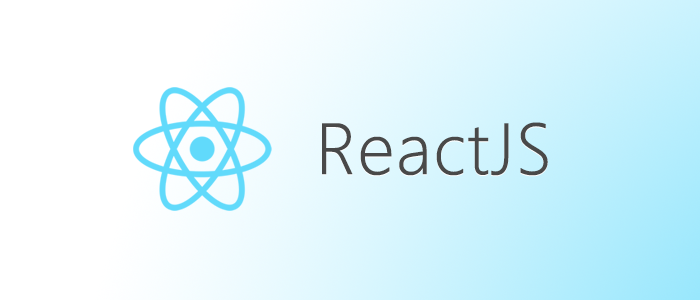
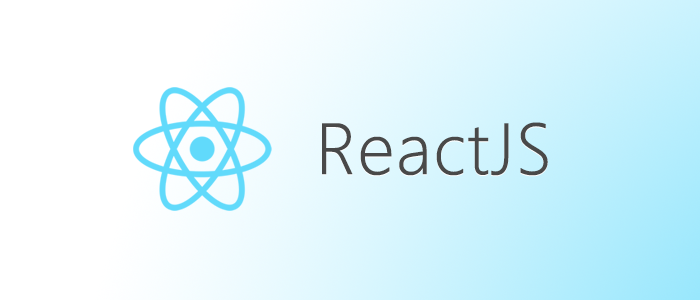
**Exercise07\_01\_01 – Step 1**

1. Create a project folder named ***Exercise07\_01\_01***. Open the project folder with your IDE. Using your terminal, we will first use ***npm init*** to initialize a Node.js project with a ***package.json***. The responses should all be the defaults except the following:  
   ***package name: fullstack  
   description: React.js Full Stack  
   entry point: index.js  
   author: Yourself***Check your package.json for correctness in the IDE.
2. Now let’s install our initial basic project ***dependencies***. Use npm as follows, either in a terminal or in your IDE:  
   ***npm install express –save  
   npm install mongodb –save***Check your package.json for correctness in the IDE.
3. Now let’s install our React.js project ***dependencies***. This will consist of the basic ***react*** library, along with the ***react-dom*** module, which will be used to render user ***interfaces***. It works on both the ***front-end*** and ***back-end***:  
   ***npm install react react-dom –save***Check your package.json for correctness in the IDE.
4. Now let’s install our project ***development dependencies***. We will need ***webpack*** as a bundler to transform code into something that all browsers will understand. It will use ***Babel*** as a loader to transform JSX extension code for React, as well as our ES6 that might not yet be supported, so we will need a few of the Babel preset packages as well:  
   ***npm install webpack webpack-cli -save-dev  
   npm install @babel/core @babel/node-@babel/preset-env –save-dev  
   npm install @babel/preset-react–save-dev  
   npm install react @babel/plugin-proposal-class-properties–save-dev***Check your package.json for correctness in the IDE.
5. Now let’s install a really interesting ***development dependency***. We will use ***nodemon*** as a utility for node.js that can monitor changes and automate node.js server restarts for us:  
   ***npm install nodemon -save-dev***Check your package.json for correctness in the IDE.
6. Now we need a few more ***development dependencies***. That will give us the eslint support we will need. This may need to be run as an administrator in the command prompt:  
   ***npm install eslint –save-dev  
   npm install eslint-plugin-react –save-dev  
   npm install babel-eslint –save-dev***Check your package.json for correctness in the IDE.
7. Download the ***Exercise07\_01\_01.zip*** file and unzip the contents into the ***Exercise07\_01\_01*** project folder at the top level. Included should be ***.eslintrc.js***, ***.babelrc***, and ***webpack.config.js***. Open the IDE and set up the following folders and files at the top level. ***/public*** with ***index.html***, ***/src*** with ***index.js***, ***/api*** with ***index.js***.
8. Open ***package.json*** and we will prepare some ***npm*** ***scripts***. First remove the ***test*** script. First let’s add a specialized ***npm start*** script. It will run ***nodemon***, which will execute ***babel-node*** instead of node so our server understands JSX. We willS ignore the files in the ***/public*** directory when it is monitoring. Enter the script as follows:  
    ***"scripts": {  
    "start": "nodemon --exec babel-node server.js --ignore public/",  
    },***
9. Now let’s add a ***dev*** script just below it. It will run ***webpack*** and the parameter sets it to run in ***watch*** mode and ***development*** mode:   
    ***"dev": "webpack –wd”***
10. Examine the ***webpack.config.js*** file. This instructs webpack to start from ***/src/index.js*** file. We are going to have it ***bundle*** all of our modules into the ***/public*** directory. The ***loaders*** property instructs webpack to run the ***babel-loader*** on all .js files to transform the JSX and ES6 to be compatible with all browsers.
11. Examine the ***.babelrc*** file. The instructions have Babel use two ***presets***: ***env*** gets the runtime environment, and ***react*** transforms the JSX code and the latest ES6. Open the ***.eslintrc.js*** file, and you will find the eslint settings. I set my indent to 4.

