ReactJS - Quick Guide

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*.

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>
```

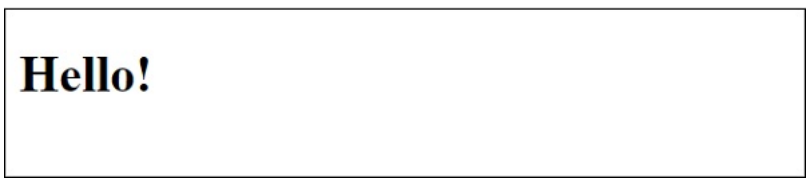
Next, run the *serve* command.

```
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.

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
```

Example

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
```

Output

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 −

Example

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

Output

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>
```

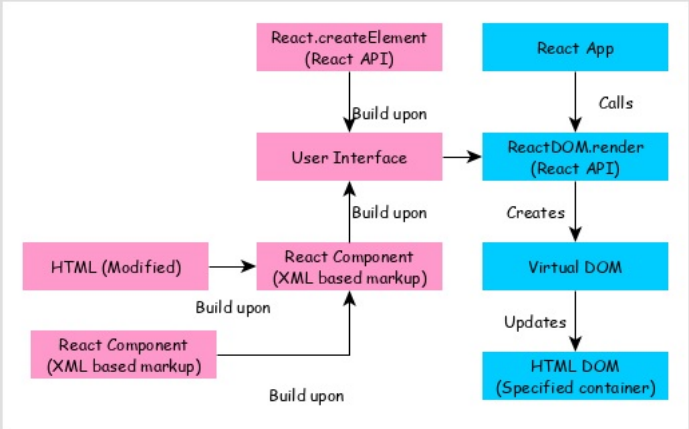
Next, run the application and open the browser. The output of the application is as follows −

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 −

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



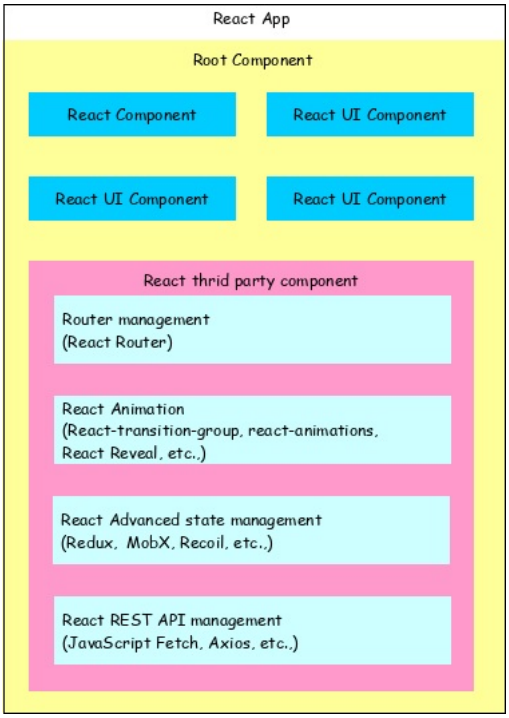
React app calls ***ReactDOM.render*** method by passing the user interface created using React component (coded in either JSX or React element format) and the container to render the user interface.

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.

ReactJS - 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 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
```

ReactJS - 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.

Example

```
<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>
```

Output

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

```
The Current time is 21:19:56 GMT+0530(India Standard Time)
```

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.

Example

```
<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>
```

Output

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

```
The Current time is 21:19:56 GMT+0530(India Standard Time)
```

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*

Example

```
<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>
```

Output

The output is as follows −

```
The Current time is 22:36:55 GMT+0530(India Standard Time)
```

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>
```

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')
);
```

Example

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
```

Output

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.

ReactJS - 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 component. 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.

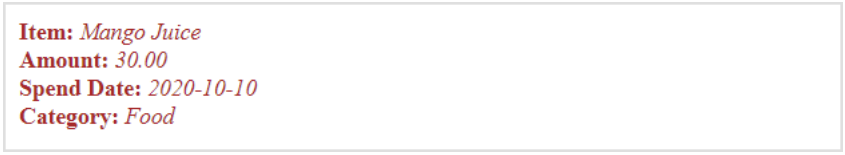
```
import React from 'react';
import './ExpenseEntryItem.css';

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.



