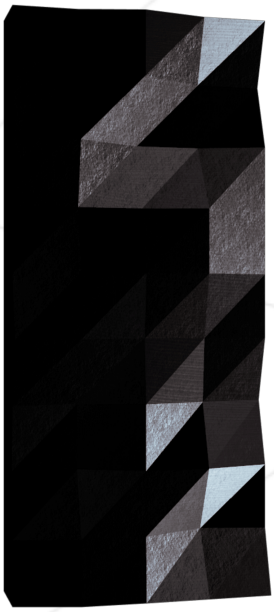


Node.js

Donald Stolz dstolz@student.42.us.org

Summary: Learn backend development with Node JS.

Contents

Chapter I

Foreword

"`Programming is like magic.` You write very specific instructions in arcane languages to invoke commands, and if you get it even a little bit wrong you risk unleashing demons and destroying everything."

Chapter II

Introduction

What is a server? A server is a computer/device/program that is dedicated to managing network resources. Most any computer is capable of serving as a network server. A server serves information to our client. The information we need is stored in our database. How do we communicate with the server in order to get the proper information? We communicate with our application server through an Application Programming Interface, or an API. An API is a set of functions and procedures, it receives requests and returns responses.

Learn more about the internet, servers, and APIs:

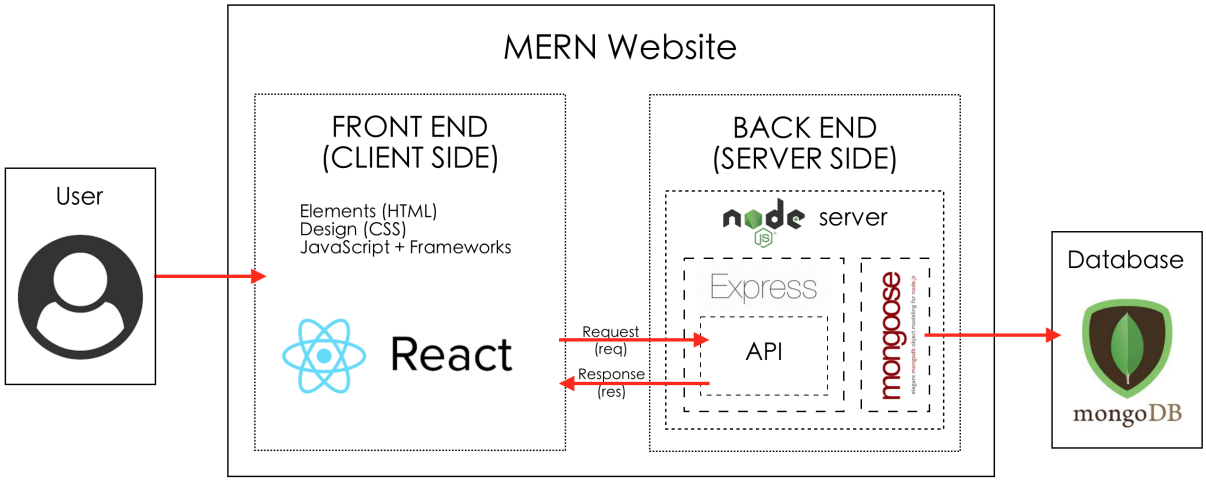
- How the internet works
- What is a Computer Server?
- What is an API?

Chapter III

Goals

By the end of this project, you'll have built a server. You'll learn how to create servers by writing backend services with Javascript and develop an understanding of how APIs work. We'll be creating the backend for a To-do list app that can be hooked up to a frontend that we'll be making later down the line. Later, using HTML, CSS, and Javascript, we'll create a view that will show our to-do's as list items that can be checked off.

The server we'll be writing is simple but scalable and can be bigger, a lot bigger. You can imagine even websites like Netflix, PayPal, Medium, and many more are using Node.js with more complex routes. Learning these skills will enable you to build many different apps.



We'll be building our app using the MERN stack, which stands for
MongoDB
Express.js
React.js
Node.js.
There are many different stacks but this is the one we'll be using. In this project we'll be building the Backend/Server-side part represented in this diagram, and connecting it to MongoDB.

Chapter IV

General instructions

We will start with some simple guided lessons on the basics of setting up and testing a Node.js server. Once you've completed the basic lessons you will apply your new knowledge towards creating your own API for tracking Todos. You'll then connect it to MongoDB and have the option to host your server on Heroku so it won't just be hosted locally and will be externally accessible.

