<https://www.youtube.com/watch?v=O3BUHwfHf84>

To get started, I have created an empty folder on my desktop called "M Crash Course," and then I just open it up in VS Code. Now we're going to need two different folders: one for the front end and one for the back end. In the first section of this course, we're going to focus on the back end part, where we build an API. Once we build it, we'll go ahead and build the front end part, but first let's start with this one.

I will open a new terminal using the shortcut `Ctrl + J`, and from here, I’d like to run `npm init -y`. Make sure you're running this under the root, so you don't accidentally `cd` into the back end and run `npm init -y` there. This will create a `package.json` file in the root directory, which will help us deploy this application easily. We’ll put all of our scripts here to build and start the application, which will make sense once we get into the deployment part.

Next, we need to install a few packages for the back end. Whenever you install a package for the back end, run it in the root. Let me clear the console for clarity and run:

```bash

npm install express mongoose dotenv

```

This command will install these packages and add them to the dependencies in the `package.json` file. We’ll be using Express as our web framework, which will make building the API and handling routing easy. Mongoose will help us interact with our MongoDB database, and dotenv will allow us to manage environment variables.

After running the installation, we’ll have a `node\_modules` folder and a `package-lock.json` file created. Now, we can create a file called `server.js`. You can call this file anything—some people use `index.js`, `main.js`, or `app.js`. I prefer to use `server.js` as the entry point for our API.

Let’s import Express and create an Express app. Instead of using the traditional `require` syntax, we can use a more modern ES modules syntax. First, we need to set `"type": "module"` in the `package.json`. After that, we can write:

```javascript

import express from 'express';

const app = express();

const PORT = 5000;

app.listen(PORT, () => {

console.log(`Server started at http://localhost:${PORT}`);

});

```

To run this file, use:

```bash

node backend/server.js

```

You should see the message indicating that the server has started. To simplify running the server, we can add a script in `package.json`:

```json

"scripts": {

"dev": "node backend/server.js"

}

```

Now you can run `npm run dev`. If you make changes to `server.js`, you'll need to restart the server manually. To avoid this inconvenience, we can install `nodemon`:

```bash

npm install nodemon --save-dev

```

After installing, update the script in `package.json` to use `nodemon`:

```json

"scripts": {

"dev": "nodemon backend/server.js"

}

```

Now, with `nodemon`, any changes you make will automatically restart the server.

Next, let's create a route. We’ll set up a `GET` method for the root route:

```javascript

app.get('/', (req, res) => {

res.send('Server is ready');

});

```

You can test this by visiting `http://localhost:5000`. If you change the response message, `nodemon` will restart the server automatically, and you’ll see the updated response.

Now, the first thing we’d like to create is a `products` route to return all products. We’ll set it up like this:

```javascript

app.get('/products', (req, res) => {

res.send('Here are all the products');

});

```

With these steps, you now have a basic back-end setup ready for further development!

To create our MongoDB database, let's start by going to mongodb.com. If you don’t have an account, you can create one for free without needing a credit card. After signing in, you’ll be taken to the dashboard.

From there, select “Create a New Project” and name it something like “M Course.” Once the project is created, proceed to create a cluster. Choose the free tier and click “Create the Deployment.” This will take you to a new page where you can copy the password. Make sure to save it, as we’ll need it soon.

Next, you’ll need to create a user for the database. Click to create the user, then select “Connect to this Database” and close that dialog. You should now see an option to get the connection string. Click on that and copy the connection string.

Open a file called `.env` and paste the connection string there. You’ll need to replace the placeholder for the password with the actual password you copied earlier. Set the database name to “products” in the connection string, placing it before the question mark. Save this file.

Additionally, go to “Network Access” in your MongoDB dashboard. By default, your IP address may not be allowed to connect to the database. You can enable access from anywhere for development purposes by confirming that option.

Now that the access is set up, go to the database section. At this point, we haven’t created any collections yet, so it should be empty. To connect to the database in our application, we’ll use the connection string we saved in the `.env` file.

In `server.js`, let’s log the connection string to ensure it works:

```javascript

console.log(process.env.MONGODB\_URI);

```

To access environment variables, make sure to import the `dotenv` package and call its configuration function:

```javascript

import dotenv from 'dotenv';

dotenv.config();

```

Once this is set up, you should be able to see the contents of your `.env` file in the console.

Next, create a new folder called `config` and inside it, create a file called `db.js`. In this file, we’ll create an asynchronous function to connect to our MongoDB database:

```javascript

import mongoose from 'mongoose';

export const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGODB\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log("MongoDB connected");

} catch (error) {

console.error(error.message);

process.exit(1);

}

};

```

Now, call this `connectDB` function in `server.js` right after you set up your Express app:

```javascript

import { connectDB } from './config/db.js';

const app = express();

connectDB();

```

If everything is set up correctly, you should see a message in the console indicating that the server has started and that MongoDB has connected successfully.

Now that we’ve connected to our MongoDB database, it’s time to create some collections, specifically for products. In our application, we’ll be storing product information, so let’s clarify the difference between tables and collections.

MongoDB is a NoSQL database, unlike SQL databases such as MySQL or PostgreSQL, which store data in tables organized into rows and columns. In MongoDB, we have collections, and inside each collection, we store documents.

For example, let’s say we create a collection called “products.” Within this collection, we can have multiple documents representing different products. We could have documents for a smartwatch, earbuds, and shoes. Each of these items is treated as a document within the “products” collection.

Additionally, you might want to have another collection for users, where each user’s information is stored as a document. For instance, we could have documents for users like John, Jane, and Bob.

This distinction highlights how data is organized differently in SQL versus NoSQL databases. In SQL, data is structured in tables, whereas in MongoDB, we have collections that can contain multiple documents.

could have multiple different documents. Let’s just say we’ll have one, two, and three different documents. One would be a smartwatch, and we could have earbuds and some shoes. Each of these is called a document. So what we refer to as products will be the collection.

You might also want to have another collection for users. For each user, we would have a document that stores their information. For example, we could have documents for John, Jane, Bob, etc.

This clarifies the difference between SQL databases and NoSQL databases. SQL databases store data in tables with rows and columns, while in MongoDB, we have collections, and each collection can contain multiple documents.

To get started, we’ll be creating our “products” collection, allowing us to store product details effectively.

products collection, and to be able to do so, we're going to create a model using Mongoose, the package we have installed.