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 −

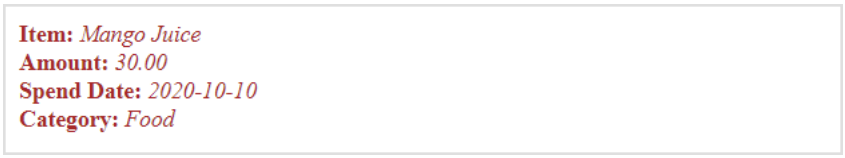
```
render() {
   return (
      <div style={
         {
            color: 'brown',
            fontSize: '14px'
         }
      }>
         <div><b>Item:</b> <em>Mango Juice</em></div>
         <div><b>Amount:</b> <em>30.00</em></div>
         <div><b>Spend Date:</b> <em>2020-10-10</em></div>
   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



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.

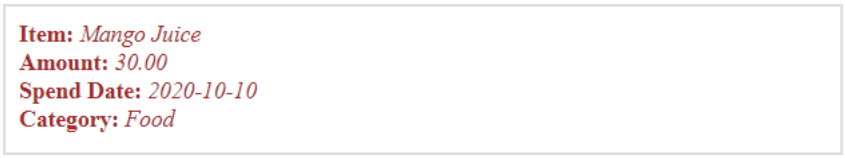
The final and complete code is −

```
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.



ReactJS - 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
- Nested Components
- Use Component
- Component Collection

ReactJS - Event management

Event management is one of the important features in a web application. It enables the user to interact with the application. React support all events available in a web application. React event handling is very similar to DOM events with little changes. Let us learn how to handle events in a React application in this chapter.

Let us see the step-by-step process of handling an event in a React component.

- Define an event handler method to handle the given event.

```
log() {
   cosole.log("Event is fired");
}
```

React provides an alternative syntax using lambda function to define event handler. The lambda syntax is −

```
log = () =;> {
   cosole.log("Event is fired");
}
```

If you want to know the target of the event, then add an argument **e** in the handler method. React will send the event target details to the handler method.

```
log(e) {
   cosole.log("Event is fired");
   console.log(e.target);
}
```

The alternative lambda syntax is −

```
log = (e) => {
   cosole.log("Event is fired");
   console.log(e.target);
}
```

If you want to send extra details during an event, then add the extra details as initial argument and then add argument **(e)** for event target.

```
log(extra, e) {
   cosole.log("Event is fired");
   console.log(e.target);
   console.log(extra);
   console.log(this);
}
```

The alternative lambda syntax is as follows −

```
log = (extra, e) => {
   cosole.log("Event is fired");
   console.log(e.target);
   console.log(extra);
   console.log(this);
}
```

Bind the event handler method in the constructor of the component. This will ensure the availability of *this* in the event handler method.

```
constructor(props) {
   super(props);
   this.logContent = this.logContent.bind(this);
}
```

If the event handler is defined in alternate lambda syntax, then the binding is not needed. *this* keyword will be automatically bound to the event handler method.

Set the event handler method for the specific event as specified below −

```
<div onClick={this.log}> ... </div>
```

To set extra arguments, bind the event handler method and then pass the extra information as second argument.

```
<div onClick={this.log.bind(this, extra)}> ... </div>
```

The alternate lambda syntax is as follows −

```
<div onClick={this.log(extra, e)}> ... </div>
```

Here,

- Create a event-aware component
- Introduce events in Expense manager app

ReactJS - State Management

State management is one of the important and unavoidable features of any dynamic application. React provides a simple and flexible API to support state management in a React component. Let us understand how to maintain state in React application in this chapter.

What is state?

State represents the value of a dynamic properties of a React component at a given instance. React provides a dynamic data store for each component. The internal data represents the state of a React component and can be accessed using this.state member variable of the component. Whenever the state of the component is changed, the component will re-render itself by calling the *render()* method along with the new state.

A simple example to better understand the state management is to analyse a real-time clock component. The clock component primary job is to show the date and time of a location at the given instance. As the current time will change every second, the clock component should maintain the current date and time in it's state. As the state of the clock component changes every second, the clock's *render()* method will be called every second and the *render()* method show the current time using it's current state.

The simple representation of the state is as follows −

```
{
   date: '2020-10-10 10:10:10'
}
```

Let us create a new *Clock* component later in this chapter.

