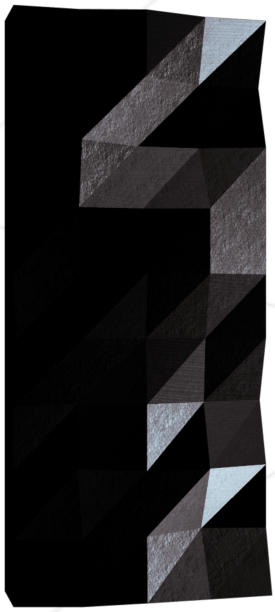


React.js

Zhag M zmagauin@student.42.us.org

Summary: Learn front-end development and build a well designed website view with React.js

Contents

Chapter I

Foreword

Humorous intro

Chapter II

Introduction

When a user opens a webpage, views the images, reads the content, and clicks the links, they are interacting with the front-end (or client-side) of the website. Front-end developers write the code to create views, essentially bridge the connection between a regular person on their browser, and the API's, services, and content that are being served on the backend. We have now created a backend server, but what can we do with it other than use postman to make requests? That's where this project comes in. Having a good user interface is important in making a website useable. We'll be learning the essentials of front-end using React.js and creating a view for your to-do list app.

What is React.js? React is a JavaScript library for building user interfaces. All it does is manage the view, or the front-end. It was developed by Facebook in 2011, and is currently open-sourced and supported by Facebook and a big community of developers.

Part of what makes React so cool, aside from the speed and ease of learning, is the format. Thanks to the JSX syntax, React code feels very tangible. It feels like we're combining both JavaScript and HTML, which makes it easy to conceptualize. It's also component-based, making the code you write in one React project pretty reusable for another.

Read through the React docs for more information on React, how it works, and what makes it so special.

Take a look at these links to get a better grasp of React. Get excited about it! This is a really fun library to learn, and also one that's in high demand.

- What is React? (Watch this! It's a very useful introduction)
- Thinking in React
- A list of some of the big companies using React
- A general overview and advantages of using React

Chapter III

General instructions

We'll start with setting up our environment and making sure Node and NPM are up to date on our monitors using NVM (Node Version Manager). Once we have that set up, we'll start with building our first app.

We'll be building a front-end for our To-do list API from the Node.js project. In this course, we'll be focusing purely on the front-end side of the website so you'll want to have deployed your API to Heroku beforehand. If you have not done this, you should go back and do that now.

After we walk through and build our To-do list app, you'll be using your new skills to build a project of your choice.

Chapter IV

Prerequisites and Setup

Before starting this project, you want to make sure you have your backend API from the previous, and that you have it deployed to Heroku. You can find the Heroku PDF (ADD LINK), if you haven't already done it go do that now.

Before we jump in, we need to check that all the tools we'll be using are installed and up-to-date. Type `nvm` into your terminal. If it tells you the command was not found, continue onto the following section. Otherwise, skip this section and continue onto the 'Updating Node' section.

IV.1 Installing nvm

NVM is a Node Version Manager. Node has many versions and many new updates, so it's good practice to have a version manager. Here we'll be installing that using Homebrew (which you should have installed while following along with the Heroku deployment tutorial). Run the following command in your terminal:

```
brew install nvm
```

Then follow the instructions provided after the installation (mkdir ~/.nvm, then open ~/.zshrc and add the necessary lines to it).

IV.2 Updating Node with NVM

Next we'll use nvm to install the version of Node we want to be working with. Right now we'll be installing v.10.15.3 because it's the last version with LTS (Long Term Support). To do this run the command:

```
nvm install 10.15.3
```

Once that's done, run:

```
nvm use 10.15.3
```

Check that you're running the right version of node by running:

```
node -v
```

That's it for our set up. Now that our environment is ready and our backend API is running, we're ready to start actually building our app.

Chapter V

Exercise 00 : Making Our First Web App

V.1 Create-React-App

In this section we'll be building the first version of our app. To do this, we'll be using Create-React-App, which will allow us to get into the actual front-end development instead of worrying about the details of configuration. As the official docs state,
`"Create React App is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration."`

In your terminal, run the command:

```
npx create-react-app todo
```

Once the installation is done:

```
cd todo
npm start
```

Go to localhost:3000 to check out the React App. This is the basic setup, now that you've seen it in action we'll be deleting all the contents of the src directory (where our source files are located) and rewriting it to understand how React works. Go ahead and clear the source directory:

```
rm src/*
```

Now we need to make our index.js file in the src directory:

```
touch src/index.js
```

V.2 Hello, World!

Inside the index.js file, import 'react' and 'react-dom'. We'll be using React for the React components we'll be building, and react-dom for rendering our app into the dom.

ReactDOM's render method takes in two arguments, a react element and the container into which we want to render the element. We'll give the render method this container by using the document.getElementById method. If you check out the index.html file (in

the public directory), you'll see we have a div with an id "root". That div is where we'll be rendering our App.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
      <!-- this is where our App will be rendered-->
    </div>
  </body>
</html>
```

With these 3 lines of code, we'll build our first Hello, World web app. Run 'npm start' and check localhost:3000 to see the result.

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<h1>Hello, World!</h1>, document.getElementById('root'));
```

V.3 React Elements and JSX

The <h1> element above looks like html, but it's actually a React element. What's the difference? React elements are plain javascript objects. They are the smallest building blocks of a React app. The reason the React element above looks like html is because of JSX, or JavaScript XML.