First, create a folder called models and give it a name. We can call this the product model, so let's name it product.model.js. You can use any convention; some might prefer just product.js, but I’ll use product.model.js.

Now, let's create our product model. I will import Mongoose from Mongoose. First, we need to create a schema, so I'll say productSchema and use Mongoose's schema function. Each product is going to have a name, a price, and an image.

For the name, we’ll open an object and set the type to string and required to true. This ensures that if a user wants to create a product, they must provide the name; otherwise, it will throw an error. Next, we’ll add a price field, where the type will be number and also required. Lastly, we’ll have the image, which will again be of type string and required.

Additionally, we can add an option for timestamps, which will automatically create createdAt and updatedAt fields on each document.

Now that we have a schema, we’ll create our product model. I’ll say const Product equals Mongoose.model and pass in product and productSchema. Finally, we’ll export this Product model, as we’ll be using it in different files later.

This tells Mongoose to create a model or collection called Product based on the schema we defined, which includes the fields we provided.

You might wonder why we use "Product" instead of "Products." Mongoose will handle this and convert it to be plural, creating a collection named "products" from the singular model name.

This file is now our product model. Next, we’d like to create product documents, so let's create an endpoint for this. If we visit...

Here is the text without timestamps or formatting:

MERN is an acronym that uses the first letter of four complementary technologies.  
M is for MongoDB,  
E is for Express JS,  
R is for React, and  
N is for NodeJS.

So if the MERN stack is full stack, that leads us to ask what is full stack and why is the MERN stack considered to be full stack?

A full stack application means it requires code that runs on the server and code that runs in the browser.  
The code that runs on the server is referred to as the back end and the code that runs in the browser is referred to as the front end.

The front end and the back end are typically two completely separate code repositories.  
In a large enterprise full stack project, there may be a team of developers that work on the front end and another separate team of developers that work on the back end.  
As a full stack developer, you should be able to work on both the front end and the back end if needed.

The back end for the MERN stack is a REST API.  
A REST API, also known as a RESTful API, is an interface that two computer systems use to exchange information securely over the internet.

The back end will receive requests from the front end.  
Those requests can be classified as CRUD operations.  
CRUD is another four-letter acronym like MERN.  
The letters of CRUD stand for Create, Read, Update, and Delete.

These terms also indicate which type of HTTP request methods will be used in the application.  
For example, POST relates to Create.  
GET relates to Read.  
PATCH and PUT requests relate to Update, and Delete has an exact match.

This information is not only an intro but also somewhat of a review because this is not a beginner's tutorial series.  
I already have many beginners courses available and I recommend those courses as prerequisites to this series.

Specifically, I suggest my Node JS course before the first lessons of this MERN series where we're going to build the back end REST API.  
I also suggest completing my React course, my Redux course, and my React login series playlist before the lessons in this course where we build the front and React app.

We're going to build a Tech Notes app for a small computer repair shop, specifically Dan D's Repair Shop.  
So our stakeholder is Dan D, or full name Dan Davidson, but he goes by Dan D.  
And this isn't the type of project where I'm going to give you a tour of the full project before we begin.  
We're more taking a real-world approach here for this project and to do that we start by gathering some user stories from our stakeholder Dan D.  
I've got VS Code open.

Here is the text without timestamps or formatting:

I've got VS Code open and I've already done a preliminary interview with Dan D, our stakeholder for the Tech Notes project, and I've come up with 20 user stories that Dan wants.

Now, if you've never done an interview for user stories, you need to think that the person you're interviewing with will probably describe what they want as a user, but they're not that technical, so it can be difficult to get those technical requirements out of that interview, and of course, that does take practice.

So I definitely wanted to start the project with this, just so you can see the highlights that I've organized for this, and we can derive our technical requirements from the descriptions that I have received from Dan D.

The main goal for this application, the first thing Dan told me, is he wants to replace his current sticky note system. He has a small computer repair shop, and right now, they use yellow sticky notes to write the problem, and they slap it on the side of the computer or whatever other technology, an iPhone, whatever somebody brings in, and it goes on their shelf, and that's their whole system.

And he knows they need to get a better system than that, so that's what this Tech Notes program's main goal is, is to have a local database, something they can refer to, and everybody knows what everybody else is working on, and Dan can manage the whole thing.

The second thing he wants is just a public-facing web page with basic contact info. Again, he's from a small town, small computer shop, he doesn't need anything extra for his website, he just basically wants it to be a business card, and we're okay with that because we want to focus on building this back-end application for Dan.

Okay, he wants to add an employee login to the notes app because he doesn't want just anybody to be able to access it, he just wants his employees to access it.

And then he wants to provide a welcome page after the login, thinks that would just be nice to, of course, show the username, maybe the current day time, and of course, what they have to work on, or what's available to them, maybe their current level of administration, whether they're an employee, a manager, Dan has at least one manager, and then he's the admin, of course.

He wants easy navigation, he wants to display the current user and assigned role at any time throughout the application, and then he wants to provide a log out option, of course.

He wants to require users to log out or log in at least once per week, so that is a big requirement that we need to think about when it comes to authorization. Again, it's not a public-facing site, it's not a financial site, it doesn't need ultimate security, but he wants some security, but he doesn't want his employees to have to log in all the time either, so we need to think about that with the authorization.

He wants to provide a way to remove employee access as soon as possible, possibly he fires someone and needs to remove that access so they do not disrupt the notes application.

Notes are assigned to specific employees, so everybody has their own responsibility. Notes have a ticket number, title, note body, created and updated dates. Notes are either open or completed.

Notes can be employees, managers, or admins. Notes can be deleted by managers and admins only, so that's a consideration when we're applying the roles and permissions of the application.

Anyone can create a note because a customer may come in to check in, and Dan doesn't know who may be at the counter, but they all need to be able to check in and create a note for a customer.

Employees can only view and edit their assigned notes, however, managers and admins can view, edit, and delete all notes, so that's another thing with the roles and permissions.

Only managers and admins can access the user settings, and only managers and admins can create new users.

And then desktop mode is the most important because that's where they'll be using it, but it should also be available in mobile.

Now, notice this is a markdown file that I created a checklist in. If you're not that familiar with markdown files, they end with a dot md, and that's also what the readme files for GitHub are usually created with.

