Lecture Notes: React Part I and Part II

Introduction to React

In this module, you will learn about ReactJS, which is one of the most popular JavaScript libraries. You will develop an understanding of all the basic concepts of this library while building a single-page application, **Phone Directory**, which would maintain a list of subscribers and allow you to add or delete a subscriber's details. Later in the next module, you will learn how to migrate the **Phone Directory** application from class-based components to function-based components using **React Hooks**. After building the front end of this application using ReactJS, you will learn how to **integrate** it with the back end and the database.

React JS: "React is a JavaScript library for building user-interfaces."
React has been designed in such a way that it can be adopted gradually. You are free to use React in an application to the extent of your requirements. You can use as little of it as required or as much as you want of it.

History of React

React was developed by the engineers at Facebook and was initially used in their Newsfeed feature. Today, numerous features of Facebook are backed by React. React has also popularly grown significantly over the years. Apart from Instagram, React is being used by many other applications.

Advantages of React

Many developers across the world are using React today. React is the foundation of some of the popular businesses as well. So, if React is so popular and is adopted by so many companies, it must have many advantages to offer. Some of these advantages can be summarised as follows:

- React helps to build interactive user-interfaces called web applications.
- With React, you can use the latest features introduced in the JavaScript language.
- React does not require you to rewrite or ship your existing code in it.
- React follows the concept of Virtual DOM, which makes DOM manipulation super fast and easy.
- React uses a huge ecosystem of open-source libraries that can be used to perform relevant tasks.

React Environment: Role of Babel

React follows the **ECMAScript** specification. You will be using the **ES6** edition of the ECMAScript specification while writing React code. This edition brought some important features and updates in the JavaScript language. We recommend you use these features while writing code. However, as mentioned before, not all the features of ES6 are supported by many browsers. Most browsers still support ES5. So, if you code in ES6, you can use the latest features, but all browsers may not support them. This is where Babel comes into the picture.
Babel is a JavaScript compiler that converts ES6 code into ES5 code, which can be run on any browser.

React Environment: Role of Webpack and ESLint

Webpack

Bundling reduces the number of HTTP requests sent by the client to the server.

For example, suppose a web page references 10 stylesheets and 20 script files, which is a total of 31 files (= 1 web page + 10 stylesheets + 20 scripts) to be referenced. A batch size is the number of files that a browser can reference in parallel. Considering this batch size to be 6, you need to make a total of 31/6, which is equal to 6 (ceil of 5.17) HTTP requests.

If you use the bundling technique, you can bundle all the stylesheets into one file and all the scripts into one file. Thus, after bundling, you would need to reference 3 files, which is equal to the sum of 1 web page + 1 bundled stylesheet (containing all 10 stylesheets) + 1 bundled script (containing all 20 scripts). Now, with an ideal batch size as 6, you would need to make only 3/6, which is equal to 1 (ceil of 0.5) HTTP request. Therefore, based on this example, we can conclude that bundling reduces the number of HTTP requests from 6 to 1.

Also, **minification helps in the process of shortening a file's contents**, resulting in **faster response time** and **lower bandwidth cost.** Webpack is an open-source JavaScript module bundler that bundles your files into a single file in a React application.

ESLint

ESLint helps you follow the coding guidelines and principles while making you commit fewer errors and mistakes.

For instance, consider the following code snippet:

```
var name = "UpGrad";
var name = "UpGrad Education";
console.log(name);
```

When you write this code in a React application that has ESLint configured in its environment, you can view the following message:

```
'name' is already defined no-redeclare
```

no-redeclare is one of the **rules in ESLint**. Such rules are applied to many different code style use cases.

If you do not want to specify your own set of rules every time you write code, there are plenty of recommendations that you can utilise. One such recommendation is specified in the *Airbnb Style Guide*. Airbnb provides a bunch of ESLint presets that cover ES6, JSX, etc., which makes it a great choice for React projects. It also helps by providing performance tips. Airbnb open-sourced its own ESLint configuration so that it can be used by anyone.

Single-Page Application vs Multi-Page Application

Advantages of SPAs over MPAs

1. Faster loading of page; no need of downloading resources all over again
2. Effective caching; easy local data storage
3. Easy debugging; technologies provide their own debugger tools
4. Decoupling of front end and back end
5. Simplified mobile development; the same back end can be used for web applications as well as native mobile applications
6. Rich in responsiveness; better user experience

Advantages of MPAs over SPAs

1. Better search engine optimisation (SEO); architecture is native to search engine crawlers; flexibility to add meta tags to each page
2. Better in terms of analytics
3. Unlimited scalability; new features can be added easily
4. JavaScript not mandatory
5. Better security; access control at a functional level

Each architecture has its pros and cons and is well-suited to a particular type of project and specific business goals. It is recommended that you choose the architecture according to your business needs and requirements.

Understanding React Concepts: Part I

Understanding React Concepts: Part I and **Understanding React Concepts: Part II** are aimed at introducing to you all the basic concepts of React. These concepts will serve as the foundation for your understanding of React.

Particularly, in this session, you will learn how to set up your environment to run a React application. You will set up your codebase to create the Phone Directory application. You will also gain an understanding of the following concepts in React and implement the same inside this application:

- JSX
- Differences between JSX and HTML
- Injecting data using curly braces {} in React
- React.createElement() method
- Rendering elements into DOM in React
- Components
- Styling components and elements in React
- Rendering content dynamically in React

Codebase Set-Up: Text Instructions (Windows)

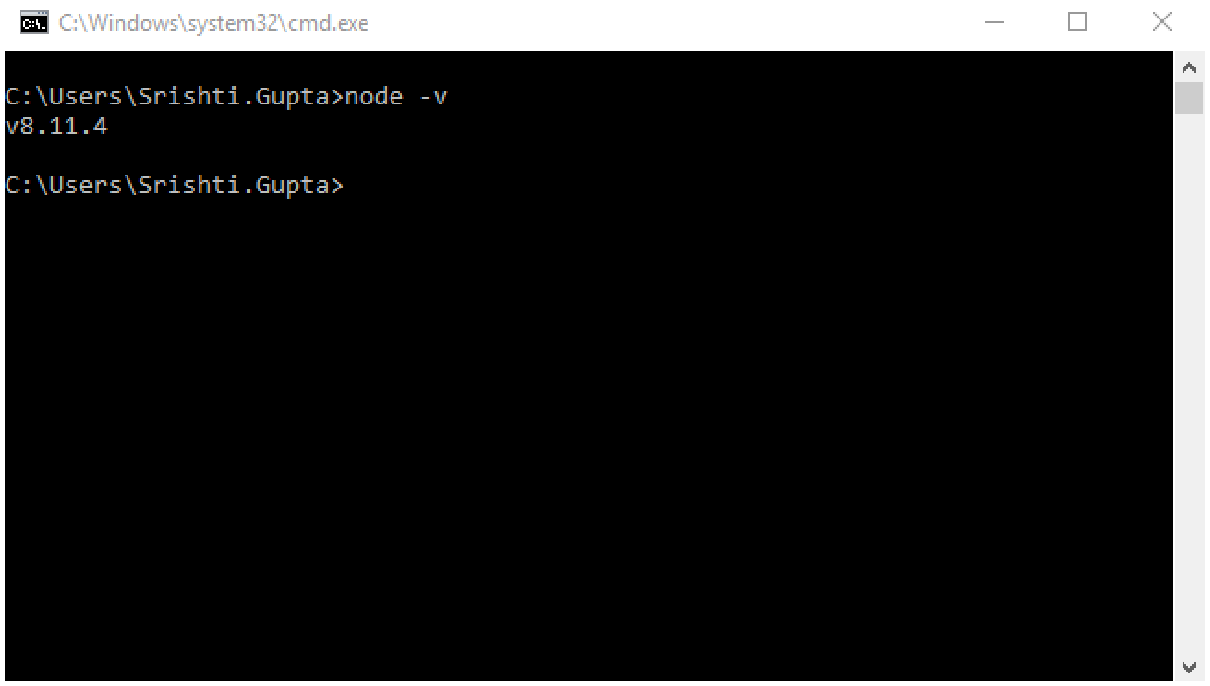
You can set up a React application using the following command-line instructions:

Instructions for Windows

Step 1: In the search bar, type ‘Command Prompt’. Click on the first search result to open it. Alternatively, press Win+R. In the Run window that pops up, type ‘cmd’ inside the open text field and press OK.

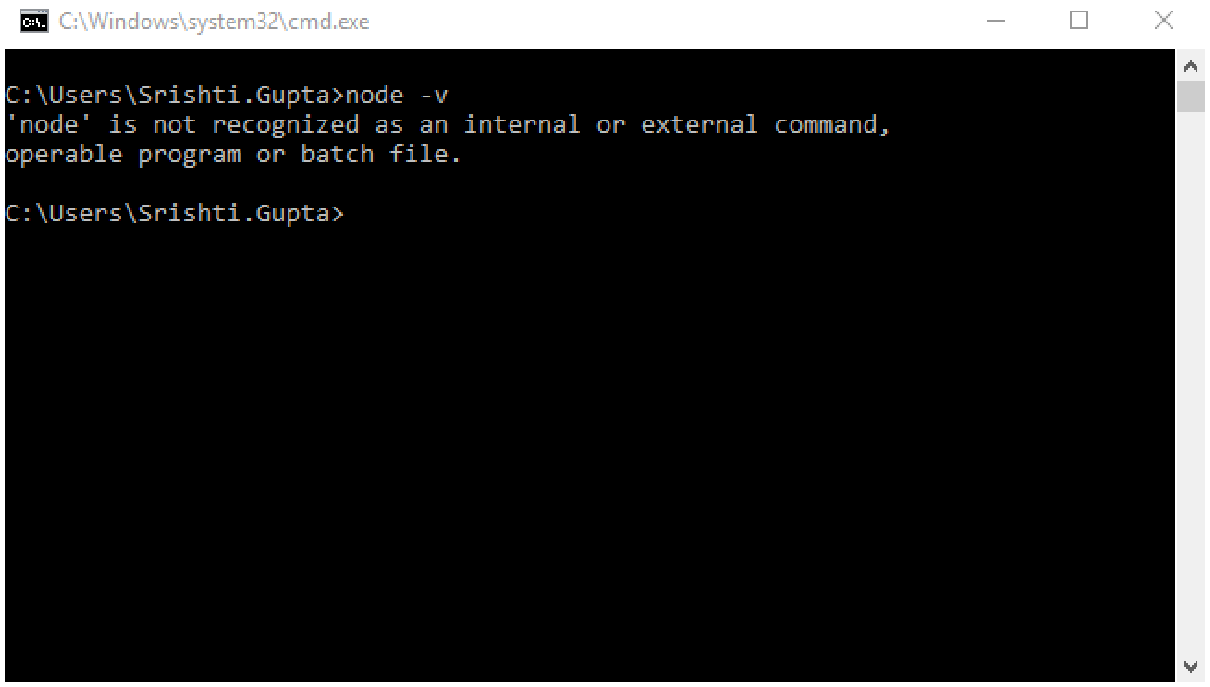
Step 2: Check if Node.js is installed on your machine by writing the following command:
node -v

If any version is returned to you, it means that *Node.js* is installed on your machine. If this version is greater than 6, you can skip steps 3 and 4 and directly jump to proceed to Step 5:


C:\Windows\system32\cmd.exe

```
C:\Users\Srishti.Gupta>node -v
v8.11.4

C:\Users\Srishti.Gupta>
```

If the version is not greater than 6 or if you see the message "*'node' is not recognized as an internal or external command, operable program or batch file.*", it means that Node.js is not installed on your computer, so you can close your *Command Prompt* and proceed to the next step.


C:\Windows\system32\cmd.exe

```
C:\Users\Srishti.Gupta>node -v
'node' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Srishti.Gupta>
```

Step 3: Go to **this** link to download Node.js.

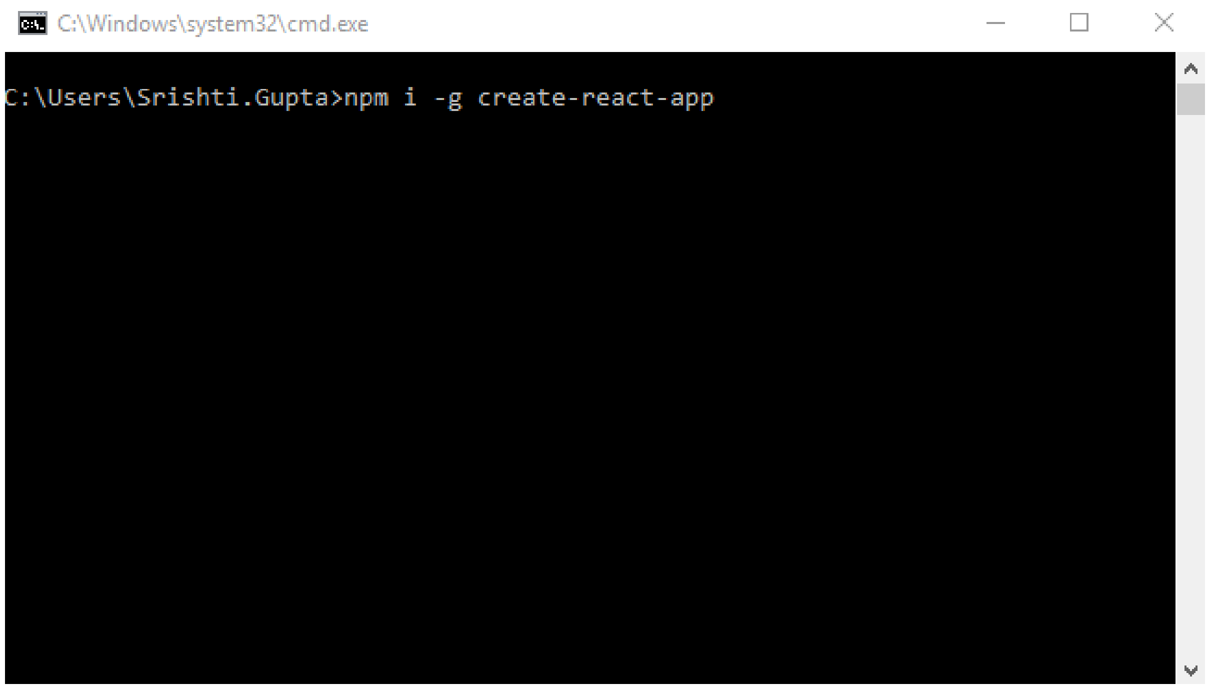
On the web page under the *LTS (Recommended For Most Users)* tab, click on *Windows Installer (.msi),* either 32-bit or 64-bit, depending on your processor. This will download the latest version for you.