Try printing out the value of our <h1> element to see what is actually happening behind the scenes:

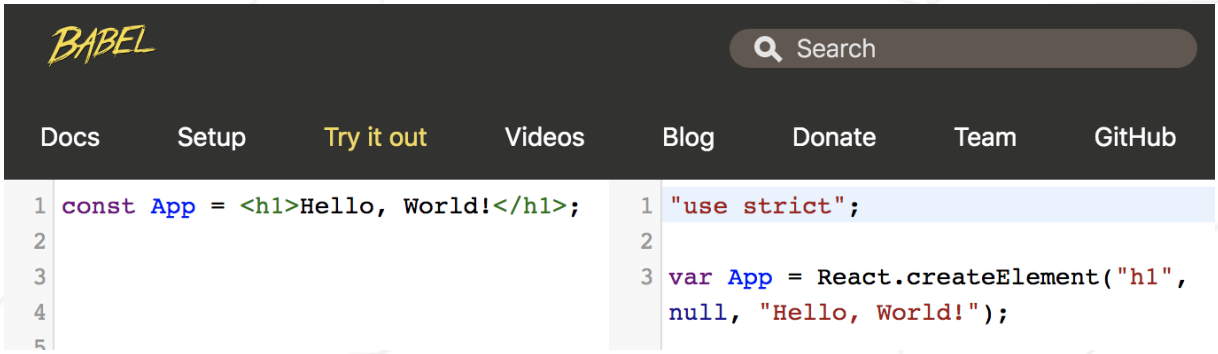
index.js

```
const App = <h1>Hello, World!</h1>
console.log("React Element: ", JSON.stringify(App));

ReactDOM.render(App, document.getElementById('root'));
```

Pull up the devtools (alt-command-j) in your browser and check the console to see what's printed. You'll see that the <h1> element is a Javascript object.

JSX looks like html but it gets compiled into Javascript before we run our app. Behind the scenes of our create-react-app, there's a transpiler called Babel. All Babel does is take our fancy, nice, syntactically sugared code and turns it into plain old Javascript that can run on any browser. Check out this Babel Repl and play around with it. Notice how the JSX code gets converted to the React.createElement method.



React elements make up the nodes of the Virtual DOM, the same way html elements make up the nodes in the real DOM. This is a big part of why React is so fast. Check out this video for a great visual explanation.

Chapter VI

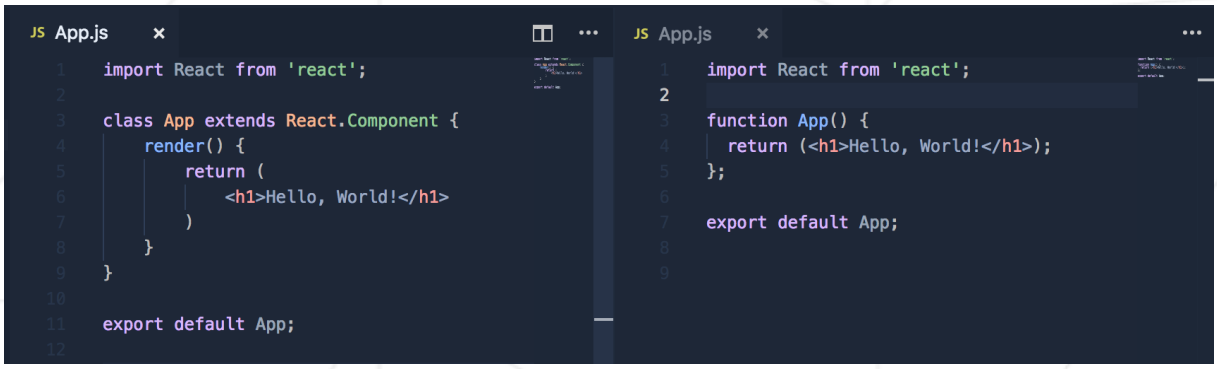
Exercise 01 : Components

One of React's key features is that it's component-based. This means we write modularized code, where we have separate pieces (the components) that can be put together to make complex UI's. Each of the pieces also has its own state, so we can store data specific to a component. The component logic is written in JS, so we can pass data easily throughout the app and keep state out of the DOM, where it's harder to pass around and expensive to update.

Check out the Thinking in React Step 1 section on component based thinking for an overview

VI.1 Making Our First Component

To start out, let's export our Hello World app into a component of its own. `mkdir src/components` to create the directory where we'll store out components. Then create a new file inside the components directory called `App.js`. This will be our root component. Inside this file, you want to import React. React components can be defined as classes or functions.
`Class` components need to return the root React element in their render method.
`Function` components need to return the root React element.
Components defined as classes have more features, but the two versions below do the same thing.