Now, in Visual Studio Code, I can preview this file by pressing Ctrl Shift and the letter V, and we can see this markdown file as it would appear possibly on GitHub or somewhere else.

And the nice thing is we can also use it as a check. Well, it's not checking right now, let me go back and if we put an x in here, it would check.

But then also, we can preview this a different way with Control K and then just press the letter V.

And now let's look at this, and we have this on the right and the edit on the

Here is the text without timestamps or formatting:

Yes, it does check and we see the x over here, so you can work with these markdown files within Visual Studio Code and they're sometimes very useful for checklists, so I just wanted to highlight that fact as well.

But those are our user stories and we'll create our technical requirements from these descriptions.

Go ahead and create an empty folder for your back-end code for this MERN stack project for Tech Notes.

And then, if you do not already have it, I sure hope you do, but if you don't, install Node.js from (link unavailable) and you can download that right on the home page.

They have currently, as of the making of this video, 16.16 LTS, which stands for Long Term Support, and is recommended for most users. Of course, get whatever the most current is, that should give you npm and Node.

I already said I hope you do, because again, this is not for beginners. You might get through today's lesson, but I don't feel like you will be comfortable and get that far if you're an absolute beginner.

Go back to that Node.js course if you are. Now, after you've installed Node and npm, or maybe you already have them, you can control backtick inside of Visual Studio Code and it should open up a terminal window.

You can check your versions as well by typing node -v. It says I have 16.16. I can type npm -v as well, because that also installs and we need that, and it's currently 8.15.1.

Here is the text without timestamps or formatting:

Today we're going to build the back end, or at least start building the back end REST API for this MERN Tech Notes project, and that will use Node, Express, and so three of the four technologies in the stack.

So what can we take away from these user stories? Already, just the basic general thing that we would need for our server is we need to be able to create notes, and as we mentioned in here, view, which would be read as related to CRUD, edit, and that would be update, delete.

And then there's also the ability to create notes as well, so we need to perform all CRUD operations for notes and also for users as we read through that. So we know our basic REST API is going to need to complete the CRUD operations for both notes and for users.

And it's eventually going to need to support authentication, although we'll do that last. We want to get the application working first and then apply the authentication as the last step.

So we're now ready to get our server up and running in Node.js. To do that, I'm going to once again open up the terminal window with control and backtick, then I'm going to type npm init and -y, which will help it avoid all of the questions it usually wants to ask.

We can see some of that being completed here, and after we did that, we get a package.json file. Now this is where our dependencies will be listed.

I need to install two dependencies today, so I just opened up the terminal window again. I'm going to use npm and then type i for install, and then I want to install Express, which is one of our MERN stack technologies.

And then I'll just press enter and I will let that install, and we should see it listed here as a dependency inside of our package.json.

After that, there's a dev dependency, which is different from a normal dependency, because we'll only use it during development. So I'll type npm i for install again, and then I want nodemon.

Nodemon will let us run our code and as soon as we save the changes, it will continue to run our server with those updated changes, so it's very useful.

Okay, after we've applied both of those, we should be able to close out the terminal. Once again, I'm going to scroll to the top of the package.json, and you can see it probably named your project whatever you named your folder.

Mine's in Lesson One, it's version 1.0.0, that's fine. Let's go ahead and put a description in here and say TechNotes MERN Project.

After that, we're going to use a server.js file. This is really just kind of preference; index.js is what it defaults to.

For scripts, I'm going to go ahead and remove this test script, and instead, I'm going to put in a start script, and there I'll have node space server, a comma after that, because this is an object, I'm going to type dev, and then I'm going to have nodemon space server.

So we'll use nodemon during development, and that's the script that will go ahead and let us start nodemon.

One other thing I like to do right at the beginning, so I don't forget, is create a .gitignore file. We haven't initialized Git for our repository yet, but we could at any point.

Inside of the Git ignore, we list the node\_modules, because we do not want to send those to GitHub or wherever we might keep our code repository.

Now that we've completed those things, let's create our server.js file. At the top of our server file, we'll define Express, and we'll set this equal to require, and we'll just require Express right inside of there.

Underneath that, we need to define the app, so we'll say const app, set that equal to Express, and we'll call Express.

And then I'm going to define a constant called port, and this will help set what port we are running our server on in development, but also when we deploy it somewhere.

Here, we'll get a .env port, if the place we would deploy it would have a port number saved in the environment variables, then it would grab that, otherwise, we're going to run it here locally at port 3500.

Okay, that's the initial imports. Now let's just tell our app to start listening, so we'll say app dot listen, and then we'll pass in the port, and then we have a function.

In this function, we'll say console.log, and we'll go ahead and make a template literal here. We'll say server running on port, and inside of this, we can pass in our value for port.

Let's go ahead and save our server file. Let's press control back tick to open up the terminal window

Here is the text without timestamps or formatting:

Type np, let me get lowercase npm, run dev, and after I press enter, notice our server has started running. This is the console for Node.js and it says server running on port 3500. It's not really doing anything if we sent a request there, we wouldn't have any luck, but the server is up and running with just those few lines of code.

Let's not stop there though. Let's go ahead and make our server serve a few bits of information to us. The first thing we need to do for that is import path from the Node.js system. So there will say require path.

And after we have path, now we can go ahead and use it here in the body of our server file. We're going to say app dot use, and we're going to listen for the root route, just the slash, which would normally be the root or the index of a web page.

Here we'll say express dot static, and then we'll use path dot join, which is a method of path, and then we have two underscores and dir name, which is a global variable that Node.js understands. It says look inside of the folder that we're in. After that, we'll put a comma and then say look inside of the slash public folder.

We're telling Express where to find static files like a CSS file or other resources like an image that we would use on the server. Now if we save this, it won't be able to find that if something was looking for it, because we haven't created it.

But notice Nodemon went ahead and restarted, and we're still running on port 3500. So now let's go ahead and come over here to the file tree and create that public folder. And inside the public folder, I'm going to create a CSS folder.

Normally with our REST API, we're just going to be receiving requests and sending back JSON data that would be requested, and we'll be receiving JSON data. However, a REST API can still have a splash page. It could still also return information about requests that cannot be fulfilled.

And so we can at least set that much up as we start. Now inside the CSS folder, you might guess we'll go ahead and create a style.css file. Not being a CSS tutorial, I'm just going to paste in some basic styles here.

