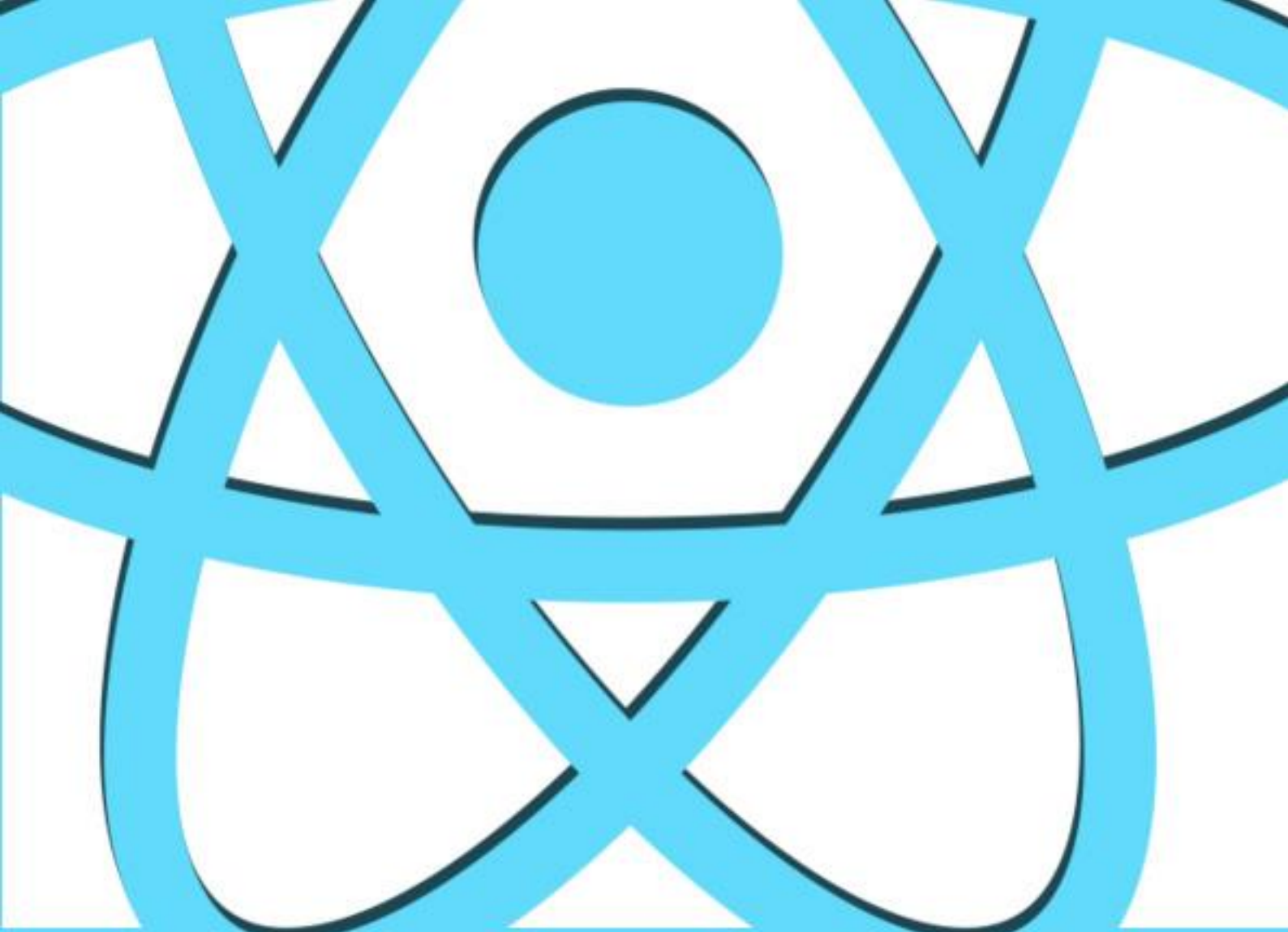


ReactJS

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

React is an open source, JavaScript library for developing user interface (UI) in web application. React is developed and released by **Facebook**. Facebook is continuously working on the *React* library and enhancing it by fixing bugs and introducing new features.

This tutorial starts with the architecture of React, how-to guide to setup projects, creating components, JSX and then walks through advanced concepts like state management, form programming, routing and finally conclude with step by step working example.

Audience

This tutorial is prepared for professionals who are aspiring to make a career in the field of developing front-end web application. This tutorial is intended to make you comfortable in getting started with the React concepts with examples.

Prerequisites

Before proceeding with the various types of concepts given in this tutorial, we assume that the readers have the basic knowledge in HTML, CSS and OOPS concepts. In addition to this, it will be very helpful, if the readers have a sound knowledge in JavaScript.

Copyright & Disclaimer

Table of Contents

1. ReactJS — Introduction

ReactJS is a simple, feature rich, component based JavaScript UI library. It can be used to develop small applications as well as big, complex applications. ReactJS provides minimal and solid feature set to kick-start a web application. React community compliments React library by providing large set of ready-made components to develop web application in a record time. React community also provides advanced concept like state management, routing, etc., on top of the React library.

React versions

The initial version, *0.3.0* of React is released on May, 2013 and the latest version, *17.0.1* is released on October, 2020. The major version introduces breaking changes and the minor version introduces new feature without breaking the existing functionality. Bug fixes are released as and when necessary. React follows the *Sematic Versioning (semver)* principle.

Features

The salient features of *React library* are as follows:

- Solid base architecture
- Extensible architecture
- Component based library
- JSX based design architecture
- Declarative UI library

Benefits

Few benefits of using *React library* are as follows:

- Easy to learn
- Easy to adept in modern as well as legacy application
- Faster way to code a functionality
- Availability of large number of ready-made component
- Large and active community

Applications

Few popular websites powered by *React library* are listed below:

- *Facebook*, popular social media application
- *Instagram*, popular photo sharing application
- *Netflix*, popular media streaming application

- *Code Academy*, popular online training application
- *Reddit*, popular content sharing application

As you see, most popular application in every field is being developed by *React Library*.

2. ReactJS — Installation

This chapter explains the installation of React library and its related tools in your machine. Before moving to the installation, let us verify the prerequisite first.

React provides CLI tools for the developer to fast forward the creation, development and deployment of the React based web application. React CLI tools depends on the *Node.js* and must be installed in your system. Hopefully, you have installed Node.js on your machine. We can check it using the below command:

```
node --version
```

You could see the version of *Nodejs* you might have installed. It is shown as below for me,

```
v14.2.0
```

If *Nodejs* is not installed, you can download and install by visiting https://nodejs.org/en/download/.

Toolchain

To develop lightweight features such as form validation, model dialog, etc., React library can be directly included into the web application through content delivery network (CDN). It is similar to using jQuery library in a web application. For moderate to big application, it is advised to write the application as multiple files and then use bundler such as webpack, parcel, rollup, etc., to compile and bundle the application before deploying the code.

React toolchain helps to create, build, run and deploy the React application. React toolchain basically provides a starter project template with all necessary code to bootstrap the application.

Some of the popular toolchain to develop React applications are:

- Create React App - SPA oriented toolchain
- Next.js - server-side rendering oriented toolchain
- Gatsby - Static content oriented toolchain

Tools required to develop a React application are:

- The *serve*, a static server to serve our application during development
- Babel compiler
- Create React App CLI

Let us learn the basics of the above mentioned tools and how to install those in this chapter.

The *serve* static server

The *serve* is a lightweight web server. It serves static site and single page application. It loads fast and consume minimum memory. It can be used to serve a React application. Let us install the tool using *npm* package manager in our system.

```
npm install serve -g
```

Let us create a simple static site and serve the application using *serve* app.