Follow the exercises and write out the code yourself (instead of just copy-pasting). This will help active learning. When starting out you won't understand every single thing you see because these apps take many pieces to build and when learning you need to start somewhere. Don't feel overwhelmed, you'll learn what you need to as you go. You got this, and don't forget to have fun with it!

Chapter V

Exercise 00 : Hello World!

It's time to setup our first Node.js project. What is Node.js? Node.js is a JavaScript runtime. Basically Node allows us to run JavaScript outside of a web browser. This is exciting because now you can build a full stack application using just Javascript, when previously you'd have had to learn another language for the backend.

Let's start by creating a new directory called `Basics` from the Terminal. Now `cd` into our new directory and run `npm init`. Go ahead and just press enter through all the prompts, they're not important for our project.

Once we have our project intialized we can add our first Node Package. Run `npm install -save express`. The "npm install" command add the express package to our project, using the "–save" flag saves it as a project dependency. A dependency is a package your app installs and requires. One of the advantages of Node.js is the large, active, open-source community, and with NPM you can access a large selection of packages. You should be able to find all the project dependencies in the package.json file.

Our directory should now consist of the `package.json` file and our `node_modules` folder. We will also add a `.gitignore` file. Inside this file simply write `node_modules` and git will now know to ignore our `node_modules` directory.

Now to actually build our server! Create a file called `server.js`. Inside this file write the following code:

server.js

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

app.get('/', (req, res) => {
    console.log("Hello world!");
    res.status(200).send("Hello world!");
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}`);
});
```

Check out the Express.js docs to understand to understand what exactly is happening in the code above.

Let's setup our start script. Open up `package.json` and under the scripts object replace `"test"` with `"start"` and then replace the script's value with the command `"node server.js"`. Your package.json should look like this:

{} package.json ×

```
{
    "name": "basics",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "node server.js"
    },
    "author": "Don Stolz",
    "license": "ISC",
    "dependencies": {
        "express": "^4.16.4"
    }
}
```

Finally our server is ready to run. From the root of your `Basics` project run `npm run start` and head to http://localhost:3000/ to check out your server.

Use "control + c" to quit your server.

Check out this link for more information on npm.

Chapter VI

Exercise 01 : Setting up additional endpoints

Now that we have our basic server setup we can start adding more endpoints to expand our functionalty. What is an endpoint? An endpoint is a URL pattern used to communicate with an API. Endpoints can also be differntiated by their HTTP methods. An HTTP method allows us to indicate the type of action we are trying to preform. Our current server consists of a single ‘GET‘ endpoint.

Check out MDN for more information on the different types of HTTP methods.

Let’s first add a database so that we have some information for our new endpoints to access. Create a file called `"db.json"` in this file create a simple array of user objects with the fields `"name"` and `"id"`. It should look something like this.

db.json

```
[
    {
        "name": "Don",
        "id": 0
    },
    {
        "name": "Dan",
        "id": 1
    }
]
```

Now we can access our database by importing it the same way we imported the express library. Simply add `const db = require("./db.json");` to the top of our `server.js` file.

Setup a new ‘GET‘ endpoint called `"/users"`. It will be setup similarly to our ‘Hello World¡ example, except it will return our database:

```
JS server.js ×
    // Get all users
    app.get("/users", (req, res) => {
        res.status(200).send(db);
    });
```

Notice that along with the response we send back we also set the status code to 200. Status code 200 indicates that the request has succeeded. Check out the different status code values here.

Next, we will setup a route to retrieve an individual user from our database. Setup another 'GET' endpoint and call it `"/users/:id"`. Using `":id"` will allow us to retrieve `"id"` as a variable from the request parameters. Follow the snippet below:

```
JS server.js ×
    // Get user by ID
    app.get("/users/:id", (req, res) => {
        const { id } = req.params;
        const user = db.find(item => item.id == id);
        if (user) {
            res.status(200).send(user); }
            else {
            res.status(404).send("Could not find user");
        }
    });
```

Notice the new status code 404. This indicates an error, specifically that the server could not find the requested resources. Try testing your endpoints by going to http://localhost:3000/users and http://localhost:3000/users/0.

Restart your server to see the changes. We'll set it up to automatically update in Exercise 03.

Chapter VII

Exercise 02 : Setting up POST endpoints

We can fetch our entire db and individual users, now we will learn how to set up a 'POST' endpoint so we can add new users to our database. In order to do this we will be adding a few additonal libraries. First run `npm install -save body-parser`. Then add the body-parser middleware to our express app. We will also be adding the `"fs"` library to access the file system, this library comes with Node. Node has some built in libraries that don't need to be installed and can simply be required in. The top of your `server.js` file should look like this:

server.js