Step 4: Double-click on the downloaded file. Proceed with the default instructions to install *Node.js*.

In order to verify that it has been correctly installed, open *Command Prompt* again and go back to step 2. If a version (greater than 6) of *Node.js* is returned to you, it means that you have successfully installed its required version on your machine.

Step 5: After *Node.js* is successfully installed on your system, type the following command in your *Command Prompt*:

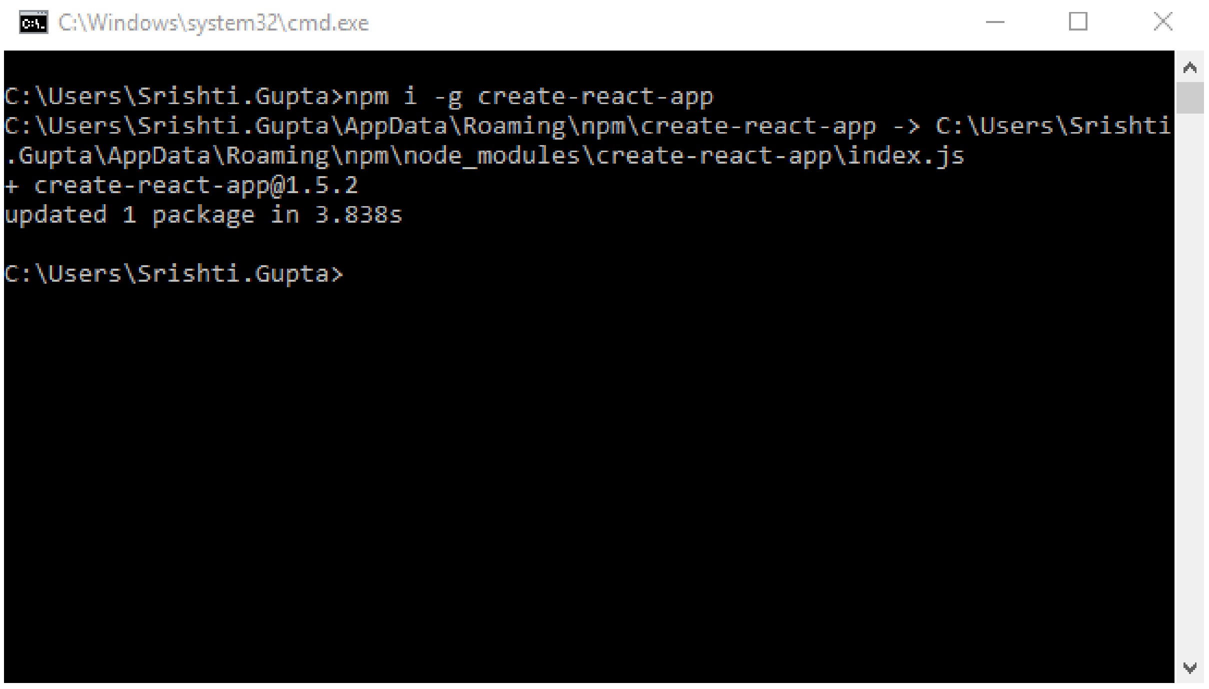
npm i -g create-react-app



```
C:\Users\Srishti.Gupta>npm i -g create-react-app
```

npm stands for *Node Package Manager*, which gets installed when you install *Node.js. i* stands for install, which is a command. You can also write *install* in place of *i. g* stands for global and is preceded with a hyphen because it is a flag. *create-react-app* is the name of the package that you need to install. It is a CLI tool that allows you to quickly create and run React applications, with no configuration required by the user.

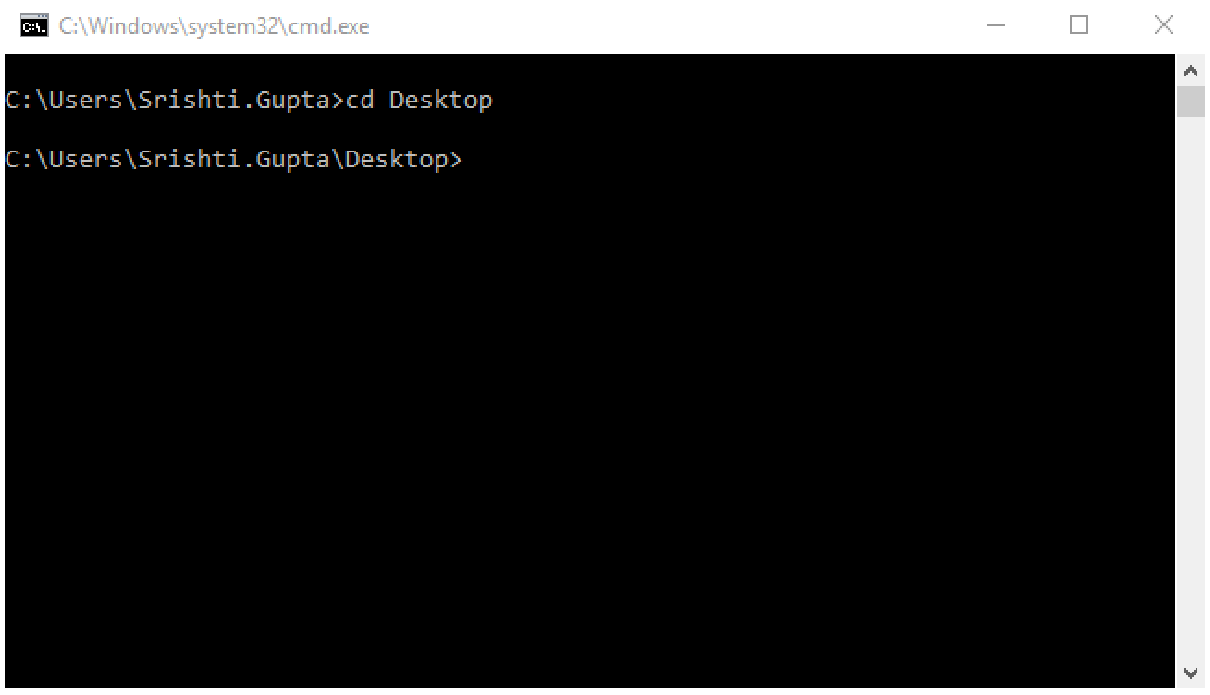
This command will instruct npm to install the *create-react-app* package globally on your machine so that you can create a React application at any valid path on your system.



```
C:\Users\Srishti.Gupta>npm i -g create-react-app
C:\Users\Srishti.Gupta\AppData\Roaming\npm\create-react-app -> C:\Users\Srishti
.Gupta\AppData\Roaming\npm\node_modules\create-react-app\index.js
+ create-react-app@1.5.2
updated 1 package in 3.838s

C:\Users\Srishti.Gupta>
```

Step 6: Now, let's complete one more step to create your first React application. In the *Command Prompt*, go to the path on your system where you want to create your application.

For example, if you want to create the application on the Desktop, go to the path in your *Command Prompt* that leads you to the Desktop.

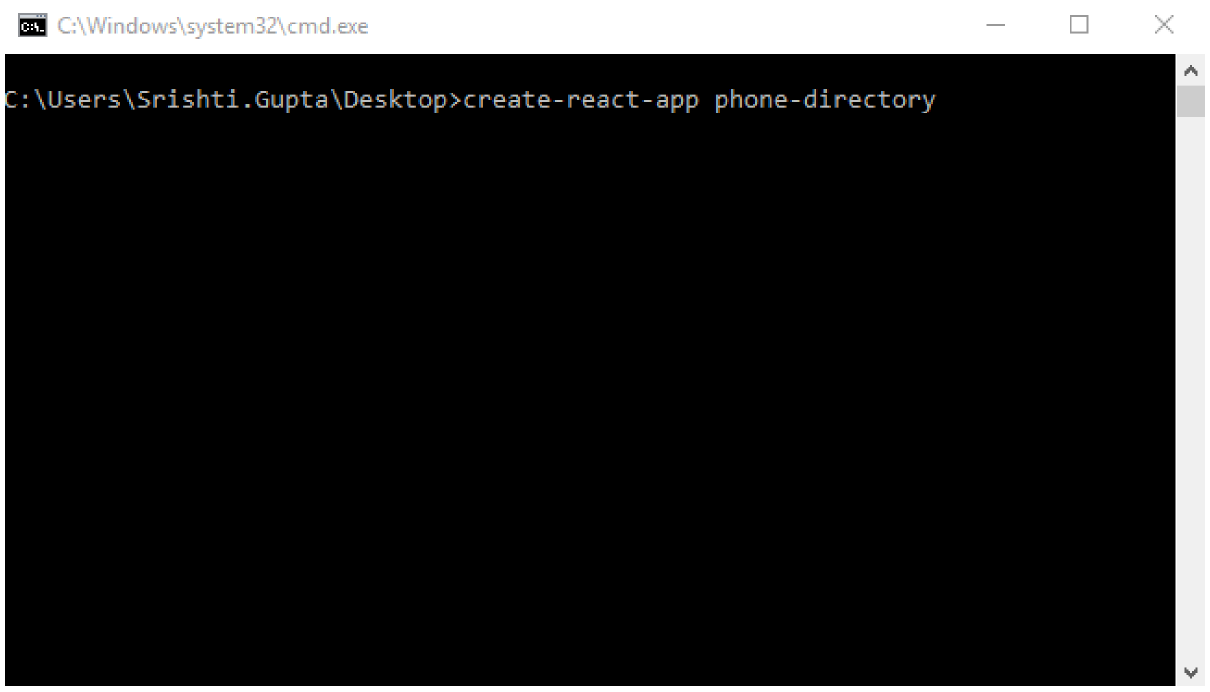


```
C:\Users\Srishti.Gupta>cd Desktop

C:\Users\Srishti.Gupta\Desktop>
```

Step 7: Create your React application using the *create-react-app* package, which you installed on your machine in the previous step. For this, you would need to type a command that mentions *create-react-app,* followed by the name of your application.

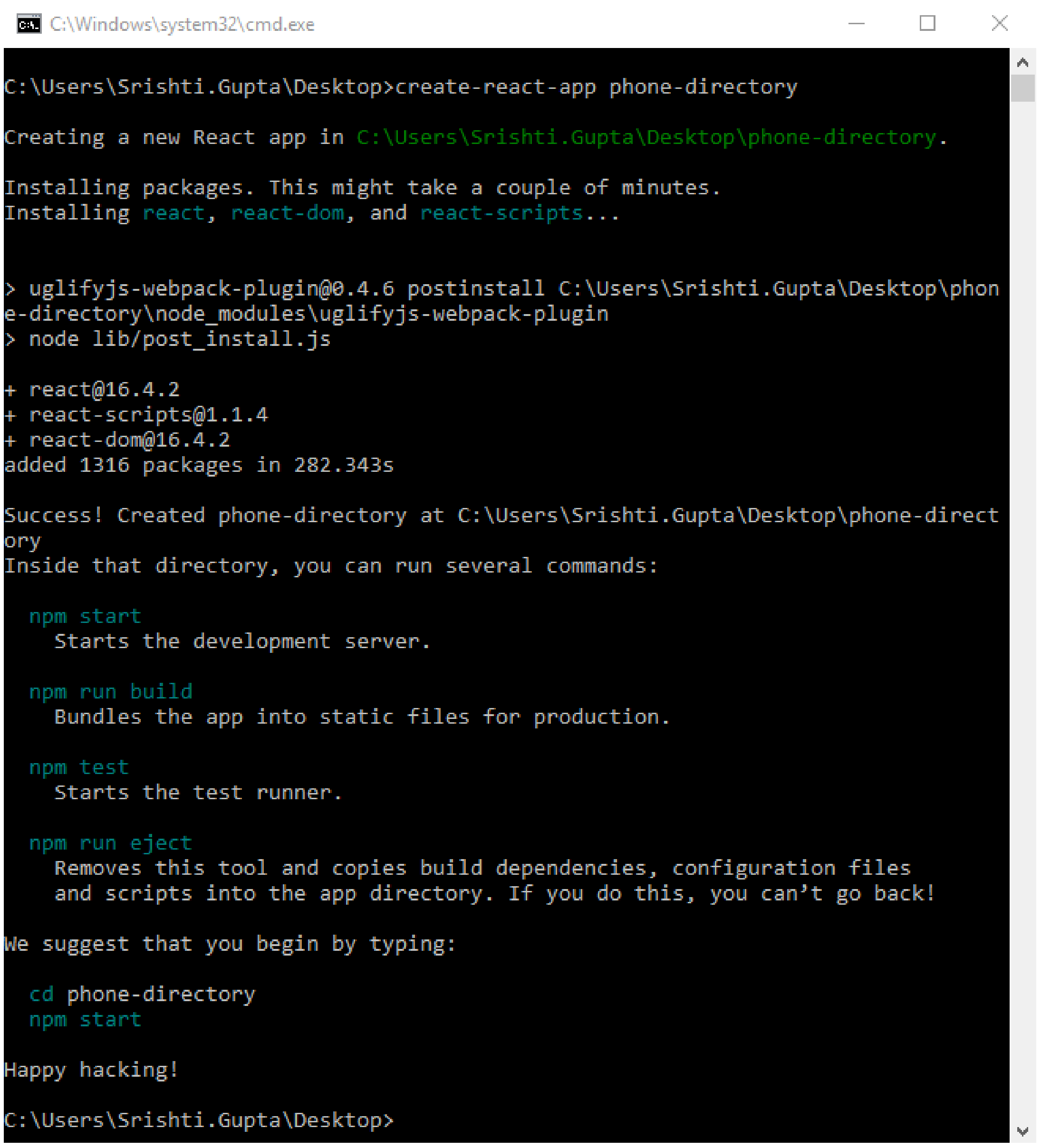
For example, here, the name of your application is ***phone-directory***. Thus, the command can be written as follows:

```
create-react-app phone-directory
```



```
C:\Users\Srishti.Gupta\Desktop>create-react-app phone-directory
```

This command will create a folder with the name that you mentioned inside your current path in the *Command Prompt*. This folder will consist of all the necessary configuration files that you need as the starter code for a React application.