**Exercise07\_01\_01 – Step 2**

1. Let’s build out our server with ExpressJS. We can start off by building a configuration file to use at the top level of the project, named ***config.js***:  
   ***const env = process.env;  
     
   export const nodeEnv = env.NODE\_ENV || 'development';  
     
   export const logStars = function(message) {  
    console.info('\*\*\*\*\*\*\*\*\*\*');  
    console.info(message);  
    console.info('\*\*\*\*\*\*\*\*\*\*');  
   };  
     
   export default {  
    port: env.PORT || 8080  
   };***
2. Now we can start to build out our server at the top level in ***server.js***. The import command caches, so multiple imports in multiple files, do not repeat:  
   ***import config from './config';  
   const express = require('express');  
   const server = express();  
     
   server.listen(config.port, () => {  
    console.info('Server listening on port: %s', config.port);  
   });***Give this a test by running ***npm start***. This should run our server under ***nodemon***, which will both check for bugs and bounce our server on changes.
3. Let’s build a server ***endpoint*** for testing:  
   ***server.get('/', (req, res) => {  
    res.send("Hello from the server!");  
   });***When the file is saved, the server should bounce for us. Bring up a browser and go to ***localhost:8080*** to test the endpoint.
4. Let’s build a second server ***endpoint*** for testing the ***About*** page:  
   ***server.get('/about.html', (req, res) => {  
    res.send("This is the about page!");  
   });***When the file is saved, the server should bounce for us. Bring up a browser and go to ***localhost:8080/about.html*** to test the endpoint.
5. Now we can scaffold out a simple web page at the top level named ***about.html***:  
   ***<!doctype html>  
   <html>  
   <head>  
    <title>About This Project</title>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="initial-scale=1.0">  
   </head>  
     
   <body>  
    <h1>About This Project</h1>  
    <p>This is the About page of the project</p>  
   </body>  
   </html>***
6. Let’s play around with how Node.js can use the file system to read this page. This will be temporary so we can recall how node uses the file system. We will use the Node.js ***fs*** core module:  
   const server = express();  
   ***const fs = require('fs');***
7. Now let’s modify the code in our route to use the module:  
   server.get('/about.html', (req, res) => {  
    ***fs.readFile('./about.html', (err, data) => {  
    res.send(data.toString());  
    });  
   });***Refresh the page with the URL ***localhost:8080/about.html*** in the browser to test the code.
8. Let’s do this a better way by mounting some middleware. First, move ***about.html*** to the ***/public*** folder. Let’s also remove the ***require(‘fs’)*** statement, as we will not be needing it. Now we can remove the ***about.html*** endpoint, and mount some ***middleware*** that can serve our ***static*** files:  
   ***server.use(express.static('public'));***  
   ***server.listen(config.port, () => {***Refresh the page with the URL ***localhost:8080/about.html*** in the browser to test the code.
9. Now let’s modularize a little bit and split off our routing in a different way. We can handle a group of routes in a different module. Open up the ***/api/index.js*** file, and we will implement the ExpressJS ***router***. We implement a ***get()*** call with an empty object as debug. We will ***export*** that router so we can use it in our server as follows:  
   ***const express = require('express');  
   const router = express.Router();  
     