Open a command prompt and go to your workspace.

```
cd /go/to/your/workspace
```

Create a new folder, *static_site* and change directory to newly created folder.

```
mkdir static_site
cd static_site
```

Next, create a simple webpage inside the folder using your favorite html editor.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Static website</title>
    </head>
    <body>
        <div><h1>Hello!</h1></div>
    </body>
</html>
```

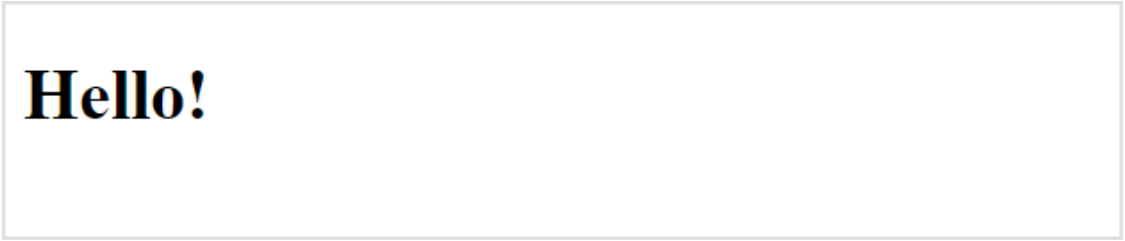
Next, run the *serve* command.

```
serve .
```

We can also serve single file, *index.html* instead of the whole folder.

```
serve ./index.html
```

Next, open the browser and enter *http://localhost:5000* in the address bar and press enter. *serve* application will serve our webpage as shown below.



The *serve* will serve the application using default port, 5000. If it is not available, it will pick up a random port and specify it.

```
|   Serving!                                          |
    |                                                   |
    |   - Local:            http://localhost:57311      |
    |   - On Your Network:  http://192.168.56.1:57311   |
    |                                                   |
    |   This port was picked because 5000 is in use.    |
    |                                                   |
    |   Copied local address to clipboard!
```

Babel compiler

Babel is a JavaScript compiler which compiles many variant (es2015, es6, etc.,) of JavaScript into standard JavaScript code supported by all browsers. React uses JSX, an extension of JavaScript to design the user interface code. Babel is used to compile the *JSX* code into JavaScript code.

To install *Babel* and it's React companion, run the below command:

```
npm install babel-cli@6 babel-preset-react-app@3 -g

...
...

+ babel-cli@6.26.0
+ babel-preset-react-app@3.1.2
updated 2 packages in 8.685s
```

Babel helps us to write our application in next generation of advanced JavaScript syntax.

Create React App toolchain

Create React App is a modern CLI tool to create single page React application. It is the standard tool supported by React community. It handles babel compiler as well. Let us install *Create React App* in our local system.

```
> npm install -g create-react-app

+ create-react-app@4.0.1
added 6 packages from 4 contributors, removed 37 packages and updated 12
packages in 4.693s
```

Updating the toolchain

React Create App toolchain uses the *react-scripts* package to build and run the application. Once we started working on the application, we can update the react-script to the latest version at any time using *npm* package manager.

```
npm install react-scripts@latest
```

Advantages of using React toolchain

React toolchain provides lot of features out of the box. Some of the advantages of using React toolchain are:

- Predefined and standard structure of the application.
- Ready-made project template for different type of application.
- Development web server is included.
- Easy way to include third party React components.
- Default setup to test the application.

3. ReactJS — Architecture

React library is built on a solid foundation. It is simple, flexible and extensible. As we learned earlier, React is a library to create user interface in a web application. React's primary purpose is to enable the developer to create user interface using pure JavaScript. Normally, every user interface library introduces a new template language (which we need to learn) to design the user interface and provides an option to write logic, either inside the template or separately.

Instead of introducing new template language, React introduces three simple concepts as given below:

React elements

JavaScript representation of HTML DOM. React provides an API, ***React.createElement*** to create *React Element*.

JSX

A JavaScript extension to design user interface. JSX is an XML based, extensible language supporting HTML syntax with little modification. JSX can be compiled to *React Elements* and used to create user interface.

React component

React component is the primary building block of the React application. It uses *React elements* and *JSX* to design its user interface. React component is basically a JavaScript class (extends the ***React.component*** class) or pure JavaScript function. React component has properties, state management, life cycle and event handler. React component can be able to do simple as well as advanced logic.

Let us learn more about components in the *React Component* chapter.

Workflow of a React application

Let us understand the workflow of a React application in this chapter by creating and analyzing a simple React application.

Open a command prompt and go to your workspace.

```
cd /go/to/your/workspace
```

Next, create a folder, *static_site* and change directory to newly created folder.

```
mkdir static_site
cd static_site
```

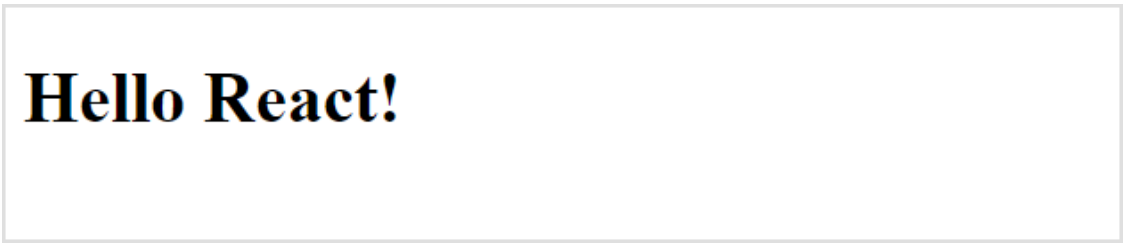
Next, create a file, *hello.html* and write a simple React application.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>React Application</title>
    </head>
    <body>
        <div id="react-app"></div>

        <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
        <script language="JavaScript">
            element = React.createElement('h1', {}, 'Hello React!')
            ReactDOM.render(element, document.getElementById('react-app'));
        </script>
    </body>
</html>
```

Next, serve the application using serve web server.

```
serve ./hello.html
```

Next, open your favorite browser. Enter *http://localhost:5000* in the address bar and then press enter.



Let us analyse the code and do little modification to better understand the React application.

Here, we are using two API provided by the React library.

React.createElement

Used to create React elements. It expects three parameters:

- Element tag
- Element attributes as object
- Element content - It can contain nested React element as well

ReactDOM.render

Used to render the element into the container. It expects two parameters:

- React Element OR JSX
- Root element of the webpage

Nested React element

As ***React.createElement*** allows nested React element, let us add nested element as shown below:

```
<script language="JavaScript">
    element = React.createElement('div', {},
        React.createElement('h1', {}, 'Hello React!'));
    ReactDOM.render(element, document.getElementById('react-app'));
</script>
```

It will generate the below content:

```
<div><h1>Hello React!</h1></div>
```

Use JSX

Next, let us remove the React element entirely and introduce JSX syntax as shown below:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>React Application</title>
    </head>
    <body>
        <div id="react-app"></div>

        <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
        <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
        <script type="text/babel">
            ReactDOM.render(
                <div><h1>Hello React!</h1></div>,
                document.getElementById('react-app') );
        </script>
    </body>
</html>
```

Here, we have included babel to convert JSX into JavaScript and added *type="text/babel"* in the script tag.

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
<script type="text/babel">
...
...
</script>
```

Next, run the application and open the browser. The output of the application is as follows:

Hello JSX!