C:\Windows\system32\cmd.exe

```
C:\Users\Srishti.Gupta\Desktop>create-react-app phone-directory

Creating a new React app in C:\Users\Srishti.Gupta\Desktop\phone-directory.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...


> uglifyjs-webpack-plugin@0.4.6 postinstall C:\Users\Srishti.Gupta\Desktop\phon
e-directory\node_modules\uglifyjs-webpack-plugin
> node lib/post_install.js

+ react@16.4.2
+ react-scripts@1.1.4
+ react-dom@16.4.2
added 1316 packages in 282.343s

Success! Created phone-directory at C:\Users\Srishti.Gupta\Desktop\phone-direct
ory
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd phone-directory
  npm start

Happy hacking!

C:\Users\Srishti.Gupta\Desktop>
```

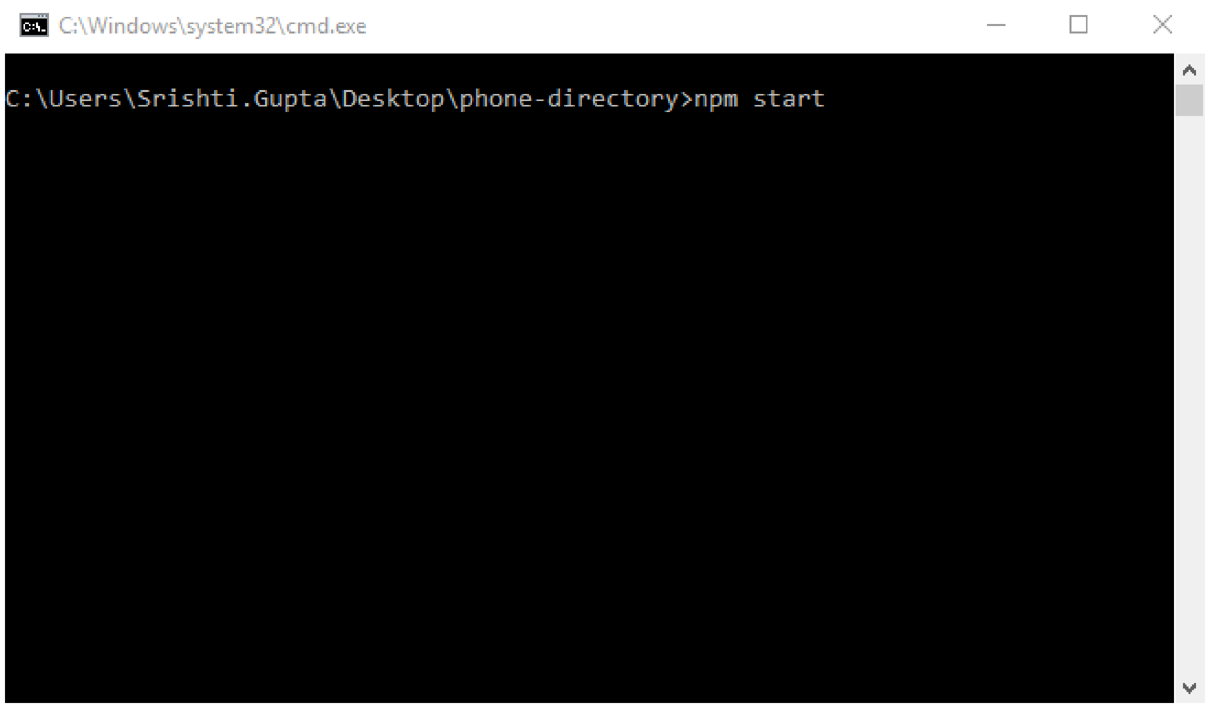
Step 8: Go to the application folder that you created on your current path inside the *Command Prompt*.

For example, here, the name of your application is *phone-directory*. Thus, you can write the following command to go into this application folder:

cd phone-directory

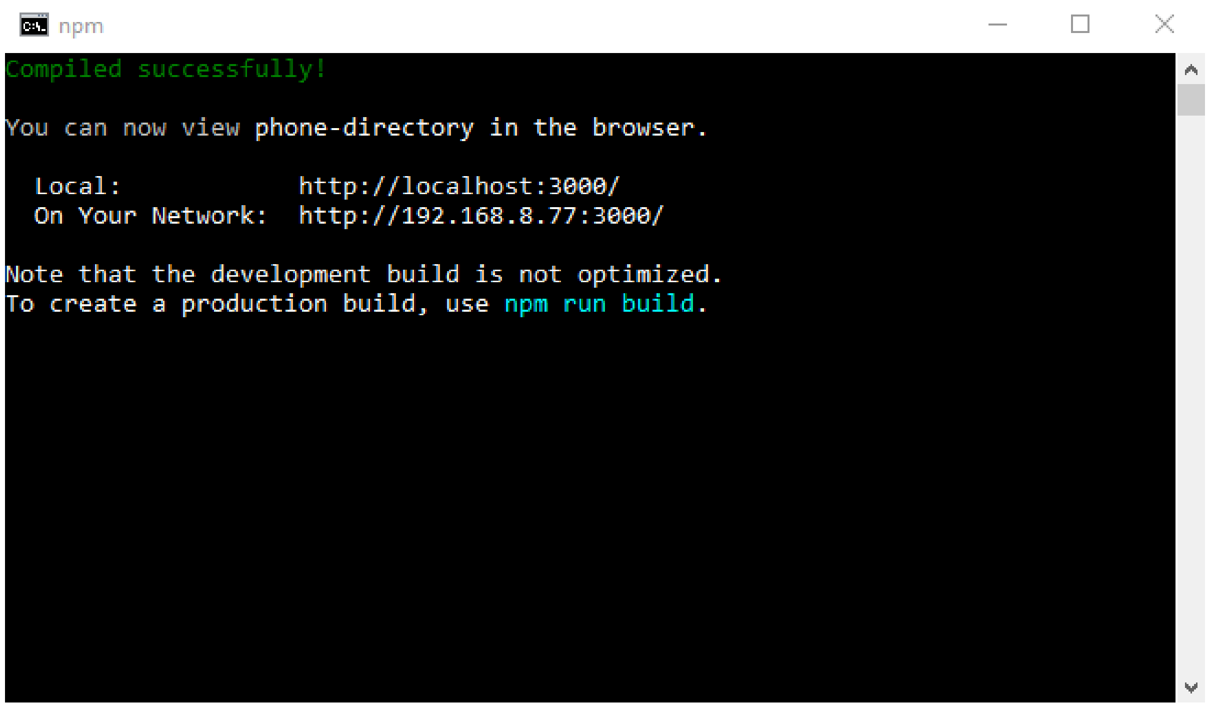
Step 9: Now, you are ready to start a development server locally on your machine and run your application in the development mode. For this, you need to write the following command in the *Command Prompt*:

npm start



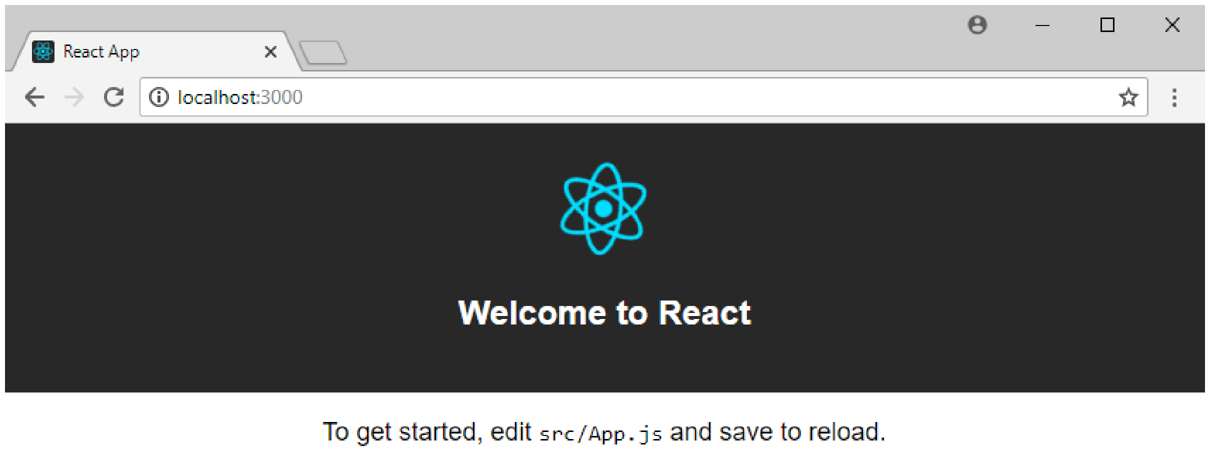
```
C:\Users\Srishti.Gupta\Desktop\phone-directory>npm start
```

As you can observe, the development server has been started on the mentioned URL on your local machine. Here, the server has been started on the localhost with port number 3000.



```
Compiled successfully!

You can now view phone-directory in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.8.77:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

This will open a browser tab/window displaying the landing page of the application, which will look similar to the one given below.



Welcome to React

To get started, edit `src/App.js` and save to reload.

Step 10: Now, you can start making changes to your *phone-directory* application. Any code change that you make will be immediately reflected on the browser, where the application is running.

Codebase Set-Up: Text Instructions (macOS)

You can set up any React application using the following command-line instructions:

Instructions for macOS:

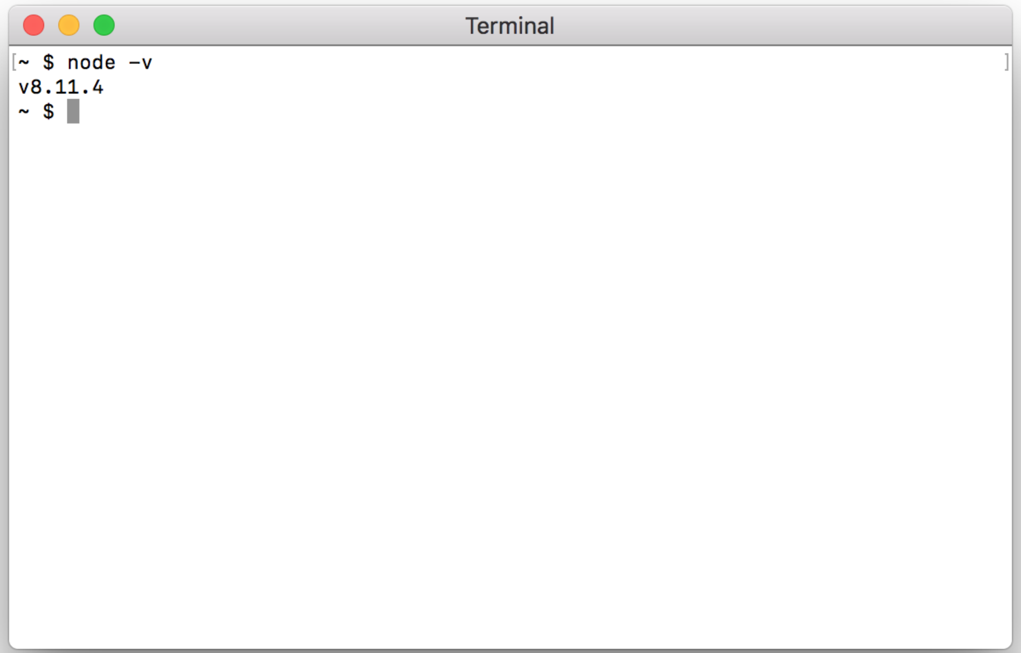
Step 1: Press *Cmd+Space* to open *Spotlight Search*, and type *Terminal*. Click on the first search result to open it.

Step 2: Check if *Node.js* is installed on your machine by writing the following command:

```
node -v
```

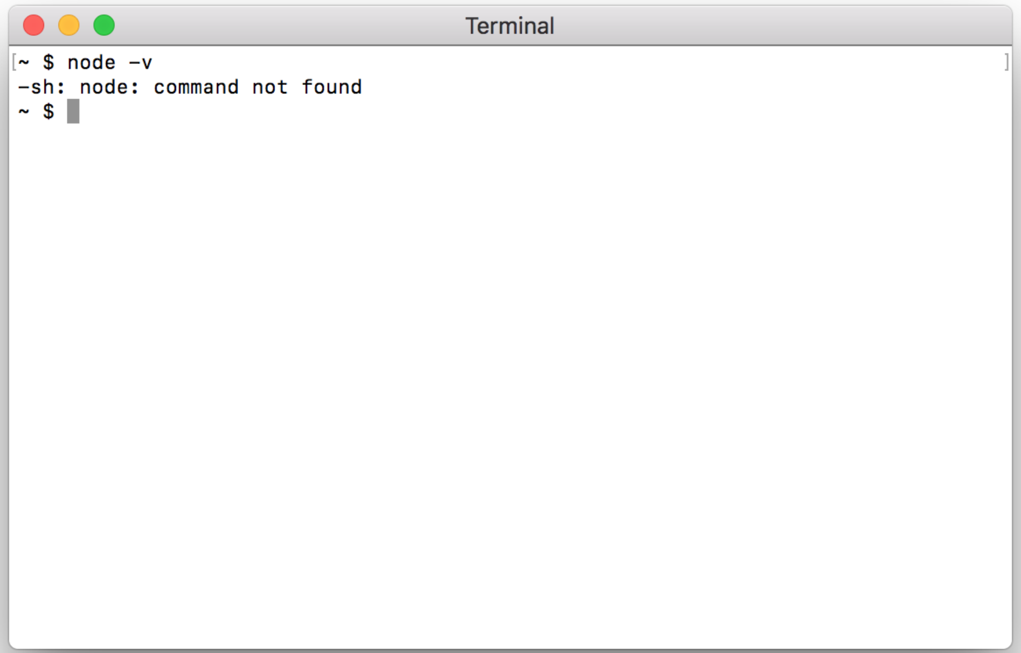
And, press the Enter key.

If any version is returned to you, it means that *Node.js* is installed on your machine. If this version is greater than 6, you can skip steps 3 and 4 and directly proceed to step 5.