   router.get('/', (req, res) => {  
    res.send({ data: [] });  
   });  
     
   export default router;***
10. Now let’s import the router into ***server.js***:  
    import config from './config';  
    ***import apiRouter from './api';***
11. We can mount the router as middleware in server.js:  
    ***server.use('/api', apiRouter);  
    server.use(express.static('public'));***Refresh the page with the URL ***localhost:8080/api*** in the browser to test the router and we should see our debug empty object.

**Exercise07\_01\_01 – Step 3**

1. First we will need to install the EJS package for node as follows:  
   ***npm install ejs -save***
2. Let’s implement ***EJS*** as our template language. First, let’s enable it in ***server.js*** as follows:  
   const server = express();  
    ***server.set('view engine', 'ejs');***
3. EJS looks for templates in a ***/views*** folder, so let’s build that at the top level and build a new file in it called ***index.ejs***. Scaffold a new HTML template into it as follows:  
   ***<!doctype html>  
   <html>  
   <head>  
    <title>Naming Contests</title>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="initial-scale=1.0">  
   </head>  
   <body>  
     
   <body>  
   </html>***
4. To test it, let’s put in some EJS tags with a little bit of debug:  
   <body>  
    ***<%= Math.random() %>***<body>
5. Let’s modify our ***get()*** route in ***server.js*** to render the template:  
   ***server.get('/', (req, res) => {  
    res.render('index');  
   });***Bring up a browser and go to ***localhost:8080*** to test the endpoint. We should see our debug from the EJS template.
6. Now we will try to pass some ***content*** into the template as follows:  
   server.get('/', (req, res) => {  
    ***res.render('index', {  
    content: “Hello Express and EJS!”  
    })***});
7. We will now modify the ***index.ejs*** EJS tag to use the content:  
   <body>  
    ***<%= content %>***<body>  
   Regenerate the browser and we should see the new content.
8. Let’s add some HTML to the ***content*** as follows:  
    res.render('index', {  
    content: 'Hello Express and ***<em>EJS</em***>!'  
    })  
   Regenerate the browser and we should see the new content and see the HTML tags, as the HTML has been ***escaped***.
9. We will now modify the ***index.ejs*** EJS tag to ***unescape*** the content with an EJS tag modification, which will render the HTML:  
   <body>  
    ***<%- content %>***<body>  
   Regenerate the browser and we should see the new content with the HTML rendered.
10. Now let’s make use of EJS ***partials***. Create a new file in ***/views*** named ***header.ejs***. In index.ejs, cut out the code from the top through and including the opening ***<body>*** tag and paste it into ***header.ejs*** as follows. Change the <title> to ***Hello EJS***:  
    ***<!doctype html>  
    <html>  
    <head>  
     <title>Hello EJS</title>  
     <meta charset="UTF-8">  
     <meta name="viewport" content="initial-scale=1.0">  
    </head>  
    <body>***
11. Now add the following EJS tag to the top of index.ejs:  
    ***<%- include('header') %>*** <%- content %>  
    <body>  
    </html>  
    Regenerate the browser and we should see the same content with the partial now being used.
12. Let’s do the same thing with the footer, by making a file ***footer.ejs***. Cut out the bottom part of index.ejs and place it in the file as follows:  
    ***</body>  
    </html>***
13. Now let’s put another ***include*** statement into index.ejs:  
    ***<%- include('footer') %>***Regenerate the browser and we should see the same content with another partial now being used.
14. Let’s add a ***Bootstrap stylesheet*** into ***header.ejs***, which I will put into ***Slack*** for you. Also add a class container to the <body> element:  
     ***<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">***</head>  
    <body ***class="container"***>  
    Give this a browser test and we should see font and positioning differences.