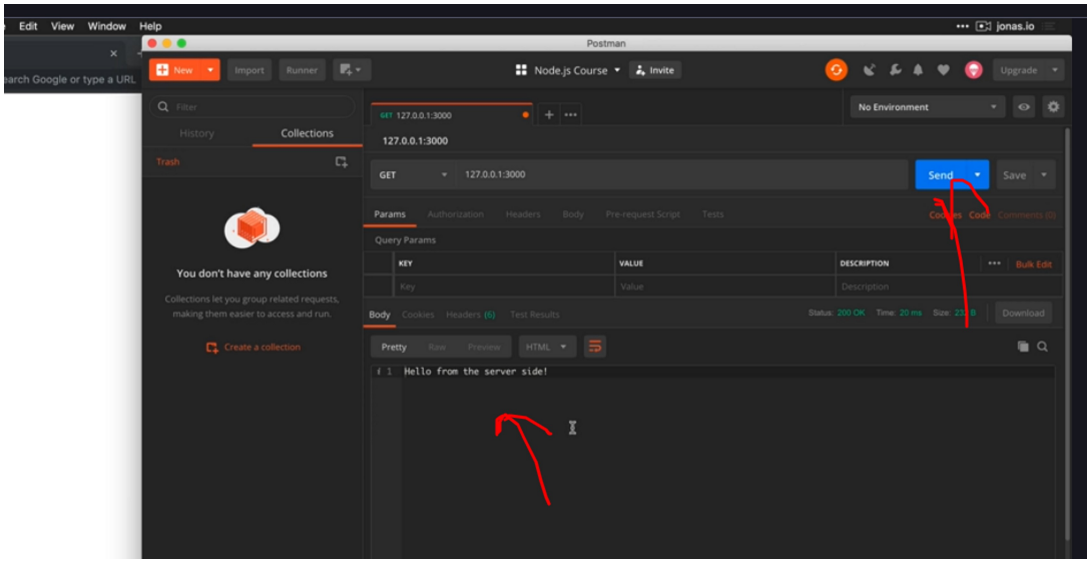




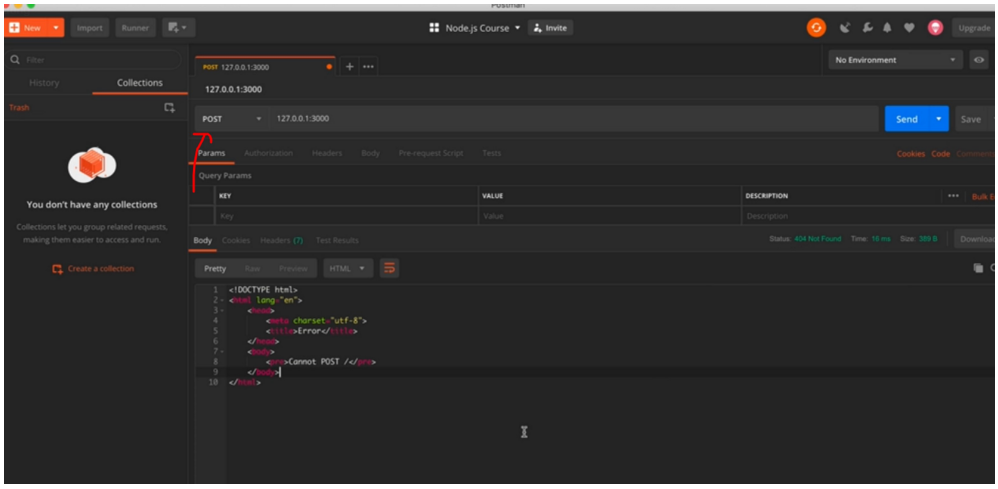
It's convention to have all kinds of configuration in the 'app.js' file.

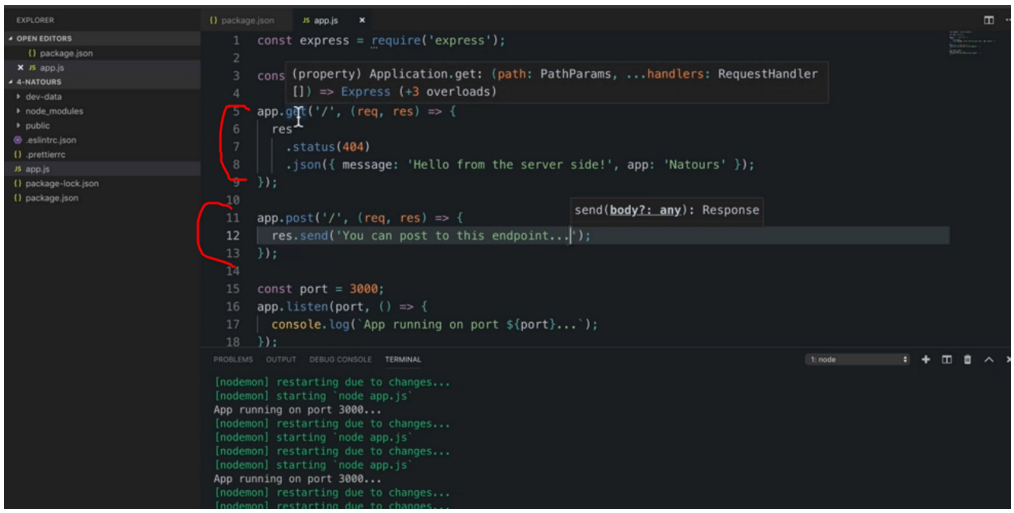




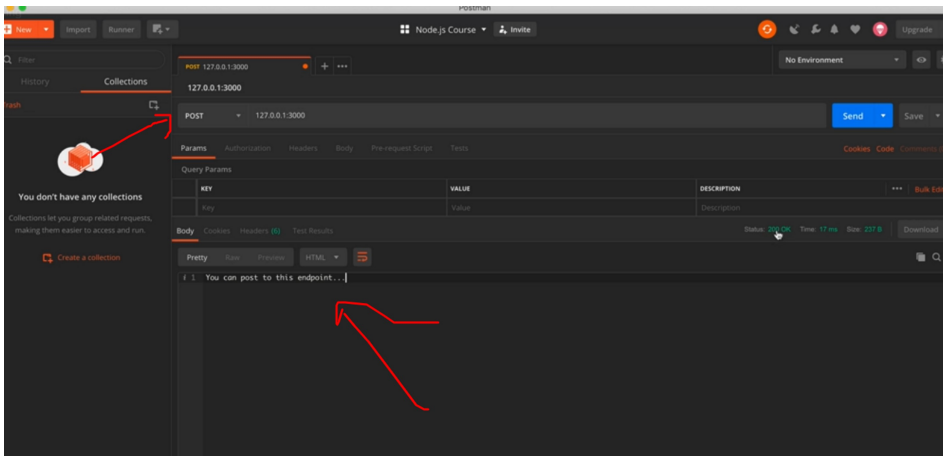


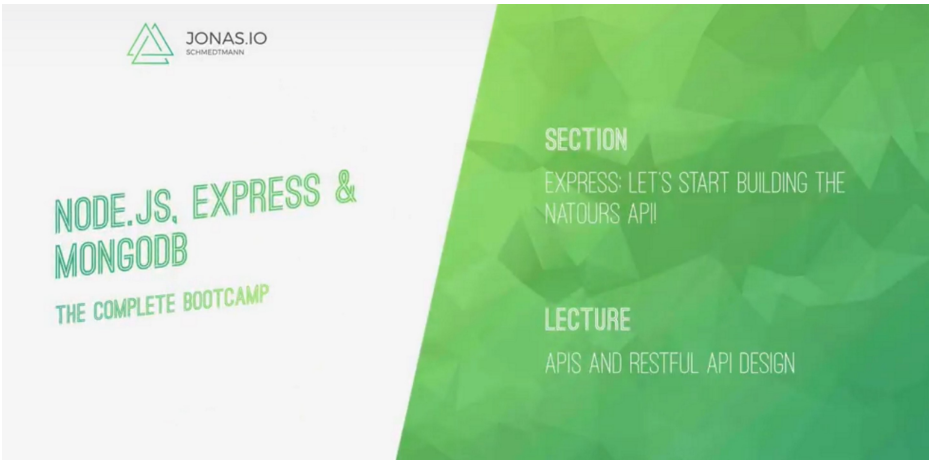
We have used get method. So won't work. Express automatically sends this response if you try to send.

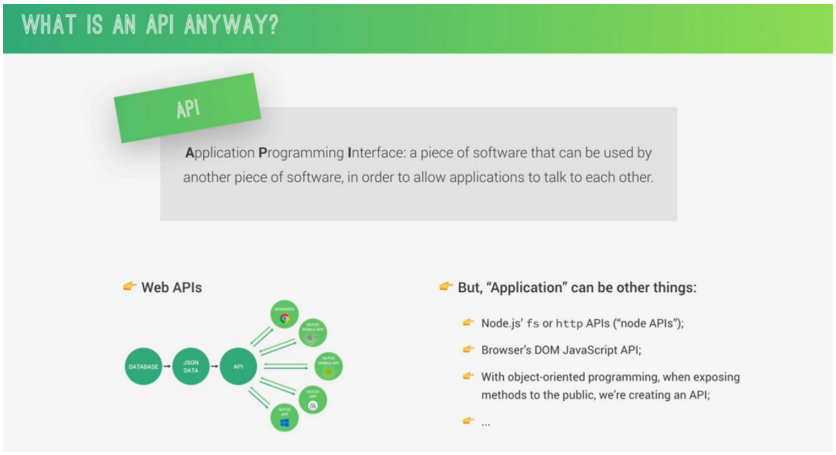




Now you can post







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.

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:

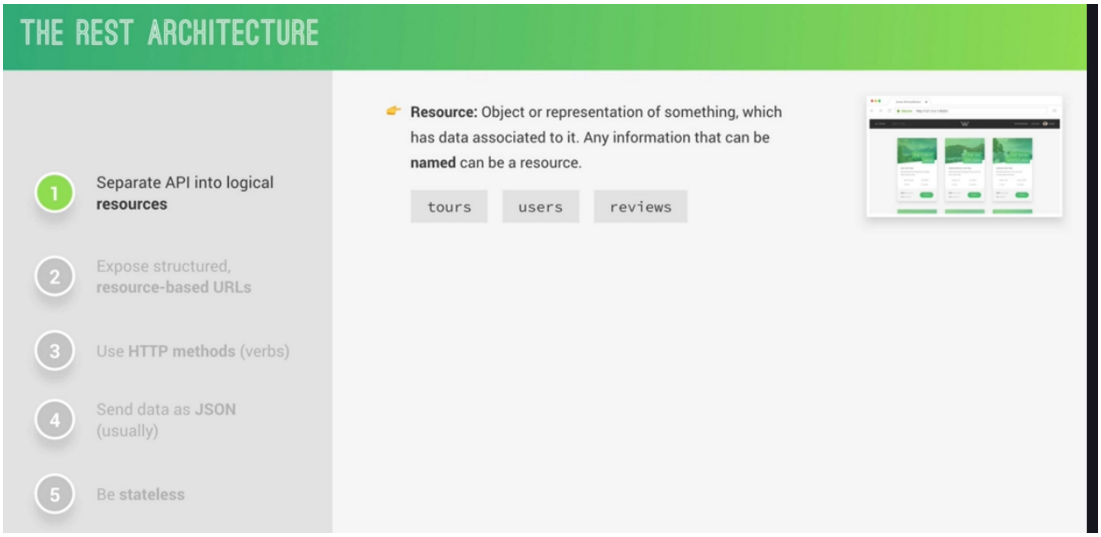
- GET – Read data
- POST – Create data
- PUT/PATCH – Update data
- DELETE – Delete data

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.



🔷 **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?

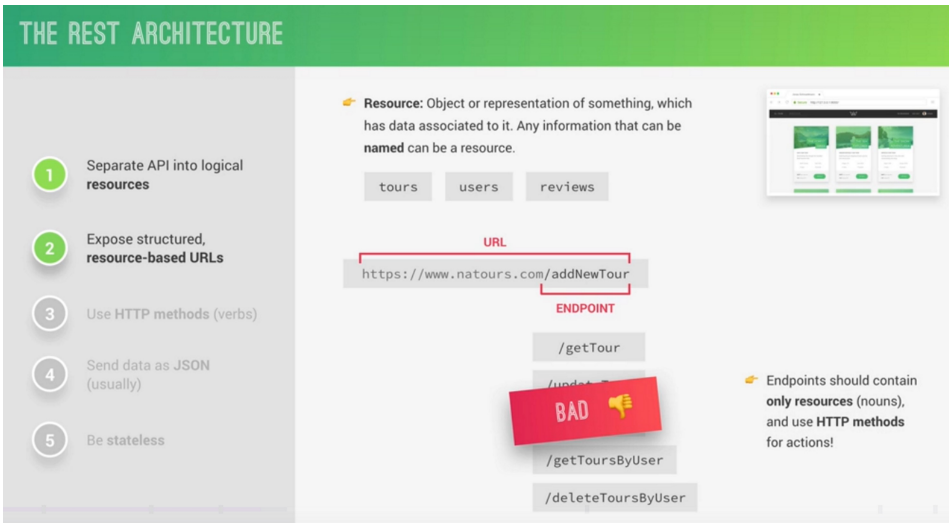
In the context of REST, a **resource** is simply an object or representation of something that holds data. Think of examples like:

- tours
- users
- reviews

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

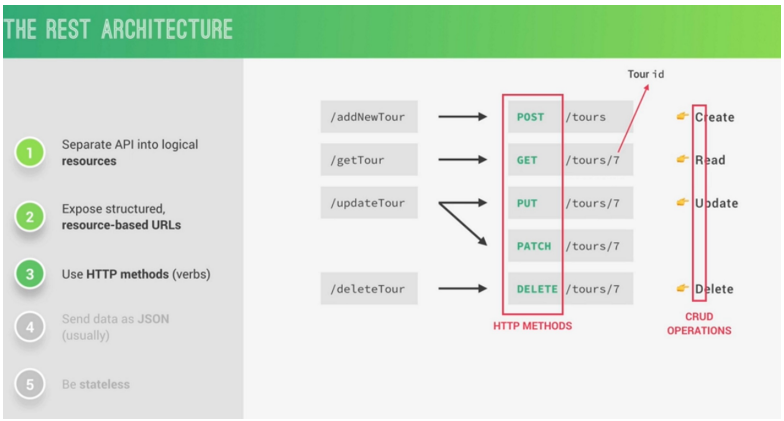
👉 The key rule? It should be a **noun**, not a **verb**.

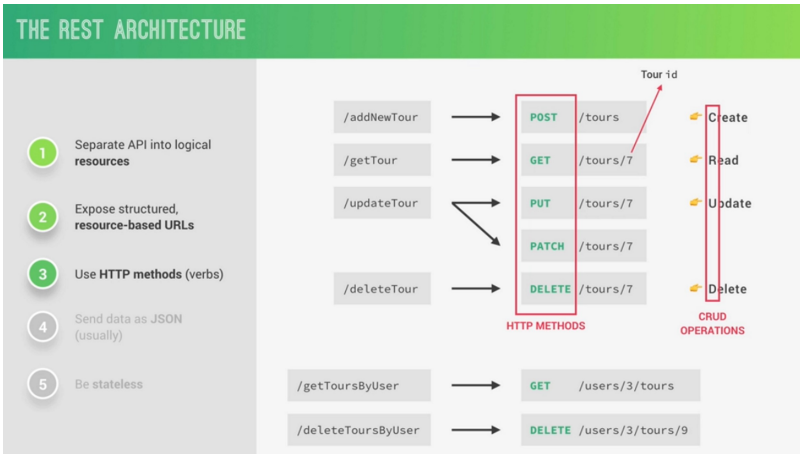
So rather than thinking in terms of actions (like getTours), we think in terms of **entities** (tours) and then perform actions using the appropriate **HTTP method**.



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.





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:

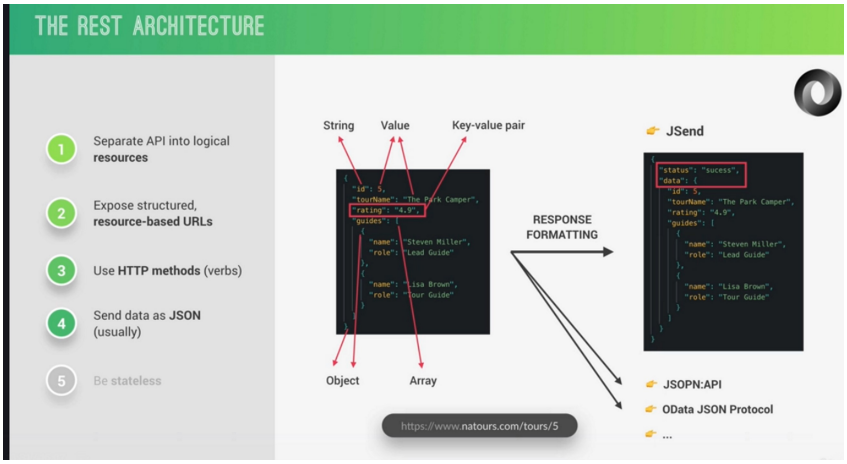
- **Easy for humans to read and write**
- **Easy for machines to parse and generate**

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.



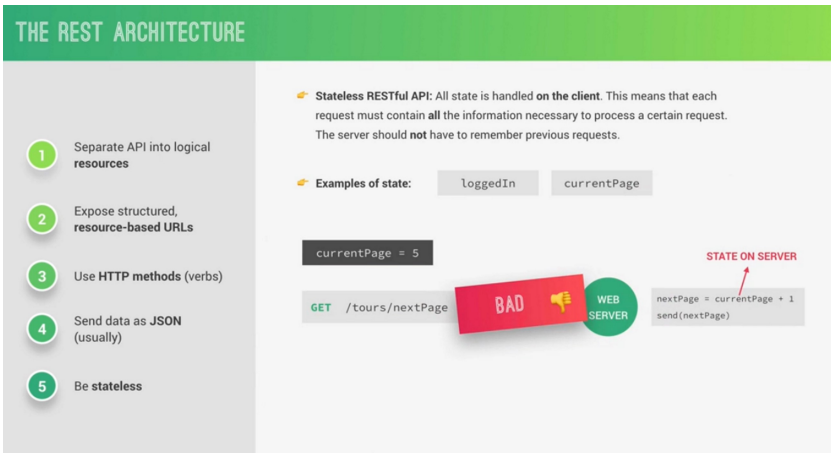
Before sending data back to the client, we usually apply **simple response formatting**. One common and easy-to-use standard is called **Jsend**.
With **Jsend**, we wrap our response in a new object that includes:

- 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.

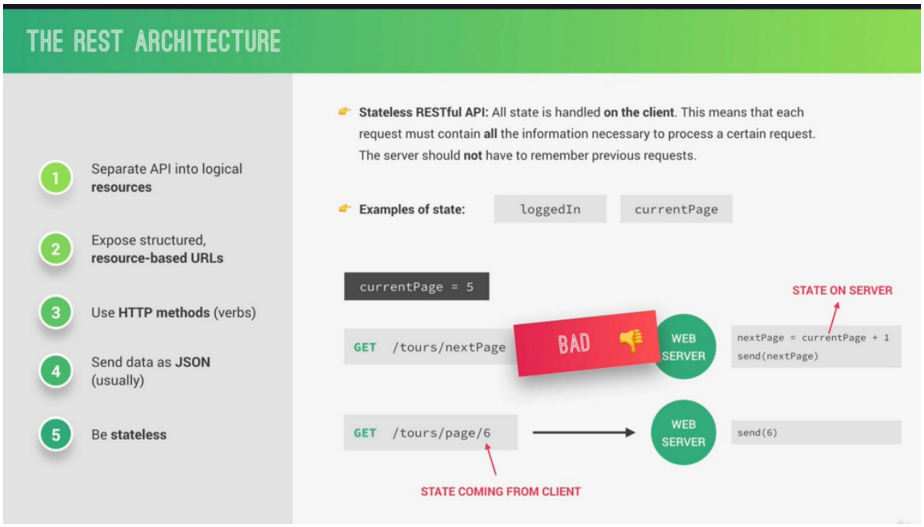


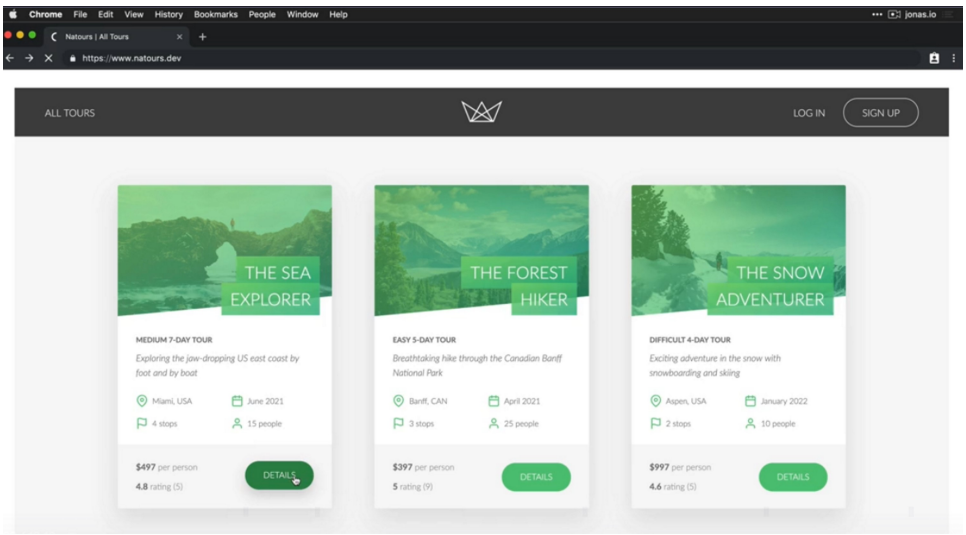
A **RESTful API should always be stateless**—but what does that actually mean?
In a **stateless API**, all state (like user login status or current pagination page) is handled **on the client**, *not* on the server.