```
~ $ node -v
v8.11.4
~ $ 
```

If the version is not greater than 6 or if you see the message '*node: command not found*', which means that *Node.js* is not installed on your computer, then close your *Terminal* and proceed to the next step.



```
~ $ node -v
-sh: node: command not found
~ $
```

Step 3: Go to **this** link to download Node.js.

On the web page under the *LTS (Recommended For Most Users)* tab, click on *macOS Installer (.pkg)* (64-bit). This will download the latest version for you.

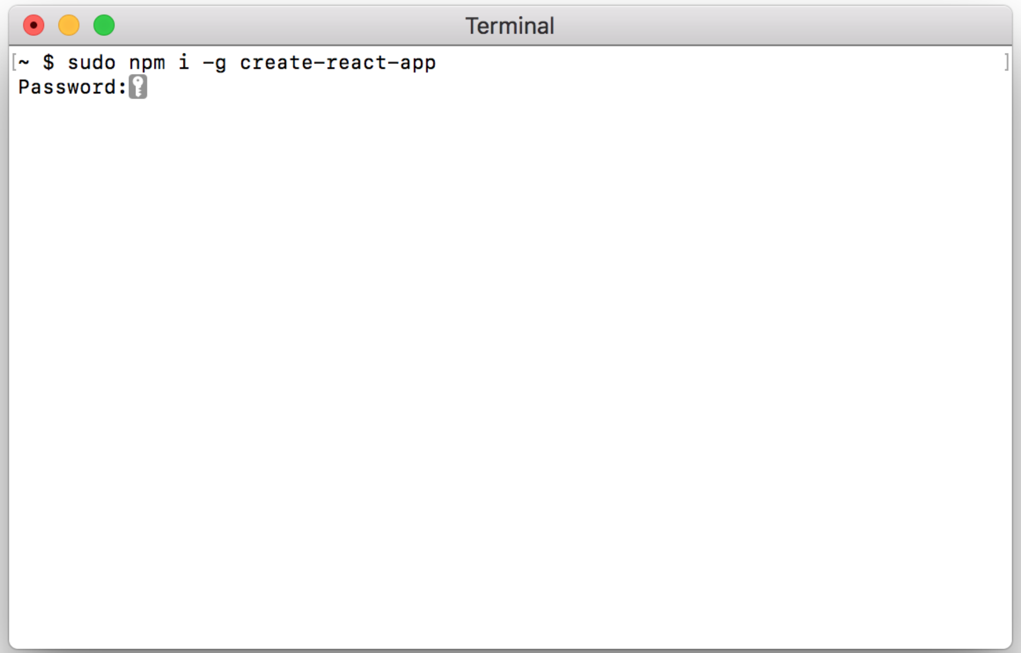
Step 4: Double-click on the downloaded file. Proceed with the default instructions to install *Node.js*.

To verify that Node.js is correctly installed, open *Terminal* again and repeat step 2. If a version (greater than 6) of *Node.js* is returned to you, it means that you have successfully installed the required version of *Node.js* on your machine.

Step 5: After *Node.js* is successfully installed on your system, type the following command in your *Terminal*:

sudo npm i -g create-react-app

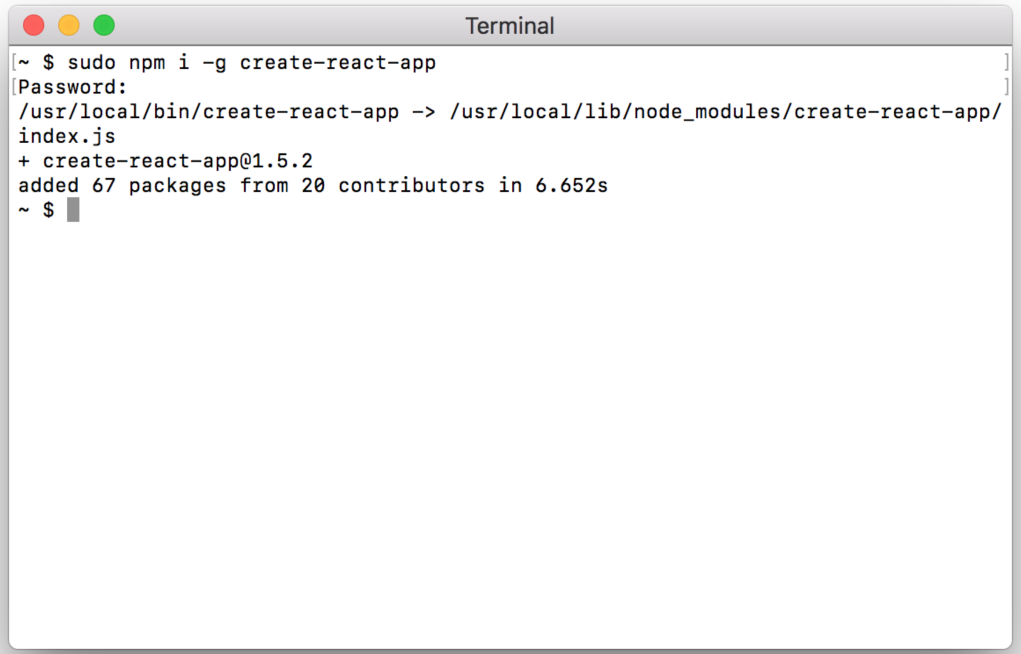
And, press the Enter key.



```
~ $ sudo npm i -g create-react-app
Password:
```

npm stands for *Node Package Manager*, which gets installed when you install *Node.js*. *i* stands for install, which is a command. You can also write *install* in place of *i*. *g* stands for global and is preceded with a hyphen because it is a flag. *create-react-app* is the name of the package that you need to install. It is a CLI tool that allows you to quickly create and run React applications, with no configuration required by the user.

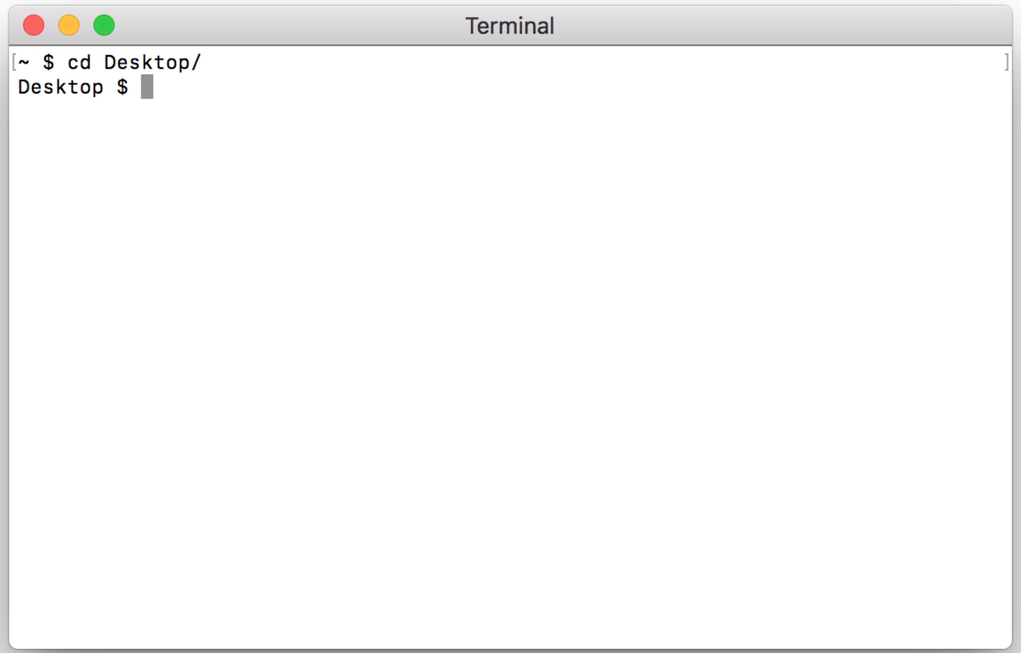
This command will instruct npm to install the *create-react-app* package globally on your machine so that you can create a React application at any valid path on your system.



```
~ $ sudo npm i -g create-react-app
Password:
/usr/local/bin/create-react-app -> /usr/local/lib/node_modules/create-react-app/
index.js
+ create-react-app@1.5.2
added 67 packages from 20 contributors in 6.652s
~ $ 
```

Step 6: Now, let's complete one more step to create your first React application. In the *Terminal*, go to the path on your system where you want to create your application.

For example, if you want to create the application on the Desktop, go to the path in your *Terminal* that leads you to the Desktop.



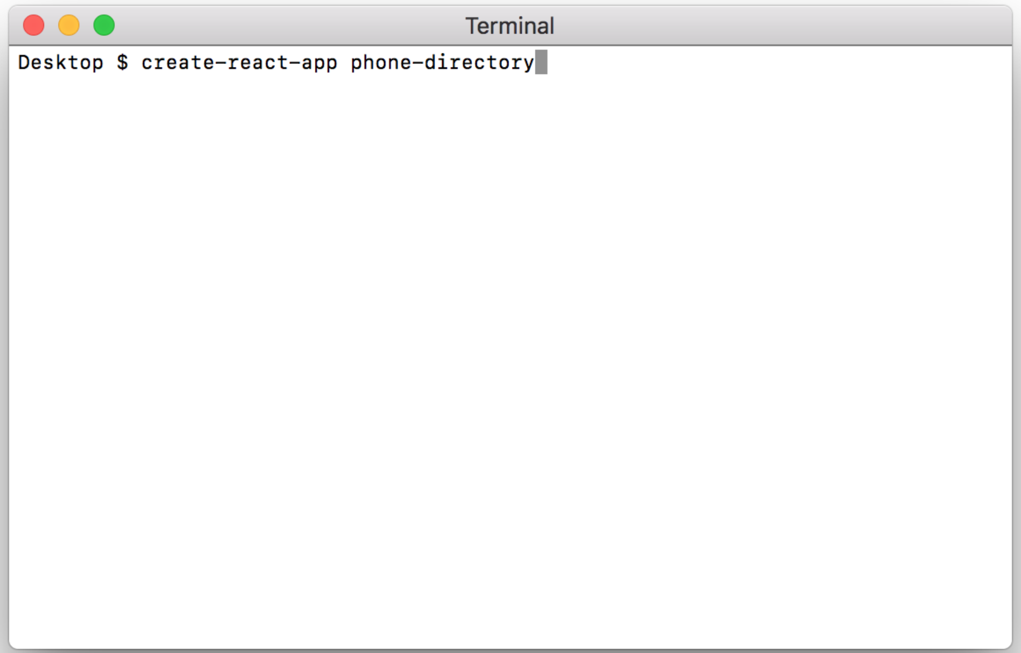
```
~ $ cd Desktop/
Desktop $ 
```

Step 7: Create your React application using the *create-react-app* package, which you installed on your machine in the previous step. For this, you would need to type a command that mentions *create-react-app,* followed by the name of your application.

For example, here, the name of your application is ***phone-directory***. Thus, the command can be written as follows:

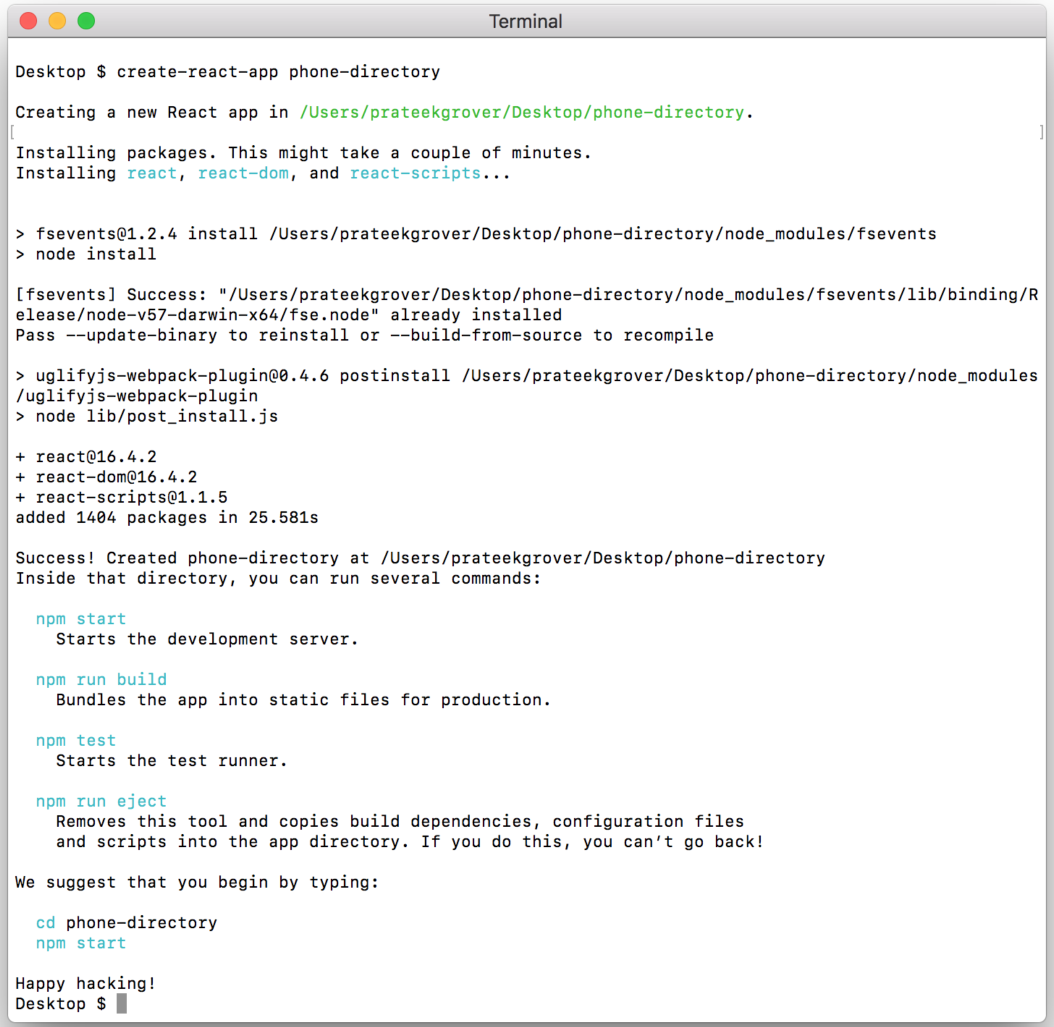
create-react-app phone-directory

And, press the Enter key.