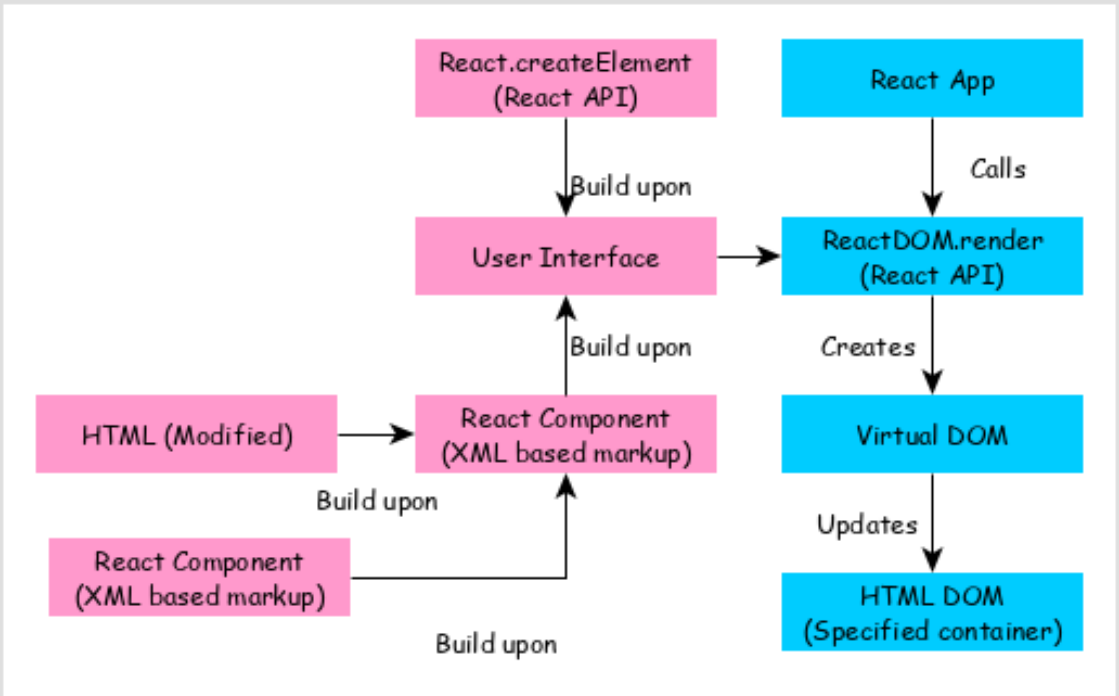
Next, let us create a new React component, *Greeting* and then try to use it in the webpage.

```
<script type="text/babel">
    function Greeting() {
        return <div><h1>Hello JSX!</h1></div>
    }
    ReactDOM.render(
        <Greeting />,
        document.getElementById('react-app') );
</script>
```

The result is same and as shown below:

Hello JSX!

By analyzing the application, we can visualize the workflow of the React application as shown in the below diagram.



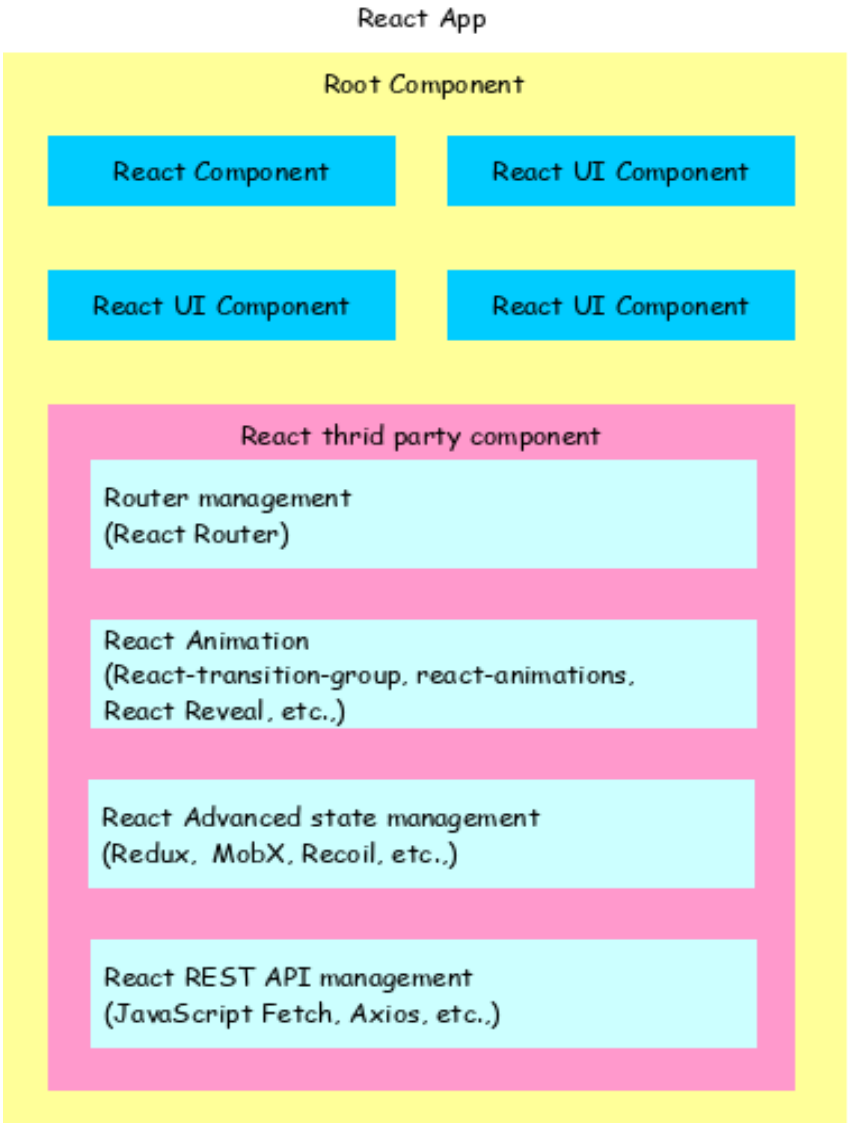
React app calls ***ReactDOM.render*** method by passing the user interface created using React component (coded in either JSX or React element format) and the container to render the user interface.

ReactDOM.render processes the JSX or React element and emits Virtual DOM.

Virtual DOM will be merged and rendered into the container.

Architecture of the React Application

React library is just UI library and it does not enforce any particular pattern to write a complex application. Developers are free to choose the design pattern of their choice. React community advocates certain design pattern. One of the patterns is *Flux* pattern. React library also provides lot of concepts like Higher Order component, Context, Render props, Refs etc., to write better code. React Hooks is evolving concept to do state management in big projects. Let us try to understand the high level architecture of a React application.



- React app starts with a single root component.
- Root component is build using one or more component.
- Each component can be nested with other component to any level.
- Composition is one of the core concepts of React library. So, each component is build by composing smaller components instead of inheriting one component from another component.
- Most of the components are user interface components.
- React app can include third party component for specific purpose such as routing, animation, state management, etc.

4. React — Creating a React Application

As we learned earlier, React library can be used in both simple and complex application. Simple application normally includes the React library in its script section. In complex application, developers have to split the code into multiple files and organize the code into a standard structure. Here, React toolchain provides pre-defined structure to bootstrap the application. Also, developers are free to use their own project structure to organize the code.

Let us see how to create simple as well as complex React application:

- Simple application using CDN
- Complex application using *React Create App* cli
- Complex application using customized method

Using CDN

Let us learn how to use content delivery network to include React in a simple web page.