```
const express = require("express");
const db = require("./db.json");
const fs = require("fs");
const bodyParser = require("body-parser"); const app = express();

// Add bodyparsing middleware to app
app.use(bodyParser.json());
```

Set up a new ‘POST‘ endpoint called `"/users"`, this will be exactly the same as our ‘GET‘ route except we will use `app.get('/users')`. In this route we will also be pulling the `"name"` variable out of `"req.body"`, this is what we setup the `"body-parser"` middleware for. We will check that name exists, if it doesn't we will return a `400` status. Next we will setup the new user object and assign it's `name` and `id` values. Once we have the object setup, we can push it to the database. However, simply push the new value to the database will not update the file itself. We will have to first convert our JSON array into a string, then use the `fs` module to write the stringified database to our JSON file. Follow this example:

server.js

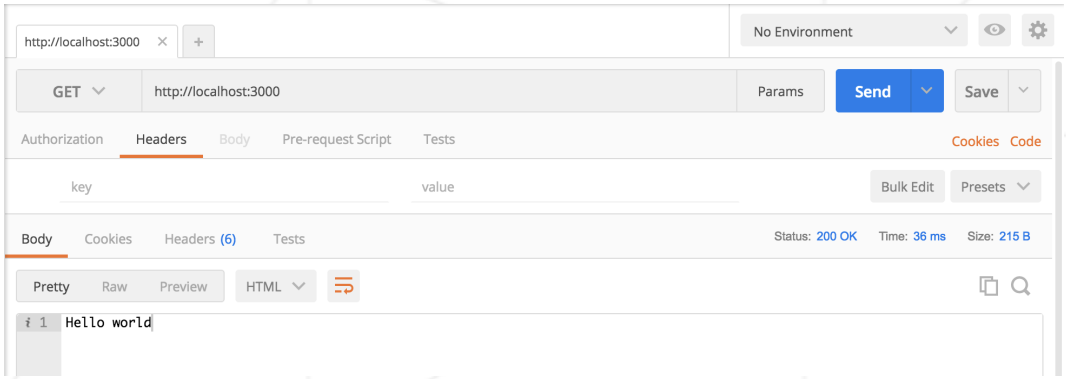
```
// Create new user
app.post("/users", (req, res) => {
    const { name } = req.body;
    if (!name) {
        res.status(400).send("Could not add user");
    }
    let user = {};
    user.name = name;
    user.id = db.length;
    db.push(user);
    // Convert DB to JSON string
    data = JSON.stringify(db);
    // Write string to file
    fs.writeFile("./db.json", data, "utf8", (err) => {
        if (err) {
            res.status(404).send(`Failed to add ${user.name} to db`);
        }
        res.status(200).send(`Added ${user.name} to db`);
    });
});

```

Chapter VIII

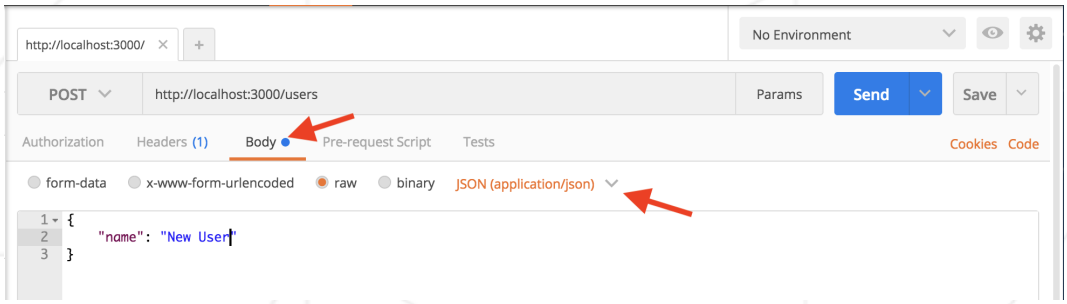
Exercise 03 : Testing with Postman

So we've setup a 'POST' route, but how do we send data with our request? We will be using a tool called Postman to test our routes. Head to the Application directory and open Postman, or use 'command + space' and search Postman.

Feel free to login with google or head straight to the app. Once you are inside you can quickly test our 'GET' requests by add `localhost:3000` as the 'request URL' like so:



Now that we know how to setup a 'GET' request we can setup our 'POST' request in a similar way. Here we will also want to add a body for sending the name of the user to add. We will head to the `body` tab and select `raw` as our format. Next we will also need to specify that our data is `JSON content`. Then we can add create a JSON object to add a new user. You should get a response that says `"Added New User to db"` with a status of 200.



Logging in with Google will allow you to save your routes which we will come in handy when your testing the Mandatory parts.

The final tool you will want to add to your project before starting the Mandatory Parts is nodemon. Nodemon is a development tool that will automatically refresh our server whenever we make file changes, this way we will avoid having to manually shutdown and restart our server everytime we make changes. To add nodemon run `npm install -save-dev nodemon`. Then open your package.json file. Add a new script called `"start-watch"` and give it the value `"nodemon server.js"`. Now we can run `npm run start-watch` and nodemon will take care of refreshing our server. Your `package.json` file should look like this:

{} package.json ×