Here,

- State management API
- Stateless component
- State management using React Hooks
- Component Life cycle
- Component life cycle using React Hooks
- Layout in component
- Pagination

- Material UI

ReactJS - Http Client Programming

Http client programming enables the application to connect and fetch data from http server through JavaScript. It reduces the data transfer between client and server as it fetches only the required data instead of the whole design and subsequently improves the network speed. It improves the user experience and becomes an indispensable feature of every modern web application.

Nowadays, lot of server side application exposes its functionality through REST API (functionality over HTTP protocol) and allows any client application to consume the functionality.

React does not provide it's own http programming api but it supports browser's built-in *fetch()* api as well as third party client library like axios to do client side programming. Let us learn how to do http programming in React application in this chapter. Developer should have a basic knowledge in Http programming to understand this chapter.

Expense Rest Api Server

The prerequisite to do Http programming is the basic knowledge of Http protocol and REST API technique. Http programming involves two part, server and client. React provides support to create client side application. Express a popular web framework provides support to create server side application.

Let us first create a Expense Rest Api server using express framework and then access it from our *ExpenseManager* application using browser's built-in fetch api.

Open a command prompt and create a new folder, *express-rest-api*.

```
cd /go/to/workspace
mkdir apiserver
cd apiserver
```

Initialize a new node application using the below command −

```
npm init
```

The *npm init* will prompt and ask us to enter basic project details. Let us enter *apiserver* for project name and *server.js* for entry point. Leave other configuration with default option.

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (apiserver)
version: (1.0.0)
description: Rest api for Expense Application
entry point: (index.js) server.js
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to \path\to\workspace\expense-rest-api\package.json:
{
   "name": "expense-rest-api",
   "version": "1.0.0",
   "description": "Rest api for Expense Application",
   "main": "server.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
   },
   "author": "",
   "license": "ISC"
}
Is this OK? (yes) yes
```

Next, install *express, nedb & cors* modules using below command −

```
npm install express nedb cors
```

- *express* is used to create server side application.
- *nedb* is a datastore used to store the expense data.
- *cors* is a middleware for *express* framework to configure the client access details.

Next, let us create a file, data.csv and populate it with initial expense data for testing purposes. The structure of the file is that it contains one expense entry per line.

```
Pizza,80,2020-10-10,Food
Grape Juice,30,2020-10-12,Food
Cinema,210,2020-10-16,Entertainment
Java Programming book,242,2020-10-15,Academic
Mango Juice,35,2020-10-16,Food
Dress,2000,2020-10-25,Cloth
Tour,2555,2020-10-29,Entertainment
Meals,300,2020-10-30,Food
Mobile,3500,2020-11-02,Gadgets
Exam Fees,1245,2020-11-04,Academic
```

Next, create a file *expensedb.js* and include code to load the initial expense data into the data store. The code checks the data store for initial data and load only if the data is not available in the store.

```
var store = require("nedb")
var fs = require('fs');
var expenses = new store({ filename: "expense.db", autoload: true })
expenses.find({}, function (err, docs) {
   if (docs.length == 0) {
      loadExpenses();
   }
})
function loadExpenses() {
   readCsv("data.csv", function (data) {
      console.log(data);

      data.forEach(function (rec, idx) {
         item = {}
         item.name = rec[0];
         item.amount = parseFloat(rec[1]);
         item.spend_date = new Date(rec[2]);
         item.category = rec[3];

         expenses.insert(item, function (err, doc) {
            console.log('Inserted', doc.item_name, 'with ID', doc._id);
         })
      })
   })
}
function readCsv(file, callback) {
   fs.readFile(file, 'utf-8', function (err, data) {
      if (err) throw err;
      var lines = data.split('\r\n');
      var result = lines.map(function (line) {
         return line.split(',');
      });
      callback(result);
   });
}
module.exports = expenses
```

Next, create a file, *server.js* and include the actual code to list, add, update and delete the expense entries.

