Part 4

React.js with Node.js

Contents

Why React?

In order to make dynamic websites more responsive, React.js provides an application frontend framework that can create a single page application (although multiple page applications are possible) that are *mostly* rendered on the client itself.

This means that a page does not need to be refreshed – reloaded completely from the server – each time a change is made to a section of the page. Furthermore, changes can be made to specific sections of the page without effecting other sections.

Importantly, a React app maintains a virtual DOM (tree structure of the page's HTML) on the client side. This virtual DOM is updated first before the actual DOM, for example in response to an event such as a button click. This 2-stage allows React to compute the minimum amount of change needed to be made to the actual DOM. The actual DOM is then modified in a minimal way.

Usually in React apps we maintain different sections of the page as separate JavaScript functions (and sometimes classes, although in new React it is almost always functions), called *components* or *function components.*

A function component is a JavaScript function that returns HTML. This HTML may be rendered on the page by calling (or invoking) the component in a base JavaScript file *index.js*. Not every component is invoked directly in index.js. A component may be called in another component, and so on, resulting in a hierarchical structure. The index.js file is associated with a base html file, called index.html. This is the final, assimilated html file corresponding to our page.

React components can fetch specific pieces of data from the server – such as from a Node.js server – and incorporate it in the html they return.

Sarim Baig
May 2023

Some of the most important features of React, which you may want to learn before other features are: Components, Hooks, Events, and JSX. I only touch on them briefly in this introduction. Moreover, there are other essential and interesting features of React which I do not mention here. These can be read about in more detail from the w3Schools tutorial linked in the references, or from any other good source on the internet.

A note on the client server architecture in React:

Although this will become more clear as the tutorial progresses, it is useful to note at this point that React components (in most cases) reside on the client side.

In a deployed React application, these components are first downloaded from the server when the client connects with it for the first time. This process will be shown in the deployment section at the end of this intro.

Section 1 React.js frontend

Install Node and npm on your machine if you haven't already.

The following steps were tested on Mac's terminal. If you're using windows, there might be a few extra things needed to be done. See here:
https://learn.microsoft.com/en-us/windows/dev-environment/javascript/react-on-windows

STEP 1 Create and start a React App

>mkdir projects
>cd projects
>npx create-react-app my-react-app

Made a directory called projects.
Inside projects, created a react app called my-react-app.
Here, npx is the package runner used by npm to execute packages. In the command above, npx is using a tool create-react-app to create a React app called my-react-app

>cd my-react-app
>cd src
>npm start

This will start the react app. It is being served in this case by a node server running locally on 127.0.0.1 (localhost) port 3000. A welcome page should pop up in your browser.

At this point, both server and client are on the same machine. By the end of this tutorial, we will have the node server running remotely on ec2. The current server, on localhost, is a 'development server'. This server is useful for testing and development purposes. The app can be fully developed using this server, and then the server may be moved. React benefits from the use of a number of tools available with the node server, as we shall see.

You may stop your react app by pressing Ctrl + C.

STEP 2 Inspect the code

All React app files are present in the **my-react-app** folder.

We will first focus on the following two files in /my-react-app/src/

(a) index.js
(b) App.js

Open these files one by one in VS Code and strip the codes down to the following basic versions:

App.js

```
function App() {
  //The following piece of code uses JSX
  //Read about JSX here:
  //https://www.w3schools.com/react/react_jsx.asp
  return (
    <div className="App">
      <p>This text is printed inside the App component</p>
    </div>
  );
}
export default App;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <App />
);
```

Start the React app with *npm start* to see the output of the modified code.

App() is a React component. Like any React component, it returns HTML. Notice that the html has been written directly into the JavaScript code. This is not possible in simple JavaScript. We're using something called JSX, a JavaScript extension. In your React app folder you have all the necessary tools available to build and link JSX into JavaScript. The w3schools tutorial provides a good introduction to JSX.

The App component has been invoked in index.js in the root.render method, which means that he html returned by App will become the entire DOM of the page index.html (which uses index.js)

Note: The file index.js is linked to index.html in the configurations maintained as JSON files in the react app folder.

With App.js, index.js, and index.html we have a basic structure of a complete React app. The App component may define and/or call further components within it. Moreover, components other than App may be directly invoked in index.js, etc.

Step 3 Add another component

This section shows some more examples of components.