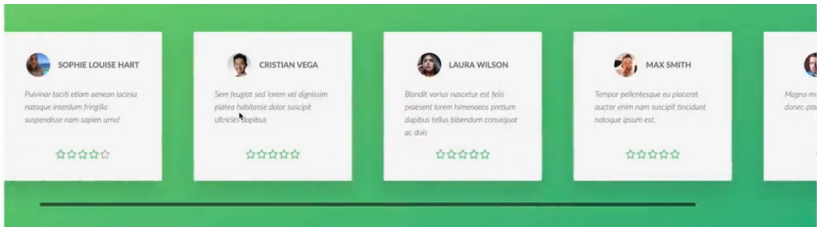
That means **every request must contain all the necessary information** for the server to process it. The server should never rely on previous requests to understand or respond to the current one.
👉 For example, imagine you're on page 5 of a tour list and want to go to page 6.
If your endpoint is /tours/nextPage, the server would need to remember you were on page 5.

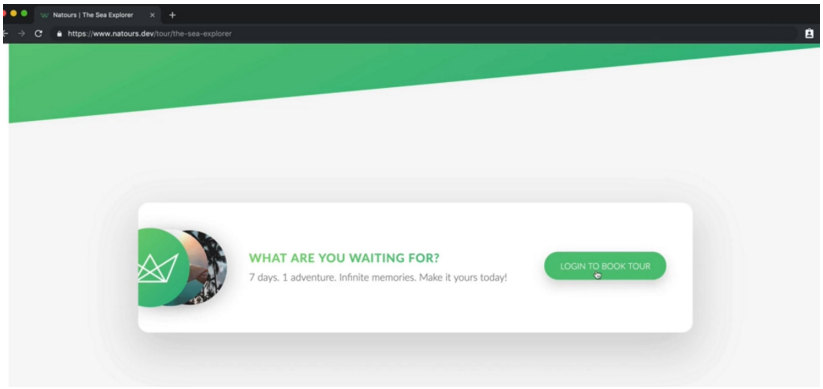
That's *stateful*—and **not allowed in REST**.
Instead, the client should directly request /tours?page=6, making it clear and independent.

No memory needed on the server = stateless = RESTful.









🗺️ **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:

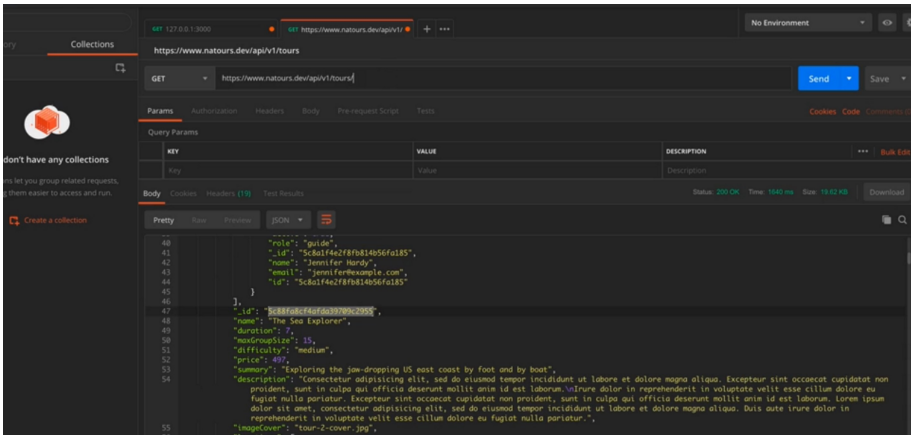
- Build out the **RESTful API** with Express.js
- Connect it to **MongoDB** using Mongoose
- Handle resources like:
  - /api/v1/tours
  - /api/v1/users
  - /api/v1/reviews

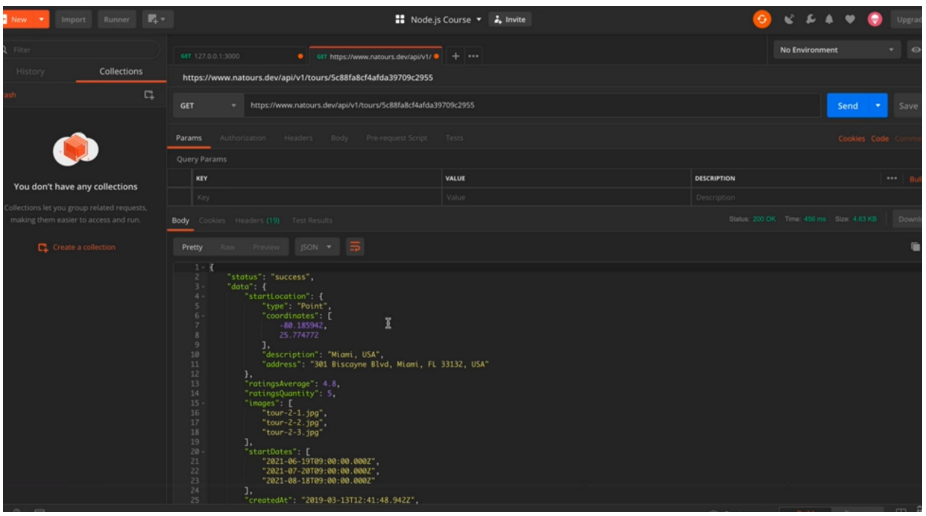
Basically: we're building the **engine** before putting the body on the car 🚗

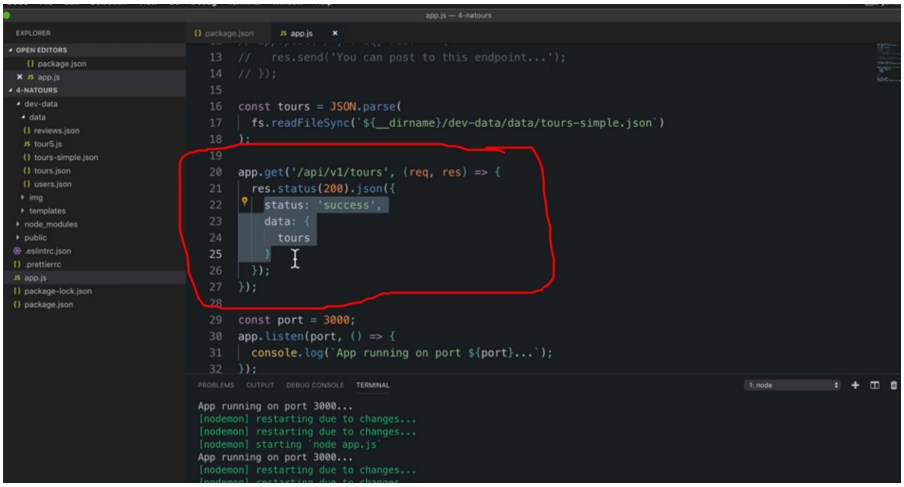
💡 **Why API First?**
"Why not build the UI first? It looks cooler!"
Totally fair, but here's the vibe:

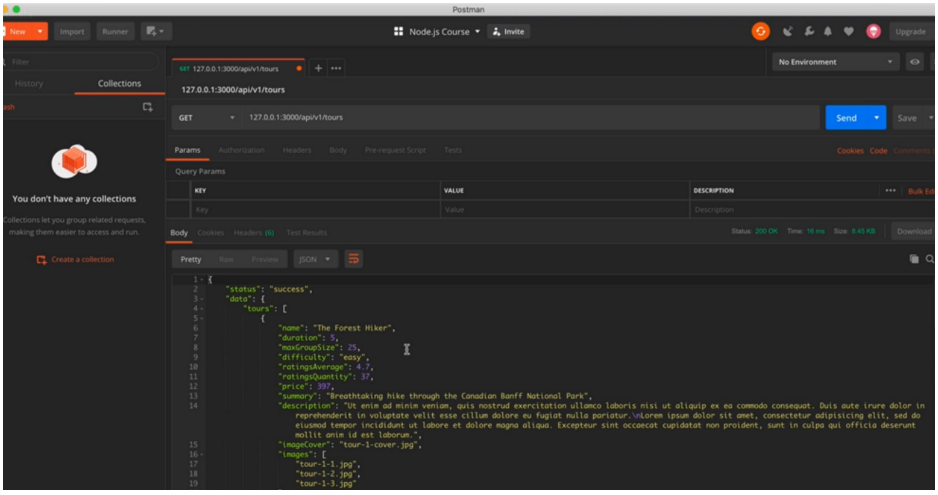
- Learning **backend** (Express + MongoDB) is easier when we just focus on **data**, not visuals.
- It keeps things clean: one thing at a time.
- And when the UI part comes later, boom 💥 — all the data handling is already solid.

Let's look at the api.

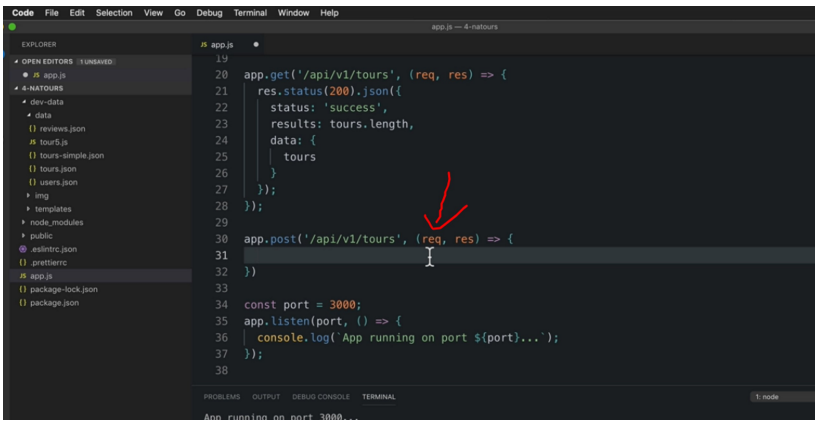



Node.js Course
Invite
No Environment
https://www.natours.dev/api/v1/tours/5c88fa8cf4afda39709c2955
GET
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Query Params
KEY
VALUE
DESCRIPTION
You don't have any collections
Create a collection
{
"status": "success",
"data": {
"startLocation": {
"type": "Point",
"coordinates": [
-80.185942,
25.774772
],
"description": "Miami, USA",
"address": "301 Biscayne Blvd, Miami, FL 33132, USA"
},
"ratingsAverage": 4.8,
"ratingsQuantity": 5,
"images": [
"tour-2-1.jpg",
"tour-2-2.jpg",
"tour-2-3.jpg"
],
"startDates": [
"2021-06-19T09:00:00.000Z",
"2021-07-20T09:00:00.000Z",
"2021-08-18T09:00:00.000Z"
],
"createdAt": "2019-03-13T12:41:48.942Z",

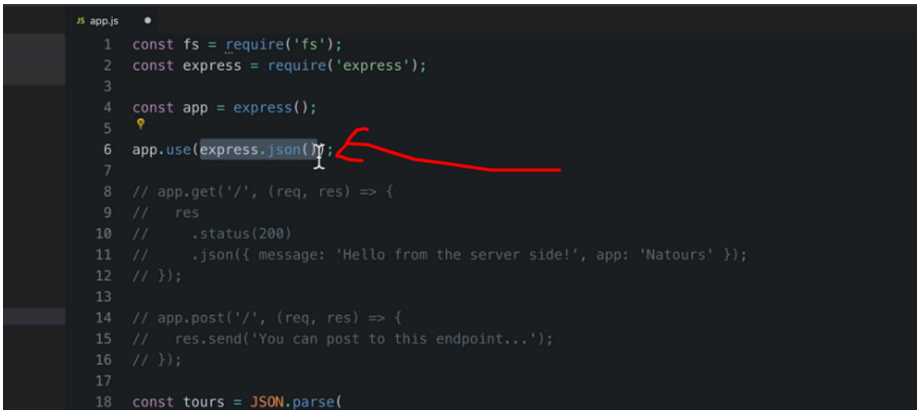

app.js — 4-natours
EXPLORER
OPEN EDITORS
package.json
app.js
4-NATOURS
dev-data
data
reviews.json
tour5.js
tours-simple.json
tours.json
users.json
img
templates
node_modules
public
.eslintrc.json
.prettierrc
app.js
package-lock.json
package.json
// res.send('You can post to this endpoint...');
// });
const tours = JSON.parse(
fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
);
app.get('/api/v1/tours', (req, res) => {
res.status(200).json({
status: 'success',
data: {
tours
}
});
});
const port = 3000;
app.listen(port, () => {
console.log(`App running on port ${port}...`);
});
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
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...
[nodemon] restarting due to changes...


Postman
Node.js Course
Invite
No Environment
127.0.0.1:3000/api/v1/tours
GET
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Query Params
KEY
VALUE
DESCRIPTION
You don't have any collections
Create a collection
{
"status": "success",
"data": {
"tours": [
{
"name": "The Forest Hiker",
"duration": 5,
"maxGroupSize": 25,
"difficulty": "easy",
"ratingsAverage": 4.7,
"ratingsQuantity": 37,
"price": 397,
"summary": "Breathtaking 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",
"tour-1-3.jpg"

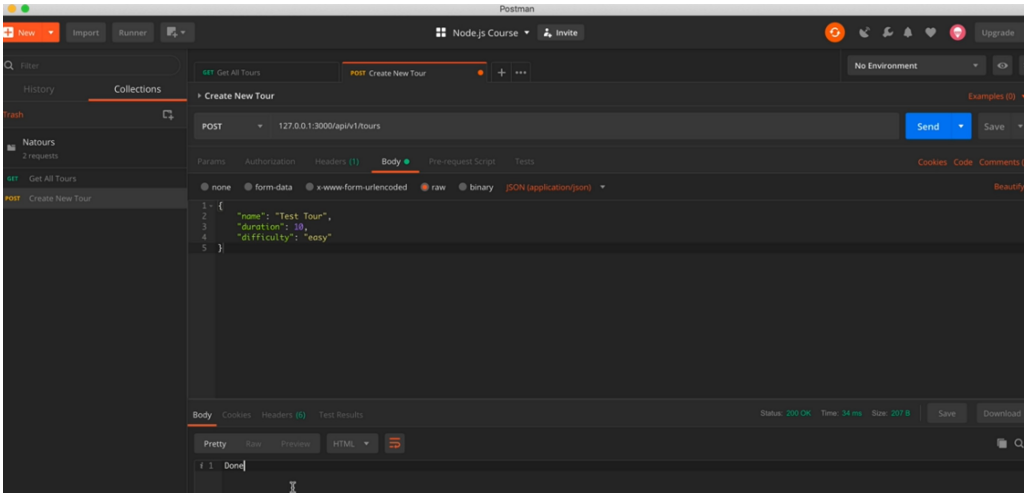
Express does not put body data to the request. In order to have that data available, we have to use something called middleware.

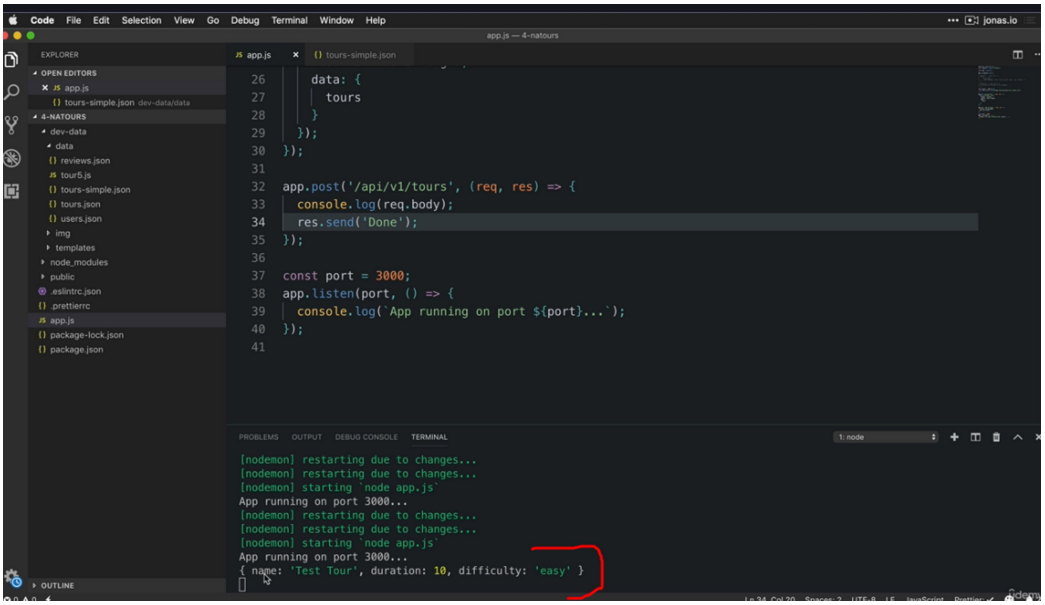



```
//express.json() is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on body-parser.
//Middleware is basically just a function that can modify the incoming request data.
```

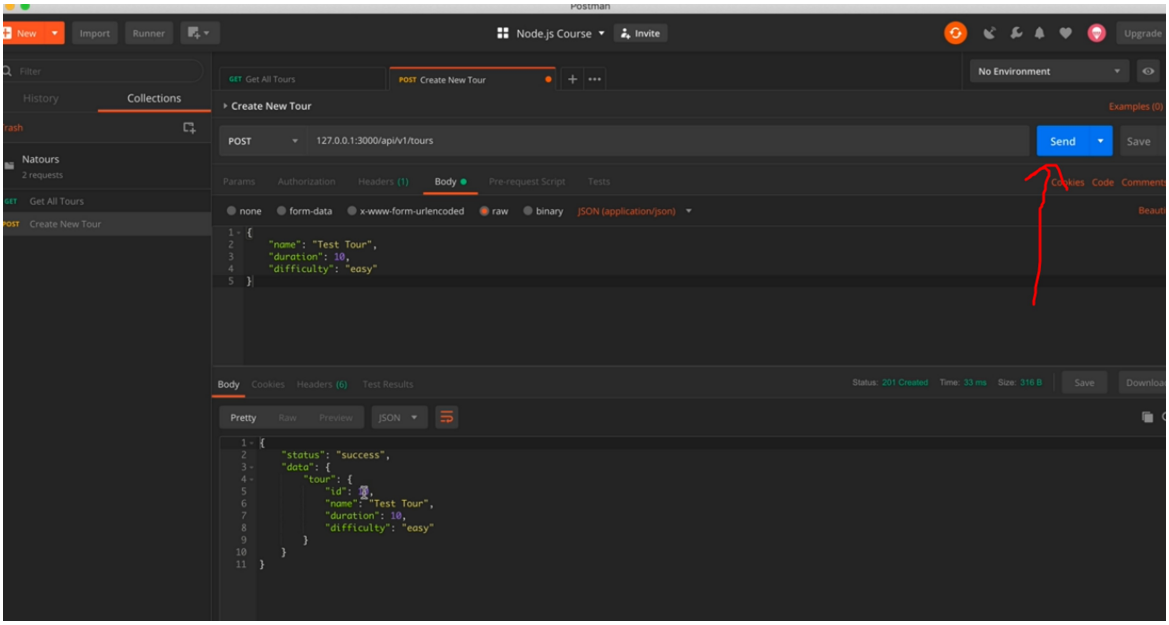



```
app.post('/api/v1/tours', (req, res) => {
    console.log(req.body); // body is the available now because we used that middleware.
    res.send('Done'); //we always need to send back something in order to finish request-response cycle.
});
```