```
Desktop $ create-react-app phone-directory
```

This command will create a folder with the name that you mentioned inside your current path in *Terminal*. This folder will consist of all the necessary configuration files that you need as the starter code for a React application.

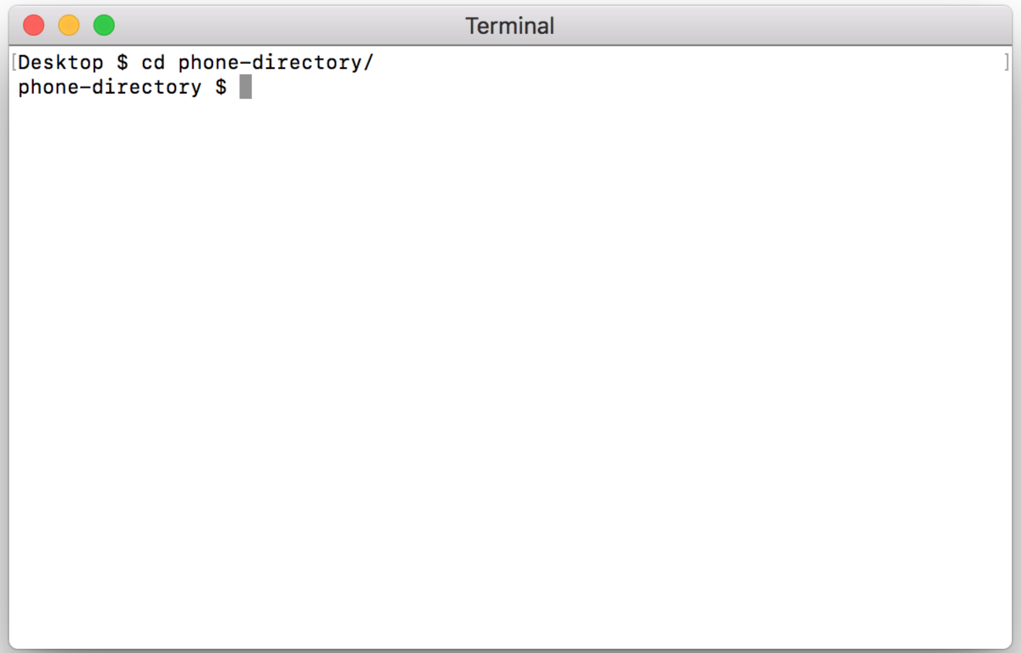


```
Desktop $ create-react-app phone-directory

Creating a new React app in /Users/prateekgrover/Desktop/phone-directory.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...


> fsevents@1.2.4 install /Users/prateekgrover/Desktop/phone-directory/node_modules/fsevents
> node install

[fsevents] Success: "/Users/prateekgrover/Desktop/phone-directory/node_modules/fsevents/lib/binding/R
elease/node-v57-darwin-x64/fse.node" already installed
Pass --update-binary to reinstall or --build-from-source to recompile

> uglifyjs-webpack-plugin@0.4.6 postinstall /Users/prateekgrover/Desktop/phone-directory/node_modules
/uglifyjs-webpack-plugin
> node lib/post_install.js

+ react@16.4.2
+ react-dom@16.4.2
+ react-scripts@1.1.5
added 1404 packages in 25.581s

Success! Created phone-directory at /Users/prateekgrover/Desktop/phone-directory
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd phone-directory
  npm start

Happy hacking!
Desktop $
```

Step 8: Go to the application folder that you created on your current path inside *Terminal*.

For example, here, the name of your application is *phone-directory*. Thus, you can write the following command to go into this application folder:

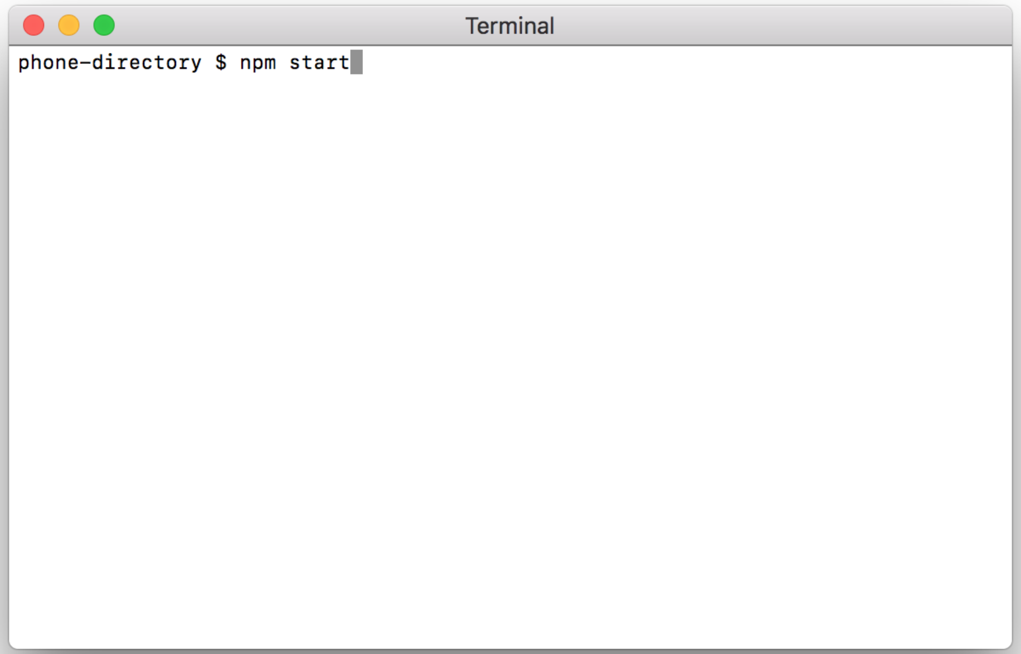
cd phone-directory



```
Desktop $ cd phone-directory/
phone-directory $ 
```

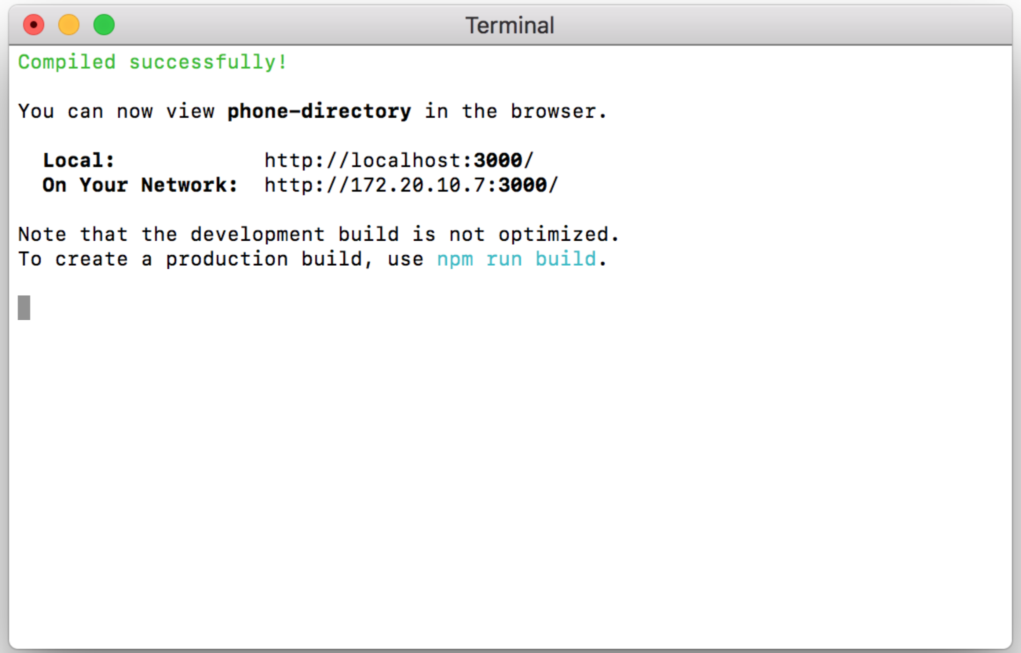
Step 9: Now, you are ready to start a development server locally on your machine and run your application in the development mode. For this, you need to write the following command in *Terminal*:

npm start



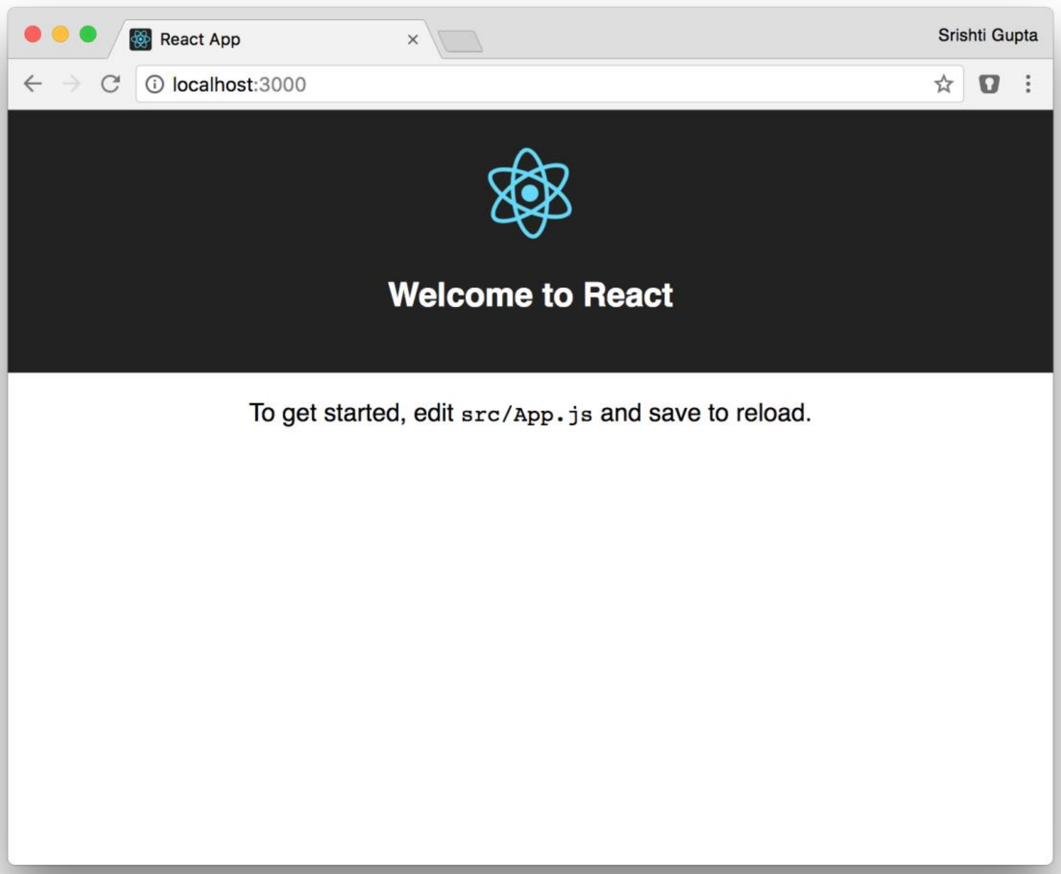
```
phone-directory $ npm start
```

As you can observe, the development server has been started on the mentioned URL on your local machine. Here, the server has been started on the localhost with port number 3000.



```
Compiled successfully!

You can now view phone-directory in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://172.20.10.7:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

This will open a browser tab/window that will display the landing page of the application, which will look similar to the one given below.