Open a terminal and go to your workspace.

```
cd /go/to/your/workspace
```

Next, create a folder, *static_site* and change directory to newly created folder.

```
mkdir static_site
cd static_site
```

Next, create a new HTML file, *hello.html.*

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Simple React app</title>
    </head>
    <body>
    </body>
</html>
```

Next, include *React library.*

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Simple React app</title>
    </head>
    <body>
```

```
        <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
    </body>
</html>
```

Here,

- We are using ***unpkg*** CDN. ***unpkg*** is an open source, global content delivery network supporting ***npm*** packages.
- ***@17*** represent the version of the *React library*
- This is the development version of the *React library* with debugging option. To deploy the application in the production environment, use below scripts.

```
<script src="https://unpkg.com/react@17/umd/react.production.min.js"
crossorigin></script>
<script src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"
crossorigin></script>
```

Now, we are ready to use *React library* in our webpage.

Next, introduce a ***div*** tag with id ***react-app***.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>React based application</title>
    </head>
    <body>
        <div id="react-app"></div>

        <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
    </body>
</html>
```

The ***react-app*** is a placeholder container and React will work inside the container. We can use any name for the placeholder container relevant to our application.

Next, create a script section at the end of the document and use React feature to create an element.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>React based application</title>
    </head>
```

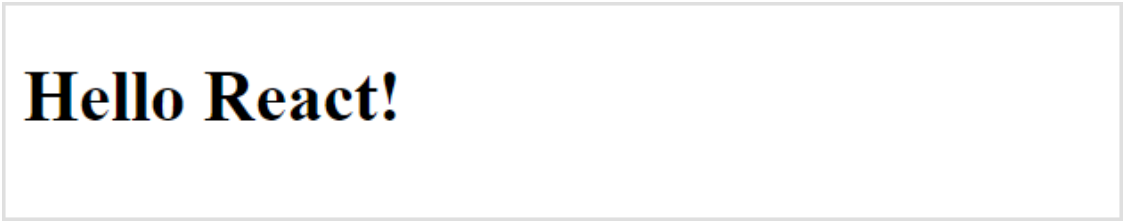
```
    <body>
        <div id="react-app"></div>

        <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
        <script language="JavaScript">
            element = React.createElement('h1', {}, 'Hello React!')
            ReactDOM.render(element, document.getElementById('react-app'));
        </script>
    </body>
</html>
```

Here, the application uses *React.createElement* and *ReactDOM.render* methods provided by *React Library* to dynamically create a HTML element and place it inside the **react-app** section.

Next, serve the application using *serve* web server.

```
serve ./hello.html
```

Next, open the browser and enter *http://localhost:5000* in the address bar and press enter. *serve* application will serve our webpage as shown below.



We can use the same steps to use React in the existing website as well. This method is very easy to use and consume React library. It can be used to do simple to moderate feature in a website. It can be used in new as well as existing application along with other libraries. This method is suitable for static website with few dynamic section like contact form, simple payment option, etc., To create advanced single page application (SPA), we need to use React tools. Let us learn how to create a SPA using React tools in upcoming chapter.

Using Create React App tool

Let us learn to create an expense management application using *Create React App* tool.

Open a terminal and go to your workspace.

```
> cd /go/to/your/workspace
```

Next, create a new React application using *Create React App* tool.

```
> create-react-app expense-manager
```

It will a create new folder ***expense-manager*** with startup template code.

Next, go to ***expense-manager*** folder and install the necessary library.

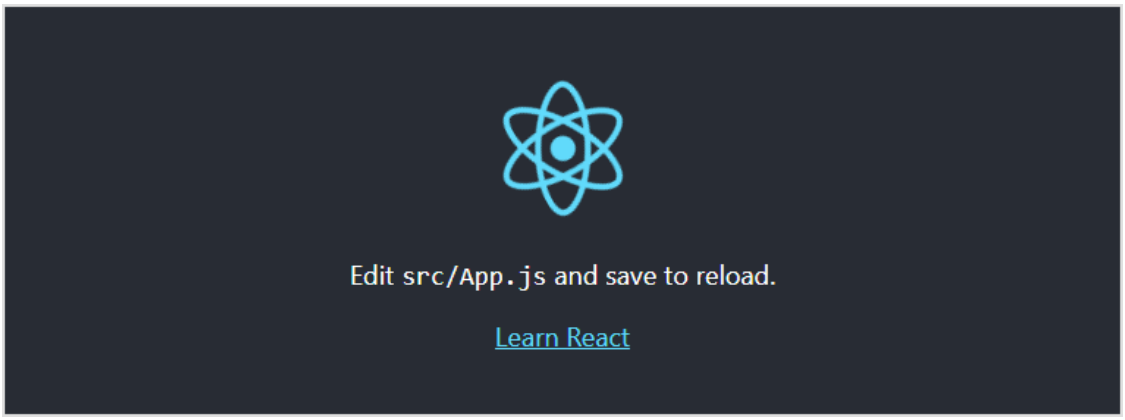
```
cd expense-manager
npm install
```

The *npm install* will install the necessary library under *node_modules* folder.

Next, start the application.

```
npm start

Compiled successfully!

You can now view react-cra-web-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

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

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter. The development web server will serve our webpage as shown below.



Let us analyse the structure of our React application.

Files and folders

The content of the React application is as follows:

```
|-- README.md
|-- node_modules
|-- package-lock.json
|-- package.json
|-- public
|   |-- favicon.ico
|   |-- index.html
|   |-- logo192.png
|   |-- logo512.png
|   |-- manifest.json
|   `-- robots.txt
`-- src
```

```
    |-- App.css
    |-- App.js
    |-- App.test.js
    |-- index.css
    |-- index.js
    |-- logo.svg
    |-- reportWebVitals.js
    `-- setupTests.js
```

Here,

The *package.json* is the core file representing the project. It configures the entire project and consists of project name, project dependencies, and commands to build and run the application.

```
{
  "name": "expense-manager",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.6",
    "@testing-library/react": "^11.2.2",
    "@testing-library/user-event": "^12.6.0",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.1",
    "web-vitals": "^0.2.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

The *package.json* refers the below React library in its dependency section.

- *react* and *react-dom* are core react libraries used to develop web application.
- *web-vitals* are general library to support application in different browser.
- *react-scripts* are core react scripts used to build and run application.
- *@testing-library/jest-dom*, *@testing-library/react* and *@testing-library/user-event* are testing libary used to test the application after development.

- The ***public* folder** - Contains the core file, *index.html* and other web resources like images, logos, robots, etc., *index.html* loads our react application and render it in user's browser.
- The *src* folder - Contains the actual code of the application. We will check it next section.

Source code of the application

Let us check the each and every source code document of the application in this chapter.

- The *index.js* - Entry point of our application. It uses *ReactDOM.render* method to kick-start and start the application. The code is as follows:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Here,

React.StrictMode is a build-in component used to prevent unexpected bugs by analysing the component for unsafe lifecycle, unsafe API usage, depreciated API usage, etc., and throwing the relevant warning.

- *App* is our first custom and root component of the application. All other components will be rendered inside the *App* component.

The *index.css* - Used to styles of the entire application. Let us remove all styles and start with fresh code.

App.js - Root component of our application. Let us replace the existing JSX and show simple hello react message as shown below:

```
import './App.css';

function App() {
  return (
    <h1>Hello React!</h1>
  );
}

export default App;
```

- ***App.css* -** Used to style the *App* component. Let us remove all styles and start with fresh code.
- ***App.test.js* -** Used to write unit test function for our component.
- ***setupTests.js* -** Used to setup the testing framework for our application.
- ***reportWebVitals.js* -** Generic web application startup code to support all browsers.
- ***logo.svg* -** Logo in SVG format and can be loaded into our application using *import* keyword. Let us remove it from the project.

Customize the code

Let us remove the default source code of the application and bootstrap the application to better understand the internals of React application.

Delete all files under *src* and *public* folder.