```
var express = require("express")
var cors = require('cors')
var expenseStore = require("./expensedb.js")
var app = express()
app.use(cors());
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
var HTTP_PORT = 8000
app.listen(HTTP_PORT, () => {
   console.log("Server running on port %PORT%".replace("%PORT%", HTTP_PORT))
});
app.get("/", (req, res, next) => {
   res.json({ "message": "Ok" })
```

```
});
app.get("/api/expenses", (req, res, next) => {
   expenseStore.find({}, function (err, docs) {
      res.json(docs);
   });
});
app.get("/api/expense/:id", (req, res, next) => {
   var id = req.params.id;
   expenseStore.find({ _id: id }, function (err, docs) {
      res.json(docs);
   })
});
app.post("/api/expense/", (req, res, next) => {
   var errors = []

   if (!req.body.item) {
      errors.push("No item specified");
   }
   var data = {
      name: req.body.name,
      amount: req.body.amount,
      category: req.body.category,
      spend_date: req.body.spend_date,
   }
   expenseStore.insert(data, function (err, docs) {
      return res.json(docs);
   });
})
app.put("/api/expense/:id", (req, res, next) => {
   var id = req.params.id;
   var errors = []
   if (!req.body.item) {
      errors.push("No item specified");
   }
   var data = {
      _id: id,
      name: req.body.name,
      amount: req.body.amount,
      category: req.body.category,
      spend_date: req.body.spend_date,
   }
   expenseStore.update( { _id: id }, data, function (err, docs) {
      return res.json(data);
   });
})
app.delete("/api/expense/:id", (req, res, next) => {
   var id = req.params.id;
   expenseStore.remove({ _id: id }, function (err, numDeleted) {
      res.json({ "message": "deleted" })
   });
})
app.use(function (req, res) {
   res.status(404);
});
```

Now, it is time to run the application.

```
npm run start
```

Next, open a browser and enter *http:///localhost:8000/* in the address bar.

```
{
   "message": "Ok"
}
```

It confirms that our application is working fine.

Finally, change the url to *http:///localhost:8000/api/expense* and press enter. The browser will show the initial expense entries in JSON format.

```
[
   ...
   {
      "name": "Pizza",
      "amount": 80,
```

```
      "spend_date": "2020-10-10T00:00:00.000Z",
      "category": "Food",
      "_id": "5H8rK8lLGJPVZ3gD"
   },
   ...
]
```

Let us use our newly created expense server in our Expense manager application through *fetch()* api in the upcoming section.

The fetch() api

Let us create a new application to showcase client side programming in React.

First, create a new react application, *react-http-app* using *Create React App* or *Rollup* bundler by following instruction in *Creating a React application* chapter.

Next, open the application in your favorite editor.

Next, create *src* folder under the root directory of the application.

Next, create *components* folder under src folder.

Next, create a file, *ExpenseEntryItemList.css* under *src/components* folder and include generic table styles.

```
html {
   font-family: sans-serif;
}
table {
   border-collapse: collapse;
   border: 2px solid rgb(200,200,200);
   letter-spacing: 1px;
   font-size: 0.8rem;
}
td, th {
   border: 1px solid rgb(190,190,190);
   padding: 10px 20px;
}
th {
   background-color: rgb(235,235,235);
}
td, th {
   text-align: left;
}
tr:nth-child(even) td {
   background-color: rgb(250,250,250);
}
tr:nth-child(odd) td {
   background-color: rgb(245,245,245);
}
caption {
   padding: 10px;
}
tr.highlight td {
    background-color: #a6a8bd;
}
```

Next, create a file, *ExpenseEntryItemList.js* under *src/components* folder and start editing.

Next, import *React* library.

```
import React from 'react';
```

Next, create a class, ExpenseEntryItemList and call constructor with props.

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

Next, initialize the state with empty list in the constructor.

```
this.state = {
   isLoaded: false,
   items: []
}
```

Next, create a method, *setItems* to format the items received from remote server and then set it into the state of the component.

```
setItems(remoteItems) {
   var items = [];
   remoteItems.forEach((item) => {
      let newItem = {
         id: item._id,
         name: item.name,
         amount: item.amount,
         spendDate: item.spend_date,
         category: item.category
      }
      items.push(newItem)
   });
   this.setState({
      isLoaded: true,
      items: items
   });
}
```

Next, add a method, *fetchRemoteItems* to fetch the items from the server.

```
fetchRemoteItems() {
   fetch("http://localhost:8000/api/expenses")
      .then(res => res.json())
      .then(
         (result) => {
            this.setItems(result);
         },
         (error) => {
            this.setState({
               isLoaded: false,
               error
            });
         }
      )
}
```

Here,

- *fetch* api is used to fetch the item from the remote server.
- *setItems* is used to format and store the items in the state.

Next, add a method, *deleteRemoteItem* to delete the item from the remote server.