Welcome to React

To get started, edit `src/App.js` and save to reload.

Step 10: Now, you can start making changes to your *phone-directory* application. Any code change that you make will be immediately reflected on the browser where the application is running.

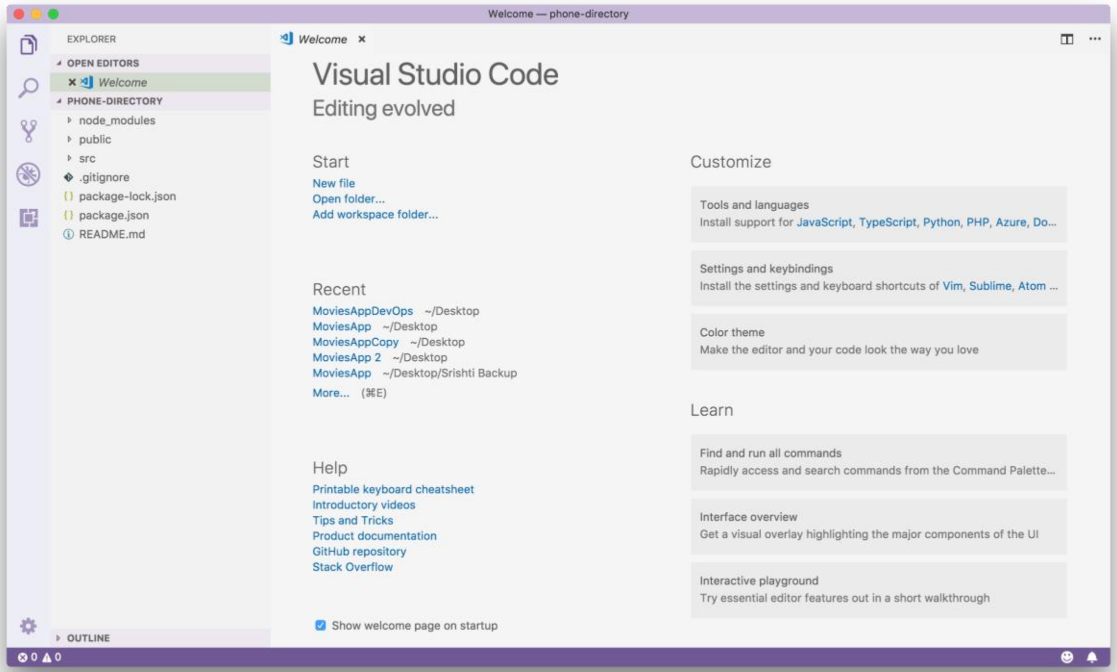
Folder Structure

You need to open the application folder inside a text editor. You will be using **Visual Studio Code** as your text editor during the course of this entire module. However, you also have the option to use any other text editor. If you do not already have *Visual Studio Code* installed on your machine, follow **this** link to download it. After downloading the file, install it on your machine.

You need to complete the following steps to open your application folder inside the text editor:

1. Open Visual Studio Code.
2. If you are using *macOS*, click on the *File* menu at the top, and then click on '*Open…*'. If you are using *Windows*, click on the *File* menu at the top, and then click on '*Open Folder*'.
3. In the dialog box that opens, go to the location where your application folder is present and choose your application folder. Here, you have created the *phone-directory* folder inside the Desktop. So, go to the Desktop and choose the *phone-directory* folder.
4. Click on *Select Folder*.

When you look at *Explorer* on the left side in *Visual Studio Code*, you will see the name of your application folder. You will also see a list of all the files and folders that are inside your application folder. Given below is a screenshot of what it will look like.


Welcome — phone-directory
EXPLORER
OPEN EDITORS
Welcome
PHONE-DIRECTORY
node_modules
public
src
.gitignore
package-lock.json
package.json
README.md
OUTLINE
Welcome
Visual Studio Code
Editing evolved
Start
New file
Open folder...
Add workspace folder...
Recent
MoviesAppDevOps ~/Desktop
MoviesApp ~/Desktop
MoviesAppCopy ~/Desktop
MoviesApp 2 ~/Desktop
MoviesApp ~/Desktop/Srishti Backup
More... (⌘E)
Help
Printable keyboard cheatsheet
Introductory videos
Tips and Tricks
Product documentation
GitHub repository
Stack Overflow
Show welcome page on startup
Customize
Tools and languages
Install support for JavaScript, TypeScript, Python, PHP, Azure, Do...
Settings and keybindings
Install the settings and keyboard shortcuts of Vim, Sublime, Atom ...
Color theme
Make the editor and your code look the way you love
Learn
Find and run all commands
Rapidly access and search commands from the Command Palette...
Interface overview
Get a visual overlay highlighting the major components of the UI
Interactive playground
Try essential editor features out in a short walkthrough
0 0

Note that the *index.html* file inside the *public* folder is the starting point of your application. **The index.html file should always be saved with the same name inside the same location; otherwise, your application will not run and throw an error**. Similarly, *index.js* is the entry point for all of your JavaScript code. **Like the index.html file, the name and location of index.js should not be altered.** You must keep this in mind in order to ensure that your code runs successfully.

Some the basic points to note about the different types of files that you will see inside your application folder are as follows:

package.json file

- It consists of the name and version of the application, the combination of which should be unique in order to publish the package.
- It comprises dependencies that list all the packages needed to be installed for the application.
- It also includes scripts that specify the commands to be run at various points in the application lifecycle.

package-lock.json file

- It is automatically generated for any operation where npm modifies either the *node_modules* tree or the *package.json* file.
- It locks the version of the full dependency tree of packages.
- It guarantees the generation of an identical dependency tree when the application is cloned by other developers.

node_modules folder

- Its contents are defined by the *package.json* file, and it consists of all the packages required for running your application.

public folder

- Nothing inside this folder is processed by *webpack*.
- This folder is used for storing small files that are not required to be bundled.
- It can be used to contain images when there are thousands of them and their paths need to be referenced dynamically.
- Any file inside this folder needs to be referenced at other places using the *%PUBLIC_URL%/* keyword, which gets replaced with the path of the public folder during the application's build process.

index.html file

- This is the starting point of the application.
- It should always remain with the name *index.html* and inside the *public* folder; otherwise, the code will fail to run.
- It can only reference the files that are inside the *public* folder.

manifest.json file

- This is a simple JSON file instructing the browser about the web application and how the application should behave when it is installed on the user's mobile device or computer.

src folder

- This folder contains the real application code.
- It consists of all the files that are needed to be bundled by *webpack*.

index.js file

- This is the entry point for JavaScript.
- The filename should remain *index.js* and the location should be inside the *src* folder; otherwise, the code will not run.

index.css file

- This is the stylesheet for *index.html*.

registerServiceWorker.js file

- This is the web browser API that is used to cache assets and other files to work passively in the background. It helps offline users or the ones who are on the slow network to view results on the screen faster.
- It adds offline capabilities to the application.

App.js file

- This is the JavaScript file for the *App* component.

App.css file

- This is the stylesheet for the *App* component.

App.test.js file

- This is the test file for the *App* component.
- It contains unit tests for the application.
- It runs test cases for all the files that are changed after the last commit of the application.

logo.svg file

- SVG refers to Scalable Vector Graphics.
- An SVG file is an XML-based vector image format for 2D graphics with support for interactivity and animation.
- It is similar to raster-based image formats such as JPEG, PNG, BMP and GIF.
- It offers a bandwidth-friendly way to render images; no matter how large a graphic gets, it transmits only the XML describing the graphic to the client.
- It helps to render resolution-independent and SEO-friendly images.

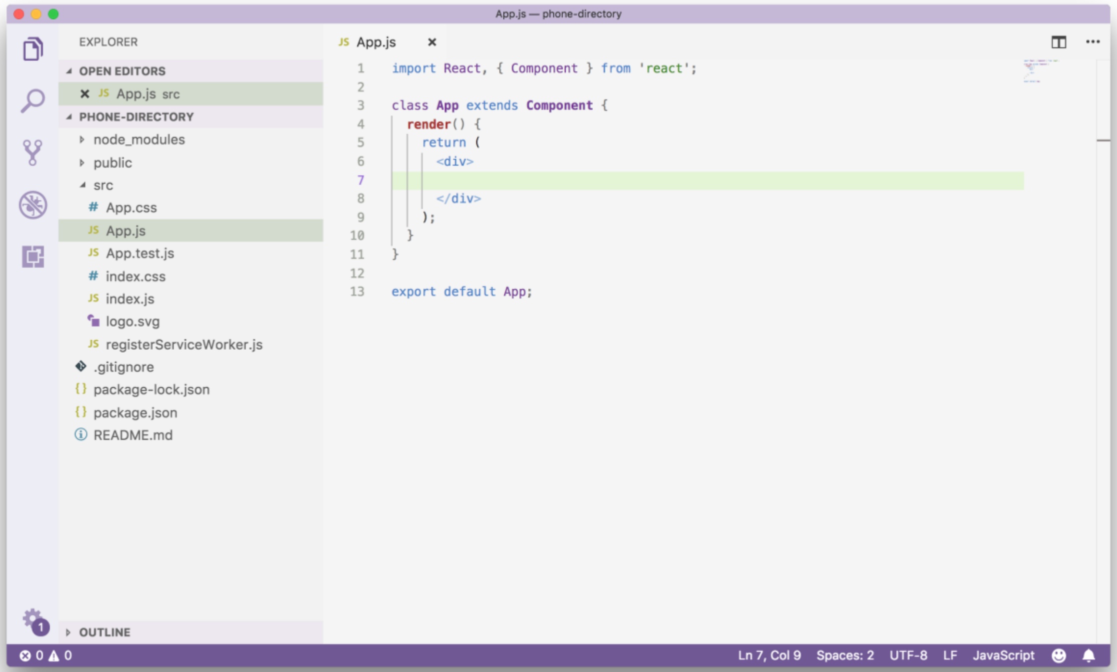
- It makes up the icon for your application and appears alongside the title in the browser tab.
- It gets saved along with the bookmark.

Code Clean-Up

Many files are created automatically for you when you create a React application using the *create-react-app* node package. Since you will be writing your own custom code, you do not need all the auto-generated code and files in your application. So, let’s clean up some unnecessary code blocks before you proceed to building the application.

Complete the following steps to remove the unwanted code and files from your application:

1. Inside the *src* folder, go to the *App.js* file.
    - Remove all the code written inside the *div o*f the return statement of the *render* method inside the *App* class.
    - Remove the *className="App"* from the outer *div* while leaving the outer *div* as it is.
    - Remove the import statements at the top of the file for the *logo.svg* and *App.css* files. After doing this, the *App.js* file should look as shown in the image given below.



```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div>

      </div>
    );
  }
}

export default App;
```

2. Inside the *src* folder, delete the *App.css* file.

3. Inside the *src* folder, delete the *logo.svg* file.

JSX

JSX is an XML extension to the ECMAScript specification. So, instead of using pure JavaScript for building DOM elements, you can use JSX, which offers flexibility to developers to use a familiar syntax, viz. HTML.

JSX vs HTML

1. **Adjacent JSX elements must be wrapped in an enclosing tag.**

While returning JSX from a function or a class, you cannot return multiple elements. You can return only a single element. This is why you need to encompass all children elements within a parent element and then return this parent element. In case you fail to do this, you will get a syntax error with the message '*Adjacent JSX elements must be wrapped in an enclosing tag*'. However, in HTML, you can return as many DOM elements as you want. There is no rule regarding returning a single element.

Note that with the introduction to React 16, one can return an array consisting of multiple elements existing at the same level. These elements are separated from each other using a comma. Thus, one can write the following code snippet, which works fine in React 16:

```
return [
  <div>
    Phone Directory
  </div>,
  <button>Add</button>,
  <div>
    <span>Name</span>
    <span>Phone</span>
  </div>
]
```

However, when React released its v16.2, it introduced *Fragment*, which allows you to return multiple elements. You can reference Fragment, as mentioned below.

```
import { Fragment } from 'react';
```

After including *Fragment* as a named import from the 'react' library, you can wrap multiple elements inside it. Finally, the entire *Fragment* enclosing multiple elements can be returned. Thus, you can simply write the following code snippet:

```
return (
  <Fragment>
    <div>
      Phone Directory
    </div>
    <button>Add</button>
    <div>
      <span>Name</span>
      <span>Phone</span>
    </div>
  </Fragment>
)
```

2. **Tags must be closed.**

In JSX, you need to close both types of tags opening-closing tags and self-closing tags. For an opening tag, you need to explicitly write a closing tag at the end. For a self-closing tag, you need to insert a forward slash before the closing angular bracket. If you fail to do this, you will get a syntax error with the message '*Expected corresponding JSX closing tag for <br>*'.

However, in HTML, the browser sometimes handles closing the tags by itself. Don't believe this? Try to create a simple webpage (HTML page) and write a paragraph tag inside the body tag inside it. For example, let's consider a paragraph consisting of the text 'Hello World', as shown below. Do not close this paragraph tag.

```
<p>Hello World
```

Now, run this on your browser. What do you see? Amazed?

3. **JSX properties are not similar to HTML attributes.**

Some attributes that you use in HTML cannot be used as JSX properties. You can go through the entire list of such attributes **here**. This is because the entire JSX code gets converted into JavaScript code at the end. As you already know, JavaScript has its own set of keywords. If you try to write these keywords or reserved words as JSX properties, it gets confusing to identify which word is being used as a JavaScript keyword (or reserved word) and when it is being used as a JSX property. To make this distinction, use alternative keywords in JSX for the HTML attributes that exist in the JavaScript language.

4. **Case sensitiveness**

React ‘reacts’ to the cases that you use. It will not allow you to write something in the case of your choice. On the other hand, HTML syntax is not case-sensitive. You can choose to write the div tag as <DIV>, <div> or <Div>. But can you do the same in React? You cannot.