I'll close the terminal quickly so you can see everything on one screen. We're importing in a Google font, we've got a basic reset, and just some styles on the HTML and body. Nothing big, but just a few styles there.

So we could have something on the splash page for the REST API or possibly a 404 page when a resource is not found. Let's go back to the server.js now and let's put in another line of code that says app dot use, and we'll once again look at that root route.

Here is the text without timestamps or formatting:

Now we'll say require and we're going to look for a routes folder and then a root file. And now we crashed the app because we do not have a routes folder and we required it in the file. So let's go ahead and create that routes folder and then inside of routes, we're going to need to create a file named root.js.

Inside of root, we need to require express again. Now after that, we need to define a router. We'll say const router and we'll set this equal to express dot router. Now we need the path once again. We'll say const path and will require path from Node.js.

And then after we have required those three things, we're ready to say router dot get. This would be a get request that relates directly back to our HTTP methods. The nice thing about Express in these routes is they recognize regex.

So we can use a regex here. The first thing I'm going to type is the carrot and that says at the beginning of the string only, and then we'll put a slash and then the dollar sign, which says at the end of the string only. So that means this will only match if the requested route is only a slash and that would be for the root.

But then I'm going to put in a pipe, which is an or for regex, then I'll have slash index, because maybe they would request more than just the slash as they put that in. And after that, I'm going to have the HTML optional, so they could request just the slash or maybe just the slash index without the dot HTML or the user could request the full index.html, which would also work.

And after that, we'll have a request and a response for our function. And then inside the function, we will send the file back. So it'll be response dot send file, then we'll say path dot join.

We'll use that directory name variable that Node.js recognizes, and now we need to tell it where to find the file. The file's going to be up out of the routes folder, so that's what the two dots indicate. And then we're going to tell it to look into a views folder and then we're going to have it look for the index dot HTML file.

But before we leave this file, we need to say module.exports and set that equal to router. And now this file is complete. Also, notice our server is once again running. Nodemon restarted it as soon as the errors were fixed.

We can go ahead and do that. This is not a require. The require is actually what crashed before when the required thing was not available. But now this does not, of course, crash the application.

Okay, now we'll go over here to our file tree once again, create a new folder, we'll call this views. And then inside of the views folder, we'll create a new file and call it index.html.

I'm going to type an exclamation mark, which is an Emmet abbreviation in VS Code, then I can press tab and we instantly get the foundation of an HTML page here. I'm going to replace the title where it says document with Tech Notes API.

And then inside of the body, I'm just going to use an h1 element and type Tech Notes. And then we need to bring in the CSS, so that is a link rel is a style sheet, and then the href is going to be set to CSS slash style dot CSS.

And then we close out that link tag. Actually, I think I need the slash close there. We go. And after we save that, now it can find this CSS because that is a static public file that we set up earlier with our route back here in the server.

Our server is still running. Let's drag VS Code to the left. I'll go to my new tab. We want to go to localhost colon 3500 and press enter. And there we get our Tech Notes homepage at the root.

However, we could request something that doesn't exist, and we might not get the best results. If I put in slash Dave, we just get the basic Express response cannot get Dave. So let's fix that by going ahead and taking care of basically 404 errors.

And those are for resources that are not found. Let's start by creating a 404 page inside of the views since we're already there. So another new file, I'll type 404 dot HTML.

Again with the Emmet abbreviation to get the foundation here. We'll just say 404 not found. I think that's a little bit better than error. But then inside of an h1, we'll just say sorry.

And then we can put a paragraph as well. And inside the paragraph, we'll say the resource you have requested does not exist. And of course, we once again need to bring in the link. That would be rel equals stylesheet and href equals CSS style dot CSS

Here is the text without timestamps and formatting:

Close that out and save. Let's go back to the server and let's handle anything that's not found. We want to put this after all the other routes, and of course, I'll put it before the app listens for the server. But here we'll just say app.all instead of app.use.

And now we'll listen for this asterisk, which essentially means all. Everything that reaches it to app.all will be put through this instead of being routed to anything that's above, so that it's the catch-all that goes at the end.

So then we'll once again have our request and our response, and we'll have our function. Now and the first thing we know is the status is a 404, so we can set that right away, although we're not sending the response yet.

And now we can look at the headers from the requests that come in and determine what type of response to send. We can say if the request has an accepts header that is HTML, then we can base our response on that.

So here inside of this if we'll say res.send file, and now this will basically be what we did before to send our index.html, but we're going to send our 404. And of course, we have to route to it correctly.

So now we're at the server level, so we don't need to go up out of a folder. We're just going to go down into the views folder, and from there we're going to get the 404.html file.

After that, we can put an elsif, and here we could say request.accepts, and let's look for JSON, which would be very common sent to a REST API.

So if there's a JSON request that wasn't routed properly and didn't get stopped by any of the expected routes, this would be the response. We'll say response.json, and now inside of this, we'll have a message, and then we'll say 404 not found.

Very basic generic message. And finally, we'll have our last else that will be sent no matter what if HTML or JSON was not matched in the accept header.

And here we'll say response.type text is fairly safe, just about everything can receive text, and we will send once again our 404 not found.

Now we'll test this out in the browser, and as you might expect, we should get HTML back. So if I'll highlight this and just press enter again, now we get sorry, the resource you have requested does not exist.

So we got our 404 page when it didn't match any of the other routes. We are well on our way to having our REST API up and running for our full-stack MERN stack application.

Remember to keep striving for progress over perfection, and a little progress every day will go a very long way. Please give this video a like if it's helped you, and thank you for watching and subscribing. You're helping my channel grow. Have a great day, and let's write more code together very soon.

MIDDLEWARE REST API

Here is the text without timestamps and formatting:

Welcome to today's tutorial where we're adding middleware to our MERN stack project. I'll provide links to example source code and all resources in the description below.

What is Middleware?

Middleware is just one or more functions that are placed in the path of requests that are received by our backend API. Middleware can add additional functionality to our backend REST API. Middleware can also apply some preliminary processing to requests before they get to the controller where the request processing will be completed.

We'll be adding three types of middleware today: built-in middleware, custom middleware, and third-party middleware.

Quick Corrections & Starter Code

I'm on the (link unavailable) site in the first tutorial and I said the latest version of Node to use, and really what it stands for, LTS stands for Long Term Support. I had latest on the brain from installing the latest packages, but I just wanted to highlight that as I've been asked about that a couple of times.