```
deleteRemoteItem(id) {
   fetch('http://localhost:8000/api/expense/' + id, { method: 'DELETE' })
      .then(res => res.json())
      .then(
         () => {
            this.fetchRemoteItems()
         }
      )
}
```

Here,

- *fetch* api is used to delete and fetch the item from the remote server.
- *setItems* is again used to format and store the items in the state.

Next, call the *componentDidMount* life cycle api to load the items into the component during its mounting phase.

```
componentDidMount() {
   this.fetchRemoteItems();
}
```

Next, write an event handler to remove the item from the list.

```
handleDelete = (id, e) => {
   e.preventDefault();
   console.log(id);

   this.deleteRemoteItem(id);
}
```

Next, write the render method.

```
render() {
   let lists = [];
   if (this.state.isLoaded) {
      lists = this.state.items.map((item) =>
         <tr key={item.id} onMouseEnter={this.handleMouseEnter} onMouseLeave={this.handleMouseLeave}>
            <td>{item.name}</td>
            <td>{item.amount}</td>
            <td>{new Date(item.spendDate).toDateString()}</td>
            <td>{item.category}</td>
            <td><a href="#" onClick={(e) => this.handleDelete(item.id, e)}>Remove</a></td>
         </tr>
      );
   }
   return (
      <div>
         <table onMouseOver={this.handleMouseOver}>
            <thead>
               <tr>
                  <th>Item</th>
                  <th>Amount</th>
                  <th>Date</th>
                  <th>Category</th>
                  <th>Remove</th>
               </tr>
            </thead>
            <tbody>
               {lists}
            </tbody>
         </table>
      </div>
   );
}
```

Finally, export the component.

```
export default ExpenseEntryItemList;
```

Next, create a file, *index.js* under the *src* folder and use *ExpenseEntryItemList* component.

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

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

Finally, create a *public* folder under the root folder and create *index.html* file.

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

Next, open a new terminal window and start our server application.

```
cd /go/to/server/application
npm start
```

Next, serve the client application using npm command.

```
npm start
```

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

Item	Amount	Date	Category	Remove
Java Programming book	242	Thu Oct 15 2020	Academic	Remove
Exam Fees	1245	Wed Nov 04 2020	Academic	Remove
Dress	2000	Sun Oct 25 2020	Cloth	Remove
Cinema	210	Fri Oct 16 2020	Entertainment	Remove
Mobile	3500	Mon Nov 02 2020	Gadgets	Remove
Tour	2555	Thu Oct 29 2020	Entertainment	Remove
Mango Juice	35	Fri Oct 16 2020	Food	Remove
Grape Juice	30	Mon Oct 12 2020	Food	Remove
Pizza	80	Sat Oct 10 2020	Food	Remove
Meals	300	Fri Oct 30 2020	Food	Remove

Try to remove the item by clicking the remove link.

Item	Amount	Date	Category	Remove
Exam Fees	1245	Wed Nov 04 2020	Academic	Remove
Mango Juice	35	Fri Oct 16 2020	Food	Remove
Tour	2555	Thu Oct 29 2020	Entertainment	Remove
Mobile	3500	Mon Nov 02 2020	Gadgets	Remove
Java Programming book	242	Thu Oct 15 2020	Academic	Remove
Dress	2000	Sun Oct 25 2020	Cloth	Remove
Grape Juice	30	Mon Oct 12 2020	Food	Remove
Pizza	80	Sat Oct 10 2020	Food	Remove
Meals	300	Fri Oct 30 2020	Food	Remove
Cinema	210	Fri Oct 16 2020	Entertainment	Remove

ReactJS - Form Programming

The nature of form programming needs the state to be maintained. Because, the input field information will get changed as the user interacts with the form. But as we learned earlier, React library does not store or maintain any state information by itself and component has to use state management api to manage state. Considering this, React provides two types of components to support form programming.

- **Controlled component** − In controlled component, React provides a special attribute, value for all input elements and controls the input elements. The value attribute can be used to get and set the value of the input element. It has to be in sync with state of the component.
- **Uncontrolled component** − In uncontrolled component, React provides minimal support for form programming. It has to use Ref concept (another react concept to get a DOM element in the React component during runtime) to do the form programming.

Let us learn the form programming using controlled as well as uncontrolled component in this chapter.

- Controlled component
- Uncontrolled Component
- Formik

ReactJS - Routing

In web application, Routing is a process of binding a web URL to a specific resource in the web application. In React, it is binding an URL to a component. React does not support routing natively as it is basically an user interface library. React community provides many third party component to handle routing in the React application. Let us learn React Router, a top choice routing library for React application.

Install React Router

Let us learn how to install *React Router* component in our Expense Manager application.

Open a command prompt and go to the root folder of our application.