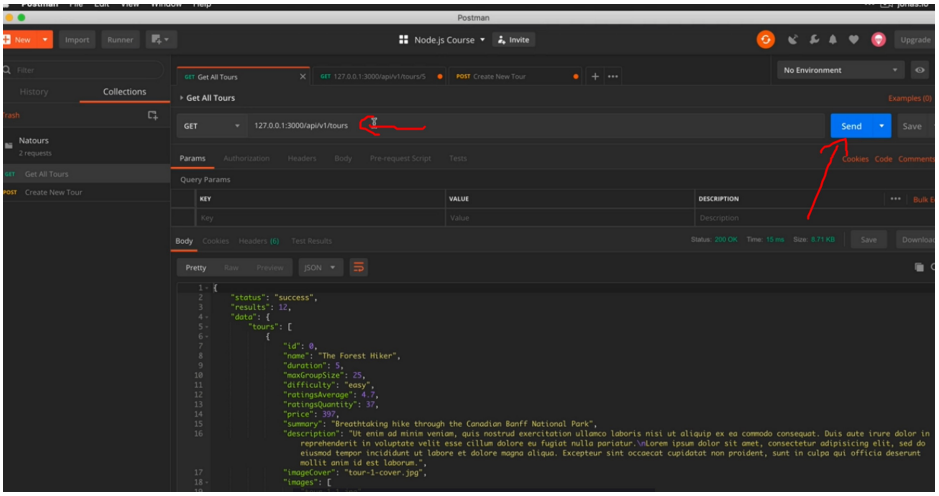


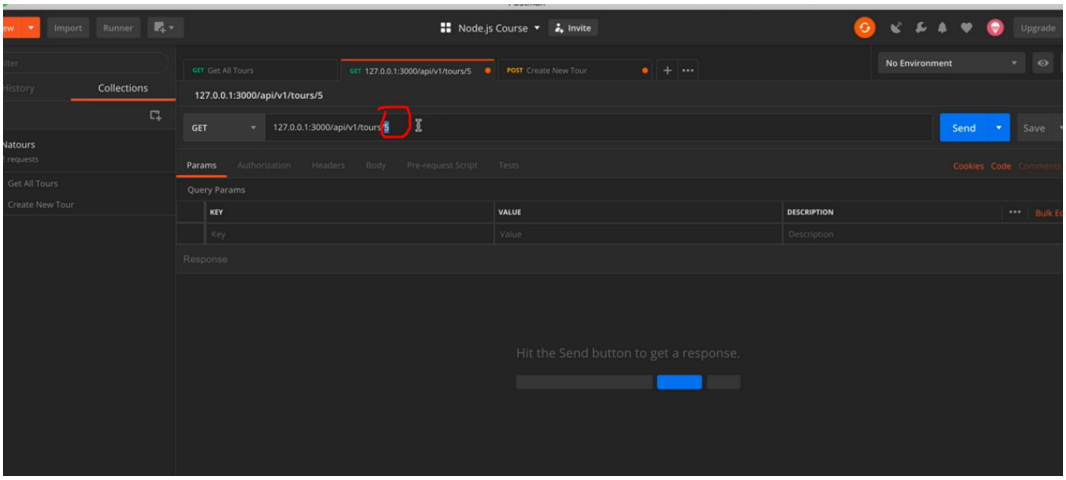

```
app.post('/api/v1/tours', (req, res) => {
    // console.log(req.body); // body is the available now because we used that middleware.
    const newId = tours[tours.length - 1].id + 1;
    const newTour = Object.assign({ id: newId }, req.body); // create a new object with the newId and the body of the
    request.
    tours.push(newTour); // push the new tour to the array.
    fs.writeFile(`${__dirname}/dev-data/data/tours-simple.json`, JSON.stringify(tours), err => {
        res.status(201).json({
            status: 'success',
            data: {
                tour: newTour,
            }
        });
    }); // why async function?  why not just use fs.writeFileSync? because we want to make sure that the server is not
    blocked while writing the file. We want to be able to handle other requests while this one is being processed. We
    inside a callback function that is gonna run in the event loop.

    res.send('Done'); //We always need to send back something in order to finish request-response cycle.
});
```



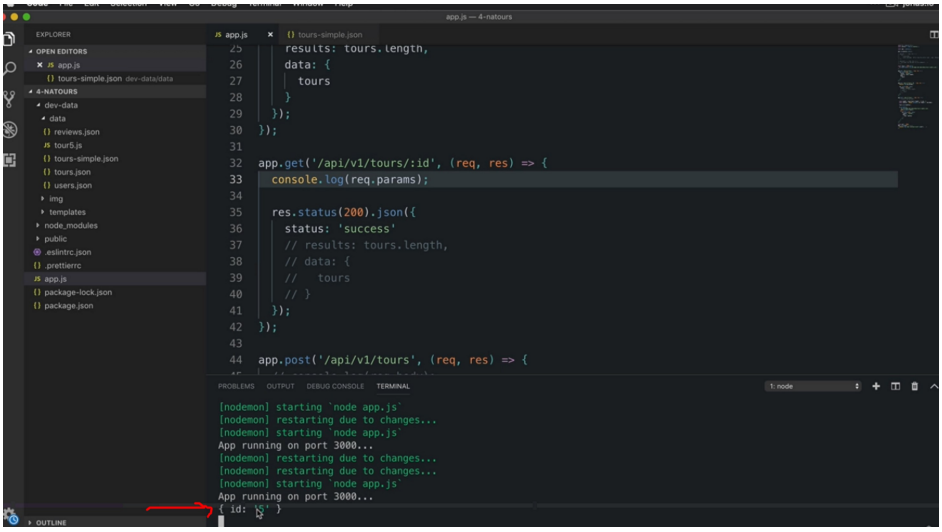
If we hit

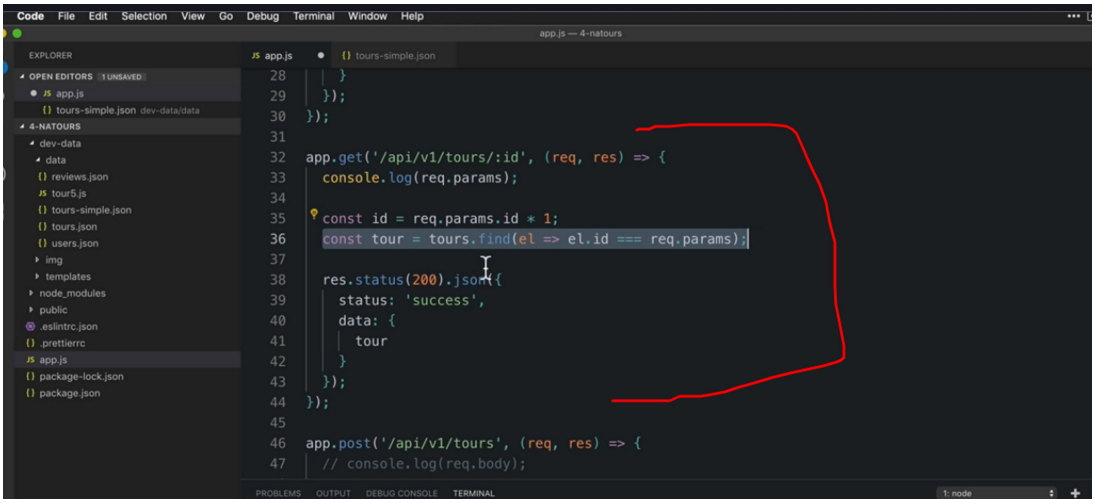
=> api/v1/tours ; we would get all the tour

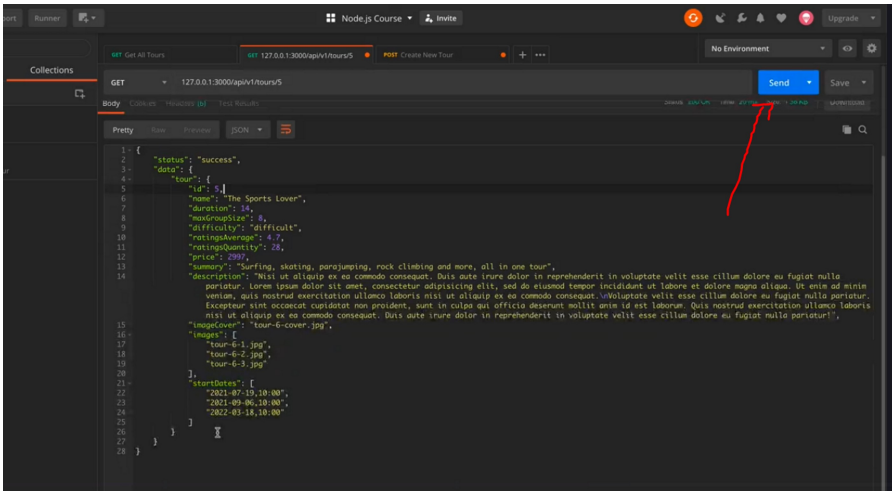


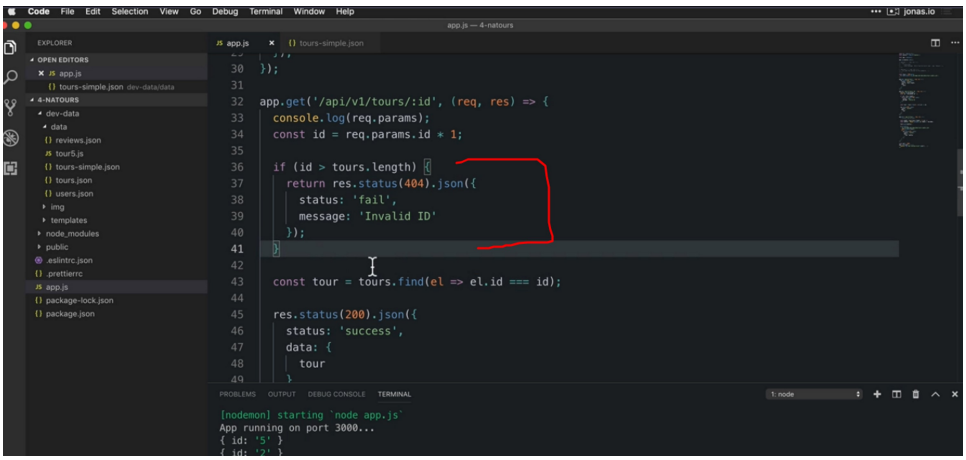


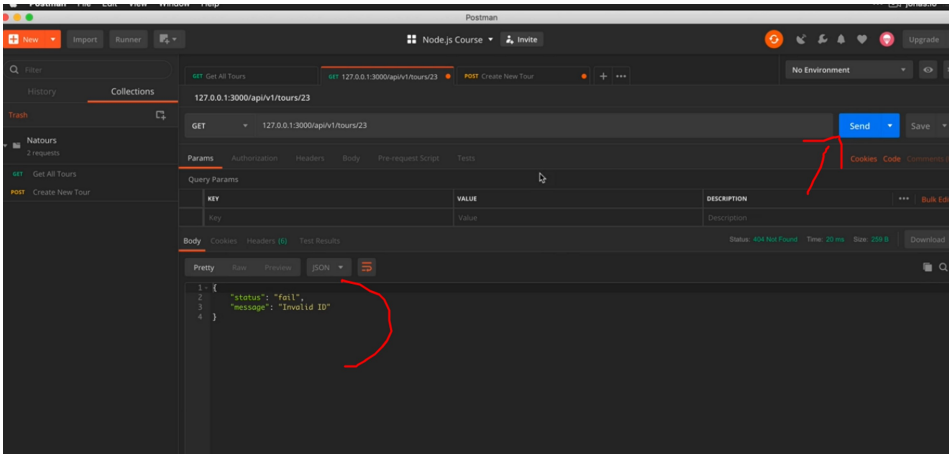
Hit from the postman and we have got id to the console.

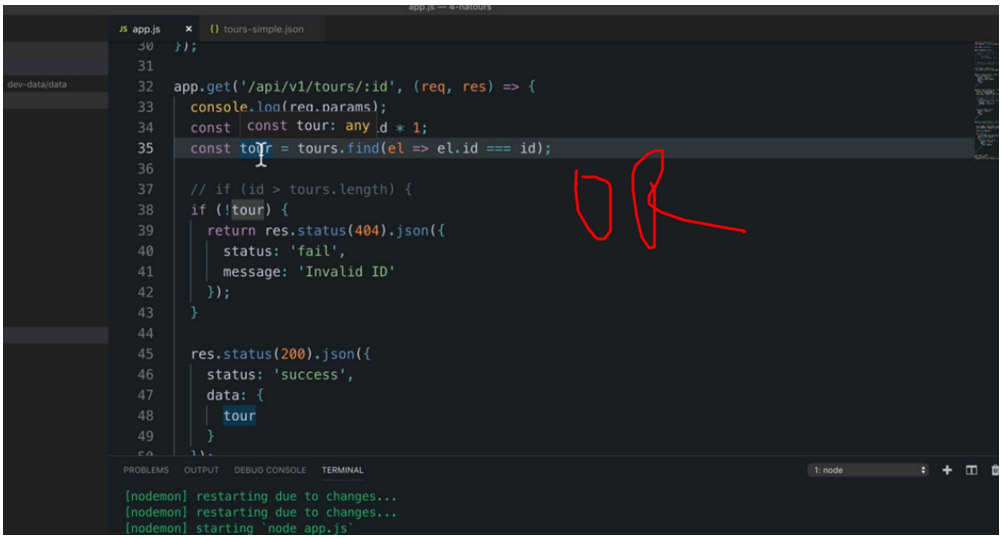


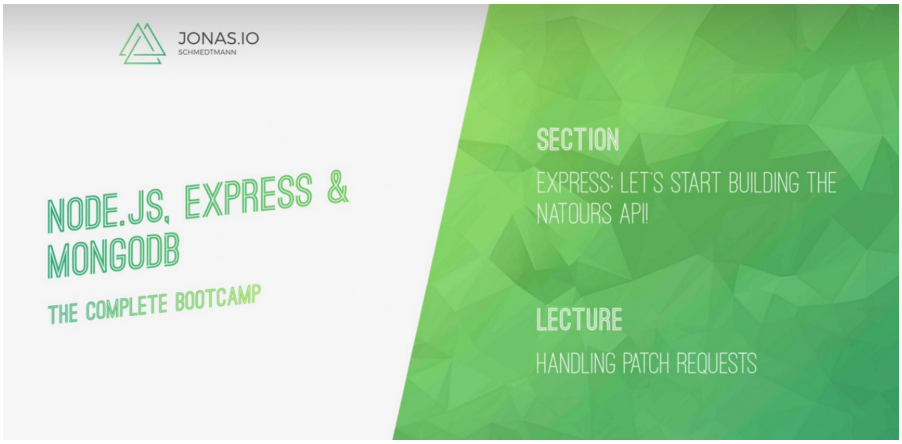

app.js — 4-natours
app.get('/api/v1/tours/:id', (req, res) => {
console.log(req.params);
const id = req.params.id * 1;
const tour = tours.find(el => el.id === req.params);
res.status(200).json({
status: 'success',
data: {
tour
}
});
});
app.post('/api/v1/tours', (req, res) => {
// console.log(req.body);


Node.js Course
Collections
GET
127.0.0.1:3000/api/v1/tours/5
Send
Save
{
"status": "success",
"data": {
"tour": {
"id": 5,
"name": "The Sports Lover",
"duration": 14,
"maxGroupSize": 8,
"difficulty": "difficult",
"ratingsAverage": 4.7,
"ratingsQuantity": 28,
"price": 2997,
"summary": "Surfing, skating, parajumping, rock climbing and more, all in one tour",
"description": "Nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.\nVoluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. 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!",
"imageCover": "tour-6-cover.jpg",
"images": [
"tour-6-1.jpg",
"tour-6-2.jpg",
"tour-6-3.jpg"
],
"startDates": [
"2021-07-19,10:00",
"2021-09-06,10:00",
"2022-03-18,10:00"
]
}
}
}

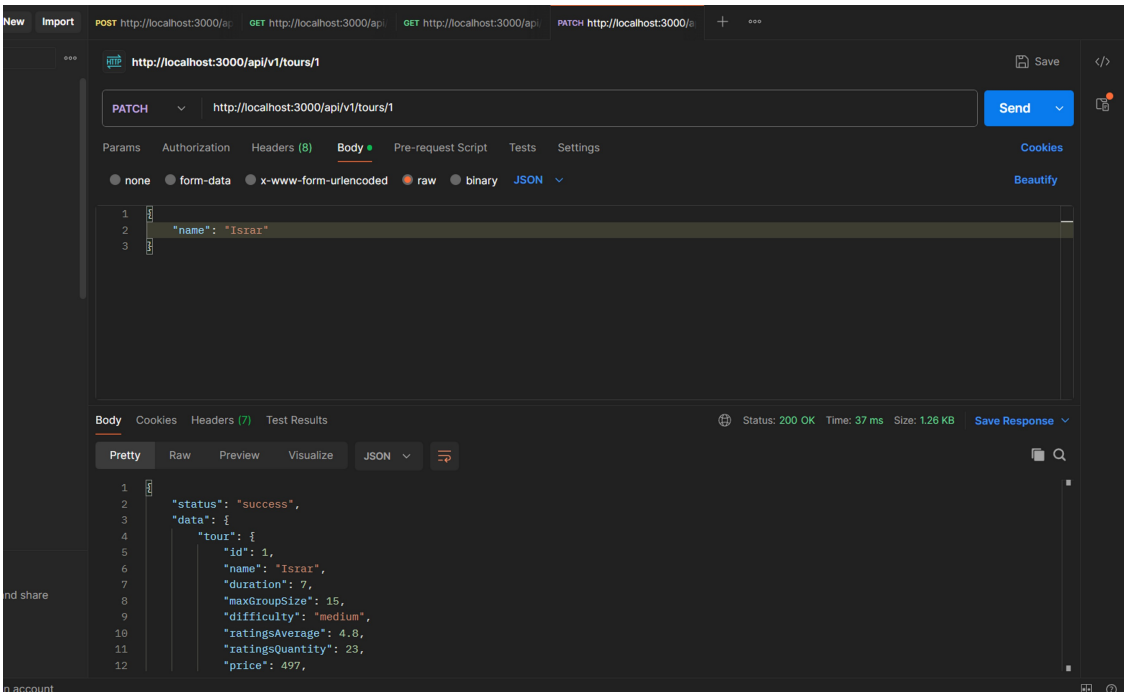

app.js — 4-natours
app.get('/api/v1/tours/:id', (req, res) => {
console.log(req.params);
const id = req.params.id * 1;
if (id > tours.length) {
return res.status(404).json({
status: 'fail',
message: 'Invalid ID'
});
}
const tour = tours.find(el => el.id === id);
res.status(200).json({
status: 'success',
data: {
tour
[nodemon] starting `node app.js`
App running on port 3000...
{ id: '5' }
{ id: '2' }

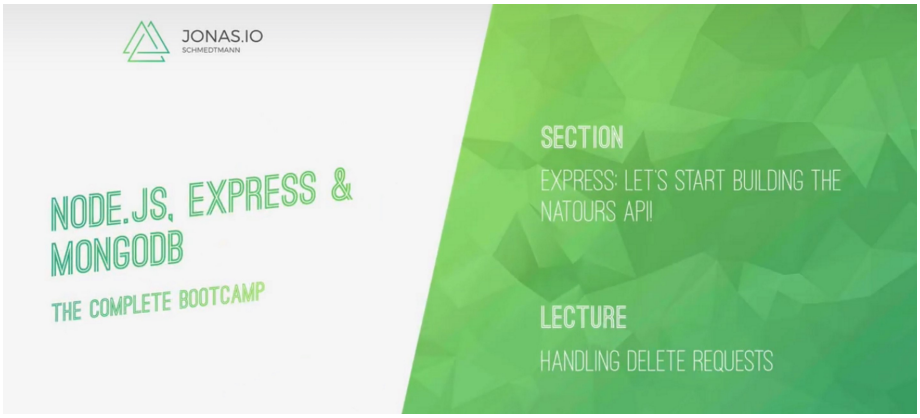

Postman
Node.js Course
Invite
Upgrade
No Environment
History
Collections
Trash
Natours
2 requests
GET Get All Tours
POST Create New Tour
127.0.0.1:3000/api/v1/tours/23
GET
127.0.0.1:3000/api/v1/tours/23
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Query Params
KEY
VALUE
DESCRIPTION
Body
Cookies
Headers (6)
Test Results
Status: 404 Not Found
Time: 20 ms
Size: 259 B
Download
Pretty
Raw
Preview
JSON
{
"status": "fail",
"message": "Invalid ID"
}


app.js
tours-simple.json
});
app.get('/api/v1/tours/:id', (req, res) => {
console.log(req.params);
const id = req.params.id * 1;
const tour = tours.find(el => el.id === id);
// if (id > tours.length) {
if (!tour) {
return res.status(404).json({
status: 'fail',
message: 'Invalid ID'
});
}
res.status(200).json({
status: 'success',
data: {
tour
}
OR
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: node
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`