Next, create a folder, *components* under *src* to include our React components. The idea is to create two files, *<component>.js* to write the component logic and *<component.css>* to include the component specific styles.

The final structure of the application will be as follows:

```
|-- package-lock.json
|-- package.json
`-- public
    |-- index.html
`-- src
    |-- index.js
    `-- components
    |   |-- mycom.js
    |   |-- mycom.css
```

Let us create a new component, HelloWorld to confirm our setup is working fine. Create a file, HelloWorld.js under components folder and write a simple component to emit Hello World message.

```
import React from "react";

class HelloWorld extends React.Component {
  render() {
    return (
      <div>
```

```
        <h1>Hello World!</h1>
      </div>
    );
  }
}
export default HelloWorld;
```

Next, create our main file, *index.js* under *src* folder and call our newly created component.

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

ReactDOM.render(
  <React.StrictMode>
    <HelloWorld />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Next, create a html file, *index.html* (under *public* folder*), which will be our entry point of the application.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Expense Manager</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Run the application

Let us run the application by invoking the *start* script configured in *package.json* file.

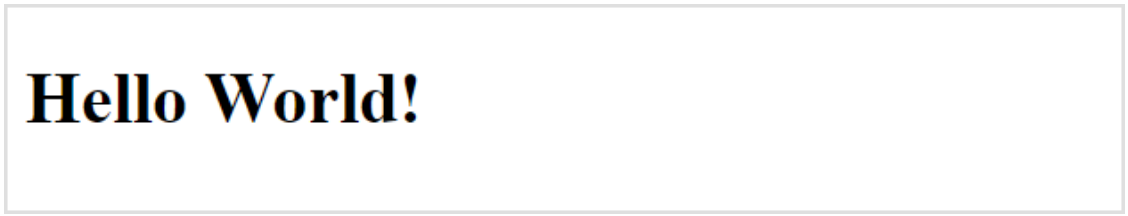
```
> npm start
```

It will start the application in the local system and can be accessed through browser @ http://localhost:3000/.

```
> expense-manager@0.1.0 start D:\path\to\expense-manager
> react-scripts start

i ｢wds｣: Project is running at http://192.168.56.1/
i ｢wds｣: webpack output is served from
i ｢wds｣: Content not from webpack is served from D:\path\to\expense-
manager\public
i ｢wds｣: 404s will fallback to /
```

```
Starting the development server...
Compiled successfully!

You can now view expense-manager in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

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

Open your favorite browser and go to http://localhost:3000. The result of the application is as shown below:



Using custom solution

As we learned earlier, *Create react app* is the recommended tool to kick-start the React application. It includes everything to develop React application. But sometimes, application does not need all the feature provided by *Crzzeate React App* and we want our application to be small and tidy. Then, we can use our own customized solution to create React application with just enough dependency to support our application.

To create a custom project, we need to have basic knowledge about four items.

- **Package manager -** High level management of application. We are using *npm* as our default package manager.
- **Compiler -** Compiles the JavaScript variants into standard JavaScript supported by browser. We are using *Babel* as our default compiler.
- **Bundler -** Bundles the multiple sources (JavaScript, html and css) into a single deployable code. *Create React App* uses webpack as its bundler. Let us learn how to use *Rollup* and *Parcel* bundler in the upcoming section.
- **Webserver -** Starts the development server and launch our application. *Create React App* uses an internal webserver and we can use *serve* as our development server.

Using Rollup bundler

Rollup is one of the small and fast JavaScript bundlers. Let us learn how to use rollup bundler in this chapter.

Open a terminal and go to your workspace.

```
cd /go/to/your/workspace
```

Next, create a folder, *expense-manager-rollup* and move to newly created folder. Also, open the folder in your favorite editor or IDE.

```
mkdir expense-manager-rollup
cd expense-manager-rollup
```

Next, create and initialize the project.

```
npm init -y
```

Next, install React libraries (*react* and *react-dom*).

```
npm install react@^17.0.0 react-dom@^17.0.0 --save
```

Next, install babel and its preset libraries as development dependency.

```
npm install @babel/preset-env @babel/preset-react @babel/core @babel/plugin-
proposal-class-properties -D
```

Next, install rollup and its plugin libraries as development dependency.

```
npm i -D rollup postcss@8.1 @rollup/plugin-babel @rollup/plugin-commonjs
@rollup/plugin-node-resolve @rollup/plugin-replace rollup-plugin-livereload
rollup-plugin-postcss rollup-plugin-serve postcss@8.1 postcss-modules@4 rollup-
plugin-postcss
```

Next, install corejs and regenerator runtime for async programming.

```
npm i regenerator-runtime core-js
```

Next, create a babel configuration file, *.babelrc* under the root folder to configure the babel compiler.

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage",
                "corejs": 3,
                "targets": "> 0.25%, not dead"
            }
        ],
        "@babel/preset-react"
    ],
    "plugins": [
        "@babel/plugin-proposal-class-properties"
    ]
}
```

Next, create a *rollup.config.js* file in the root folder to configure the rollup bundler.

```
import babel from '@rollup/plugin-babel';
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
import replace from '@rollup/plugin-replace';

import serve from 'rollup-plugin-serve';
import livereload from 'rollup-plugin-livereload';

import postcss from 'rollup-plugin-postcss'

export default {
    input: 'src/index.js',
    output: {
        file: 'public/index.js',
        format: 'iife',
    },
    plugins: [
        commonjs({
            include: [
                'node_modules/**',
            ],
            exclude: [
                'node_modules/process-es6/**',
            ],
        }),
        resolve(),
        babel({
            exclude: 'node_modules/**'
        }),
        replace({
            'process.env.NODE_ENV': JSON.stringify('production'),
        }),
        postcss({
            autoModules: true
        }),
        livereload('public'),
        serve({
            contentBase: 'public',
            port: 3000,
            open: true,
        }), // index.html should be in root of project
    ]
}
```

Next, update the *package.json* and include our entry point (*public/index.js* and *public/styles.css*) and command to build and run the application.

```
...
"main": "public/index.js",
"style": "public/styles.css",
"files": [
  "public"
],
"scripts": {
```

```
  "start": "rollup -c -w",
  "build": "rollup"
},
...
```

Next, create a *src* folder in the root directory of the application, which will hold all the source code of the application.

Next, create a folder, *components* under *src* to include our React components. The idea is to create two files, *<component>.js* to write the component logic and *<component.css>* to include the component specific styles.

The final structure of the application will be as follows:

```
|-- package-lock.json
|-- package.json
|-- rollup.config.js
|-- .babelrc
`-- public
    |-- index.html
`-- src
    |-- index.js
    `-- components
    |   |-- mycom.js
    |   |-- mycom.css
```

Let us create a new component, *HelloWorld* to confirm our setup is working fine. Create a file, *HelloWorld.js* under components folder and write a simple component to emit *Hello World* message.

```
import React from "react";

class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello World!</h1>
      </div>
    );
  }
}
export default HelloWorld;
```

Next, create our main file, *index.js* under *src* folder and call our newly created component.