Also, in the code from lesson one, which is our starter code by the way for lesson two, just always go back one previous lesson and that completed code will be the starter for the next.

I just wanted to highlight that in path join, as you see that I'm using right here, path dot join, you do not have to use a slash, and notice I didn't do that when I talked about the views directory, but I did it with public. You can remove that if you want to.

Built-in Middleware

Okay, with those corrections out of the way, let's get started by talking about middleware. We're in that starter code, which is the completed code from lesson one, already. So we already have our server set up, and now we're going to add middleware.

We've already added one piece of middleware that we really didn't discuss, and that was built-in middleware, express.static. It's telling our server where to grab static files.

We put in the app dot use, and then we said slash, and then we have express dot static, and I use path join to get the directory name, which is this global variable, and then I said in the public folder, which we created.

You might see this used without quite so much explicit information, which you can also do. But I like to leave mine with a little bit more explicit information.

We've already added one piece of built-in middleware, but now we can add some more. The other one that we need is the ability to process JSON in our application.

So we're going to say app.use, then we're going to have express.json, and we need to call that, and that will let our app receive and parse that JSON data, and that's what we expect to use.

Now with that complete, you can see how easy it is to add built-in middleware to your application. But now we need to go ahead.

Here is the text without timestamps and formatting:

Let's create some custom middleware. I'm going to create another folder. Let's first create the logs folder because a server needs to be able to log some things, errors possibly, requests, and whatever we want to log.

We'll have that logs folder. We're also going to create a middleware folder. And then, while I'm thinking of it, inside our .gitignore file, let's add the logs directory because we don't want to send those logs up to GitHub.

There'll be development logs, but even in the future with another repository, the logs wouldn't really help our code base, so they don't need to be sent to the repository at all.

Now, let's go to the package.json file. I'm going to change the name here to Lesson Two. Now we need to add some dependencies that we're going to use as we write some custom middleware.

I'm going to press Ctrl and the backtick, then I'm going to type npm i date-fns and also uuid. Two separate packages, and we can add them on the same line.

When they are finished installing, we should see them listed in our dependencies. Now we have date-fns, express, and uuid.

Inside our middleware directory, let's create a new file and name it logger.js.

At the top of logger, we need to destructure. I'm going to destructure format and get that from our date-fns dependency.

Then I'm going to do another destructure, and this is going to be v4, and I'm going to rename it uuid. That's going to be coming from uuid.

After that, I need the fs module for the file system, so that comes directly from Node. It's already built-in.

Now I'm going to create a helper function called logEvents. This will equal an async function that receives a message and also a log file name.

Inside this function, I'm going to create a date time variable and use that format function. Notice this is a template literal.

I'm creating a new date object and then formatting it. This goes from the docs of the date-fns package.

Then I have a log item, and for this log item, I'm passing in this date time that we just created.

Notice the /t's, those are tabs, which will make the logs easy to import into Excel or something similar.

I'm calling uuid here, which creates a specific id for each log item.

Then we have the actual message that is passed in. This last /n creates a new line, so each log item gets its own line in the log file.

Let's go ahead and create the rest of this function. We've got a try.

Here we're going to check if the directory exists first. If not, and we try to save there, we'll have an error.

So we say if not, which is the exclamation mark, then fs.exists sync.

Then inside here, I'll use path.join.

We'll use the global der name variable, and then we'll go up out of the folder with two dots and then look for a logs folder.

We're saying if that doesn't exist, then we'll need to create it.

So we'll say await fs.promises.mkdir.

Then we'll just need that same directory that we're looking for if it doesn't exist.

That's what we're creating.

So now we have either created the directory or it already existed.

And now let's say await fs.promises.appendFile.

We are appending to our log file or creating that log file as well if it doesn't exist.

This is once again that same path.

We're not finished after that. I need a comma after logs and need to say the log file name.

Then we can have the parentheses and then specify our log item.

Here is the text without timestamps and formatting:

We're ready to add one more piece of custom middleware.

Let's create another new file inside of the middleware directory and we'll call this error handler.js.

I'm going to close the terminal just so we don't have to look at that right now.

And we'll start out by importing in that helper function that we created called log events.

So that will equal require and we're in the same directory so it should just come from logger.

From there we can create our error handler and this is going to overwrite the default Express error handling.

And we can do that just by creating this middleware that starts with an error and then it also has the request response and next.

And now inside of this function, this middleware, I am going to once again paste in a line that's just a little bit difficult for me to type without typos.

And we'll go over this, but we're calling that log events function and it's a big template literal here.

I'm including the error name and message and then there's a tab and then it's just like the previous log events that we were logging.

So it has the request method, the URL, the origin, and this is going to the error log file, so error log.log.

Otherwise, basically the same.

I'm also going to put console.log and log the error.stack.

This will be a pretty large message inside of our console, several lines worth, but it will give us a lot of details about an error and tell it specifically where it is, which is always helpful.

Then I'm going to define a status and this is going to look to see if the response we receive up here as one of the parameters already has a status code set.

And this is a ternary, so if it does have that status code set, then we'll just return that status code.

If not, it's going to be a 500, which is a server error.

And then we'll set the status to whatever our ternary determined.

And then we'll have a response that is JSON data and we'll say message.

And then we'll have the error dot message.

Let's not forget to do the module exports at the bottom.

So we'll say module.exports, set that equal to error handler.

And now back in the server.js file, we can import our error handler as well.

So we'll say const error handler and this will come from that middleware folder also.

And this is from the error handler file.

But now instead of using it at the top like the logger, we want to use it at the very end.

Essentially at least right before we tell our app to start listening.

So here we'll say app use and we'll pass in that error handler right there.

Now we will test this out.

Here is the text without timestamps and formatting:

A moment, but I also want to get started with our third-party middleware now that we've written custom middleware.

So let's go to the package.json one more time, bring up the terminal, I'm going to press Ctrl+C to stop the server temporarily.

Type npm i and I need cookie-parser, our REST API is going to need to be able to parse cookies.

And that's because we're going to use them in this MERN application.

We can see we've added the dependency right up here.

So I am ready to once again type npm run dev and keep the server running.

And it will restart with changes.

That's good.

Let's go back to the server.js.

I'm going to scroll back to the top for the imports.