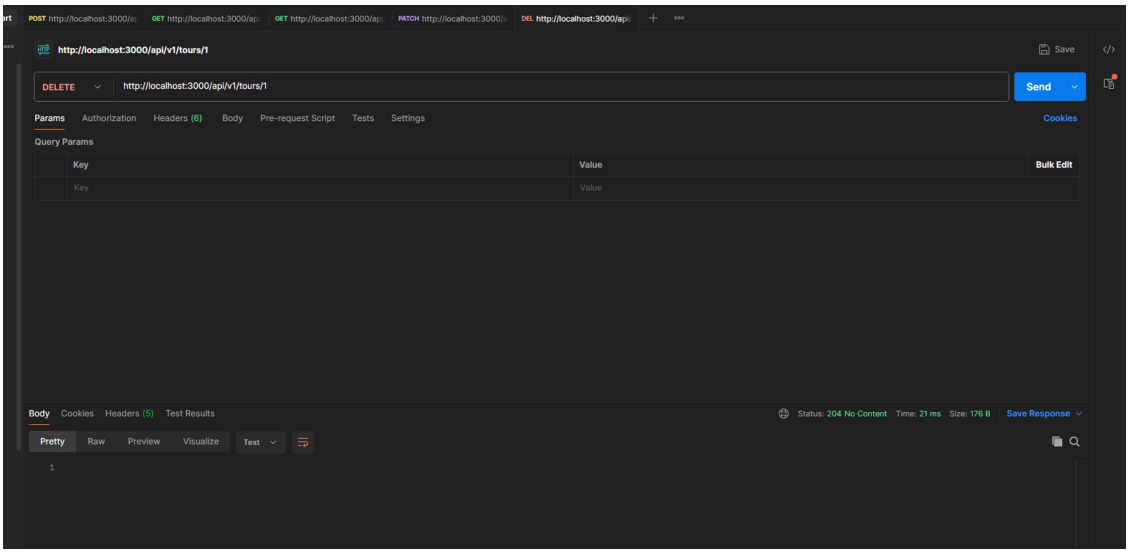

JONAS.IO
SCHMEDTMANN
NODE.JS, EXPRESS & MONGODB
THE COMPLETE BOOTCAMP
SECTION
EXPRESS: LET'S START BUILDING THE NATOURS API!
LECTURE
HANDLING PATCH REQUESTS

```
    // res.send('Done'); //we always need to send back something in order to finish request-response cycle.
});

app.patch('/api/v1/tours/:id', (req, res) => {
    const id = req.params.id * 1; // convert the string to a number
    if(id > tours.length) {
        return res.status(404).json({
            status: 'fail',
            message: 'Invalid ID'
        });
    }

    const tour = tours.find(tour => tour.id === id); // find the tour with the given id
    const updatedTour = Object.assign(tour, req.body); // update the tour with the body of the request

    fs.writeFile(`${__dirname}/dev-data/data/tours-simple.json`, JSON.stringify(tours), err => {
        res.status(200).json({
            status: 'success',
            data: {
                tour: updatedTour,
            }
        });
    });       You, 2 minutes ago • Uncommitted changes
});
```





```
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 });       You, 1 second ago • Uncommitted changes
124
```

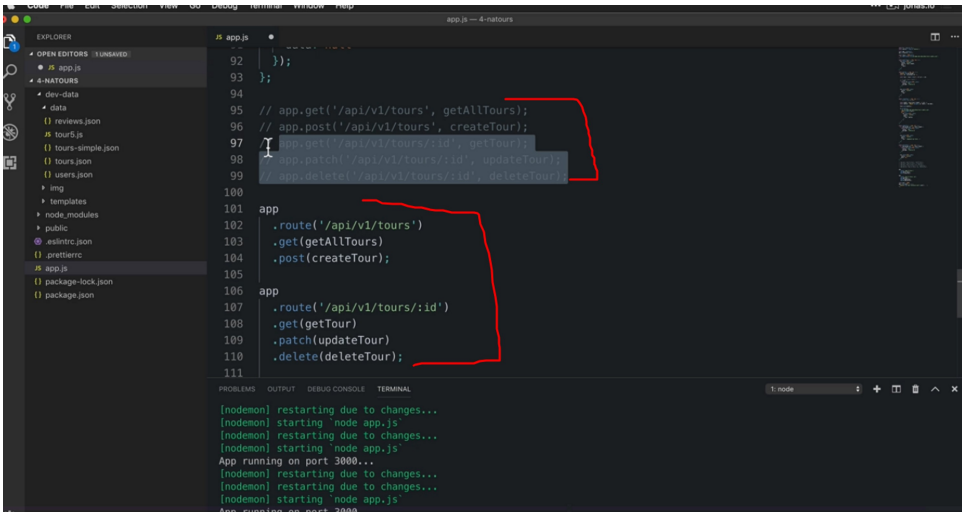


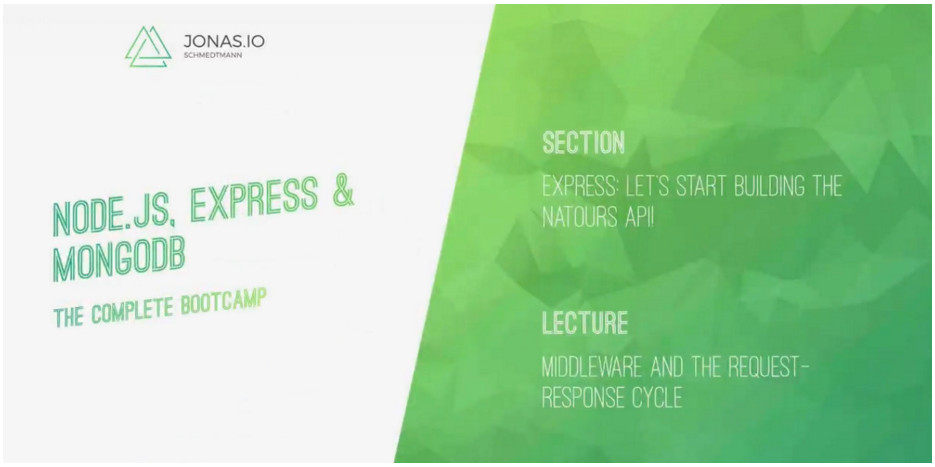
Let’s now refactor our code a little bit. So basically reorganize some of our route to make the code a lot better.

```
};

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 chage It in all of these five places (red marked).

We can do something better.





THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

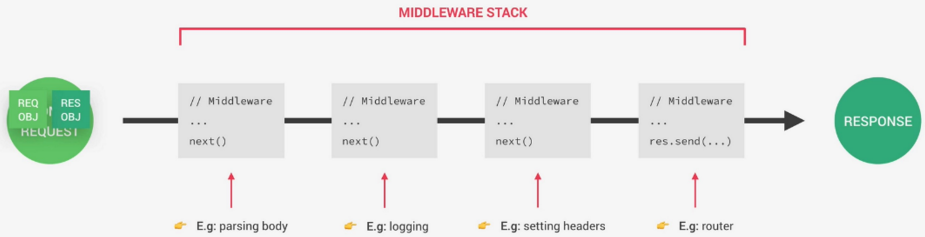
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

👉 *"Everything is middleware" (even routers)*

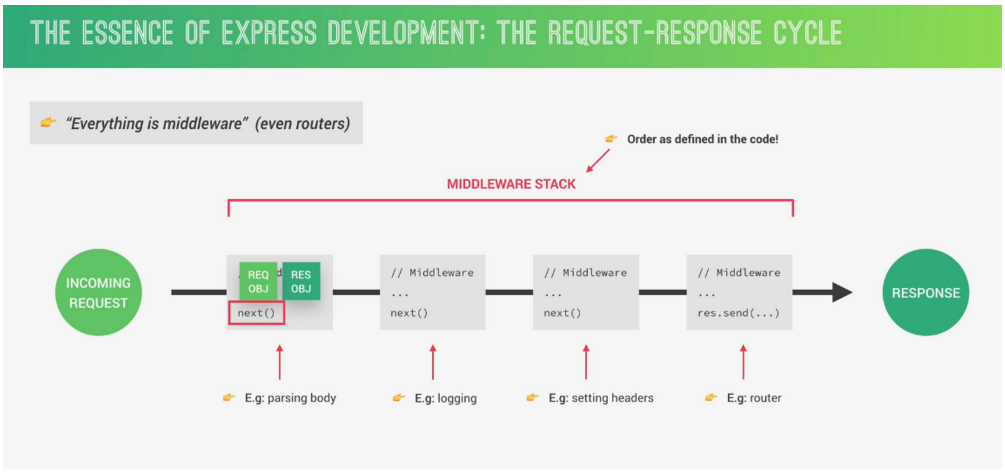
👉 E.g: parsing body 👉 E.g: logging 👉 E.g: setting headers 👉 E.g: router

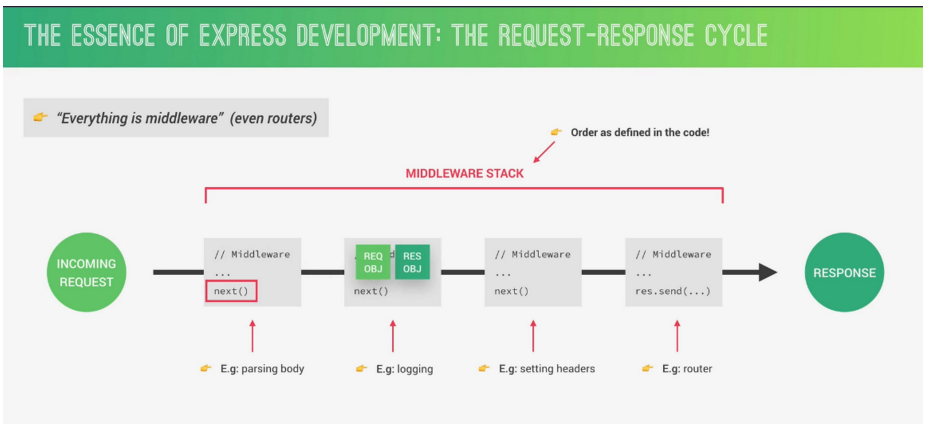
THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

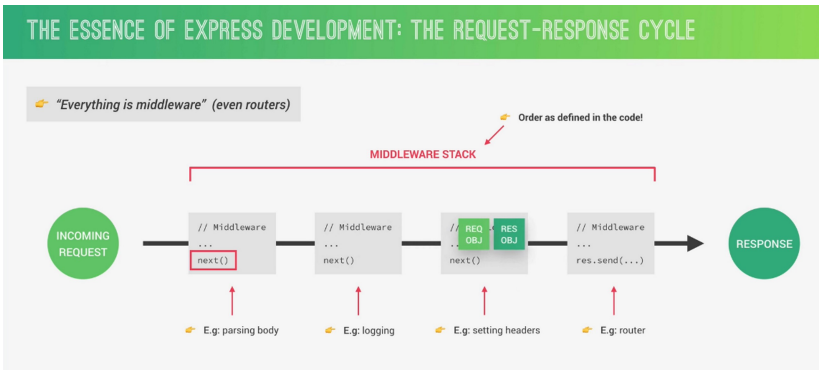
👉 *"Everything is middleware" (even routers)*

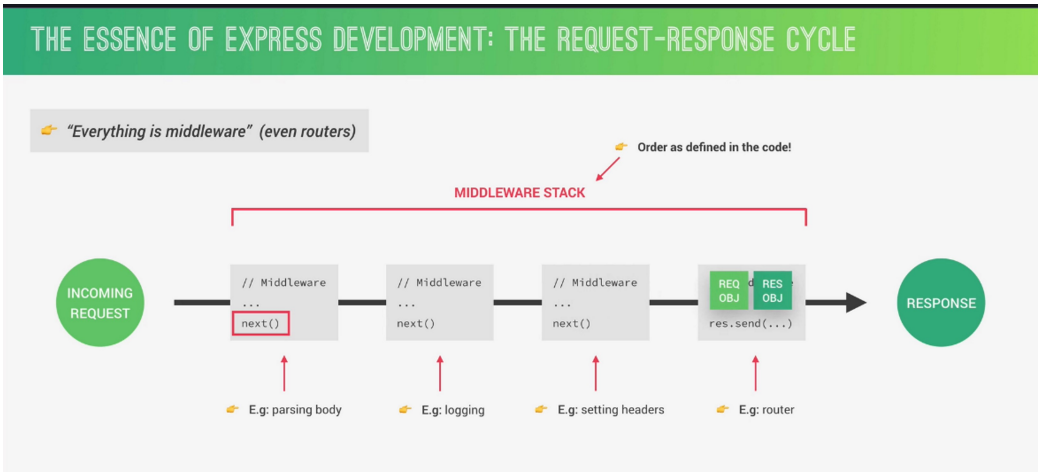


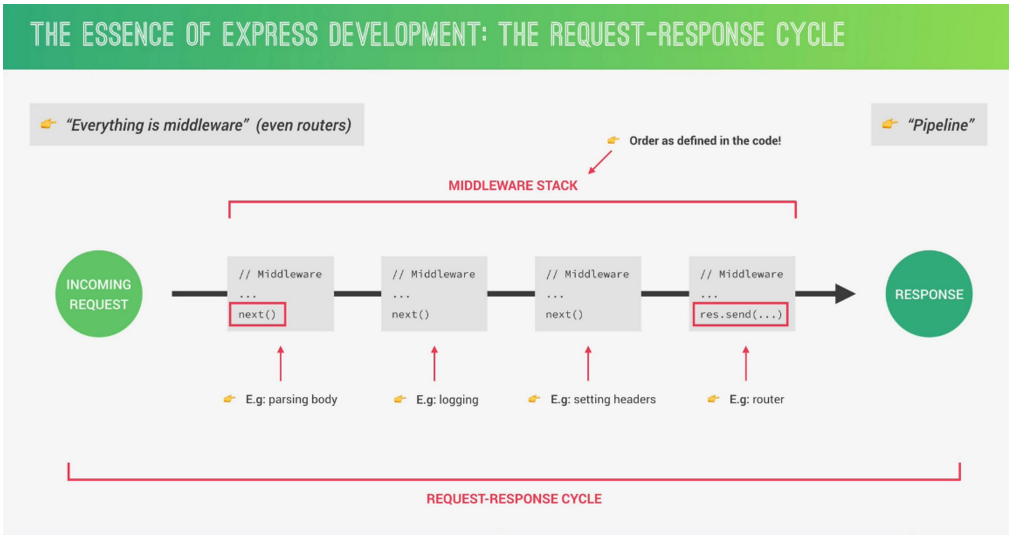
👉 E.g: parsing body 👉 E.g: logging 👉 E.g: setting headers 👉 E.g: router


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE
"Everything is middleware" (even routers)
Order as defined in the code!
MIDDLEWARE STACK
INCOMING REQUEST
REQ OBJ
RES OBJ
next()
// Middleware
...
next()
// Middleware
...
next()
// Middleware
...
res.send(...)
RESPONSE
E.g: parsing body
E.g: logging
E.g: setting headers
E.g: router


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE
"Everything is middleware" (even routers)
Order as defined in the code!
MIDDLEWARE STACK
INCOMING REQUEST
// Middleware
...
next()
REQ OBJ
RES OBJ
next()
// Middleware
...
next()
// Middleware
...
res.send(...)
RESPONSE
E.g: parsing body
E.g: logging
E.g: setting headers
E.g: router


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE
"Everything is middleware" (even routers)
Order as defined in the code!
MIDDLEWARE STACK
INCOMING REQUEST
// Middleware
...
next()
// Middleware
...
next()
REQ OBJ
RES OBJ
next()
// Middleware
...
res.send(...)
RESPONSE
E.g: parsing body
E.g: logging
E.g: setting headers
E.g: router


THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE
"Everything is middleware" (even routers)
Order as defined in the code!
MIDDLEWARE STACK
INCOMING REQUEST
// Middleware
...
next()
// Middleware
...
next()
// Middleware
...
next()
REQ OBJ
RES OBJ
res.send(...)
RESPONSE
E.g: parsing body
E.g: logging
E.g: setting headers
E.g: router



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.

🧩 **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 🚀

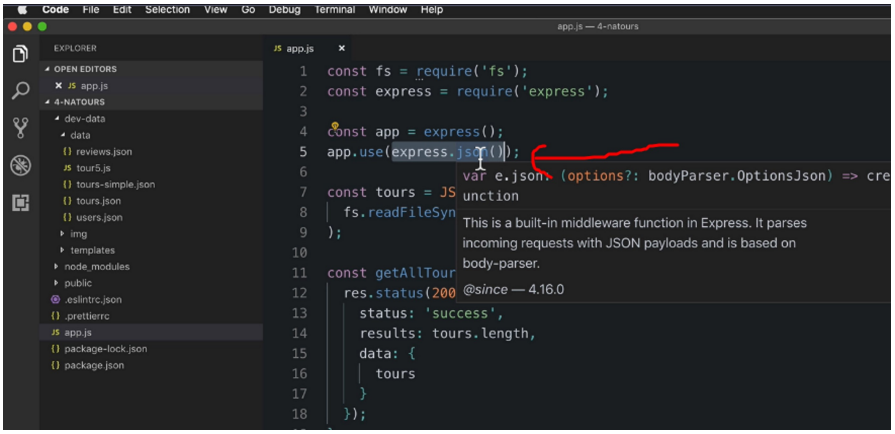
Alright, let's now go ahead and **create our own middleware functions**!
But first, quick reminder—we've actually already used middleware before. Let's take a look.
Somewhere earlier in our code, we used:

app.use(express.json());

This line is key!

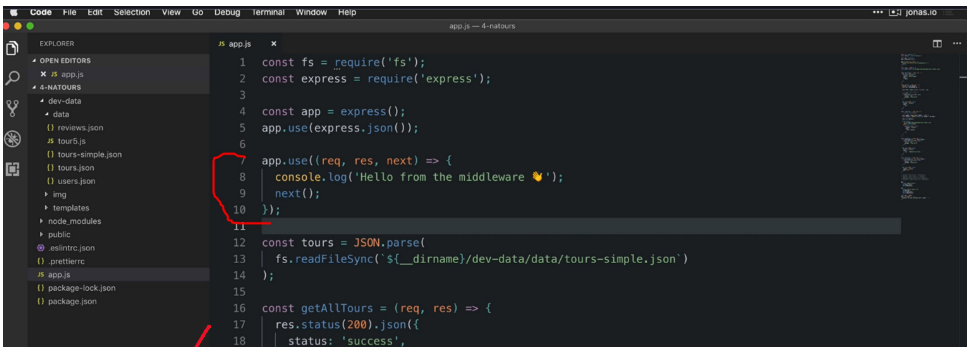
- app.use() is the method we use to **add middleware** to our **middleware stack**.
- express.json() returns a **middleware function**.
- That function is then registered and executed for every incoming request.

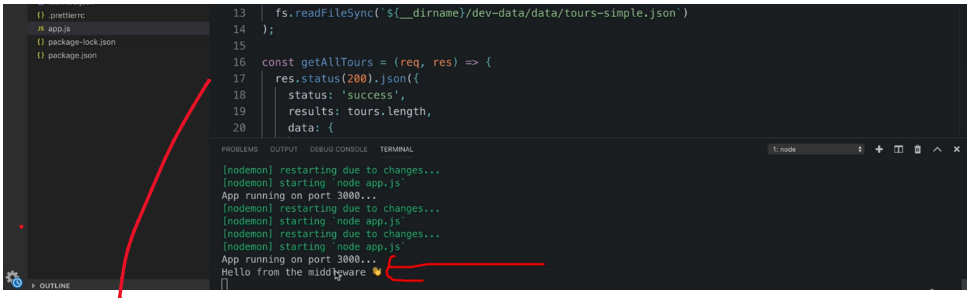
So yeah, we've already been working with middleware—now it's time to build our own! 🙌



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.



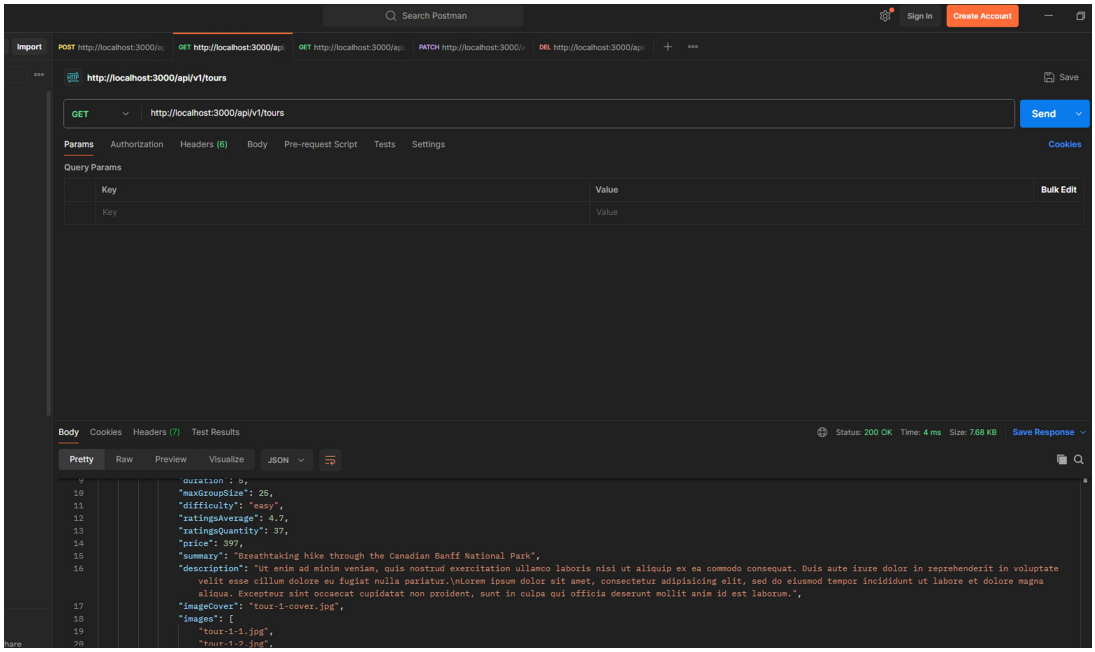


This middleware is applied to each and every request. That's because we didn't specify any route.

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

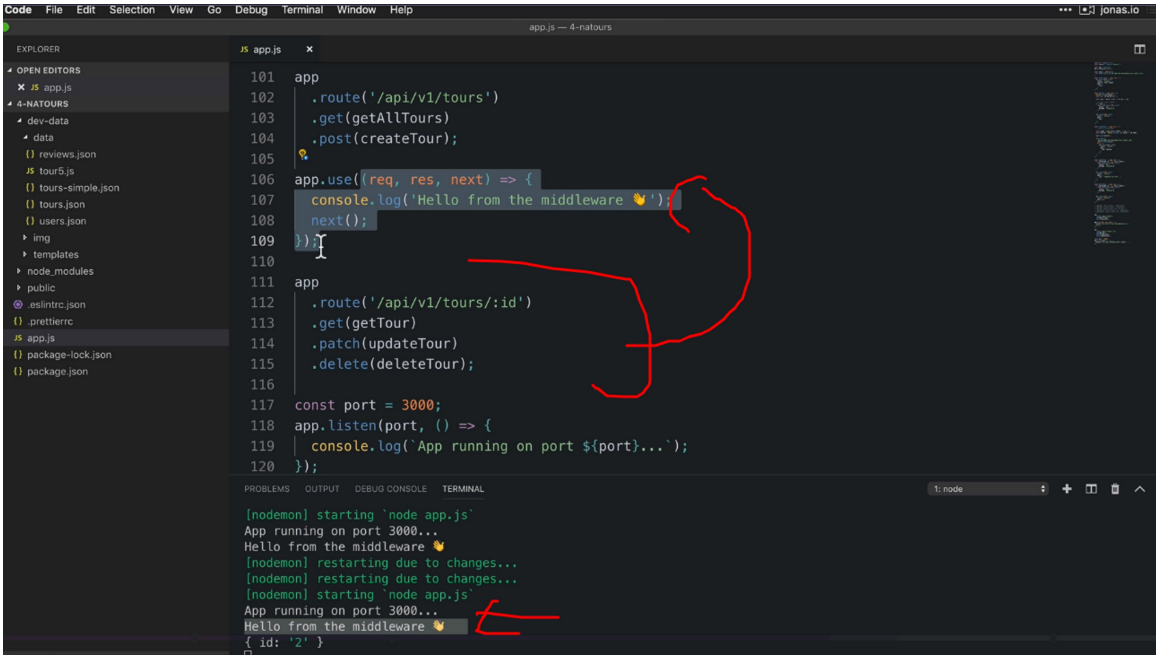
Let's change the order of the middleware.

From postman, hit this endpoint.



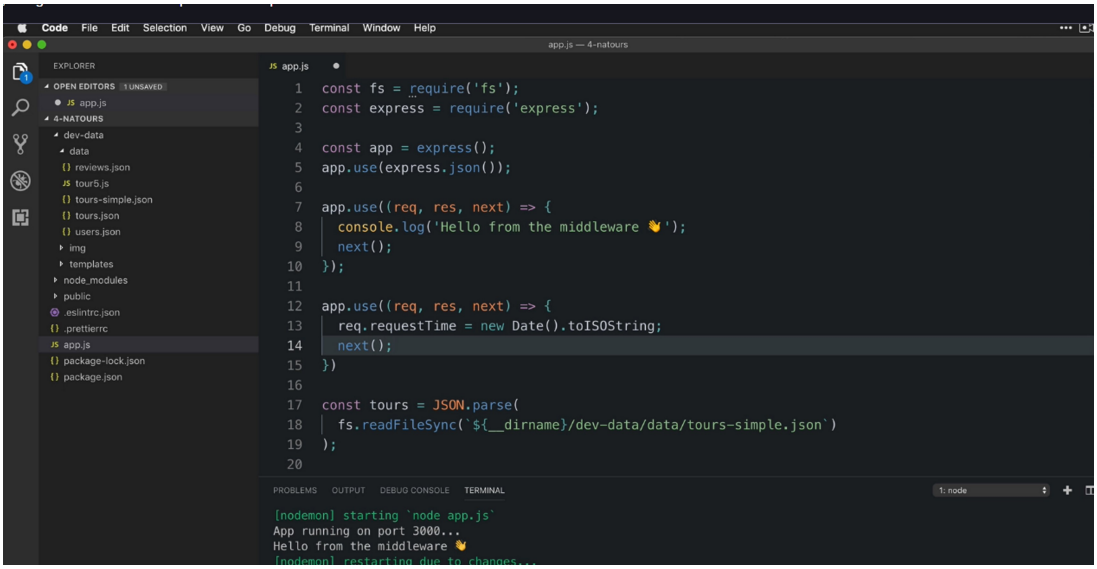
Now the middleware code does not get executed.

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



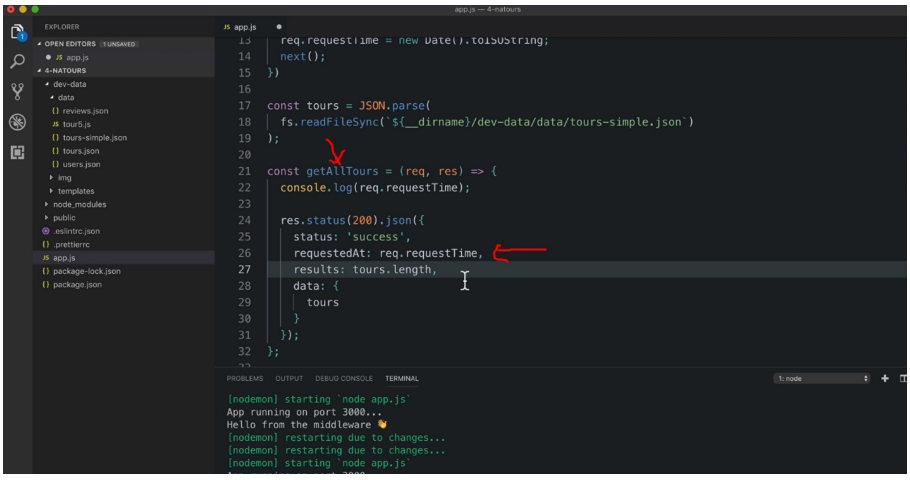
```
app
  .route('/api/v1/tours')
  .get(getAllTours)
  .post(createTour);