```
{
    "name": "test",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "node server.js",
        "start-watch": "nodemon server.js"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "body-parser": "^1.18.3",
        "express": "^4.16.4"
    },
    "devDependencies": {
        "nodemon": "^1.18.10"
    }
}
```

Chapter IX

Mandatory part 1

Now you are equiped with enough knowledge and tools to build your own API. You will be building an API to keep track of Todos. Your endpoints should be as followed:

- GET `"/todos"` -> Gets all Todos in database
- GET `"/todos/:id"` -> Gets single Todo by ID
- POST `"/todos"` -> Add new Todo to database
- POST `"/todos/:id"` -> Update single Todo by ID
- DELETE `"/todos/:id"` -> Delete single Todo by ID

Use a `db.json` like we did in our previous example for storage.

Chapter X

Mandatory part 2

Now that we are pros at writing APIs we will connect our API to a proper database instance. Your challenge will be to `FIRST COPY` your existing API, then replace our current file system database with a newly setup Mongo database.

`First` follow the MongoDB PDF for instructions on setting up a cloud instance of MongoDB. Once we have our DB setup, we can set up Mongoose. Mongoose is an Object Document Modeler, it allows us to define a schema and easily connect to MongoDB. We will install it using `npm install -save mongoose` and require it at the top of server.js as usual. Then we can add the following snippet to connect our API to MongoDB through

server.js

```
// Connect to MongoDB
mongoose.connect("mongodb+srv://dstolz:42marvin@cluster0-4bipl.mongodb.net/test?retryWrites=true",
    { useNewUrlParser: true }
);

// Test MongoDB connection
const connection = mongoose.connection;
connection.once("open", function() {
    console.log("MongoDB database connection established successfully");
});
```

In order to use MongoDB with Mongoose we will also have to create a schema. Create a seperate file called `todo.model.js` and add the following code:

todo.model.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

let Todo = new Schema({
  task: { type: String, required: true },
  completed: { type: Boolean, required: true }
});

module.exports = mongoose.model("Todo", Todo);
```

We can import our todo model at the top of our `server.js` file.

server.js

```
const Todo = require("./todo.model");
```

Read through the Getting Started section of the mongoose docs and get acquainted with mongoose. After you do that, let's walk through an example here and make a route for getting all users. You don't need to write out this code, this is just to familiarize you with Mongoose. If you remember, this was the code we had before:

```
server.js
// Get all users
app.get("/users", (req, res) => {
    res.status(200).send(db);
});
```

Our mongoose schema for the user would look something like this, and be stored in a file named `user.model.js`:

```
user.model.js
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

let User = new Schema({
  name: { type: String, required: true },
  id: { type: Number, required: true }
});

module.exports = mongoose.model("User", User);
```

Next we will import the User model into the server.js:

```
server.js
const User = require("./user.model");
```

Now our server.js has access to the User model and all of it's methods. The method we'll be using is the find method. The find method takes an object with the properties it's looking for, so we could try to find someone by their name. If we don't pass it any object, it'll return all the model items, which is what we want to do for our '/users' route. Our new code would look something like this:

```
server.js
// Get all users
app.get("/users", (req, res) => {
    User.find((err, users) => {
        if (err) {
            console.error(err);
            res.status(400).send("Error: failed to get users");
        } else {
            res.status(200).json(users);
        }
    });
});
```

That's all for the walk-through, now it's your turn to implement Mongoose in your To-do list API. Good luck!

Check out the Mongoose model API docs to find the other methods that will help you rewrite your routes with Mongoose.

Chapter XI

Bonus Parts

XI.1 Bonus Part 1

Follow the Heroku PDF to remotely host your server.

XI.2 Bonus Part 2

Add routes that expand on the existing functionalty of your API.

Chapter XII

Turn-in and peer-evaluation

Always be ready to explain your code. Also be prepared to demonstrate your routes in Postman.