Example 1 Adding a 3x3 table component to the App component

App.js

```
function App() {
  function Table33() {
    return (
      <div className="App">
        <table>
          <tr>
            <th>Name</th>
            <th>Age</th>
            <th>Gender</th>
          </tr>
          <tr>
            <td>Ed</td>
            <td>19</td>
            <td>Male</td>
          </tr>
          <tr>
            <td>Mia</td>
            <td>19</td>
            <td>Female</td>
          </tr>
          <tr>
            <td>Max</td>
            <td>25</td>
            <td>Male</td>
          </tr>
        </table>
      </div>
    );
  }

  return (
    <div className="App">
      <Table33/>
    </div>
  );
}
```

Example 2 Making the table component more versatile

(i) We move the Table33 component to a separate file and make it available publicly.

(ii) We pass the contents of the table in parameters to the component.

Modified App.js and TableComp.js

App.js

```
import React from "react";
//import logo from "./logo.svg";
import "./App.css";
import Table33 from './TableComp';


//App is a React Function Component
//Read about it, and React Components in
general, here:
//https://www.w3schools.com/REACT/react_compo
nents.asp

function App() {

  let tableData33 =[
    ["Ed", "19", "Male"],
    ["Mia", "19", "Female"],
    ["Max", "25", "Male"]
  ];


  return (
    <div className="App">
      <Table33 td = {tableData33}/>
    </div>
  );
}
export default App;
```

TableComp.js

```
//React props are used to pass parameters
//to reach components
//Read more about React Props here
//https://www.w3schools.com/react/react_p
rops.asp
function Table33(props) {
    return (
      <div className="Table33">
        <table>
          <tr>
            <th>Name</th>
            <th>Age</th>
            <th>Gender</th>
          </tr>
          <tr>
            <td>{props.td[0][0]}</td>
            <td>{props.td[0][1]}</td>
            <td>{props.td[0][2]}</td>
          </tr>
          <tr>
            <td>{props.td[1][0]}</td>
            <td>{props.td[1][1]}</td>
            <td>{props.td[1][2]}</td>
          </tr>
          <tr>
            <td>{props.td[2][0]}</td>
            <td>{props.td[2][1]}</td>
            <td>{props.td[2][2]}</td>
          </tr>
        </table>
      </div>
    );
  }
  export default Table33;
```

Note the use of export default in TableComp.js and import Table33 in App.js

At the moment, tableData33 is a local variable in App. Later, we will fetch this data from the server.

(iii) We make the table component more versatile by allowing it to be of any dimensions of data supplied in the props. In the process we add two more components and use JavaScript's map.

App.js

```
import React from "react";
import TableComp from './TableComp';
//App is a React Function Component
//Read about it, and React Components in
general, here:
//https://www.w3schools.com/REACT/react_compo
nents.asp

function App() {

  let tableData33 =[
    ["Ed", "19", "Male"],
    ["Mia", "19", "Female"],
    ["Max", "25", "Male"]
  ];


  return (
    <div className="App">
      <TableComp td = {tableData33}/>
    </div>
  );
}
export default App;
```

TableComp.js

```
///React props are used to pass
parameters
///to reach components
///Read more about React Props here
//https://www.w3schools.com/react/react_p
rops.asp


function TableComp(props) {

///Map each col to its corresponding HTML
    function TableCols(props){
        const cols = props.tr.map(
            (col) =>
            <td>
                {col}
            </td>
        )

        return cols;
    }

 //Map each row to its corresponding HTML
    function TableRows(props){
        const rows = props.td.map(
            (row) =>
            <tr>
                <TableCols tr = {row}/>
            </tr>
        )

        return rows;

    }

    return (
      <div className="TableComp">
        <table>
          <TableRows td={props.td}/>
        </table>
      </div>
    );


  }
```

```
export default TableComp;
```

Note we have renamed the component to TableComp

JavaScript's map is used to map each element of a list or array to a different value according to a rule. In this example it has been used to map each row in props.td and then each element in the row to their corresponding HTML (JSX).

map is often used to convert data to its corresponding JSX in React components.

Read more about map here.

Step 4 Use React states, hooks and events

A React component may have an associated **state**: data maintained by the component that needs to be tracked for change. The special thing about state is that whenever it is changed React re-renders all appearances of it on the page. This does not happen entirely automatically but React provides an easy mechanism for it.