app.use((req, res, next) => {
  console.log('Hello from the middleware 👋');
  next();
});

app
  .route('/api/v1/tours/:id')
  .get(getTour)
  .patch(updateTour)
  .delete(deleteTour);

const port = 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});
```

```
[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' }
```



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

const app = express();
app.use(express.json());

app.use((req, res, next) => {
  console.log('Hello from the middleware 👋');
  next();
});

app.use((req, res, next) => {
  req.requestTime = new Date().toISOString;
  next();
})

const tours = JSON.parse(
  fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
);
```

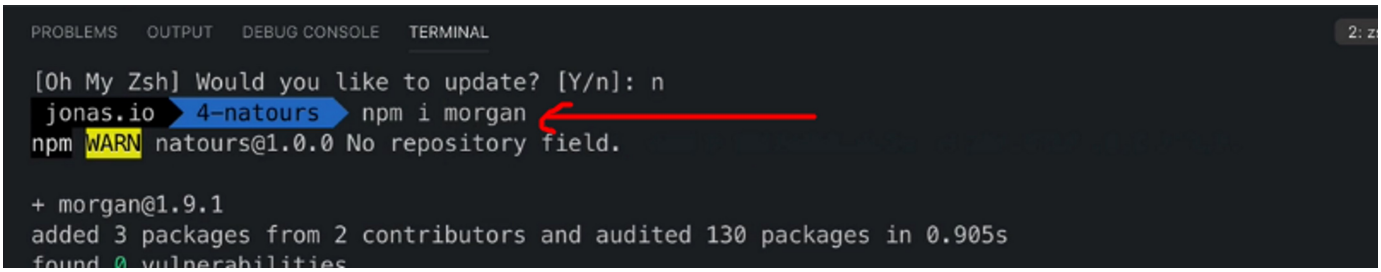
```
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 👋
```



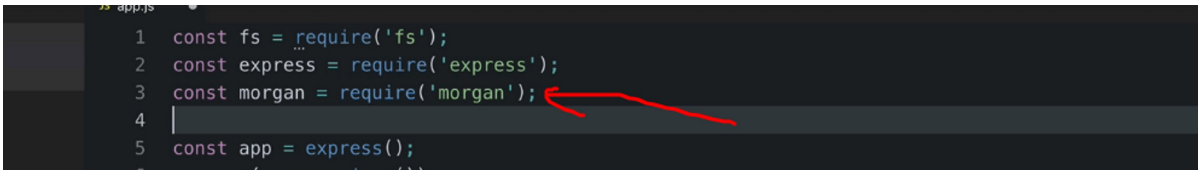
```
  req.requestTime = new Date().toISOString;
  next();
})

const tours = JSON.parse(
  fs.readFileSync(`${__dirname}/dev-data/data/tours-simple.json`)
);

const getAllTours = (req, res) => {
  console.log(req.requestTime);

  res.status(200).json({
    status: 'success',
    requestedAt: req.requestTime,
    results: tours.length,
    data: {
      tours
    }
  });
};
```

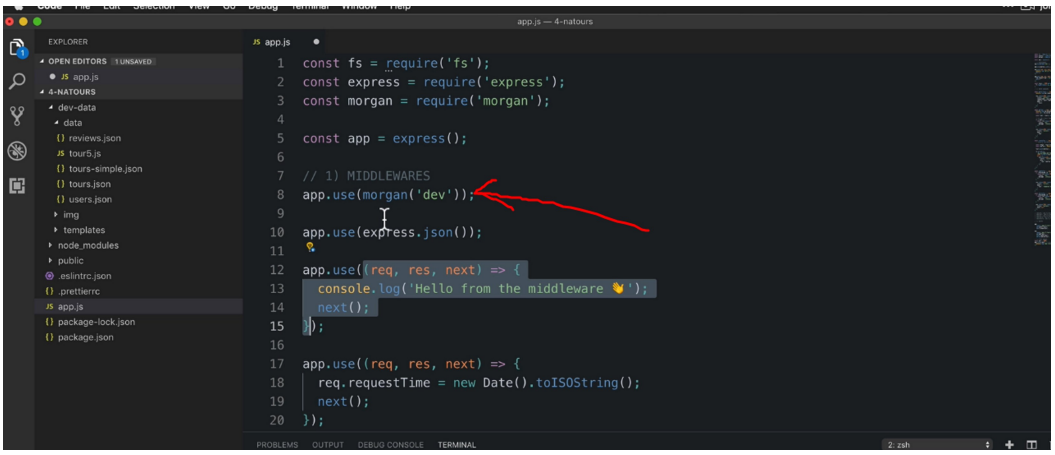
```
[nodemon] starting `node app.js`
App running on port 3000...
Hello from the middleware 👋
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
```

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



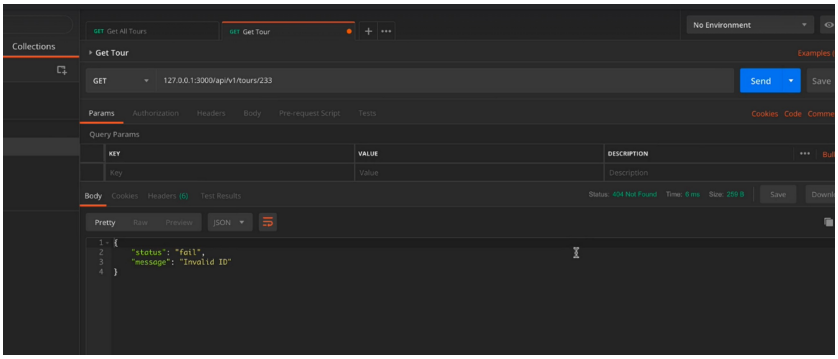
This logging middleware is gonna make our life easier. (Regular Dependency)



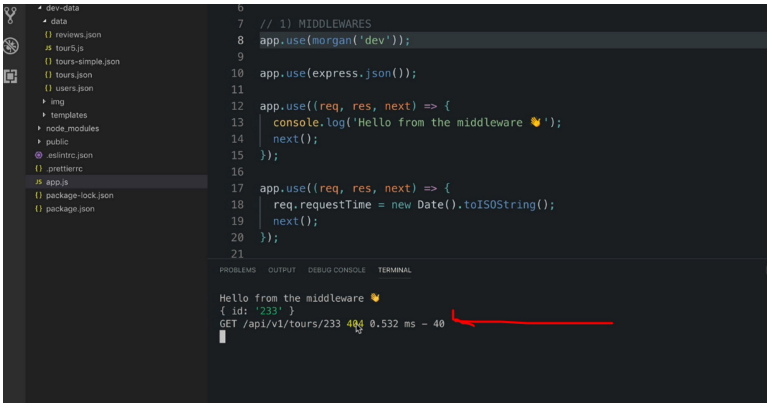


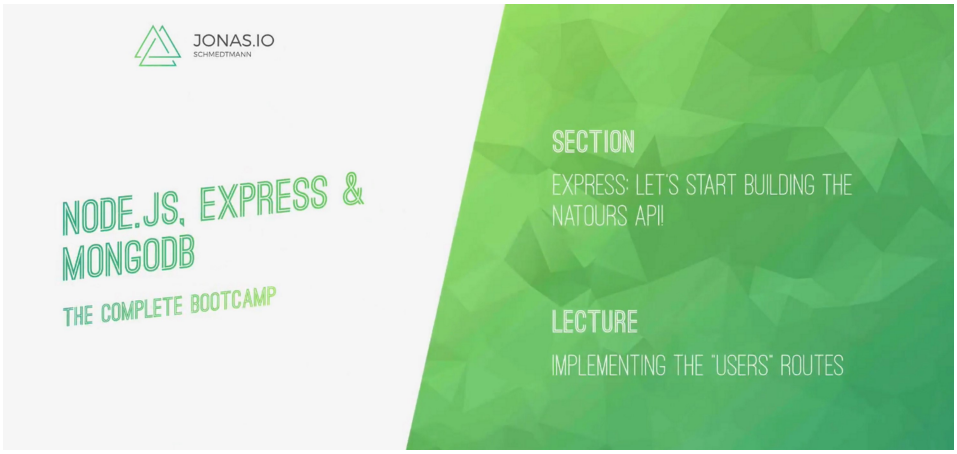
morgan() it returns a middleware function.

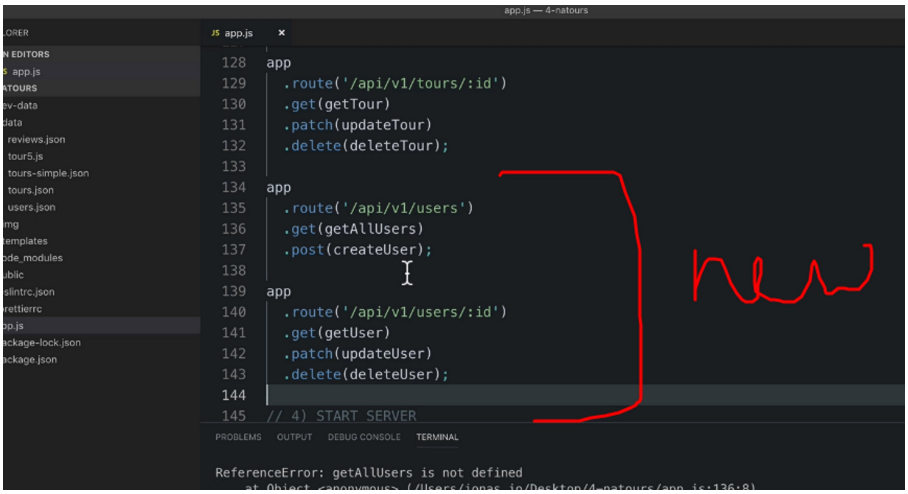
Let's pass a invalid id.

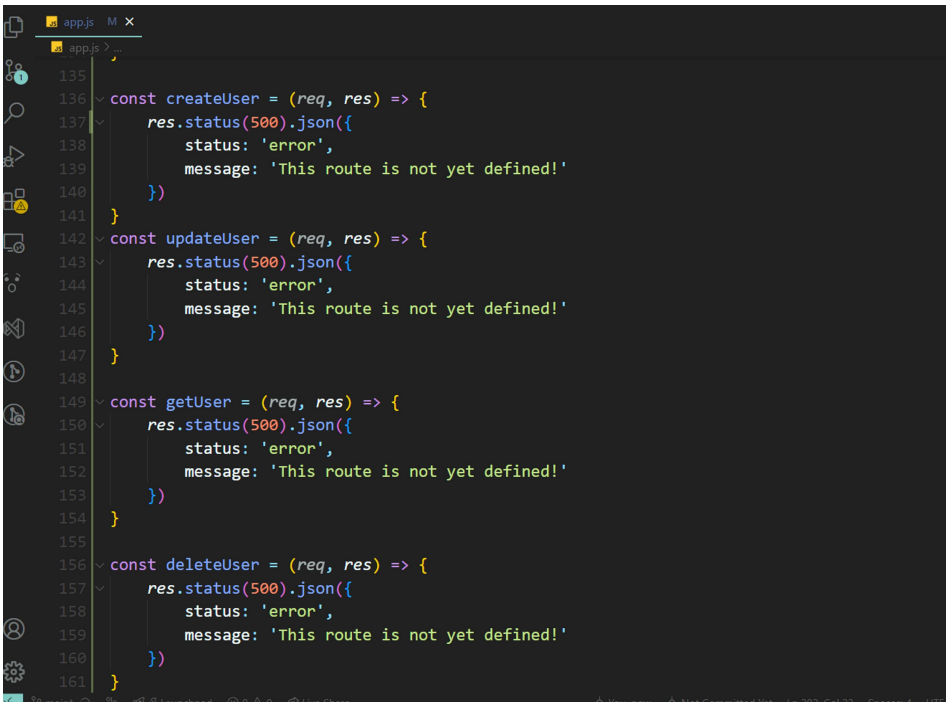


We got 404 in different color.





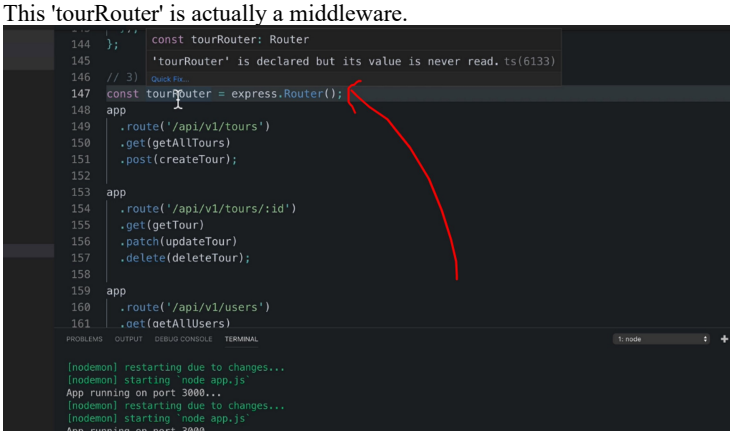




In this lecture we will now create multiple routers and use a process called 'mounting'.
The ultimate goal will be to separate all the code that we have in this file into multiple files.

We have created router and save into this variable.

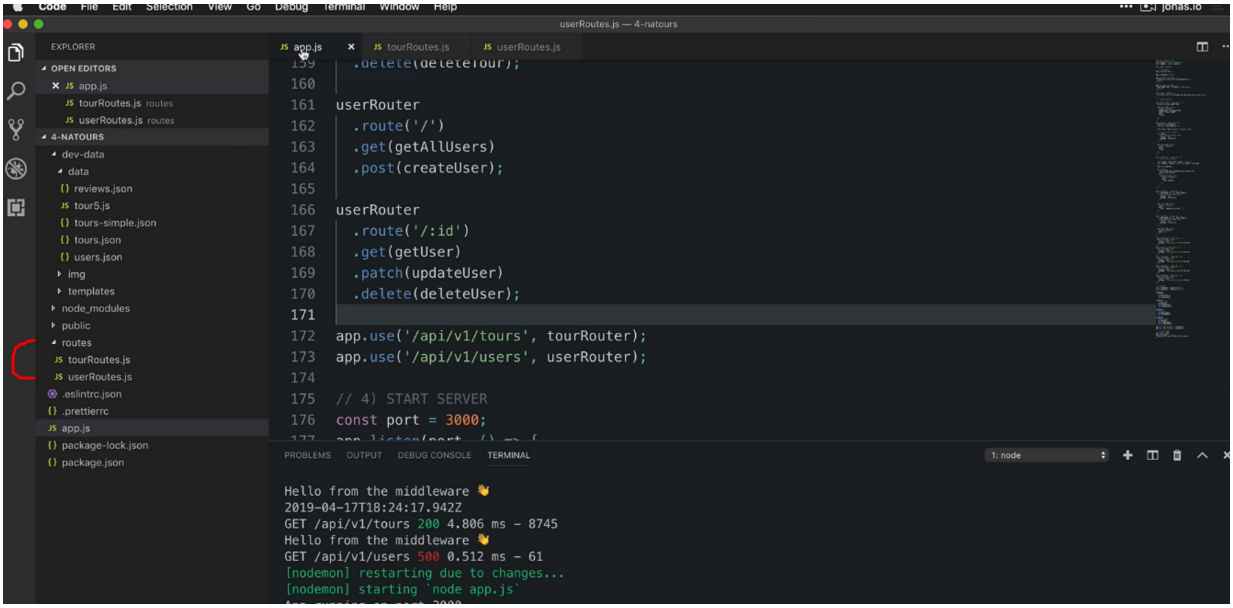
This 'tourRouter' is actually a middleware.