Injecting Data Using Curly Braces {}

Curly braces {} are used as a special syntax in React. These can be used to evaluate a JavaScript expression during compilation. The expression can be a variable, a function, an object, an arithmetic calculation, logical evaluation, or any code snippet that returns some value.

Example:

Code Snippet:

```
let moduleName="React";

<span>Learning {moduleName} is so much fun!</span>
```

Output:

```
Learning React is so much fun!
```

React.createElement() method

The syntax of the ***React.createElement()*** method is as follows:

React.createElement(element_name, element_properties, children);

The **first argument** in this method is the **name of the element that is to be rendered**. This can be either your custom component or an HTML element. The **second argument** is the object that consists of property-value pairs that can be provided as **attributes** to this component or element. After these two arguments, you can pass an **infinite number of children elements**, which will be nested inside this main component or element. These children elements can, in turn, have other children elements nested inside them.

The first argument is mandatory, while the rest of the arguments that follow are optional.

Rendering Elements into the DOM

As you already know, you have one root node inside your application that contains all of the React code. But can you have **multiple root nodes** inside one application?

Well, one root node is sufficient in a small application, but you can have multiple root nodes depending on the needs of your application.

When do you need to have multiple root nodes inside an application?

One of the advantages of using React is that it does not make assumptions about your technology stack. Imagine that you want to ship only some features in React in an application that is not primarily built using React. These features are spread across your entire application and are written at multiple places inside the application. Now, how will you achieve this without making any changes to the rest of your independently existing code? This can be possible only when you are able to choose selective pieces of your existing code to be shipped, convert them into React and plug these new pieces of React code back into different places inside your application. Having multiple root elements can help you achieve this and, thus, redefine your application using React. Each root node can contain multiple React components, and these root nodes can be plugged into different places inside your application.

But h*ow do you render elements into DOM in React?* You can use the **ReactDOM.render()** method for this purpose.

Syntax of the ***ReactDOM.render()*** method:

ReactDOM.render(argument_1, argument_2);

argument_1 tells WHAT to render.

argument_2 tells WHERE to render.

1. The **first argument** of the ReactDOM.render() method indicates **what is to be rendered**. This does not mean that only one element can be rendered in the first argument. *You can have multiple elements rendered by enclosing them inside a parent div container.* Also, this argument does not necessarily have to be a component. This argument can also take JSX code directly.
2. The **second argument** of the ReactDOM.render() method can be anything that specifies a **location on the DOM** where the element(s) in the first argument need to be rendered.

Components

How does React provide reusability of code? You can achieve this using components.

***Components* are just the JavaScript way of writing independent, reusable, and dynamic code**.

There are **two types of components** in React, **functional components** and **class components**. As the names suggest, functional components are written in the form of functions, whereas class components are written in the form of classes.

In React, you ought to write reusable code snippets; this is one of the recommended coding guidelines in software development. Thus, using components saves you the trouble of rewriting redundant code at multiple places. This is why React is all about components.

Styling

Styling makes a website look better. It is used to enhance the appearance of the DOM elements. This is analogous to how painting the walls of your house makes them look better.

React offers styling in two ways, *inline* and *external*, which can be summarised as follows:

- **Inline Styling**

1. It allows you to write styles in the same line with the element or component. Therefore, it is called 'inline' styling.
2. The *style* property is used with the element or component to be rendered into the DOM.
3. The *style* property accepts a JavaScript object enclosed in curly braces.
4. Two curly braces are used with the style property, one to evaluate the expression inside the JSX code and the other to represent a JavaScript object, which is taken as input by the *style* property, as shown below.

```
<div style={{}}>

  Phone Directory

</div>
```

5. The property names look similar to CSS property names, but they are not exactly like them. These names are actually JavaScript identifiers; they can be considered as the keys (or properties) of a JavaScript object.
6. The property names must be written in camelCase. Unlike CSS, hyphens are not allowed in JSX because the JSX code gets converted into JavaScript code, and hyphens are not allowed in JavaScript identifiers.

```
<div style={{textTransform: 'uppercase'}}>

  Phone Directory

</div>
```

This is why textTransform is written in camelCase in JSX, unlike text-transform in CSS. If you fail to follow this format, you will get an error with the message

‘*Uncaught SyntaxError: Inline Babel script: Unexpected token*’.

7. The property values look similar to CSS property values, but they are not exactly like them. These values can be considered the values corresponding to the keys (or properties) in a JavaScript object. Since all the values in JavaScript must be of a valid datatype, each value should be correctly mapped to a valid datatype in JavaScript.

```
<div style={{background: '#000'}}>
   Phone Directory
 </div>
```

Therefore, *'#000'* is written inside single quotes because it corresponds to a string value. However, in CSS, you write it without quotes for the code to work.

7. All property-value pairs are separated using the comma operator. This is because the *style* property accepts a JavaScript object with a comma, whereas in CSS style, a semicolon is used instead.

8. When a component or an element is styled by moving out style as a constant object, only one pair of curly braces is used in the style property.

- **External Styling**

1. Write all the styles in an external stylesheet. This is similar to writing external CSS.

2. Import this stylesheet in the file where the component or element is defined on which you want to apply the given style. Note that since the extension of a stylesheet is .css (not equivalent to .js or .jsx), you need to specify the file extension while writing the import statement for a stylesheet.

Rendering Content Dynamically

You can dynamically render content inside components while using JavaScript's *map()* method.

The **map()** method in JavaScript creates a new array after calling the given function on each array element in order. Note that it does not change the original array.

JavaScript's map() method can be used to iterate over an array and inject data into the React components or elements dynamically. You need not hard-code the data inside each component. This is one of the major reasons why React refers to its components as 'reusable' entities. Also, this is yet another application where curly braces are used in order to write some JavaScript code alongside JSX.

1. The entire map() method is written inside curly braces because the JavaScript code needs to be evaluated. The map() method is a JavaScript function and returns an array after applying the given function to each element of the array.

```
{
  arrayNameToIterateOver.map(
    //code here
  )
}
```

2. In React, you need to provide a unique key to each element to be rendered into the DOM. If you fail to do this, you will get a warning with the message '*Each child in an array or iterator should have a unique "key" prop*'.

To overcome this issue, you need to first assign each array element with a unique value for a property. Let's say the property is *id* and the unique values are 101 and 102, corresponding to the individual array elements.

```
let demoArray = [
  {
    id: 101, // unique
    prop1: "SomeValForProp1",
    prop2: "SomeValForProp2"
  },
  {
    id: 102, // unique
    prop1: "SomeOtherValForProp1",
    prop2: "SomeOtherValForProp2"
  }
];
```

Secondly, you need to assign this property (*id,* in this case) to the property *key* of the outermost element inside the map method.

```
{
  demoArray.map(arrayElement => {
    return <div key={arrayElement.id}>


    </div>
  })
}
```

Providing unique keys helps in distinguishing between different elements that are rendered into the DOM in React.

Understanding React Concepts: Part II

You will learn the following concepts in React:

- Props
- Event handling
- State
- Component Lifecycle
- Smart Componentsvs Dumb Components
- Routing

Props

Props help you to pass values from a parent component to a child component so that they can be accessed within the child component.

Props in a functional component:

A functional component accepts a **parameter** called props from the parent component. This parameter is an object that holds all the properties passed from the parent component to the child component. In place of props, you can use any other parameter name.

Example:

```
const Organization = function(props) {

  return (

    <div>

        <h1>{props.name}</h1>

          <h3>{props.tagline}</h3>

    </div>

  )

}

<Organization name="UpGrad" tagline="Building Careers of Tomorrow"/>
```

Props in a class component:

The properties passed from the parent component can be accessed using the **this.props** keyword. Note that in class components, you need to use the keyword props only, whereas in functional components, any parameter name can be used to represent the props of the component.

Example:

```
class Organization extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.name}</h1>
        <h3>{this.props.tagline}</h3>
      </div>
     )
  }
}
<Organization name="UpGrad" tagline="Building Careers of Tomorrow"/>
```

Event Handling

An event is an action to be taken as a result of user interactions. An **event handler is a method that is called when an event occurs.** A programmer can define a series of steps inside an event handler, which can be followed when the specified event occurs.

1. An event name should be written in camelCase.
2. It is a good practice to prefix event handlers with '*on'*, for example 'onSubmitOrder', or suffix them with '*handler'*, for example 'submitOrderHandler'. This is done to clarify what these handlers do and what events they are attached to.

To learn more about all the supported events in React, you can follow **this** link.

State

State is something that is controlled within a component, **unlike props, which are controlled by a parent component**. Also, a **change in state calls the render() method again**. The render() method is present only inside a class component because a class component extends from the Component class, which acts as the base class. This is why class components are termed 'stateful'.

Based on this, it can be concluded that functional components cannot contain state because they do not extend from the Component base class and, thus, do not have any render() method. Therefore, they are termed 'stateless'. Some of the important points regarding state can be summarised as follows:

1. State can be **maintained inside a class component only**.
2. State is **always initialised inside the class constructor**.
3. If you define the constructor of a class, you need to call the **super() method** in the first statement of the constructor definition. This method calls the constructor of the parent class.
4. **To set the state**, you must always **use the setState() method** and must **never directly manipulate the application's state**. However, **the setState() method should never be called inside the constructor**.
5. Whenever **state is changed, the render()** method of the class is **called again.**

Component's Life Cycle

The component life cycle in React varies from one process to another, and there are a total of three processes, Mounting, Updating and Unmounting.

- **Mounting** refers to the process of creating and inserting a component into the DOM
- **Updating** refers to the process of updating a component by props and state.
- **Unmounting** refers to the process of removing components component from the DOM.

Inside the **Mounting** process, a component's life cycle is defined using the following methods, which are called in the given order:

1. constructor()
2. render()
3. componentDidMount()

Smart Components vs Dumb Components

A **dumb (or presentational) component** only **presents data on the DOM**. On the other hand, **a smart (or container) component provides data and logic** to the dumb components.

Thus, as mentioned earlier, **dumb components describe how things look, whereas smart components describe how things work**.

Routing

Routing is the process that helps in loading partial content, making it a requirement for building SPAs. Based on the URL that a user visits, specific content is loaded on the page, which helps in displaying different types of content to the users without the need to refresh the page. This is when the users get to view the entire application in the form of a single page even though it consists of multiple pages.

In order to implement Routing in your application, you will need to use a node package called **‘react-router-dom’**. This package provides React components to simulate server-side router handling. This package provides a *BrowserRouter* component, which you will be using in your application.

To install the 'react-router-dom' package inside your application, you can go to your application's root folder and type the following command inside *Command Prompt* (Windows) or *Terminal* (macOS):

```
npm install react-router-dom
```

React Hooks and Redux

What is a Hook?

A Hook is a special function that permits you to ‘hook into’ React options. When can you take advantage of a Hook? Earlier, if you had to write a function component and wanted to feature a state to that function, you had to convert it into a Class.

What problems do Hooks solve?

Hooks solve a number of distinct issues in React, such as those of writing and maintaining thousands of components in large programs. Whether you are a beginner or you use React daily, you would probably acknowledge that these issues exist in React. In React, you need to maintain the components that were initially simple but grew into a complex and bulky stateful logic and resulted in various side effects. In many cases, it is not possible to break these components into smaller ones because the stateful logic is scattered across different places in a large program. So, Hooks allow you to split one component into smaller functions based on how they are related.

The differences between functional-based components and class-based components are as follows:

Functional Components	**Class Components**
These take props as input and return the JSX.	These take input as a prop and also manage the state.
These are known as stateless components.	These are known as stateful components.
You do not have the access to lifecycle hooks in functional components.	You can access lifecycle hooks in class components.

You cannot write logic in it because they are stateless.	You can write logic in this because you can manage states inside class-based components.

Considering these differences, you may observe that you cannot do much with functional components. In order to fill this gap, React Hooks were introduced. Using the features of Hooks, you can manage the state.

useState() Hook

- The useState Hook that allows you to add the React state to function components.
- You cannot add state to the function component, but if you want to, you need to convert it into the class-based component and then perform the rest of the operations.
- Using the useState() hook inside the function components, you can update/get the current state of the application.

Before learning about React hooks, you need to first gain an understanding of the rules of hooks, which are as follows:

1. Call Hooks from React functions only.
2. Do not call Hooks from regular JavaScript functions. Instead, you can:

- Call Hooks from React function components, or
- Call Hooks from custom Hooks (these will be covered later in this course).

For more information, you can refer to the following link:

https://reactjs.org/docs/hooks-rules.html

How does the useState hook work? Which functions does it conduct?

The useState hook declares a ‘state variable’, and it can be another way to ‘preserve’ some values between the function calls. Recall the functionalities that this.state performs in a class. The useState hook provides the same capabilities to you. In general, variables ‘disappear’ once the function exits. However, the state variables are preserved by React.

What do we pass to useState as an argument?

The sole argument that we can pass to the useState() Hook is the initial state.

What does useState return?

It returns a combination of values: a current state and a function that will update it. Recall the functionality of this.setState. This will make it easy for you to correlate what useState returns.

How do we use the useState Hook?

You pass the initial state to this function (useState). It returns a variable with the current state value and another function to update this value.

For example, the code snippet given below will render a button, which, when clicked, will update the counter variable; this is done using the useState hook.

```
const UpdateStateVar = () => {
  const [counter, setCounter] = useState(19)
  const handleClick = () => setCounter(counter + 1)
  return (
    <div>
      Button has been clicked {counter} no. of times!
      <div>
        <button onClick={handleClick}>Click me! </button>
      </div>
    </div>
  )
}
```

Here, useState(19) is the initial state, and you invoke the updater function that is returned by the useState invocation: const [valueOfTheState, updaterFunction], i.e., const [counter, setCounter].

useEffect() Hook

Before proceeding further, let's first understand the meaning of **side effects in React.** Some of the important points regarding side effects can be summarised as follows:

- **Side effects** are those that affect things that are outside the scope of the function that is being executed.
- They can be data fetching, manually updating the DOM, etc. A functional React component uses props or states to calculate the output.
- If the functional component performs some actions that do not target the output value, then these actions or activities can be called side effects.