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

ReactDOM.render(
  <React.StrictMode>
    <HelloWorld />
  </React.StrictMode>,
```

```
  document.getElementById('root')
);
```

Next, create a *public* folder in the root directory.

Next, create a html file, *index.html* (under *public* folder*), which will be our entry point of the application.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Expense Manager :: Rollup version</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/JavaScript" src="./index.js"></script>
  </body>
</html>
```

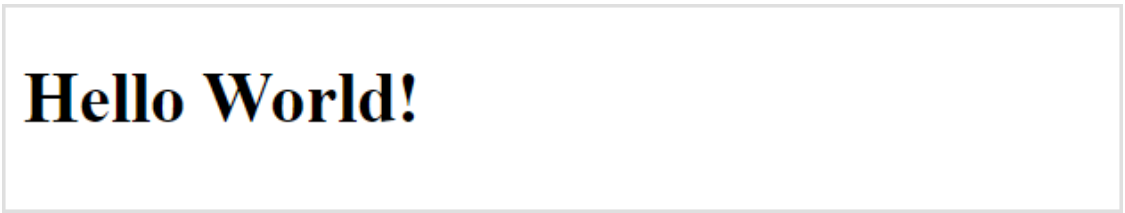
Next, build and run the application.

```
npm start
```

The *npm* build command will execute the *rollup* and bundle our application into a single file, *dist/index.js* file and start serving the application. The *dev* command will recompile the code whenever the source code is changed and also reload the changes in the browser.

```
> expense-manager-rollup@1.0.0 build /path/to/your/workspace/expense-manager-
rollup
> rollup -c


rollup v2.36.1
bundles src/index.js → dist\index.js...
LiveReload enabled
http://localhost:10001 ->  /path/to/your/workspace/expense-manager-rollup/dist
created dist\index.js in 4.7s

waiting for changes...
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter. *serve* application will serve our webpage as shown below.



Using Parcel bundler

Parcel is fast bundler with zero configuration. It expects just the entry point of the application and it will resolve the dependency itself and bundle the application. Let us learn how to use parcel bundler in this chapter.

First, install the parcel bundler.

```
npm install -g parcel-bundler
```

Open a terminal and go to your workspace.

```
cd /go/to/your/workspace
```

Next, create a folder, *expense-manager-parcel* and move to newly created folder. Also, open the folder in your favorite editor or IDE.

```
mkdir expense-manager-parcel
cd expense-manager-parcel
```

Next, create and initialize the project.

```
npm init -y
```

Next, install React libraries (*react* and *react-dom*).

```
npm install react@^17.0.0 react-dom@^17.0.0 --save
```

Next, install babel and its preset libraries as development dependency.

```
npm install @babel/preset-env @babel/preset-react @babel/core @babel/plugin-
proposal-class-properties -D
```

Next, create a babel configuration file, *.babelrc* under the root folder to configure the babel compiler.

```
{
    "presets": [
        "@babel/preset-env",
        "@babel/preset-react"
    ],
    "plugins": [
        "@babel/plugin-proposal-class-properties"
    ]
}
```

Next, update the *package.json* and include our entry point (*src/index.js*) and commands to build and run the application.

```
...
"main": "src/index.js",
"scripts": {
    "start": "parcel public/index.html",
```

```
    "build": "parcel build public/index.html --out-dir dist"
},
...
```

Next, create a *src* folder in the root directory of the application, which will hold all the source code of the application.

Next, create a folder, *components* under *src* to include our React components. The idea is to create two files, *<component>.js* to write the component logic and *<component.css>* to include the component specific styles.

The final structure of the application will be as follows:

```
|-- package-lock.json
|-- package.json
|-- .babelrc
`-- public
    |-- index.html
`-- src
    |-- index.js
    `-- components
    |   |-- mycom.js
    |   |-- mycom.css
```

Let us create a new component, *HelloWorld* to confirm our setup is working fine. Create a file, *HelloWorld.js* under components folder and write a simple component to emit *Hello World* message.

```
import React from "react";

class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello World!</h1>
      </div>
    );
  }
}
export default HelloWorld;
```

Next, create our main file, *index.js* under *src* folder and call our newly created component.

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

ReactDOM.render(
  <React.StrictMode>
    <HelloWorld />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Next, create a *public* folder in the root directory.

Next, create a html file, *index.html* (in the *public* folder), which will be our entry point of the application.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Expense Manager :: Parcel version</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/JavaScript" src="../src/index.js"></script>
  </body>
</html>
```

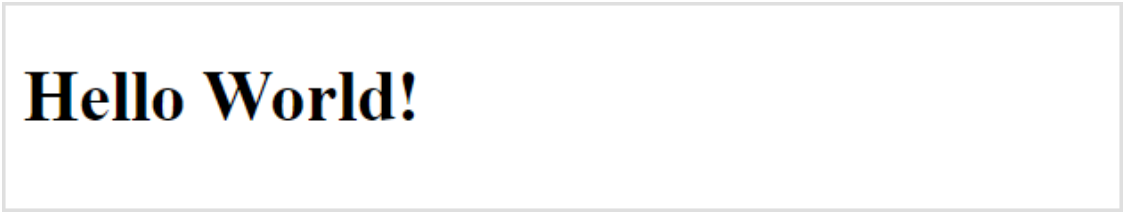
Next, build and run the application.

```
npm start
```

The *npm* build command will execute the *parcel* command. It will bundle and serve the application on the fly. It recompiles whenever the source code is changed and also reload the changes in the browser.

```
> expense-manager-parcel@1.0.0 dev /go/to/your/workspace/expense-manager-parcel
> parcel index.html

Server running at http://localhost:1234
√  Built in 10.41s.
```

Next, open the browser and enter *http://localhost:1234* in the address bar and press enter.



To create the production bundle of the application to deploy it in production server, use *build* command. It will generate a *index.js* file with all the bundled source code under *dist* folder.

```
npm run build

> expense-manager-parcel@1.0.0 build /go/to/your/workspace/expense-manager-
parcel
> parcel build index.html --out-dir dist

√  Built in 6.42s.

dist\src.80621d09.js.map    270.23 KB     79ms
```

```
dist\src.80621d09.js        131.49 KB    4.67s
dist\index.html                 221 B    1.63s
```

5. React — JSX

As we learned earlier, React JSX is an extension to JavaScript. It enables developer to create virtual DOM using XML syntax. It compiles down to pure JavaScript (*React.createElement* function calls). Since it compiles to JavaScript, it can be used inside any valid JavaScript code. For example, below codes are perfectly valid.

- Assign to a variable.

```
var greeting = <h1>Hello React!</h1>
```

- Assign to a variable based on a condition.

```
var canGreet = true;
if(canGreet) {
    greeting = <h1>Hello React!</h1>
}
```

- Can be used as return value of a function.

```
function Greeting() {
    return <h1>Hello React!</h1>
}
greeting = Greeting()
```

- Can be used as argument of a function.

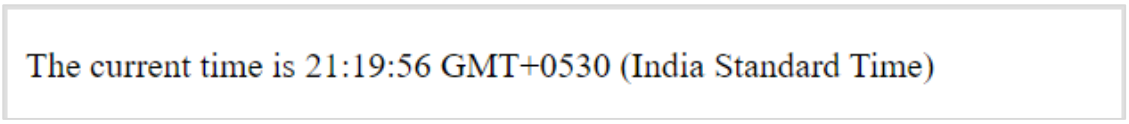
```
function Greet(message) {
    ReactDOM.render(message, document.getElementById('react-app')
}
Greet(<h1>Hello React!</h1>)
```

Expressions

JSX supports expression in pure JavaScript syntax. Expression has to be enclosed inside the curly braces, ***{}.*** Expression can contain all variables available in the context, where the JSX is defined. Let us create simple JSX with expression.

```
<script type="text/babel">
    var cTime = new Date().toTimeString();
    ReactDOM.render(
        <div><p>The current time is {cTime}</p></div>,
        document.getElementById('react-app') );
</script>
```

Here, *cTime* used in the JSX using expression. The output of the above code is as follows,



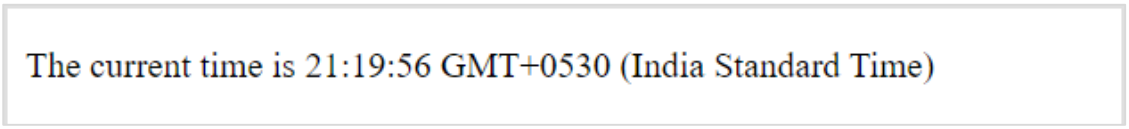
One of the positive side effects of using expression in JSX is that it prevents *Injection attacks* as it converts any string into html safe string.