So once we get up to the top, I'm going to import that cookie-parser.

We'll call this cookieParser in camel case.

We'll set this equal to require cookie-parser.

And once we have that, it's just about as easy to apply as our built-in Express JSON middleware.

So we'll say app dot use.

And then we'll say cookieParser and we call that.

And now we'll be able to parse cookies that we receive as well.

So that was very easy third-party middleware to add.

However, I want to cover something a little more complicated that must be added every time.

Or at least a consideration every time you create a REST API.

So I'm going to drag the code to the left.

I am going to go over here to the browser.

And we're at (link unavailable)

I'll just press enter to pull that up.

And then open up dev tools.

Clear out whatever warnings they have.

And what I'm going to do here is type in a fetch.

Right in the console.

And type in our development REST API.

So localhost 3500.

And now when I press enter.

We get a CORS error.

CORS policy.

That stands for Cross-Origin Resource Sharing.

And it says no Access-Control-Allow-Origin header is there.

This sends an options request.

Which is another type of HTTP request.

Much like a post, a put, a patch, a delete.

And so on.

And that options did not detect that header.

This is really the first line of security.

Also, Google should not be able to request information from our API.

Another resource at another URL essentially.

Unless we say it's okay.

So we have to enable CORS for that to happen.

And pass this what is called a pre-flight request.

Setting up CORS middleware.

So let's go ahead and do that.

And we'll do it as if we were creating a public API first.

And then we'll secure it afterwards.

So it's very easy to set it up as if it were a public API.

So we'll go ahead and control backtick.

Control C to stop the server again.

We'll type npm i cors.

And it should install quickly.

And we should be able to see that listed in the package.json now as a dependency.

So that's good.

We'll control backtick again.

And npm run dev to start the dev server back up with nodemon.

After we've done that.

Let's go back to the server.js.

And we need to import cors.

So we'll say const cors equals require cors.

And after we have that.

It's just about as easy to apply as these others.

When we just want everything to be available to the public.

We'll say app.use.

Cors.

And we call cors.

And now if we pull this back over to the left.

And we go back to Google.

And we do control shift i to open up dev tools.

We've got our error here from the fetch.

Let's try the fetch again.

And see what happens.

We get a promise pending.

Because we're not really processing anything from the fetch.

And we expect the promise.

But there's no error.

That's because our API is now essentially available to the public.

So those that are at other origins can request resources from our API.

Now what we want to do now.

Here is the text without timestamps and formatting:

So we only allow the origins that we want to access our API.

And we can do that with CORS options.

I'm going to control or close the dev tools.

Drag this back to the right Visual Studio Code full screen.

And we need to create some CORS options.

We're going to do that by creating first a directory over here called config.

And inside of config, we're going to create two files.

The first one is allowed origins.js.

Here we'll define our allowed origins.

Set this equal to an array.

And now these will just be strings that will say what origins are allowed.

And the first one we'll put in just for development will be where our React app will eventually be at.

And requesting data and that's usually localhost 3000.

After that, I'm making up a couple of fictitious URLs here.

I didn't register them, I don't think anybody else has either.

But you never know.

So I'm just saying https colon slash slash www dot.

Because that is our stakeholder for this project is Dandy.

And his repair shop.

And then we'll also say https colon slash slash.

And we'll just do this without the w's.

These are strings that need to match.

So you would typically want to do both of these for any URL.

So you'd have (link unavailable) as well.

These would be the local hosts that we would accept.

To access our REST API.

And now we'll say module dot exports.

Equals allowed origins.

And now we can use these when we create the CORS options in the next file.

So we'll create another file CORS options.js.

I'll start by importing in the allowed origins that we just created.

And it's nice to have those in a separate file.

Unless we if we want to refer to them somewhere else inside of the API as well.

So that's why they're in a separate file to begin with.

Now we'll say const CORS options.

And this is going to equal an object.

This is again third-party middleware.

So we have to follow the rules they have set up for their options.

This is like a lookup object where we have the origin.

Method here.

And this is going to receive the origin.

And a callback function.

And now inside of this function.

We're going to check that allowed origins list.

And then we'll say dot.

Index of.

And I said list, it's an array.

Then we have origin.

And then we'll say if it's not equal to minus one.

Now this would limit it to where only those in the array.

Only those origins would be able to access our backend REST API.

But then that would screen out other software like Postman.

That we might test our API with.

Or possibly desktop applications.

Or anything else that didn't provide an origin.

So you might also want to say or.

No origin.

And this would go ahead and allow Postman.

And some other things to possibly access our REST API.

So here this is if it's successful.

We'll say callback.

The first argument that is passed into the callback.

Is an error object.

So here we'll just say that's null.

Because we don't have an error.

The second is the allowed boolean.

Either true or false.

And yes this is successful.

So we'll say true.

And then here we'll have the else where it fails.

And now we'll have a callback.

And we'll use that first argument.

Which is an error.

So we'll say new error.

And we'll say not allowed by CORS.

There's also some other options.

We can set inside of CORS.

One is credentials.

And we'll set that to true.

Now this sets the access control allow credentials header.

And if you did my Node.js for beginners course.

We actually created separate middleware.

To set that header.

Which is kind of taking the long way around.

But we learned a little more.

Here you can just set this option to true.

And it handles that header for you.

So that's a great option to go ahead and set to true.

There.

And then say options success status.

And we'll set this to 200.

I believe the default is 204.

But there are some devices.

That have some problems with that.

So you might just want to set that to 200.

Makes so we will not have any problems.

Let's head back to the server js.

And go ahead and apply these options now.

So we need to import our CORS options.

So we'll say const CORS options.

Set this equal to require.

And then we'll have our config folder.

And then we'll have our CORS options file.

And then for CORS options.

We just pass them right into CORS here.

So once we've done that.

Our CORS options are now set up.

Let's go ahead and make sure the server is running.

And yes it's on port 3500.

And running as expected.

I

<https://www.youtube.com/watch?v=98BzS5Oz5E4&list=PL4cUxeGkcC9iJ_KkrkBZWZRHVwnzLIoUE>

Here is the text without timestamps and formatting:

Hey gang and welcome to your very first MERN stack tutorial.

All right, so to begin with, I just want to talk a little bit about what the MERN stack is, the different technologies involved, and how they all work together, and also just to explain exactly what we're going to be building in this series.