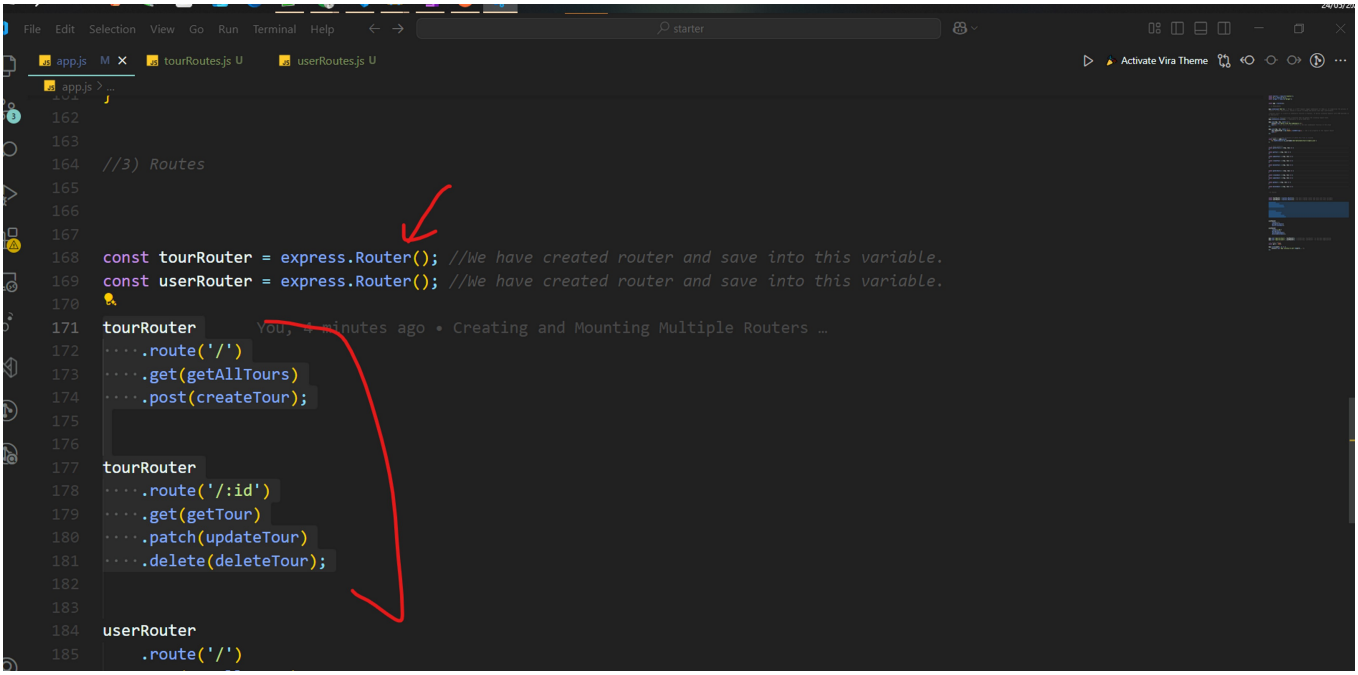
We want this 'tourRouter' middleware for this specific route.

```
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
```

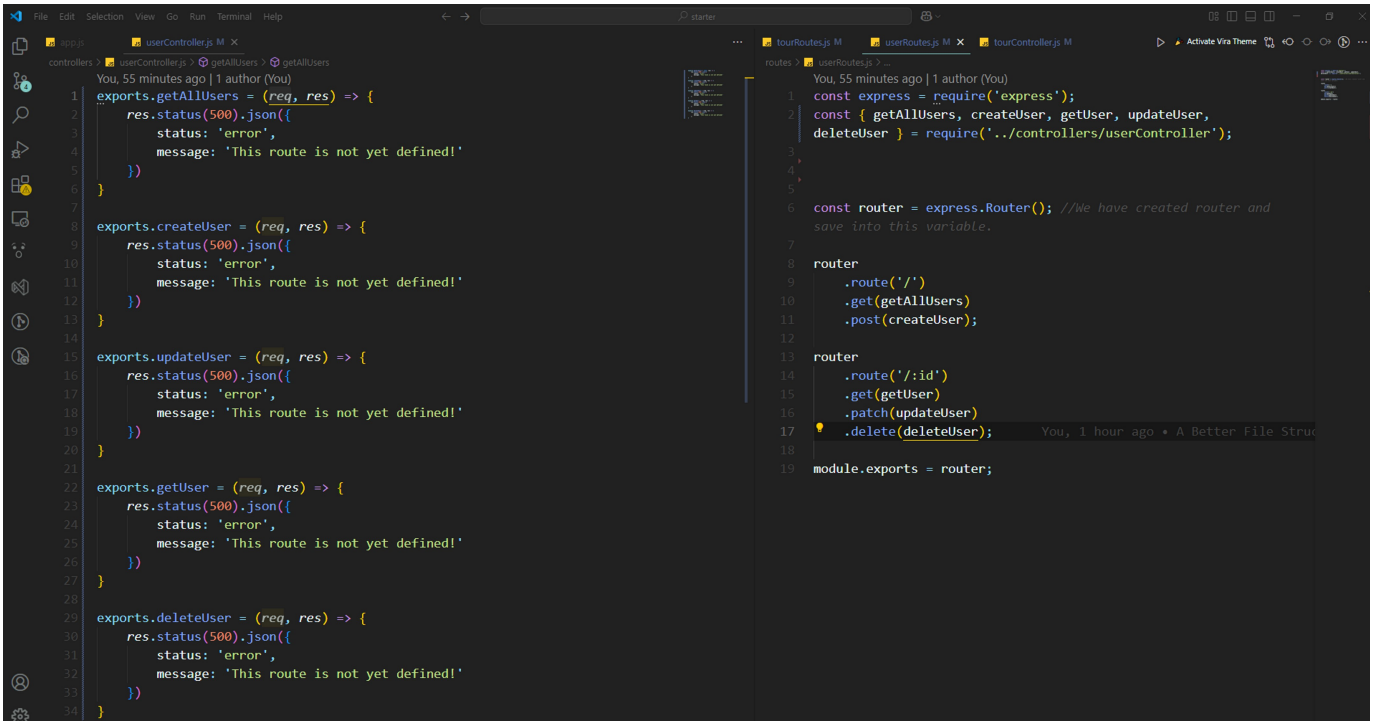
Create a new folder named 'routes'



Cut this code from here

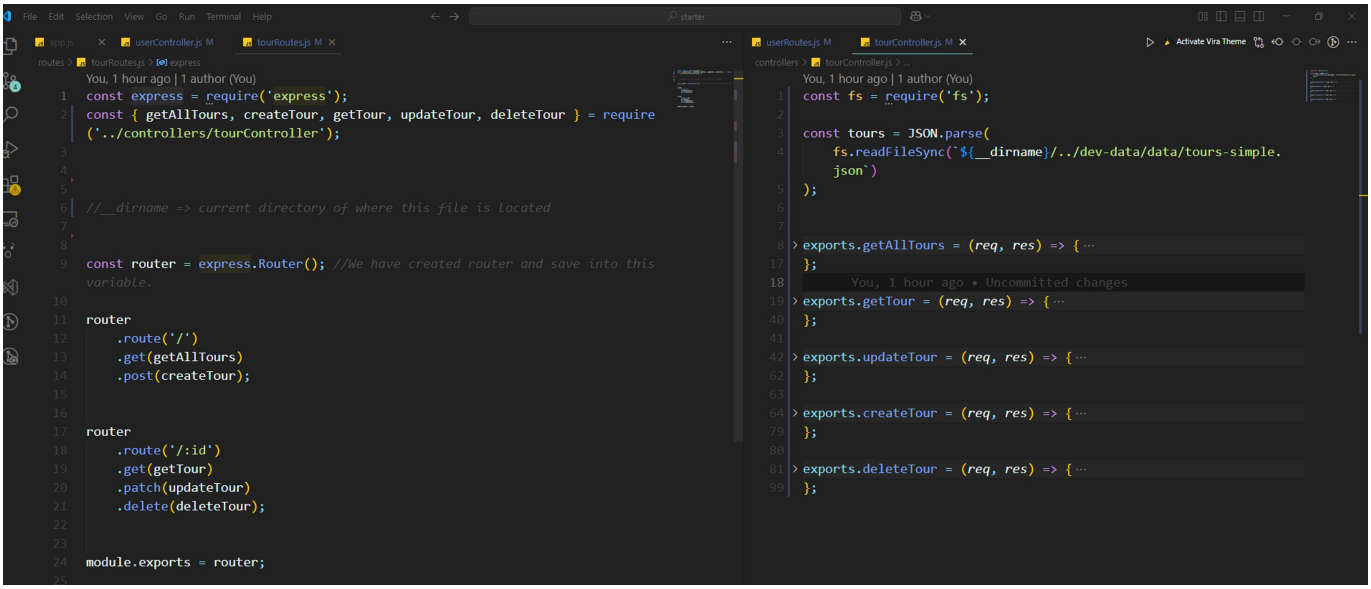


```
//3) Routes


const tourRouter = express.Router(); //We have created router and save into this variable.
const userRouter = express.Router(); //We have created router and save into this variable.

tourRouter      You, 4 minutes ago • Creating and Mounting Multiple Routers …
    .route('/')
    .get(getAllTours)
    .post(createTour);


tourRouter
    .route('/:id')
    .get(getTour)
    .patch(updateTour)
    .delete(deleteTour);


userRouter
    .route('/')
```



```
You, 55 minutes ago | 1 author (You)
exports.getAllUsers = (req, res) => {
    res.status(500).json({
        status: 'error',
        message: 'This route is not yet defined!'
    })
}

exports.createUser = (req, res) => {
    res.status(500).json({
        status: 'error',
        message: 'This route is not yet defined!'
    })
}

exports.updateUser = (req, res) => {
    res.status(500).json({
        status: 'error',
        message: 'This route is not yet defined!'
    })
}

exports.getUser = (req, res) => {
    res.status(500).json({
        status: 'error',
        message: 'This route is not yet defined!'
    })
}

exports.deleteUser = (req, res) => {
    res.status(500).json({
        status: 'error',
        message: 'This route is not yet defined!'
    })
}
```

```
You, 55 minutes ago | 1 author (You)
const express = require('express');
const { getAllUsers, createUser, getUser, updateUser,
deleteUser } = require('../controllers/userController');



const router = express.Router(); //We have created router and
save into this variable.

router
    .route('/')
    .get(getAllUsers)
    .post(createUser);

router
    .route('/:id')
    .get(getUser)
    .patch(updateUser)
    .delete(deleteUser);        You, 1 hour ago • A Better File Struc

module.exports = router;
```



routes > tourRoutes.js > express

You, 1 hour ago | 1 author (You)

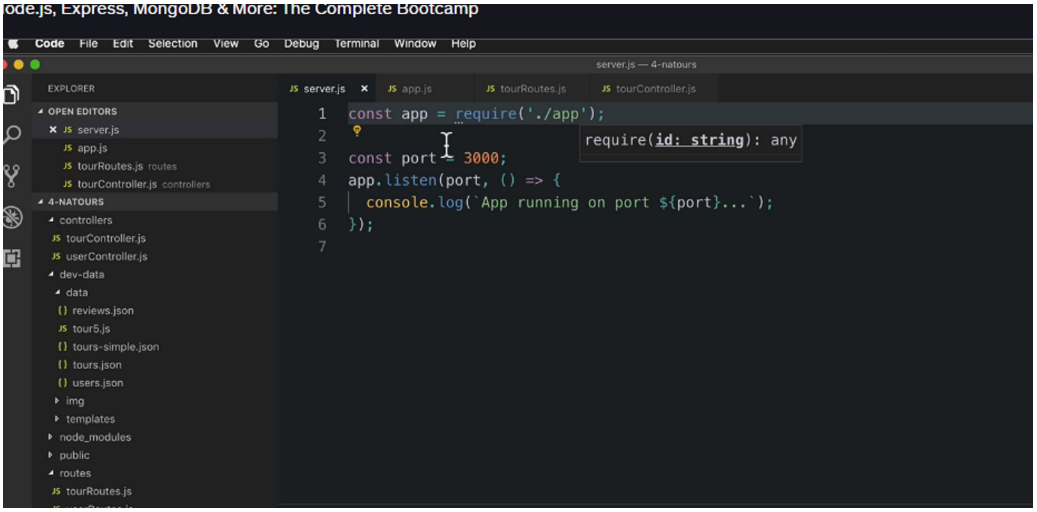
```
const express = require('express');
const { getAllTours, createTour, getTour, updateTour, deleteTour } = require
('../controllers/tourController');



//__dirname => current directory of where this file is located


const router = express.Router(); //We have created router and save into this
variable.

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


router
    .route('/:id')
    .get(getTour)
    .patch(updateTour)
    .delete(deleteTour);


module.exports = router;
```

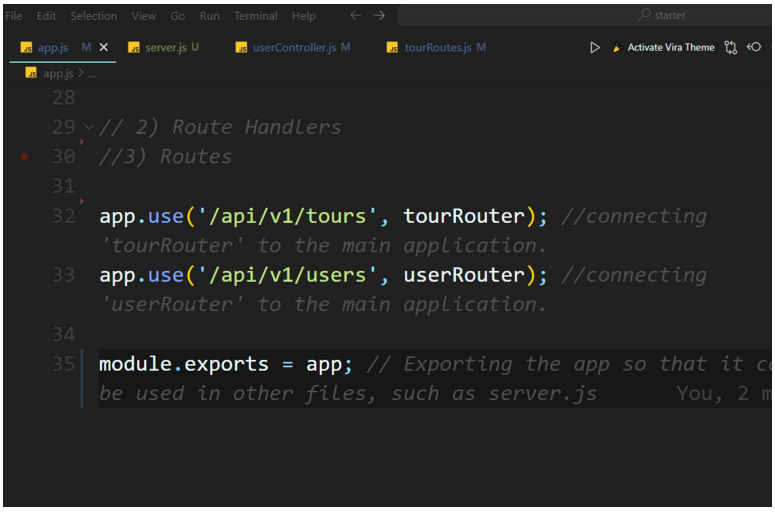
controllers > tourController.js > ...

You, 1 hour ago | 1 author (You)

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

const tours = JSON.parse(
    fs.readFileSync(`${__dirname}/../dev-data/data/tours-simple.
    json`)
);


exports.getAllTours = (req, res) => {···
};
    You, 1 hour ago • Uncommitted changes
exports.getTour = (req, res) => {···
};

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

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

exports.deleteTour = (req, res) => {···
};
```

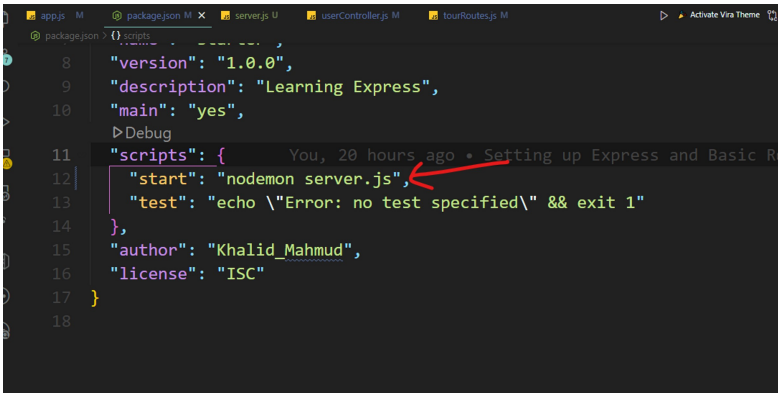
Its better to keep everything related to server in one file and express another.



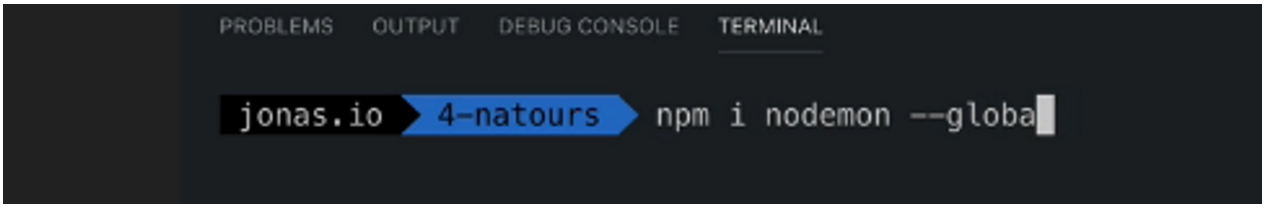


app.js

```
// 2) Route Handlers
//3) Routes

app.use('/api/v1/tours', tourRouter); //connecting
'tourRouter' to the main application.
app.use('/api/v1/users', userRouter); //connecting
'userRouter' to the main application.

module.exports = app; // Exporting the app so that it c
be used in other files, such as server.js
```



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**

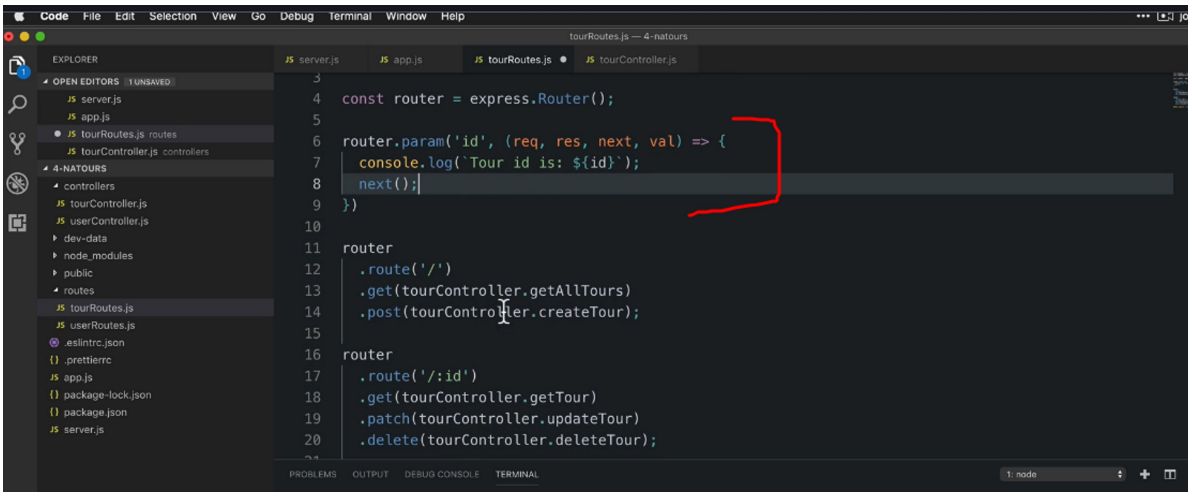
It's super simple. Use the .param() method on your router:

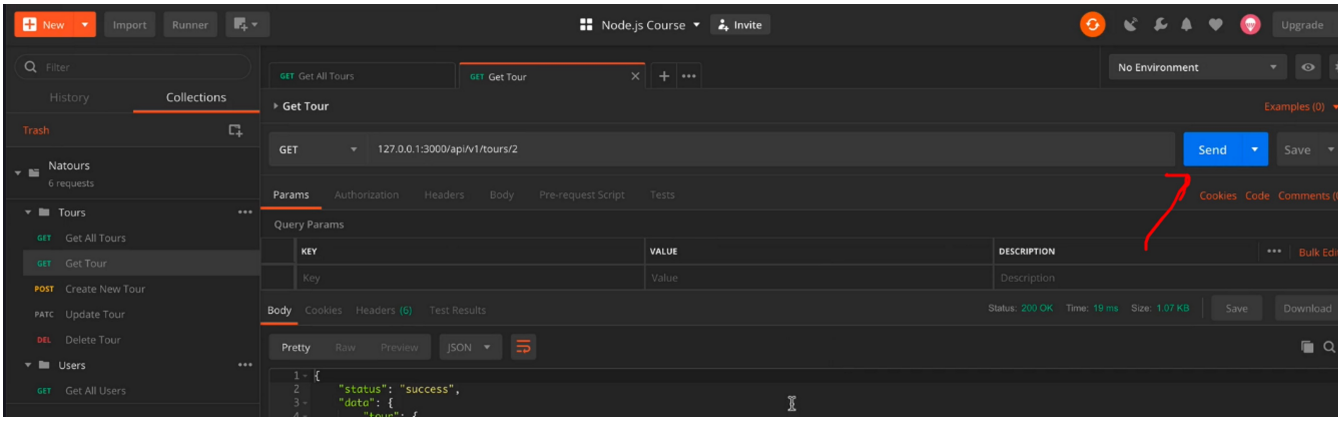
```
router.param('id', (req, res, next, val) => {
  console.log(`Tour ID is: ${val}`);
  next(); // Don't forget to call next!
});
```

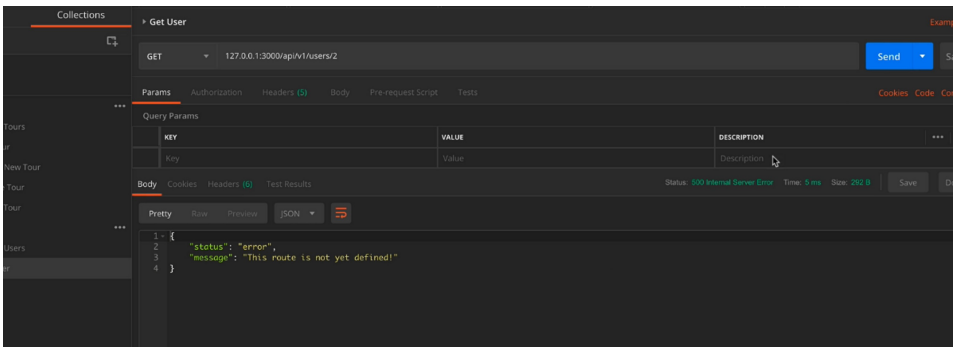
📌 **Key Points:**

- '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! 🚀







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