Functions

JSX supports user defined JavaScript function. Function usage is similar to expression. Let us create a simple function and use it inside JSX.

```
<script type="text/babel">
    function getCurrentTime() {
        return new Date().toTimeString();
    }
    ReactDOM.render(
        <div><p>The current time is {getCurrentTime()}</p></div>,
        document.getElementById('react-app') );
</script>
```

Here, *getCurrentTime()* is used get the current time and the output is similar as specified below:



Attributes

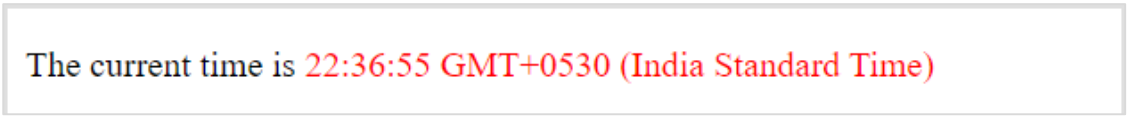
JSX supports HTML like attributes. All HTML tags and its attributes are supported. Attributes has to be specified using *camelCase* convention (and it follows JavaScript DOM API) instead of normal HTML attribute name. For example, *class* attribute in HTML has to be defined as *className*. The following are few other examples:

- *htmlFor* instead of *for*
- *tabIndex* instead of *tabindex*
- *onClick* instead of *onclick*

```
<style>
    .red { color: red }
</style>

<script type="text/babel">
    function getCurrentTime() {
        return new Date().toTimeString();
    }
    ReactDOM.render(
        <div><p>The current time is <span
className="red">{getCurrentTime()}</span></p></div>,
```

```
        document.getElementById('react-app') );
</script>
```

The output is as follows:



Expression in attributes

JSX supports expression to be specified inside the attributes. In attributes, double quote should not be used along with expression. Either expression or string using double quote has to be used. The above example can be changed to use expression in attributes.

```
<style>
    .red { color: red }
</style>

<script type="text/babel">
    function getCurrentTime() {
        return new Date().toTimeString();
    }

    var class_name = "red";
    ReactDOM.render(
        <div><p>The current time is <span
className={class_name}>{getCurrentTime()}</span></p></div>,
        document.getElementById('react-app') );
</script>
```

6. ReactJS — Component

React component is the building block of a React application. Let us learn how to create a new React component and the features of React components in this chapter.

A React component represents a small chunk of user interface in a webpage. The primary job of a React component is to render its user interface and update it whenever its internal state is changed. In addition to rendering the UI, it manages the events belongs to its user interface. To summarize, React component provides below functionalities.

- Initial rendering of the user interface.
- Management and handling of events.
- Updating the user interface whenever the internal state is changed.

React component accomplish these feature using three concepts:

- **Properties -** Enables the component to receive input.
- **Events -** Enable the component to manage DOM events and end-user interaction.
- **State -** Enable the component to stay stateful. Stateful component updates its UI with respect to its state.

Let us learn all the concept one-by-one in the upcoming chapters.

Creating a React component

React library has two component types. The types are categorized based on the way it is being created.

- Function component - Uses plain JavaScript function.
- ES6 class component - Uses ES6 class.

The core difference between function and class component are:

- Function components are very minimal in nature. Its only requirement is to return a *React element*.

```
function Hello() {
    return '<div>Hello</div>'
}
```

The same functionality can be done using ES6 class component with little extra coding.

```
class ExpenseEntryItem extends React.Component {
    render() {
        return (
            <div>Hello</div>
        );
```

```
    }
}
```

- Class components supports state management out of the box whereas function components does not support state management. But, React provides a hook, *useState()* for the function components to maintain its state.

- Class component have a life cycle and access to each life cycle events through dedicated callback apis. Function component does not have life cycle. Again, React provides a hook, *useEffect()* for the function component to access different stages of the component.

Creating a class component

Let us create a new React component (in our *expense-manager* app), *ExpenseEntryItem* to showcase an expense entry item. Expense entry item consists of name, amount, date and category. The object representation of the expense entry item is:

```
{
    'name': 'Mango juice',
    'amount': 30.00,
    'spend_date': '2020-10-10'
    'category': 'Food',
}
```

Open *expense-manager* application in your favorite editor.

Next, create a file, *ExpenseEntryItem.css* under *src/components* folder to style our component.

Next, create a file, *ExpenseEntryItem.js* under *src/components* folder by extending *React.Component.*

```
import React from 'react';
import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {
}
```

Next, create a method *render* inside the *ExpenseEntryItem* class.

```
class ExpenseEntryItem extends React.Component {
    render() {
    }
}
```

Next, create the user interface using JSX and return it from *render* method.

```
class ExpenseEntryItem extends React.Component {
    render() {
        return (
            <div>
                <div><b>Item:</b> <em>Mango Juice</em></div>
```

```
                <div><b>Amount:</b> <em>30.00</em></div>
                <div><b>Spend Date:</b> <em>2020-10-10</em></div>
                <div><b>Category:</b> <em>Food</em></div>
            </div>
        );
    }
}
```

Next, specify the component as default export class.

```
import React from 'react';
import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {
    render() {
        return (
            <div>
                <div><b>Item:</b> <em>Mango Juice</em></div>
                <div><b>Amount:</b> <em>30.00</em></div>
                <div><b>Spend Date:</b> <em>2020-10-10</em></div>
                <div><b>Category:</b> <em>Food</em></div>
            </div>
        );
    }
}

export default ExpenseEntryItem;
```

Now, we successfully created our first React component. Let us use our newly created component in *index.js*.

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

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem />
  </React.StrictMode>,
  document.getElementById('root')
);
```

The same functionality can be done in a webpage using CDN as shown below:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>React application :: ExpenseEntryItem component</title>
    </head>
    <body>
        <div id="react-app"></div>

        <script src="https://unpkg.com/react@17/umd/react.development.js"
```

```
crossorigin></script>
        <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
        <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
        <script type="text/babel">
            class ExpenseEntryItem extends React.Component {
                render() {
                    return (
                        <div>
                            <div><b>Item:</b> <em>Mango Juice</em></div>
                            <div><b>Amount:</b> <em>30.00</em></div>
                            <div><b>Spend Date:</b> <em>2020-10-10</em></div>
                            <div><b>Category:</b> <em>Food</em></div>
                        </div>
                    );
                }
            }
            ReactDOM.render(
                <ExpenseEntryItem />,
                document.getElementById('react-app') );
        </script>
    </body>
</html>
```

Next, serve the application using npm command.

```
npm start
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter.

Item: *Mango Juice*
Amount: *30.00*
Spend Date: *2020-10-10*
Category: *Food*

Creating a function component

React component can also be created using plain JavaScript function but with limited features. Function based React component does not support state management and other advanced features. It can be used to quickly create a simple component.

The above *ExpenseEntryItem* can be rewritten in function as specified below:

```
function ExpenseEntryItem() {
    return (
        <div>
            <div><b>Item:</b> <em>Mango Juice</em></div>
            <div><b>Amount:</b> <em>30.00</em></div>
            <div><b>Spend Date:</b> <em>2020-10-10</em></div>
```

```
            <div><b>Category:</b> <em>Food</em></div>
        </div>
    );
}
```

Here, we just included the render functionality and it is enough to create a simple React component.

7. React — Styling

In general, React allows component to be styled using CSS class through *className* attribute. Since, the React JSX supports JavaScript expression, a lot of common CSS methodology can be used. Some of the top options are as follows:

- CSS stylesheet - Normal CSS styles along with *className*
- Inline styling - CSS styles as JavaScript objects along with camelCase properties.
- CSS Modules - Locally scoped CSS styles.
- Styled component - Component level styles.
- Sass stylesheet - Supports Sass based CSS styles by converting the styles to normal css at build time.
- Post processing stylesheet - Supports Post processing styles by converting the styles to normal css at build time.

Let use learn how to apply the three important methodology to style our component in this chapter.

- CSS Stylesheet
- Inline Styling
- CSS Modules

CSS stylesheet

CSS stylesheet is usual, common and time-tested methodology. Simply create a CSS stylesheet for a component and enter all your styles for that particular componnt. Then, in the component, use *className* to refer the styles.

Let us style our *ExpenseEntryItem* component.

Open *expense-manager* application in your favorite editor.

Next, open *ExpenseEntryItem.css* file and add few styles.