So the MERN stack is basically a bunch of four different technologies that we can use together to make an interactive and data-driven web application or website.

And if you master the MERN stack, then technically speaking, you're a full-stack developer, meaning you build the front end of a website and the back end of a website.

That's generally what we mean when we say full stack: the front end and the back end, the whole shazam.

Now, there's loads of different variations of a full stack, and the MERN stack is just one of them.

It consists of React, which is the front-end library that we use.

Node and Express, which we use to make the back-end API that we interact with from the front end.

And then finally, MongoDB, which is a NoSQL database to store application data in, like user data or blogs or whatever other data that your website uses.

And together, they spell out the acronym MERN.

So, how these all work together is as follows:

On the front end, we have the browser where we see the website, and our React application runs in the browser to power the website and typically handles routing in the browser as well to show different website pages.

Then, when we need to show data in the website, like blogs or even just authenticate users, we'd send a request from the front end to the back end.

Now, the back end is an Express app running in a Node.js environment.

And if you don't already know, Express is just a framework for Node that lets us easily create APIs.

So the Node and Express API would handle our request on the back end and typically interact with a database to get data or update data or delete data, etc.

It would also handle authentication requests to do things like log a user in, log them out, or sign them up, as well as protect certain API endpoints from unauthenticated users.

So only logged-in users can access certain resources.

Anyway, once it has the data from the database, it would then send a response with that data back to the browser, the client.

And the React app would handle that response by outputting the data in some kind of template.

Now you might be thinking, what's the point of this middle Node API step in order to fetch the data?

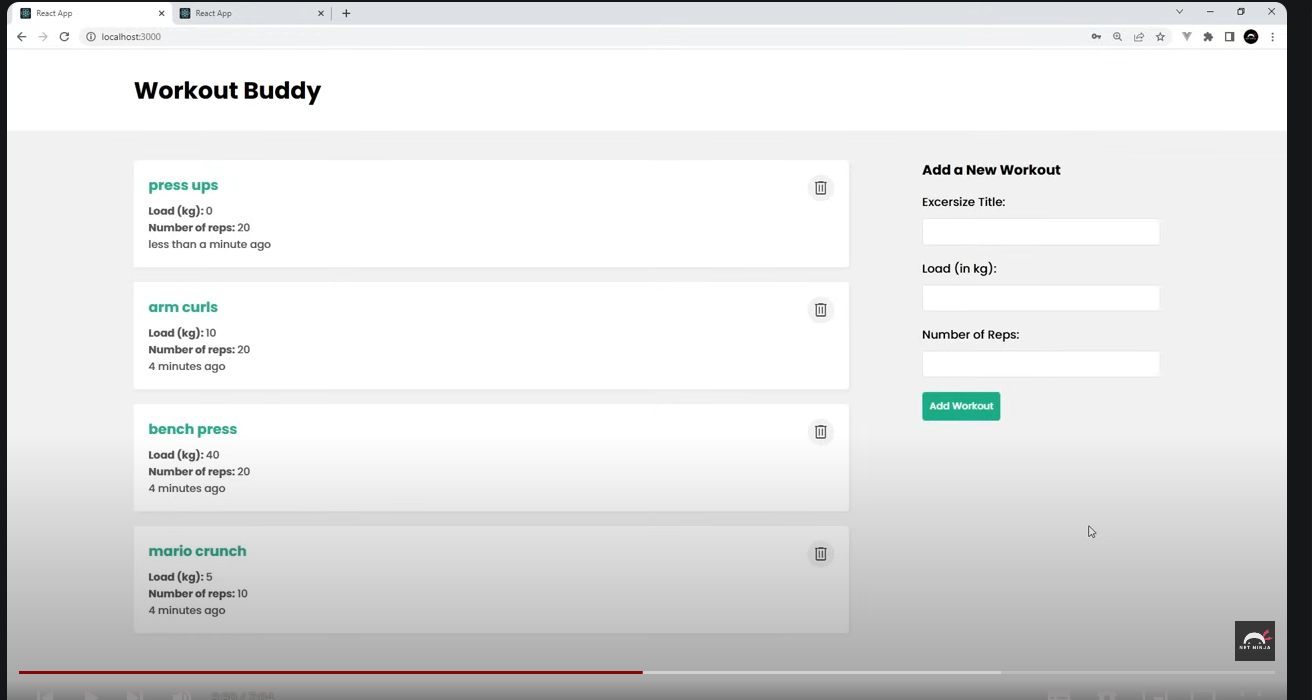
Why can't I just reach out directly to MongoDB for the data and skip this middle bit entirely?

And the answer is that if you do that, you'd be exposing sensitive data in your front-end code in the browser, such as an admin username and password to access your database.

And by putting that logic in the back end instead, you're hiding that sensitive code from people who use your website.

So that's the MERN stack from a bird's eye view.

And these are the four different technologies we'll be using to build a full-stack application in this series: React, Node, Express, and MongoDB.



Here is the text without timestamps:

It's going to be absolutely fine, we can see now listening on port 4000, so it's working.

But if I make a change to the file now, for example, if I just add some exclamations here and save it, then in order for this to work, we'd have to cancel out of this process and then run node server again, and now we can see the updated string right here.

Okay, let me just change this because I've spelled it incorrectly and save that again.

An alternative is to use something called nodemon.

So let me cancel out of this process and clear down here.

The first thing you want to do is make sure you have this nodemon package installed globally.

And to do that, you can say npm install -g and then nodemon and press enter.

That's going to install the package globally on your computer, so you can use it anywhere.

I've already done it, so I'm not going to do it again.

But once you've done that, you can then say nodemon and then the name of the file, server.js, and press enter.

And now it does the same thing, it logs this to the console down here.

But now if I make a change to the file, let me get rid of those and press save.

We can see that it detected that change and it re-ran the file, and we see that new log down here.

So that's good.

It watches our files for us and it reruns the application whenever it detects a change.

So that's cool.

But I'm going to cancel out the process again by pressing Ctrl+C and clearing this.

And what I'm going to do is go to the package.json file to create a new script.

And this is going to be a dev script.

So this is what I like to do sometimes.

Create a script inside the package.json file and then in here, I can say whatever script I want to run when I run the dev command.

So it's going to be nodemon and then server.js.

So now we can run this dev script right here down here.

I can say npm run dev and that's going to do exactly the same thing for us.