```
tourRoutes.js — 4-natours
server.js  app.js  tourRoutes.js  tourController.js

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
```

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

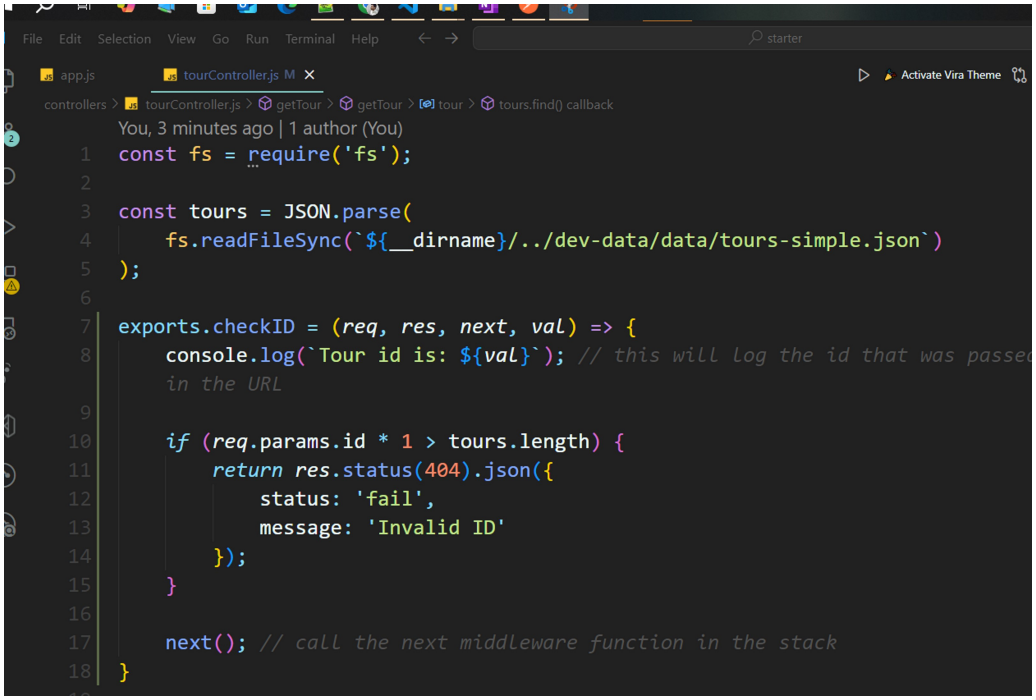
In all the handler function, we check if the id is valid. Or not

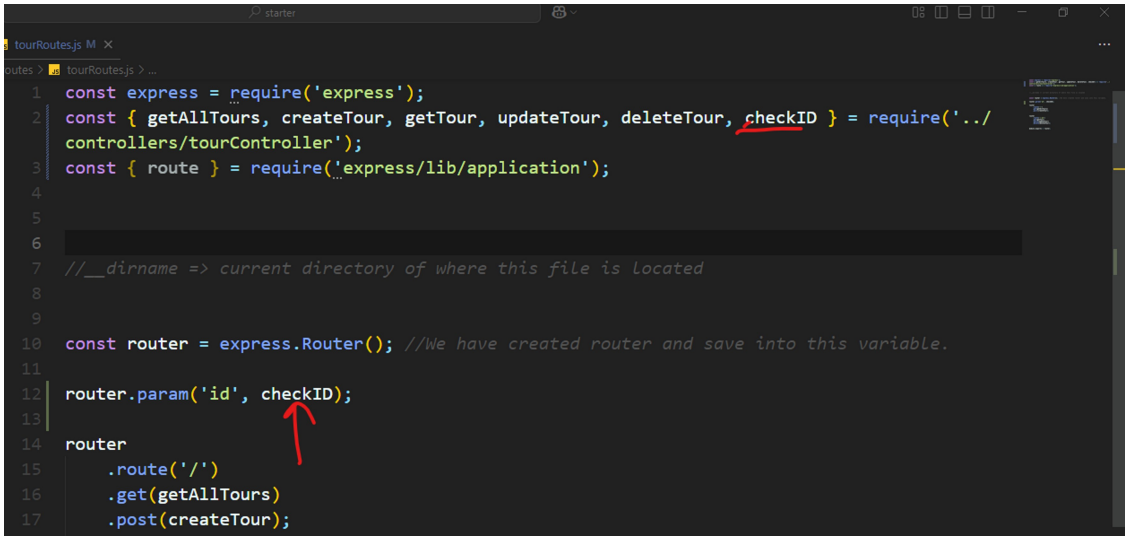
```
Go  Debug  Terminal  Window  Help
tourController.js — 4-natours
server.js  app.js  tourRoutes.js  tourController.js

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

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  1: node

[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
[nodemon] restarting due to changes...
```

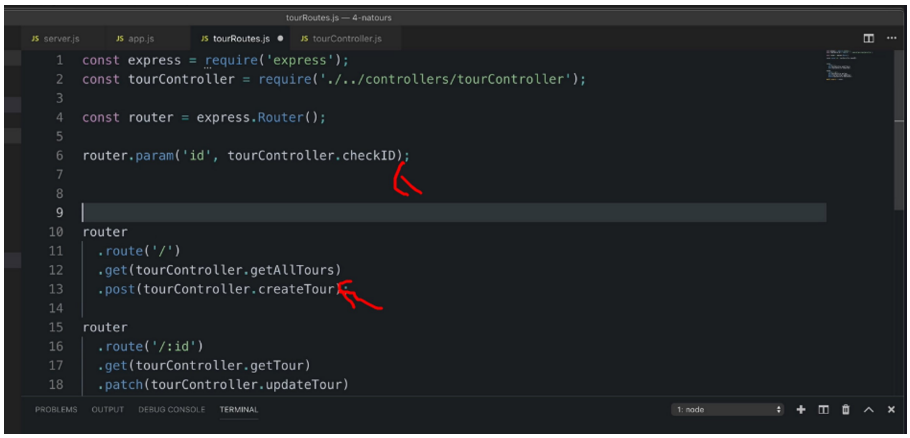
```
tourController.js — 4-natours
server.js  app.js  tourRoutes.js  tourController.js

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 };
79
```

```
79
80 exports.deleteTour = (req, res) => {
81   if (req.params.id * 1 > tours.length) {
82     return res.status(404).json({
83       status: 'fail',
84       message: 'Invalid ID'
85     });
86   }
87
88   res.status(204).json({
89     status: 'success',
90     data: null
91   });
92 };
93
```

We all know it is not good to repeat code.
Now we can apply the concept 'param middleware' and check id.







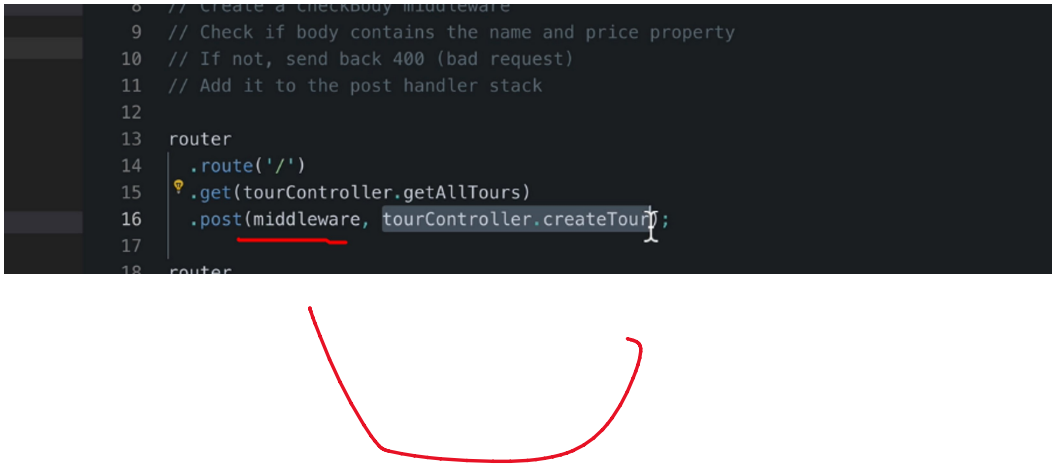
🔗 **Chaining Multiple Middleware Functions in Express**
So far, we've been using **only one middleware function** per route. For example, in a POST request, we might only use a createTour handler:

app.post('/api/v1/tours', createTour);
But what if we want to run **multiple middleware functions** before reaching the final route handler?

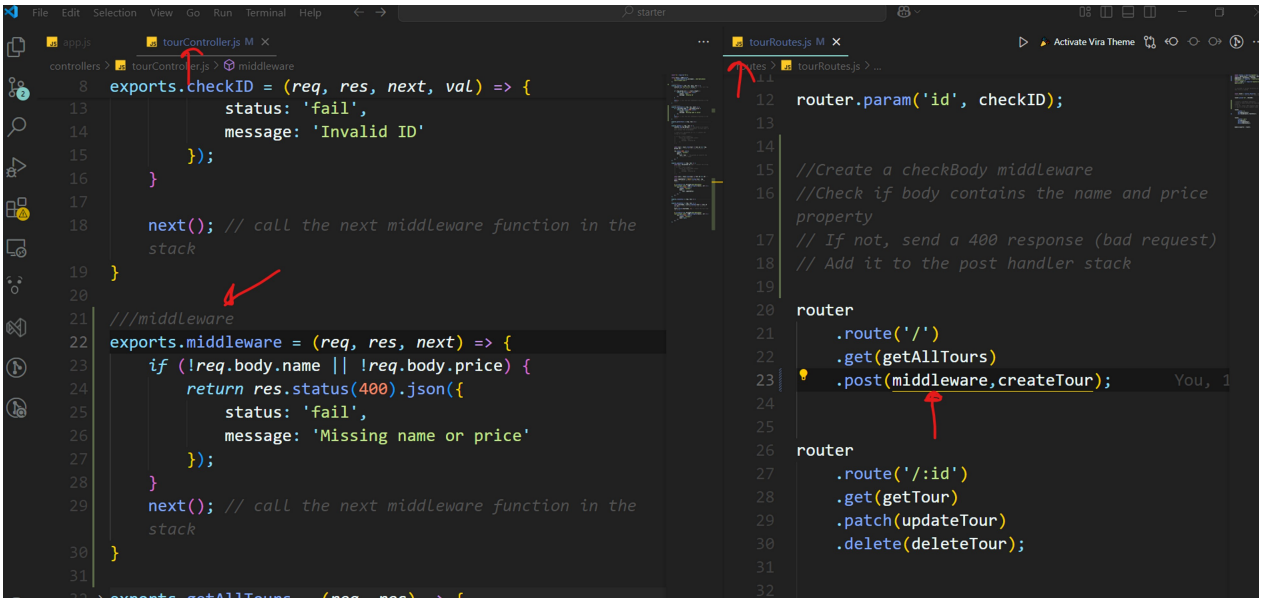
🤔 **Why Chain Middleware?**
Sometimes, we want to **pre-process** or **validate** data before running the main logic. For example:

- 🔍 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).

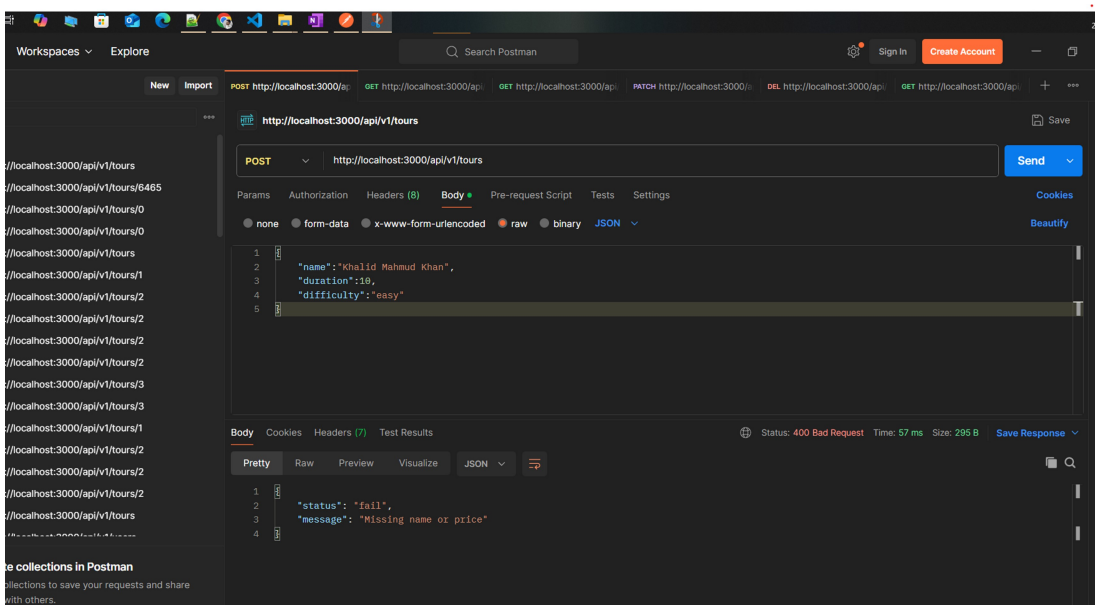


Chain



```
exports.checkID = (req, res, next, val) => {
            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
}
```

```
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);