```
div.itemStyle {
    color: brown;
    font-size: 14px;
}
```

Next, open *ExpenseEntryItem.js* and add *className* to the main container.

```
import React from 'react';
import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {
    render() {
```

```
        return (
            <div className="itemStyle">
                <div><b>Item:</b> <em>Mango Juice</em></div>
                <div><b>Amount:</b> <em>30.00</em></div>
                <div><b>Spend Date:</b> <em>2020-10-10</em></div>
                <div><b>Category:</b> <em>Food</em></div>
            </div>
        );
    }
}

export default ExpenseEntryItem;
```

Next, serve the application using npm command.

```
npm start
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter.

Item: *Mango Juice*
Amount: *30.00*
Spend Date: *2020-10-10*
Category: *Food*

CSS stylesheet is easy to understand and use. But, when the project size increases, CSS styles will also increase and ultimately create lot of conflict in the class name. Moreover, loading the CSS file directly is only supported in Webpack bundler and it may not supported in other tools.

Inline Styling

Inline Styling is one of the safest ways to style the React component. It declares all the styles as *JavaScript objects* using DOM based css properties and set it to the component through *style* attributes.

Let us add inline styling in our component.

Open *expense-manager* application in your favorite editor and modify *ExpenseEntryItem.js* file in the *src* folder. Declare a variable of type object and set the styles.

```
itemStyle = {
    color: 'brown',
    fontSize: '14px'
}
```

Here, *fontSize* represent the css property, *font-size*. All css properties can be used by representing it in *camelCase* format.

Next, set *itemStyle* style in the component using curly braces ***{}***:

```
render() {
    return (
        <div style={ this.itemStyle }>
            <div><b>Item:</b> <em>Mango Juice</em></div>
            <div><b>Amount:</b> <em>30.00</em></div>
            <div><b>Spend Date:</b> <em>2020-10-10</em></div>
            <div><b>Category:</b> <em>Food</em></div>
        </div>
    );
}
```

Also, style can be directly set inside the component:

```
render() {
    return (
        <div style={
            {
                color: 'brown',
                fontSize: '14px'
            }
        }>
            <div><b>Item:</b> <em>Mango Juice</em></div>
            <div><b>Amount:</b> <em>30.00</em></div>
            <div><b>Spend Date:</b> <em>2020-10-10</em></div>
            <div><b>Category:</b> <em>Food</em></div>
        </div>
    );
}
```

Now, we have successfully used the inline styling in our application.

Next, serve the application using npm command.

```
npm start
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter.

Item: *Mango Juice*
Amount: *30.00*
Spend Date: *2020-10-10*
Category: *Food*

CSS Modules

Css Modules provides safest as well as easiest way to define the style. It uses normal css stylesheet with normal syntax. While importing the styles, CSS modules converts all the styles into locally scoped styles so that the name conflicts will not happen. Let us change our component to use *CSS modules*

Open *expense-manager* application in your favorite editor.

Next, create a new stylesheet, *ExpenseEntryItem.module.css* file under *src/components* folder and write regular css styles.

```
div.itemStyle {
    color: 'brown';
    font-size: 14px;
}
```

Here, file naming convention is very important. React toolchain will pre-process the css files ending with *.module.css* through *CSS Module*. Otherwise, it will be considered as a normal stylesheet.

Next, open *ExpenseEntryItem.js* file in the *src/component* folder and import the styles.

```
import styles from './ExpenseEntryItem.module.css'
```

Next, use the styles as JavaScript expression in the component.

```
<div className={styles.itemStyle}>
```

Now, we have successfully used the CSS modules in our application.

The final and complete code is:

```
import React from 'react';
import './ExpenseEntryItem.css';
import styles from './ExpenseEntryItem.module.css'

class ExpenseEntryItem extends React.Component {

    render() {
        return (
            <div className={styles.itemStyle} >
                <div><b>Item:</b> <em>Mango Juice</em></div>
                <div><b>Amount:</b> <em>30.00</em></div>
                <div><b>Spend Date:</b> <em>2020-10-10</em></div>
                <div><b>Category:</b> <em>Food</em></div>
            </div>
        );
    }
}

export default ExpenseEntryItem;
```

Next, serve the application using npm command.

```
npm start
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter.

Item: *Mango Juice*
Amount: *30.00*
Spend Date: *2020-10-10*
Category: *Food*

8. React — Properties (*props*)

React enables developers to create dynamic and advanced component using properties. Every component can have attributes similar to HTML attributes and each attribute's value can be accessed inside the component using properties (props).

For example, *Hello* component with a *name* attribute can be accessed inside the component through *this.props.name* variable.

```
<Hello name="React" />

// value of name will be "Hello*
const name = this.props.name
```

React properties supports attribute's value of different types. They are as follows,

- String
- Number
- Datetime
- Array
- List
- Objects

Let us learn one by one in this chapter.

Create a component using properties

Let us modify our *ExpenseEntryItem* component and try to use properties.

Open our *expense-manager* application in your favorite editor.

Open *ExpenseEntryItem* file in the *src/components* folder.

Introduce construction function with argument props.

```
constructor(props) {
    super(props);
}
```

Next, change the *render* method and populate the value from props.

```
render() {
    return (
        <div>
            <div><b>Item:</b> <em>{this.props.name}</em></div>
            <div><b>Amount:</b> <em>{this.props.amount}</em></div>
            <div><b>Spend date:</b>
                <em>{this.props.spenddate.tostring()}</em></div>
            <div><b>Category:</b> <em>{this.props.category}</em></div>
```

```
        </div>
    );
}
```

Here,

- *name* represents the item's name of type *String*
- *amount* represents the item's amount of type *number*
- *spendDate* represents the item's Spend Date of type *date*
- *category* represents the item's category of type *String*

Now, we have successfully updated the component using properties.

```
import React from 'react'

import './ExpenseEntryItem.css';
import styles from './ExpenseEntryItem.module.css'

class ExpenseEntryItem extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return (
            <div>
                <div><b>Item:</b> <em>{this.props.name}</em></div>
                <div><b>Amount:</b> <em>{this.props.amount}</em></div>
                <div><b>Spend Date:</b>
                    <em>{this.props.spendDate.toString()}</em></div>
                <div><b>Category:</b> <em>{this.props.category}</em></div>
            </div>
        );
    }
}

export default ExpenseEntryItem;
```

Now, we can use the component by passing all the properties through attributes in the *index.js*.

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

const name = "Grape Juice"
const amount = 30.00
const spendDate = new Date("2020-10-10")
const category = "Food"

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem
```