Okay, so now it's listening on port 4000.

And now whenever we make a change to the file, it's going to re-run the script for us.

All right, cool.

So now we have our Express app right here.

We're listening to port 4000 requests.

Now we also want to react to requests.

So to do that, we need to set up a route handler.

So I can say app.get and then this is going to respond to a get request coming in.

And then we can specify a specific route for this get request.

So I'll just say forward slash.

And that means if I go to localhost port 4000, just forward slash, the root of the domain, if you like.

Then it's going to fire a function, which is the second argument right here, to handle that request.

And inside here, we can take in the request object, which has information about the request.

And also a response object, which we use to send a response back to the browser or the clients.

So what I'm going to do is use that response object to say response.json.

And this sends back a JSON string for us.

And inside here, I'm going to say we have a property called message.

And then we'll just say welcome to the app.

Like so.

Let me just do a comment right here to say routes.

Like so.

Now if all of this goes over your head, then you probably should check out my Node.js and Express course first of all.

Remember the link to that course is going to be down below the video.

But anyway, now we have this route up and running.

And let's try it out in a browser.

So if we go to localhost port 4000 right here and press enter.

That sends a get request to this URL to the server.

The server responds with this JSON right here with the message property.

So this is all working.

But at the moment, we're hard-coding the port number right here in the server.js file, which is fine.

But what I want to do is store any constants like this in an environment variable.

And the benefit of doing this is that when you eventually push your projects to repositories like on GitHub.

The environment variables can remain hidden and they're not visible in the code.

Here is the text without timestamps:

At might not be a big deal when it comes to the port number for now, but when it comes to more sensitive information like a database connection string or an authentication secret, you don't want those things visible, so they're better off going in environment variables, which will remain hidden.

So the way we're going to do this is by making a .env file in the back-end folder, and this is going to store all environment variables.

The only one that we want at the minute is the port number, so let's call this PORT, all in capitals, and let's set it equal to 4000, no spaces between the environment name or the environment variable name rather and the value.

So the idea is that when you're pushing your code up to something like GitHub, you add the .env file to your .gitignore file, and then this file won't be pushed up, so everything in here remains private.

Say we want to access now these environment variables in our code, how do we do that?

Well, we need to do it with the help of a Node package called dotenv, and we need to install that into our project down in the terminal.

You'll need to cancel out of the nodemon process first of all to do this, and you can do that by hitting Control+C and then typing y, and then install dotenv by typing npm install dotenv and then hitting enter.

So dotenv is a package that loads environment variables from a .env file into the process.env object available to us globally in a Node.js environment.

And once this package is installed, we can then go ahead and use it in the server.js file to do that for us.

All we have to do is require the package at the top and directly invoke the config method on it once it's been required, and this is the method that's going to attach those environment variables for us to the process object.

So now what we can do is come down here and instead of just saying 4000, we can say process, which is a global object available to us in Node applications, and then say .env and then another dot and then whatever the environment variable was called that we want to use in our case that's PORT, all uppercase, and this is going to still be port 4000, but now it's being pulled from the .env file.

Cool.

So now we can preview this again by running the script down in the terminal.

So let's do that, we'll say npm run dev, and this should run in turn nodemon and spin up our Express application.

All right, so in the browser, everything is still working the same way, which it should.

Awesome.

So at the minute when we send a request in the browser, this is a GET request, and since we've set up a GET request handler, it's handling that request fine.

But if we wanted to test out a POST request or a DELETE request, which we will need for this application, then we're not going to be able to do that directly from the browser by entering the address up here because, like I said, when we press enter up here, it sends a GET request.

And if we want to send POST requests or DELETE requests, we're going to have to create some JavaScript on the front end to send those requests.

Or, before we have that front end, we could use a tool called Postman to try out those different routes and requests.

So you can get that from (link unavailable) and download it right here.

And what this allows us to do is kind of simulate different types of requests to our server.

So GET request, POST request, DELETE request, PATCH request, etc.

So this is what we're going to be using for the next few videos to test out any routes that we create for our API.

So after you've downloaded and installed Postman, when you first fire it up, it might ask you to sign up or sign in or something like that.

<https://www.mongodb.com/resources/languages/mern-stack-tutorial>

Net Ninja

Dave Grey

JetBrains Ai

Google Duet AI

MERN Stack Tutorial Book Store Project

MERN Recipe App With Authentication

Food Delivery App

Blog App

Nettfix Clone

Finance Tracker With user management

Banking Web App

Doctor Appointment Booking Website

Real Estate App

Expense Tracker App

Social Media With Real TIme Chat Functionality

Also, Social media App with Auth, Pagination Comments

GFG

Airbnb clone

Real Time Forum For Teachers

3D AI Website/ AI Image Generation

Kindle Clone

Chat GPT Clone

E Commerce App With Admin Panel

Build and Deploy a Travel Companion App(Google Maps)

Inventory management dashboard

Waste Maanagement App

tls certificate checker

Beware if you are creating a clone, mention clearly if its is a web app website or an UI clone

Netflix clone would mean- streaming handling, HLS, Video handling, How are audio and video streams combined

E commerce clone- database modelling- wishlist, review rating, review rating avg calculation reviews update products updated in multiple categories- one can write controllers on them as well as save database

Maybe user doesn’t know exact name, maybe they use synonymous keywords- pagination

Payment gateway – one should know multiples gateways(Razorpay, STRIPE)

How does payment get stalled

What should be retrieved from payment, What are standard practices, best way to generate Order ID

Can we directly link UPI, or do we need an aggregator

For excel clone, the expectations are that you must have handled a million rows, -- so what techniques were used, how was the rendering done,

Google Docx- multiple people editing, And if you answer that you used a library, it is an empty explanation. You must know how that package works behind the scenes. Which is the realtime? If there are sockets how do they work. How is broadcasting of events done, how are events accepted. How are rooms made and how are they closed.

Zoom Clone-Ice framework, how are offers created, what is stun server

Web RTC

In absence of valid projects, resume round becomes DSA VIVA round

Unique projects- API based projects- Amazon Review API Portfolio based projects(mostly for dev) Full stack Projects(Clone Apps)

1) Real - Time Ticket Booking App 2) Integrating the above project with a payment Gateway. 3) Distributed File System 4) Robust Payment Retry Mechanism 5) Smart Utility Management System.