Hooks are library functions that are needed to connect (or 'hook') variables defined in components to the state of the component. The most fundamental hook is **useState**

In terms of events in React, we can perform actions based on user **events** such as button clicks, typing in edit box, mouse hover, etc.

Example 1 Adding a state, hook and event to the table example

We make changes to App.js only:

App.js

```
// client/src/App.js

import React, {useState}  from "react";
import TableComp from './TableComp';


//App is a React Function Component
//Read about it, and React Components in general, here:
//https://www.w3schools.com/REACT/react_components.asp

function App() {
  const [tableData33, updateTable33] = useState(
  [
    ["Ed", 19, "Male"],
    ["Mia", 19, "Female"],
    ["Max", 25, "Male"]
  ]); //useState hooks tableData33 to the state
```

```
    //It also provides a callback updateTable33
    //which must be invoked to transmit changes to the state




  ///Randomize ages between 15 and 50
  function randomize(){
    return (
      [
        [tableData33[0][0],15 + Math.floor(Math.random() * 35),tableData33[0][2]],
        [tableData33[1][0],15 + Math.floor(Math.random() * 35),tableData33[1][2]],
        [tableData33[2][0],15 + Math.floor(Math.random() * 35),tableData33[2][2]]
      ]
    );
  }

  return (
    <div className="App">
      <TableComp td = {tableData33}/>
      <button onClick={() => updateTable33(randomize())}>Randomize ages</button>
    </div>
  );
}
export default App;
```

In this example,
useState returns two things: an object called tableData33 and a function callback updateTable33 (names are not important).

updateTable33 is the function that receives a possibly modified value of tableData33 in the parameters and updates the state and the virtual DOM accordingly.

The intermediate function randomize is not necessary, but it makes the code more readable in this case.

The onClick attribute of the <button> tag is where we assign an event handler. In this case, the callback updateTable33 has been used as the event handler. However, in general it can be any function.

State management using hooks is versatile and powerful. Learn more about Hooks here: https://www.w3schools.com/react/react_hooks.asp

Section 2 Node.js backend

So far, we've done everything on the client. Now we will run a Node web server separately for the backend. We'll run this server on localhost:3001.

So, we will have two servers running on localhost. On port 3000: a Node server that helps us develop the frontend (React app) by using Node tools. On port 3001: a Node server that helps us develop the backend independently. These are two different development servers for the frontend and backend respectively. Eventually, there will be just the one backend server running on the cloud, while a fully developed frontend app runs on the client's browser (the frontend app will no longer require a Node server running locally).

See tutorial 3 to learn about Node and express.

Use the following commands to create and run the server:

> mkdir backend_server
> cd backend_server
> npm init -y
> npm i express

We have initialized node and installed express as a dependency.

Go to the backend_server folder and add a file called index.js with the following content:

index.js

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

const PORT = process.env.PORT || 3001;

const app = express();

app.get("/tableData33", (req, res) => {
  res.json(
  {
      "tableData33":[
        ['Ed', 15 + Math.floor(Math.random() * 35), 'Male'],
        ['Mia', 15 + Math.floor(Math.random() * 35), 'Female'],
        ['Max', 15 + Math.floor(Math.random() * 35), 'Male']
      ]
  })
  });

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
});
```

Open backend_server/package.json and add a line to scripts as shown:

"scripts": {
  "start": "node server/index.js"
},

Now run the command:
> npm start

Enter the following in your browser's address bar to access the end-point tableData33:
http://localhost:3001/tableData33

Output will be like the following, with random age values:
{"tableData33":[["Ed",19,"Male"],["Mia",29,"Female"],["Max",18,"Male"]]}

Leave the server running.

Section 3 React.js frontend with Node.js backend

We'll be adding code to App.js on the frontend and index.js on the backend. The effect of the additional code will be:

(i) an initial table with random age values will be fetched from the server.
(ii) on each click of the 'Randomize ages' button, table data with random age values will be fetched from the server.

We'll be using the fetch API for this purpose.

Following are the modified App.js and index.js.

Read comments in the code.

Note: on the server side, install cors:
> npm i cors express nodemon

App.js (frontend)

```
// client/src/App.js

import React, {useState}  from "react";
import TableComp from './TableComp';