What is the function of useEffect?

Using this Hook, you can inform React that your component needs to perform an activity after rendering. React will remember the function that you passed (we will refer to this as our 'effect') and call it later after performing the DOM updates.

How do we use useEffect Hook?

Let's take a look at a simple example code to recall how useEffect works. Consider the following code snippet:

```
const { useEffect, useState } = React
const counter = () => {
  const [count, setCount] = useState(0)
  const [name, setName] = useState('Aishwarya')
  useEffect(() => {
    console.log('Hello! ${name} You have clicked ${count} times')
  });

  return (
    <div>
      <p>Hello! {name} You have clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click here
      </button>
      <button onClick={() => setName(name === 'Aishwarya' ? 'Rai' : 'Aishwarya')}>
        Flip Name
      </button>
    </div>
  )
}
ReactDOM.render(<counter />, document.getElementById('app'))
```

Here, the function runs when the component is first rendered and re-renders or updates after that. React first updates the DOM and then calls any function passed to useEffect(). This happens without halting the UI rendering, which makes our code run and render faster.

Forms and Routing

In the module on HTML, you learnt how to take input from the user using forms. React offers a similar feature of forms. The only difference is that in HTML, form data is handled by the Document Object Model (DOM), whereas in React, the data entered by the user will come into effect when you update the state. This is because in the React state, is the ‘single source of truth’.

Let’s try to understand this with the help of the following example:

```
 <form>
    <label for="username">Name</label>
    <input type="text" name="username" id="username" placeholder="Enter Name..">
 </form>
```

This form has the default HTML form behaviour, in which the user is redirected to the new page when the user submits the form.

On the other hand, in React, this will work, but the standard way to handle the form data is to use the JavaScript function. This is where the concept of controlled components comes into the picture.

Controlled Components

In HTML, form elements maintain their own state and update the data based on the user input. In React, the mutable state is typically maintained in the state property of the components and is only updated with updating the state.

A simple example of form would be as follows:

```
class Form extends React.Component {
  constructor(props) {
    super(props);
    this.state = { username: '' };
  }
  changeEventHandler = (event) => {
    this.setState({username: event.target.value});
  }
  render() {
    return (
      <form>
      <h1>Hello There!{this.state.username}</h1>
      <p>Can I know your name?</p>
      <input
        type='text'
        onChange={this.changeEventHandler}
      />
      </form>
    );
  }
}

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

Please refer to the following link for more information on forms in React:

https://reactjs.org/docs/forms.html

Form Validation Using Material UI

In React, you need not build components from scratch and redefine the UI. What if you get all the components that you require to build for the application? In such a case, you only need to customise these components as per your needs. Thus, you do not need to create each component and write a separate CSS for it.

Also, simply having the input field in your application is not enough. You need to have a validation check regarding whether the user has entered data or has submitted the form without giving any information. This type of activity leads to an inconsistency in the application data.

In ReactJS, using Material-UI, you can validate the data entered by the user.

Material UI comes with the 'react-material-ui-form-validator' package, which enables you to create validity checks and balances over the data entered by the user.

Refer to the following link for more information:

https://www.npmjs.com/package/react-material-ui-form-validator

Material Design is a design language developed by Google in 2014. According to Google, it is a visual language that synthesises the classic principles of a good design, with the innovation of technology and science.

Material UI is the compilation of all the React components that implement Google's Material Design, thereby providing a better user experience while following user interface guidelines.

From design guidelines to developer components, Material UI consists of all the components that you need to develop efficient React applications with ease while unifying the user experience across varying platforms.

useContext() Hook

For passing the information to child components or sub child components, you use props, but this process can be cumbersome for properties such as theme and locale, which are used by many components in an application.

In such cases, you can use the useContext hook. Context provides a way to share information between components without explicitly passing props through the hierarchy.

The useContext() method accepts a context within a functional component and works with a .Provider component and a .Consumer component in one call. As an example, consider the following code snippet:

```
import React, { useContext } from "react";
import BackgroundContext from './BackgroundContext';
const MyComponent = () => {
  const colors = useContext(BackgroundContext);
  return <div style={{ backgroundColor: color.red }}>Hello There!</div>;
};
```

Here, we import the useContext() method and the BackgroundContext function and declare a functional component. The functional component MyComponent sets the value within your BackgroundContext as an argument to the useContext() method. Your return statement applies the background colour of your application. When a change is triggered, the useContext() method will have the latest context value.

Back-End Integration

So far in our application, we have the static data on which we are working. In the real world, this is not the way in which data is handled.

In a real-life scenario, we have the database at the back end, which stores our data, and we can use, modify and operate on it using the APIs exposed by the backend server.

Data cannot be handled in a static manner because we may have to perform the CRUD (Create, Replace, Update, Delete) operations on the data. Therefore, we need to integrate the application with the back end to have more dynamicity in the application.

useCallback() and useMemo() Hooks

In JavaScript, functions are first-class citizens, which means that a function is a regular object. You can perform numerous operations with the object such as return it from other functions and compare them.

Take a look at the code snippet given below.

```
function sub() {
  return (a, b) => a - b;
}

const sub1 = sub();
const sub2 = sub();

sub1(8, 6); // => 2
sub2(8, 6); // => 2

sub1 === sub2; // => false
sub1 === sub1; // => true
```

Even though sub1 and sub2 share the same source code, they are different function objects.

Similarly, function objects sharing the same code are often created inside React components.

For example:

```
function Foo() {
  // handleClick is re-created on each render
  const printWhenClick= () => {
    console.log('You Clicked!');
  };
}
```

printWhenClick is a different function object on every re-rendering of Foo.

This is when the useCallback hook comes into play. It returns the memoized function based on the dependencies. If the dependencies are the same between the two re-rendering the useCallback returns the same function object.

The useMemo hook is used in a similar manner. It returns the memoized value instead of the memoized function, as it is done in the useCallback.

Suppose you have a component that renders a long list of grocery foods, as shown below.

```
import React from 'react';
import useSearch from './fetch-foods';
function ShoppingList({ term, onFoodClick }) {
  const foods = useSearch(term);
  const map = food => <div onClick={onFoodClick}>{food}</div>;
  return <div>{foods.map(map)}</div>;
}
export default React.memo(ShoppingList);
```

The list could be big and may include hundreds of food items. To prevent futile list re-renderings, you wrap it into React.memo().

The parent component of ShoppingList provides a handler function to indicate when the user clicks on a food item.

```
import React, { useCallback } from 'react';
export default function MyParent({ term }) {
    const onFoodClick = useCallback(event => {
    console.log('You clicked ', event.currentTarget);
  }, [term]);

return (
    <ShoppingList
      term={term}
      onFoodClick={onFoodClick}
    />
  );
}
```

The onFoodClick callback is memoized by useCallback(). If the term is the same, useCallback() returns the same function object. When the MyParent component re-renders, the onFoodClick function object remains the same and does not break the memoization of ShoppingList.

We can also use useMemo as follows:

```
const List = React.useMemo(() =>
 myList.map(item => ({
           ...item,
           itemProp1: function1(props.first),
           itemProp2: function2(props.second)
  })), [myList]
)
```

useReducer() Hook

As per the official ReactJS definition, useReducer is an alternative to useState. It accepts a reducer of type (state and action) and returns the current state along with the dispatch method. A reducer returns the new state based on the action.

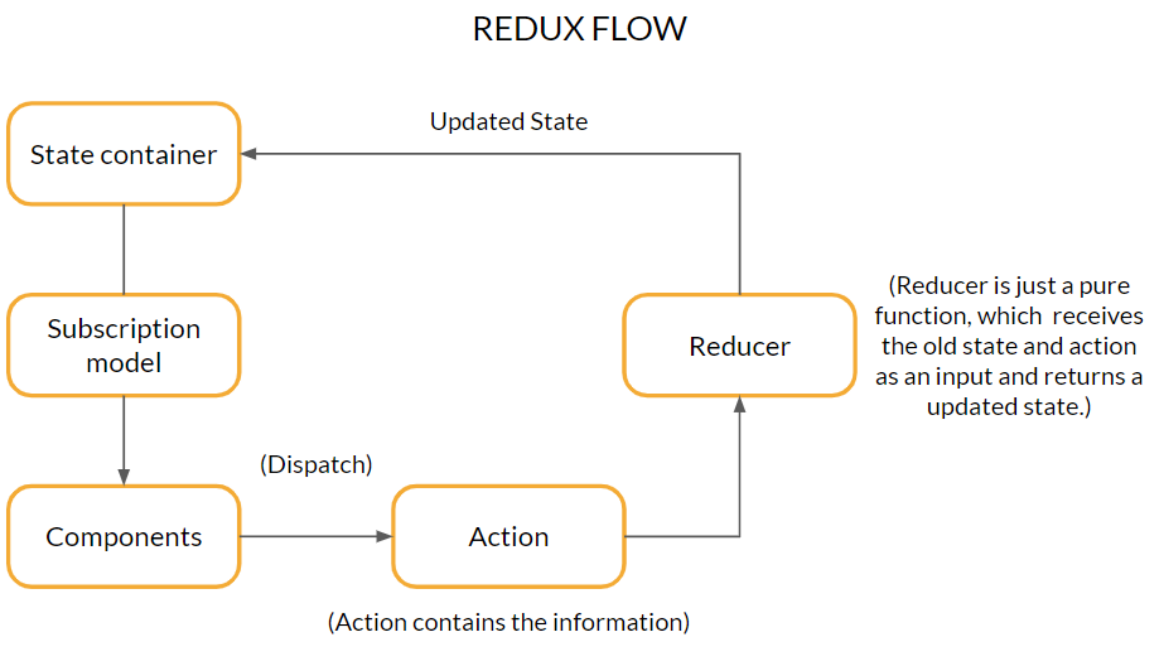
Redux

State management is one of the crucial tasks that needs to be handled in ReactJS. When the application grows and various components are introduced in our application, it becomes harder to maintain the state consistency.

So, to make state management easier, Redux is introduced.

In Redux, using the State container along with the reducer makes the state management quite convenient.

Redux is a standalone JavaScript library that is used to manage the state. You may be wondering what is the use of Redux when you already have quite a few methods to manage the state. Ideally, you should be asking the following question: How do you manage the state when your application grows and you want to share the information between two components that are in different containers?
You may think having a global JavaScript object will help you store your entire application state and allow you to use it from anywhere you want. However, this is not possible because React's reactivity system does not react to the changes that you make in the global variables. Reactivity, among other JavaScript frameworks, is the phenomenon in which changes in the application's state are automatically reflected in the DOM. However, React is not exactly a reactive framework. So, to make our lives easier with state management, we have Redux.
Let's understand the Redux flow with the help of the flow diagram given below.


REDUX FLOW
Updated State
State container
Subscription model
Components
(Dispatch)
Action
Reducer
(Reducer is just a pure function, which receives the old state and action as an input and returns a updated state.)
(Action contains the information)

In Redux, state containers store the entire application's state. As you already know, in React, there are components, which may want to update the state or get the current state. However, you can simply take or update the state directly in a state container, as it will make our application vaguer and it will be hard to find which component does what. So, to update the state in React, components dispatch the actions containing only information; it does not follow any logic as to how to update the state container. The actions are fed to the reducer, which checks the type of information provided in the actions. The reducer is a function that receives the actions and the old state as an input and returns the updated state. Which then stores in the state container and replaces the old state. And then, we have the subscription model which component will subscribe and get the updated state.

Custom Hook

Another interesting aspect of React is that it allows you to build your own custom hooks based on your needs. You can always use functions to perform a task, but what if you want to dispatch an event that changes the state of the component that calls the function? You can accomplish this only by using hooks. A custom hook does not need to have a specific signature. You can decide what it takes as arguments and what it should return.

The main advantage of building a custom hook is that you can extract the component logic into the reusable function.

Using custom hooks, you can dispatch the function that updates the component state that calls the function.

Testing

Testing always plays a crucial role in building a robust application.

Before shipping your product to production, you must always perform some rigorous testing on it so that it does not fail. In this session, you will learn how to perform testing on your application in React using jest as a tool. Here, by ‘testing’, we do not mean clicking on or checking each component’s functioning; we are referring to writing an automated test so that when you change the information in your application, if some components are broken, you get a warning message and are able to rectify that mistake immediately.

You will learn about certain methods that are used for:

- Testing JavaScript functions,
- Testing JavaScript by mocking API calls,
- Testing React Component with enzyme,
- Testing Redux Components, and
- Showing the test coverage.

For writing the test case in jest, you only need to write the test() method.

The test() method accepts the following two parameters:

- Name of the test
- Callback function

In the callback function, you call the function that you want to test and then compare the results using the expect() method.

You can refer to the following code snippet to learn more about this:

```
function add(a,b){
   return a +b;
}

test("Addition of two numbers",()=>{
   const actualResult = add(5,10);
   const expectedResult =15
```

```
    expect(actualResult).toBe(expectedResult);
})
```

In this example, if the expectedResult is equal to the value obtained by calling in the add, then the test case will pass; otherwise, an error will be thrown with a message indicating that the expected value does not match with the value that you have obtained from the function.

You can check the test coverage of your application by only adding the following script to your code:

"react-scripts test --env=jsdom --coverage".

This will display the number of test cases that were passed out of the total written test cases for the application with a detailed explanation.