router
    .route('/:id')
    .get(getTour)
    .patch(updateTour)
    .delete(deleteTour);
```



🌐 **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.

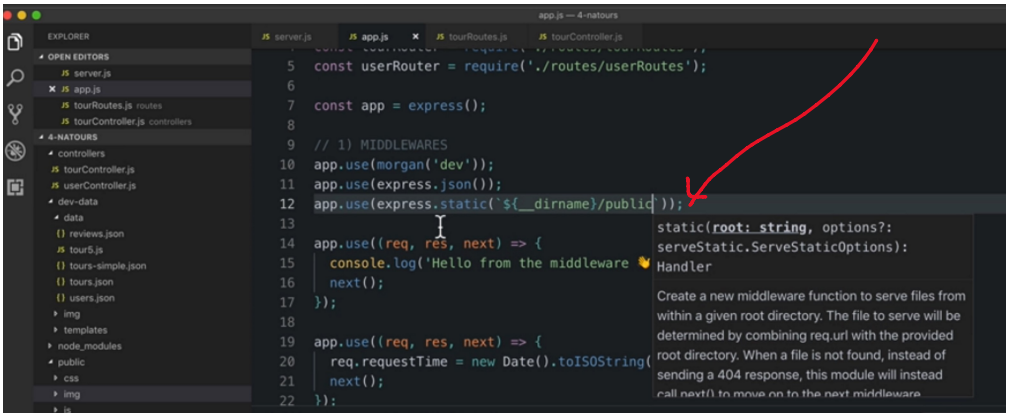
However, at this point, we can't access them from the browser—there's no route to reach them yet.

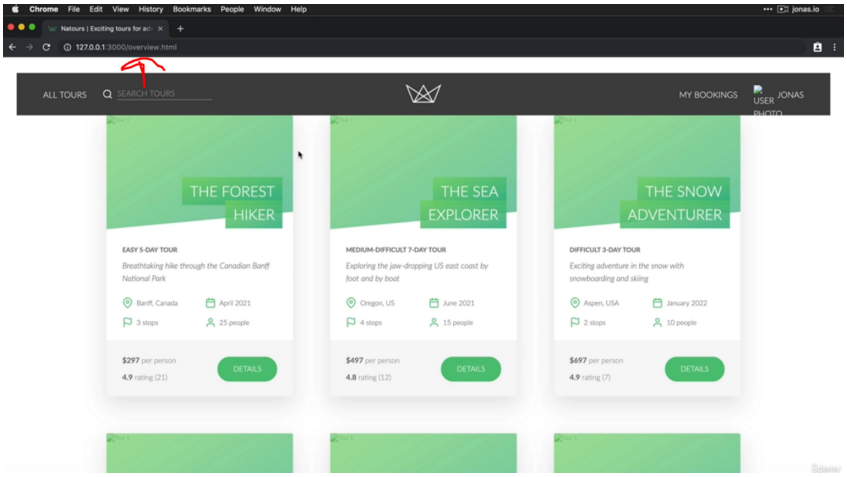
For example:

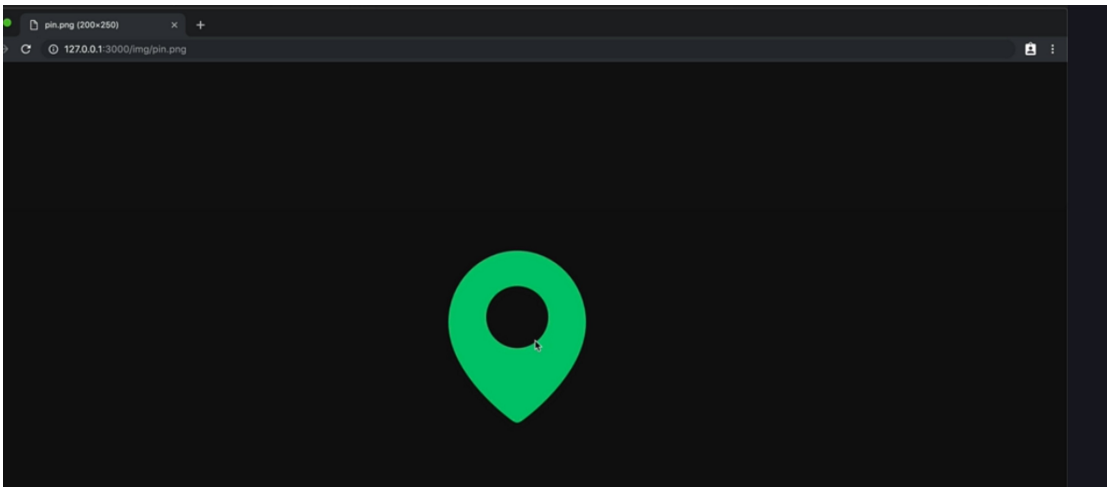
- public/overview.html ✅ exists
- But http://localhost:3000/overview.html ❌ doesn't work yet

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! 🚀







🌱 **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.

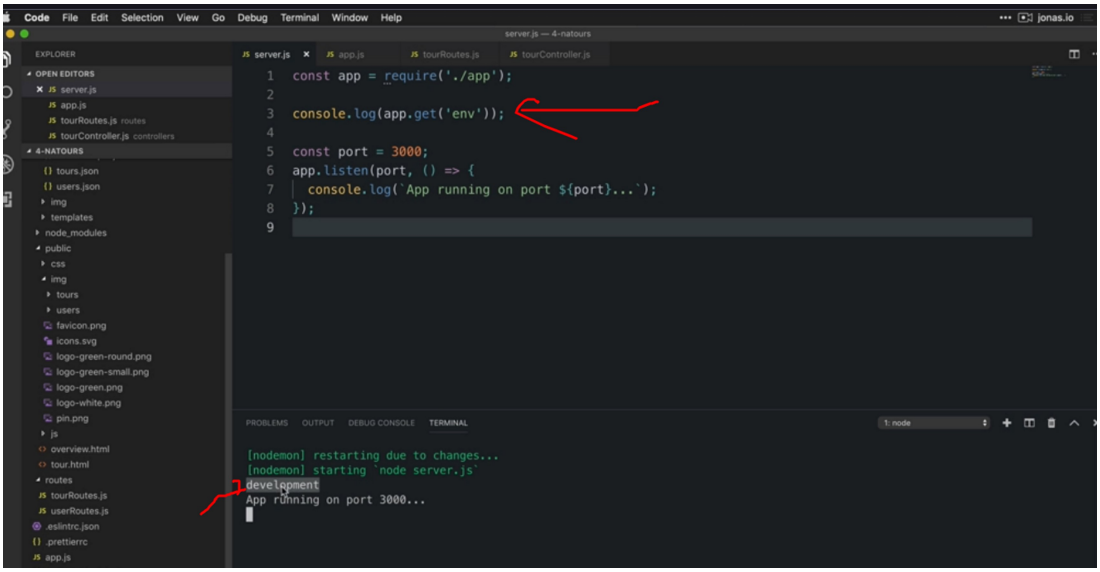
💡 **Why Use Them?**
Apps can run in **different environments** like:

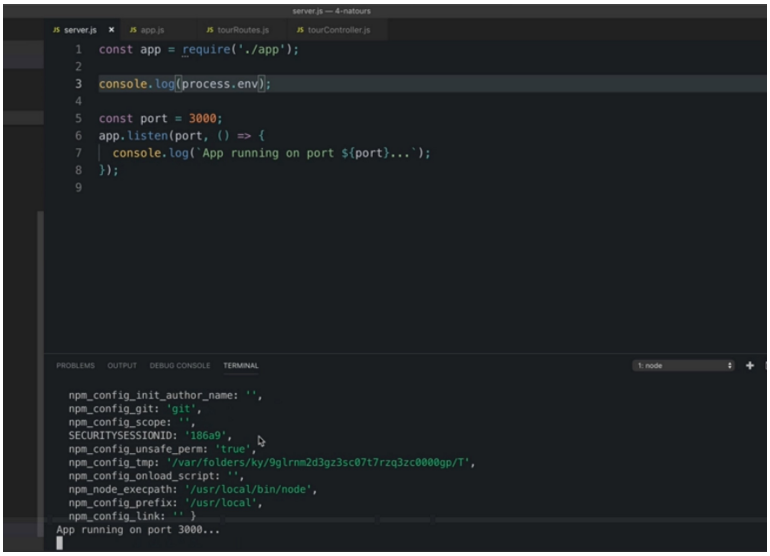
- 🧪 development (what you use while coding)
- 🚀 production (what real users see)
- 🛠️ staging, test, etc. (used in big teams or complex apps)

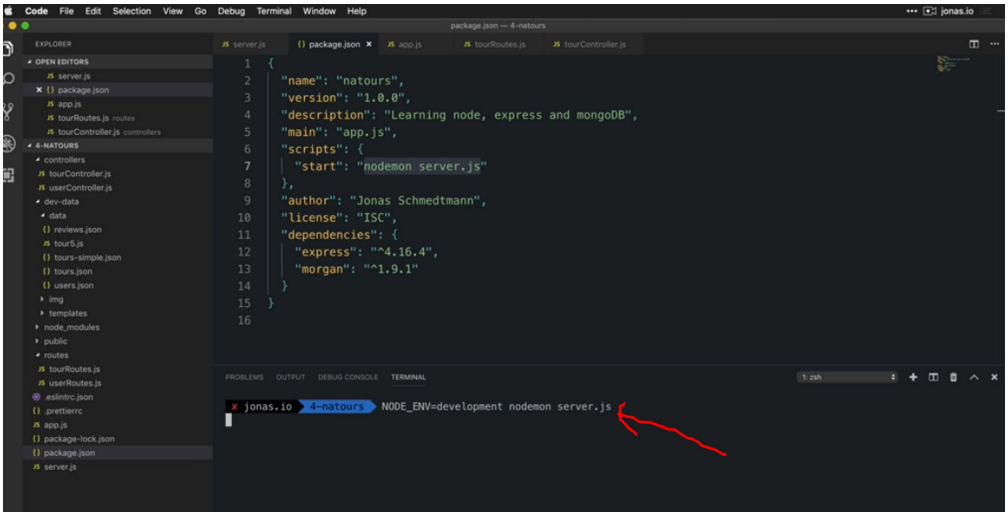
Depending on the environment, you might want:

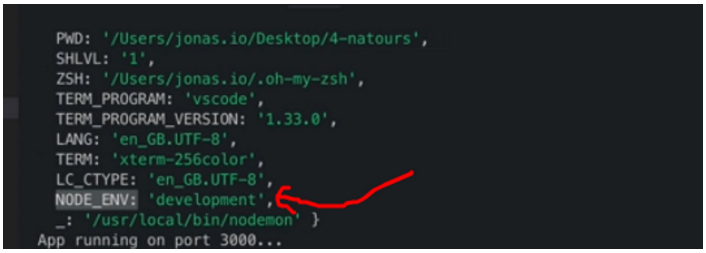
- Different **databases**
- Debugging turned **on/off**
- Logging behavior to **change**
- APIs to point to **different URLs**

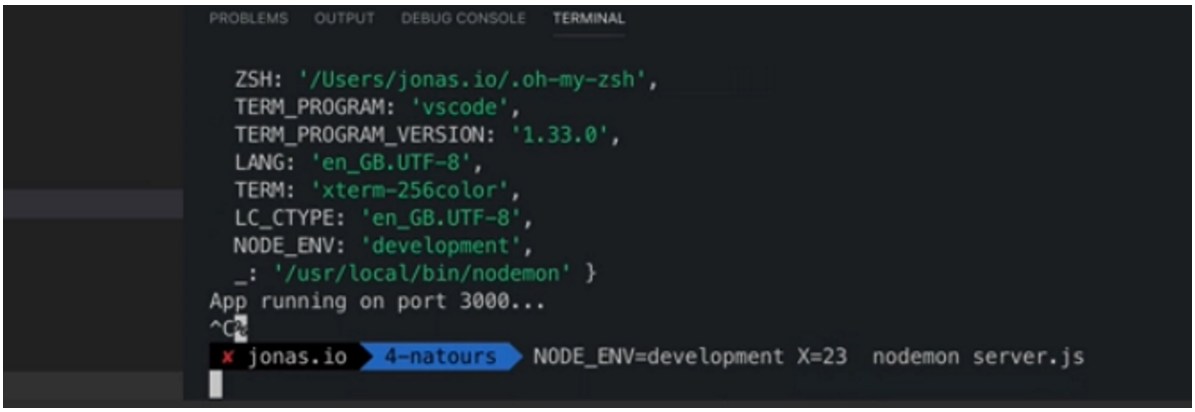
Using environment variables helps you **switch between those easily** — without rewriting your code. Neat, right?

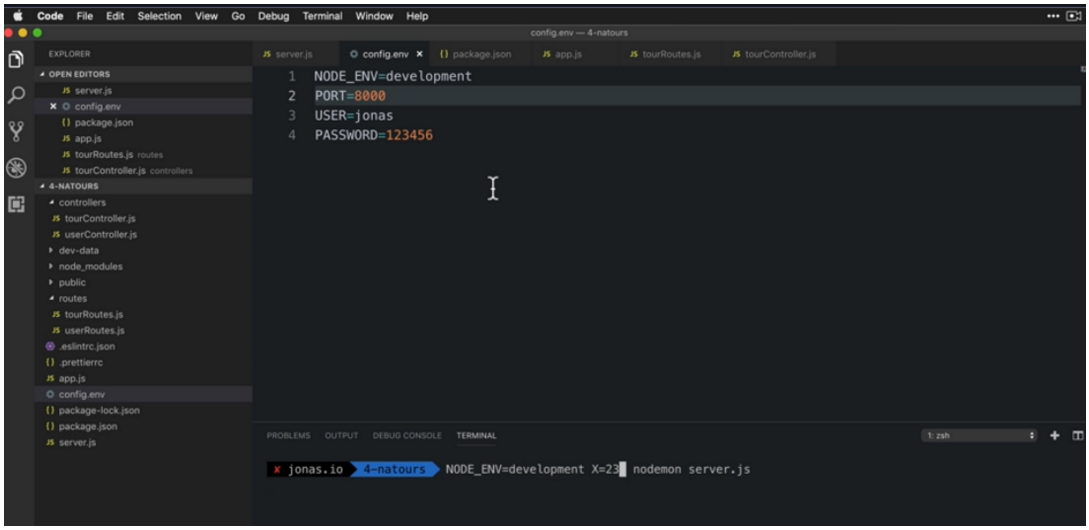




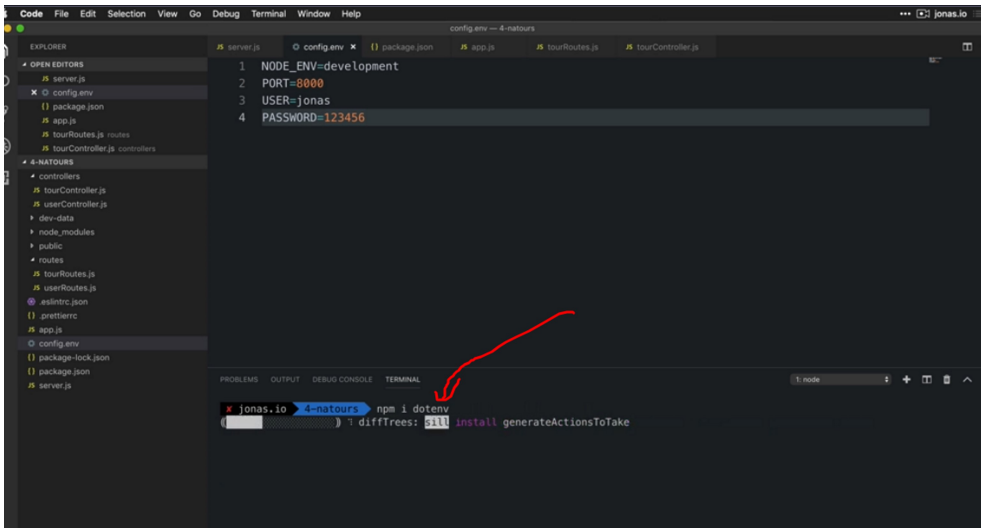

Code File Edit Selection View Go Debug Terminal Window Help
package.json — 4-natours
EXPLORER
OPEN EDITORS
server.js
package.json
app.js
tourRoutes.js routes
tourController.js controllers
4-NATOURS
controllers
tourController.js
userController.js
dev-data
data
reviews.json
tour5.js
tours-simple.json
tours.json
users.json
img
templates
node_modules
public
routes
tourRoutes.js
userRoutes.js
.eslintrc.json
.prettierrc
app.js
package-lock.json
package.json
server.js
server.js
package.json
app.js
tourRoutes.js
tourController.js
{
"name": "natours",
"version": "1.0.0",
"description": "Learning node, express and mongoDB",
"main": "app.js",
"scripts": {
"start": "nodemon server.js"
},
"author": "Jonas Schmedtmann",
"license": "ISC",
"dependencies": {
"express": "^4.16.4",
"morgan": "^1.9.1"
}
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
jonas.io 4-natours NODE_ENV=development nodemon server.js

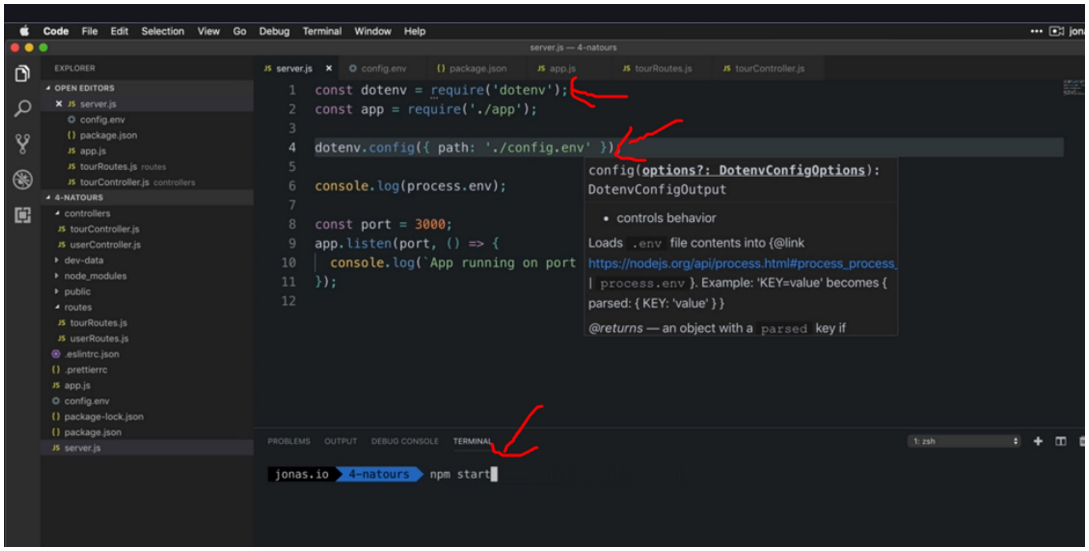

PWD: '/Users/jonas.io/Desktop/4-natours',
SHLVL: '1',
ZSH: '/Users/jonas.io/.oh-my-zsh',
TERM_PROGRAM: 'vscode',
TERM_PROGRAM_VERSION: '1.33.0',
LANG: 'en_GB.UTF-8',
TERM: 'xterm-256color',
LC_CTYPE: 'en_GB.UTF-8',
NODE_ENV: 'development',
_: '/usr/local/bin/nodemon' }
App running on port 3000...

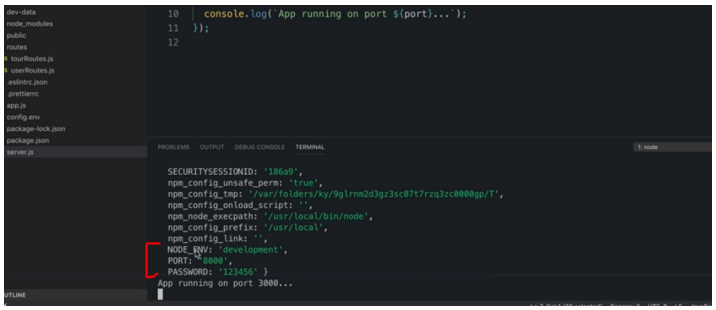

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ZSH: '/Users/jonas.io/.oh-my-zsh',
TERM_PROGRAM: 'vscode',
TERM_PROGRAM_VERSION: '1.33.0',
LANG: 'en_GB.UTF-8',
TERM: 'xterm-256color',
LC_CTYPE: 'en_GB.UTF-8',
NODE_ENV: 'development',
_: '/usr/local/bin/nodemon' }
App running on port 3000...
^C
jonas.io 4-natours NODE_ENV=development X=23 nodemon server.js



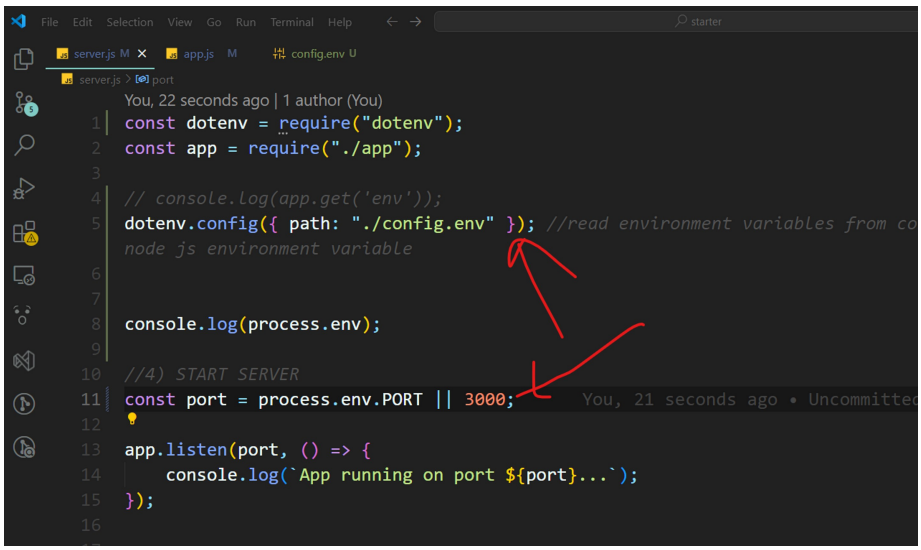
Now how do we actually connect this (.env) file with our node application?

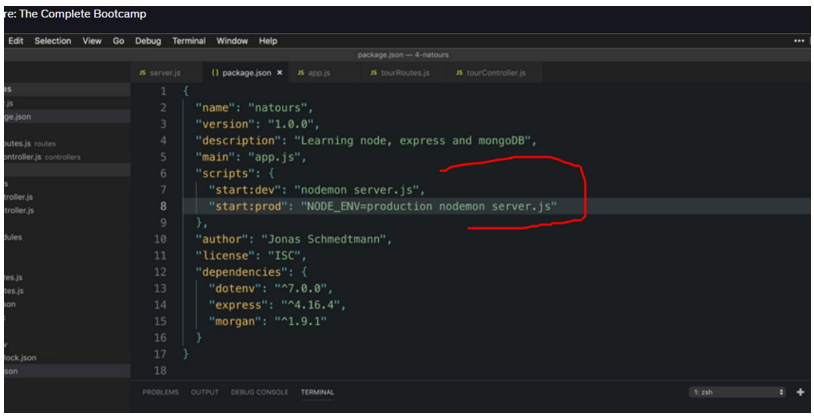


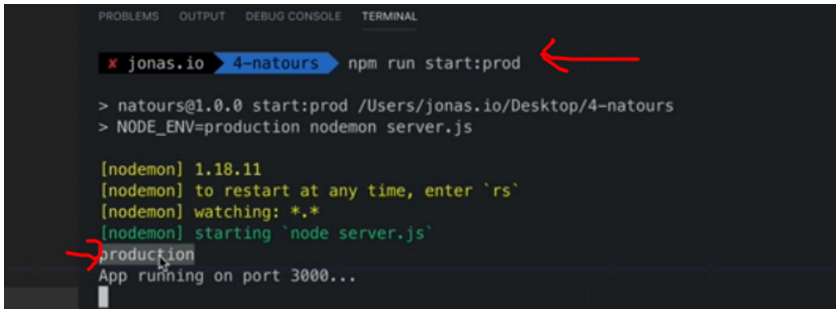



console.log(`App running on port ${port}...`);
});
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node
SECURITYSESSIONID: '186a9',
npm_config_unsafe_perm: 'true',
npm_config_tmp: '/var/folders/ky/9glrnm2d3gz3sc07t7rzq3zc0000gp/T',
npm_config_onload_script: '',
npm_node_execpath: '/usr/local/bin/node',
npm_config_prefix: '/usr/local',
npm_config_link: '',
NODE_ENV: 'development',
PORT: '8000',
PASSWORD: '123456' }
App running on port 3000...


// 1) MIDDLEWARES
if (process.env.NODE_ENV === 'development') {
app.use(morgan('dev')); // Morgan is a HTTP request logger middleware
process of logging requests to your application, making it easier to
performance.
}
You, 4 hours ago • Using 3rd-Party Middleware


File Edit Selection View Go Run Terminal Help
starter
server.js M
app.js M
config.env U
server.js > port
You, 22 seconds ago | 1 author (You)
const dotenv = require("dotenv");
const app = require("./app");
// console.log(app.get('env'));
dotenv.config({ path: "./config.env" }); //read environment variables from co
node js environment variable
console.log(process.env);
//4) START SERVER
const port = process.env.PORT || 3000;
You, 21 seconds ago • Uncommitte
app.listen(port, () => {
console.log(`App running on port ${port}...`);
});


re: The Complete Bootcamp
Edit Selection View Go Debug Terminal Window Help
package.json — 4-natours
server.js
package.json
app.js
tourRoutes.js
tourController.js
{
"name": "natours",
"version": "1.0.0",
"description": "Learning node, express and mongoDB",
"main": "app.js",
"scripts": {
"start:dev": "nodemon server.js",
"start:prod": "NODE_ENV=production nodemon server.js"
},
"author": "Jonas Schmedtmann",
"license": "ISC",
"dependencies": {
"dotenv": "^7.0.0",
"express": "^4.16.4",
"morgan": "^1.9.1"
}
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: zsh



🎯 **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:

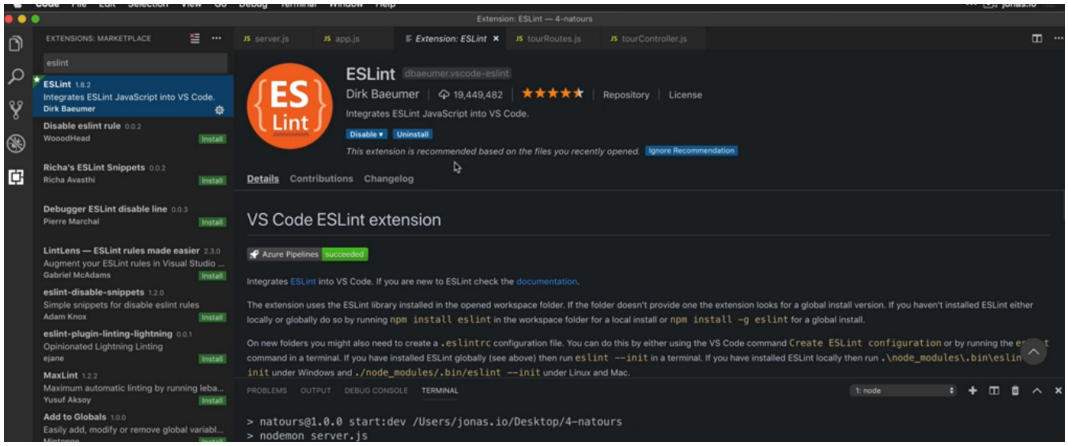
```
js
CopyEdit
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

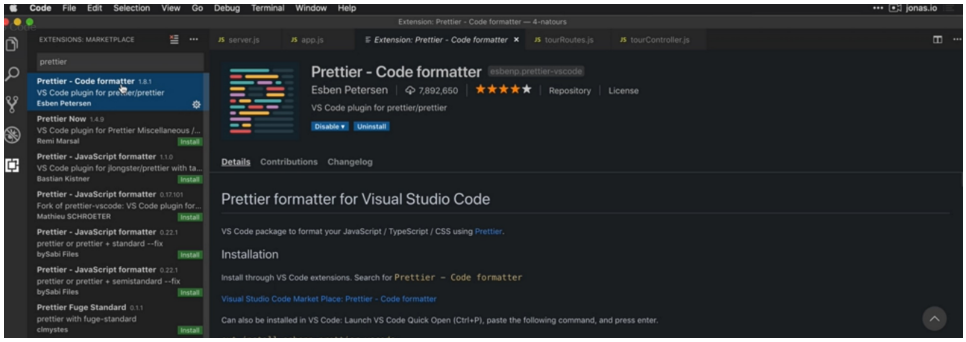
✅ **Summary Time:**

🧩 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>