//App is a React Function Component
//Read about it, and React Components in general, here:
//https://www.w3schools.com/REACT/react_components.asp

function App() {

  const [tableData33, updateTable33] = useState([[]]);


  //Get initial input from the server
  //Whenever App is invoked
  //a 'side effect' is that
  //the initial table data is fetched from the server
  //useEffect is a hook used to associate such 'side effects'
  //with components
  React.useEffect(() => {
    ///See CORS
```

```
    fetch("http://localhost:3001/tableData33/")
      .then((res) => res.json())
      .then((data) => updateTable33(data.tableData33))
      .catch((err) => alert(err)
       );
  }, [updateTable33]);


  //handleClick is our event handler for the button click
  const handleClick = (updateMethod) => {
    fetch("http://localhost:3001/tableData33/")
      .then((res) => res.json())
      .then((data) => updateMethod(data.tableData33))
      .catch((err) => alert(err)
       );
  };

  return (
    <div className="App">
      <TableComp td = {tableData33}/>
      <button onClick={() => handleClick(updateTable33)}>Randomize
ages</button>
    </div>
  );
}
export default App;
```

index.js (backend)

```
const express = require("express");
const cors = require('cors');

const PORT = process.env.PORT || 3001;
const app = express();

app.use(cors({
    origin: '*'
}));

app.use(cors({
  methods: ['GET','POST','DELETE','UPDATE','PUT','PATCH']
}));

app.get("/tableData33", (req, res) => {

  res.json(
  {

      "tableData33":[
        ['Ed', 15 + Math.floor(Math.random() * 35), 'Male'],
        ['Mia', 15 + Math.floor(Math.random() * 35), 'Female'],
        ['Max', 15 + Math.floor(Math.random() * 35), 'Male']
      ]
  })

  });

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
});
```

What's new in **App.js**:

A hook called **useEffect** has been used. This function is triggered every time the component which contains it, in this case App, is invoked. useEffect accepts two parameters. The first is a function to be executed when useEffect is triggered. The second is a list of dependencies, i.e., any state or props object of the component that have been used in the function in the first parameter. In this case, we are using the function updateTable33 hence it has been passed in the dependency list to useEffect. It is said that useEffect adds 'side effects' to the components, i.e., things that happen on the side as the main rendering functionality of the component takes place.

In useEffect as well as handleClick (our handler for the button click), we're using the fetch API. It is important to briefly note how it works beyond the obvious fact that it fetches data from the server using the api endpoint supplied in the parameter.

A call to fetch returns an object of type Promise. A detailed but simple explanation of Promises and the rationale behind them is available here. Put simply, a Promise is an object sent from a 'producer code' to a 'consumer code'. In our case, the consumer code is the useEffect method on the client; the producer code is the /tableData33 end point on the server. A Promise object has one of three possible states: 'pending,' 'fulfilled,' 'rejected'.

Let's consider our case of the Promise object returned by fetch. It will obviously take some time for an HTTP request to travel from the client to the server, the processing to take place at the server and then the response to arrive back at the client. During this time, the state of the Promise remains 'pending'. After that, the Promise object resolves; either the then or catch method of the Promise object is triggered. In particular, if the state of the promise changes from 'pending' to 'fulfilled', i.e., the fetch is successful, the then method is triggered. The catch method is triggered instead if the state changes from 'pending' to 'rejected', i.e., the fetch is unsuccessful.

We can also add a finally clause after the catch. The finally method, when present, is always triggered, and maybe used for any clean-up work.

It should be clear from the code that the parameters to the then and catch methods are JavaScript functions. These are our own callbacks, or handlers, that will execute in the case of success or failure. Notice that in our example there is a chain of two thens before a catch. This is because the first then returns another Promise: the extraction of json from the res (response from the server) object may be successful or unsuccessful. We end up in the second then if json data was successfully extracted from res. In the second then, our callback receives this data as its parameter and uses it to update the state.

Exactly the same process takes place in handleClick.

In catch I have used alert, however, you may choose to log the error or do anything else that helps you debug your code more effectively.

It should be noted that the purpose of Promise is to facilitate asynchronous programming. Consider the following scenario:

The app component is invoked. The useEffect method is triggered. Data is being fetched from the server, however, it takes rather long (suppose we have added a 6 second delay on the server side). Meanwhile, you click the button 'Randomize ages'. Let us suppose, for the sake of the point I'm trying to make, that the button click simply displays a table of all zeros. Then, the Promise methodology will allow the button click to perform this task, which will happen quickly because it happens entirely on the client side, while the result from the server side is still being awaited. In this case, useEffect has been allowed to work as an asynchronous action, i.e., an action that we initiate now but it finishes later, allowing other actions to take place in the meantime. This is an important capability because a delay on the server side could have the effect of jamming the client app if it takes too long and nothing else is allowed to happen in the meantime.

If you wish to try the experiment in the above scenario, modify the codes in App.js (frontend) and index.js (backend) as follows (only relevant sections are shown below). As soon as the page loads, press the button and then wait.

Code to demonstrate asynchronous action:

App.js