Let's use the class component for now, make sure your App.js file looks like the document on the left. Now we want to import this App.js file inside our index.js and pass that App component to our ReactDOM.render function. This is made easier by the `export default App;` line adn the end of the file, because it sets the default export to the App component. That's why we're able to `import App from './components/App';` instead of having to `import { App } from './components/App'`.

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

ReactDOM.render(<App />, document.getElementById('root'));

```

Now you can run `npm start`. You'll see our app looks the same as before, but now our code is modularized, and our App is a component of its own.

VI.2 Creating a Table Component

Next we'll be adding a <table> element that will be listing our todos. We'll set up some dumb data for the time being, eventually we'll be getting our data from the API we created in the NodeJS course. Let's change our header to label our app, our "To-Do List". Add a table element below the header. If you run your app, you should get an error:

Failed to compile

```
./src/components/App.js
  Line 7:  Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag.
Did you want a JSX fragment <>...</>?

   5 |         return (
   6 |                 <h1>To-Do List</h1>
>  7 |                 <table>
     |                 ^
```

It's important to remember that a component needs to return one parent element. You can't have adjacent elements, but this is easy to fix by wrapping the elements in a parent element. React provides `React.Fragment` for this, it's basically just a wrapper to group child elements under one parent component. After we wrap our code, your resulting component should look something like this:

App.js

```
class App extends React.Component {
    render() {
        return (
            <React.Fragment>
                <h1>To-Do List</h1>
                <table>
                    <thead>
                        <tr>
                            <th>Task</th>
                            <th>Completed</th>
                        </tr>
                    </thead>
                    <tbody>
                        <tr>
                            <td>Complete the NodeJS course</td>
                            <td>true</td>
                        </tr>
                        <tr>
                            <td>Complete Ex.01 of the React Course</td>
                            <td>false</td>
                        </tr>
                    </tbody>
                </table>
            </React.Fragment>
        );
    }
}
```

Now it's time for you to put into practice the new skills we just learned. Export the <table> element into its own component in the `components` directory. Name the file Table.js, and have it return our table element. Import the `Table` component into our App.js and put it where the table element is currently, below the header.

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. 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:

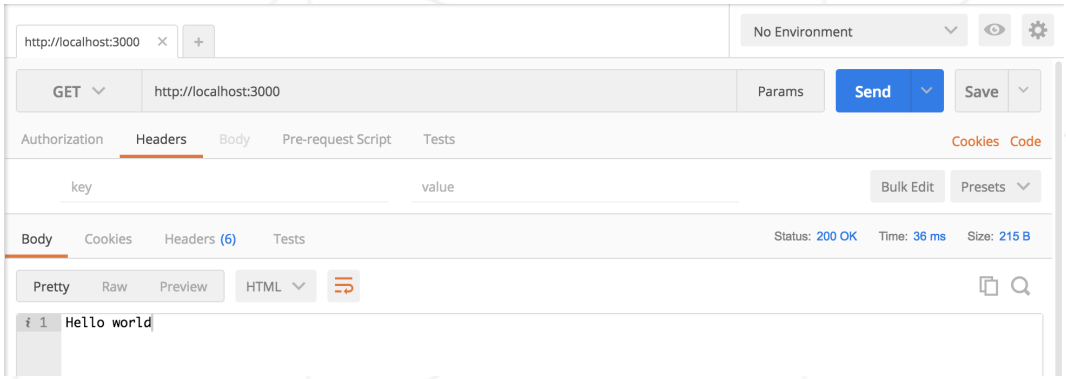
```
// Create new user
app.post("/users", (req, res) => {
    const { name } = req.body;
    if (!name) {
        res.status(400).send("Could not add user");
    }
    let user = {};
    user.name = name;
    user.id = db.length;
    db.push(user);
    // Convert DB to JSON string
    data = JSON.stringify(db);
    // 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

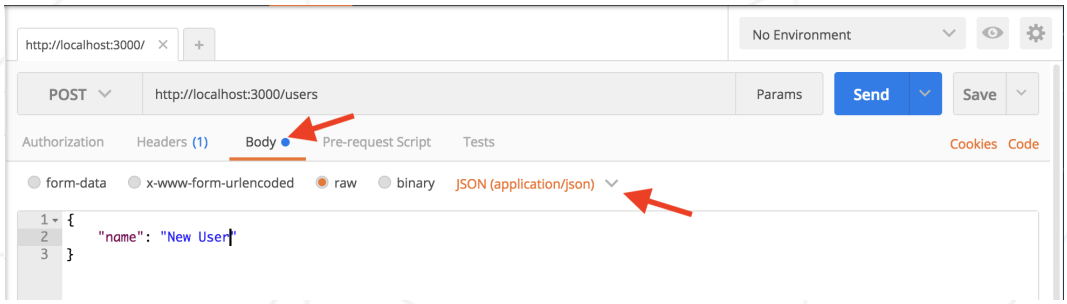
Exercise 03 : Testing with Postman

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:

```
{
  "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 as usual. Then we can add the following snippet to connect our API to MongoDB through

```
// 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:

```
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.

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

Head to the Getting Started section of the mongoose docs to learn more.

Check out the Mongoose model API docs

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.