```
//handleClick is our event handler for the
button click
  const handleClick = (updateMethod) => {
    /*

fetch("http://localhost:3001/tableData33/")
      .then((res) => res.json())
      .then((data) =>
updateMethod(data.tableData33))
      .catch((err) => alert(err)
       );
       */
      updateMethod([[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]
                  ]);
  };
```

index.js

```
app.get("/tableData33", (req, res) => {

  setTimeout(() =>{
  res.json(
  {
      "tableData33":[
        ['Ed', 15 +
Math.floor(Math.random() * 35), 'Male'],
        ['Mia', 15 +
Math.floor(Math.random() * 35), 'Female'],
        ['Max', 15 +
Math.floor(Math.random() * 35), 'Male']
      ]
  })}, 6000);
  });
```

What's new in **index.js**:

The fetch API by default uses CORS: Cross Origin Resource Sharing. This is an access restriction mechanism on top of HTTP, which restricts access to resource requests coming from different domain/port than the server's own. In index.js, in app.use, we have set origin: '*' which effectively undoes the use of cors, by allowing anyone access to the recourses. However, instead of '*' we could have listed specific hosts to allow them access. Any request from an unlisted host would be denied.

Section 4 Deploying the React App

So far, we've been having two development servers. One at localhost:3000 for frontend development. The other at localhost:3001 for backend. At this point our frontend app is complete. We can move it to localhost:3001, that will serve it to any client. localhost:3000 is no longer needed.

This can be done in a few steps:

Step 1 Build the frontend app
Navigate to my-react-app and run the following command:
>npm run build

This will create an optimized build for the app. A folder called build will appear inside my-react-app.

Step 2 Remove/move folders

- Delete the node_modules folder from my-react app.
- Delete the .git folder if there is one.
- For simplicity (though this is not a requirement) rename the my-react-app folder to client.
- Move the client folder to the backend_server folder

Step 3 Add deployment related code to index.js (backend)
Modify index.js to contain the following content:

index.js

```
const express = require("express");
const cors = require('cors');
const path = require('path');
const PORT = process.env.PORT || 3001;
const app = express();

app.use(cors({
    origin: '*'
}));

app.use(express.static(path.resolve(__dirname, './client/build')));

app.use(cors({
  methods: ['GET','POST','DELETE','UPDATE','PUT','PATCH']
}));

app.get("/tableData33", (req, res) => {

  setTimeout(() =>{
  res.json(
  {

      "tableData33":[
```

```
        ['Ed', 15 + Math.floor(Math.random() * 35), 'Male'],
        ['Mia', 15 + Math.floor(Math.random() * 35), 'Female'],
        ['Max', 15 + Math.floor(Math.random() * 35), 'Male']
      ]
  })}, 8000);

  });

  app.get('*', (req, res) => {
    res.sendFile(path.resolve(__dirname, './client/build', 'index.html'));
  });

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
```

This code first allows Node to access our built React project using the express.static function for static files.

The get.('*'... end-point is used by any request made to http://localhost:3001 and our server responds by serving the React app.

Step 4 Test the app
Run the backend server

In your browser, type in localhost:3001and enter.

Your React app should pop up.

Further Reading / References

(1) **React.js tutorial on w3schools**
https://www.w3schools.com/REACT/default.asp

(2) **How to Create a React App with a Node Backend: The Complete Guide (freeCodeCamp)**
https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/

(3) **How to use CORS in Node.js with express**
https://www.section.io/engineering-education/how-to-use-cors-in-nodejs-with-express/