```
cd /go/to/expense/manager
```

Install the react router using below command.

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

Concept

React router provides four components to manage navigation in React application.

Router − Router is th top level component. It encloses the entire application.

Link − Similar to anchor tag in html. It sets the target url along with reference text.

```
<Link to="/">Home</Link>
```

Here, **to** attribute is used to set the target url.

Switch & Route − Both are used together. Maps the target url to the component. **Switch** is the parent component and **Route** is the child component. **Switch** component can have multiple **Route** component and each **Route** component mapping a particular url to a component.

```
<Switch>
   <Route exact path="/">
      <Home />
   </Route>
   <Route path="/home">
      <Home />
   </Route>
   <Route path="/list">
      <ExpenseEntryItemList />
   </Route>
</Switch>
```

Here, **path** attribute is used to match the url. Basically, **Switch** works similar to traditional switch statement in a programming language. It matches the target url with each child route (**path** attribute) one by one in sequence and invoke the first matched route.

Along with router component, React router provides option to get set and get dynamic information from the url. For example, in an article website, the url may have article type attached to it and the article type needs to be dynamically extracted and has to be used to fetch the specific type of articles.

```
<Link to="/article/c">C Programming</Link>
<Link to="/article/java">Java Programming</Link>

...
...

<Switch>
  <Route path="article/:tag" children={<ArticleList />} />
</Switch>
```

Then, in the child component (class component),

```
import { withRouter } from "react-router"

class ArticleList extends React.Component {
   ...
   ...
   static getDerivedStateFromProps(props, state) {
      let newState = {
         tag: props.match.params.tag
      }
      return newState;
   }
   ...
   ...
}
export default WithRouter(ArticleList)
```

Here, **WithRouter** enables **ArticleList** component to access the tag information through **props**.

The same can be done differently in functional components −

```
function ArticleList() {
   let { tag } = useParams();
   return (
      <div>
         <h3>ID: {id}</h3>
      </div>
```

```
   );
}
```

Here, **useParams** is a custom React Hooks provided by React Router component.

Nested routing

React router supports nested routing as well. React router provides another React Hooks, **useRouteMatch()** to extract parent route information in nested routes.

```
function ArticleList() {
   // get the parent url and the matched path
   let { path, url } = useRouteMatch();

   return (
      <div>
         <h2>Articles</h2>
         <ul>
            <li>
               <Link to={`${url}/pointer`}>C with pointer</Link>
            </li>
            <li>
               <Link to={`${url}/basics`}>C basics</Link>
            </li>
         </ul>

         <Switch>
            <Route exact path={path}>
               <h3>Please select an article.</h3>
            </Route>
            <Route path={`${path}/:article`}>
               <Article />
            </Route>
         </Switch>
      </div>
   );
}
function Article() {
   let { article } = useParams();
   return (
      <div>
         <h3>The select article is {article}</h3>
      </div>
   );
}
```

Here, **useRouteMatch** returns the matched path and the target **url**. url can be used to create next level of links and **path** can be used to map next level of components / screens.

Creating navigation

Let us learn how to do routing by creating the possible routing in our expense manager application. The minimum screens of the application are given below −

- **Home screen** − Landing or initial screen of the application
- **Expense list screen** − Shows the expense items in a tabular format
- **Expense add screen** − Add interface to add an expense item

First, create a new react application, *react-router-app* using *Create React App* or *Rollup* bundler by following instruction in *Creating a React application* chapter.

Next, open the application in your favorite editor.

Next, create *src* folder under the root directory of the application.

Next, create *components* folder under src folder.

Next, create a file, *Home.js* under *src/components* folder and start editing.

Next, import *React library*.

```
import React from 'react';
```

Next, import **Link** from React router library.

```
import { Link } from 'react-router-dom'
```

Next, create a class, Home and call constructor with **props**.

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

Next, add *render()* method and show the welcome message and links to add and list expense screen.

```
render() {
   return (
      <div>
         <p>Welcome to the React tutorial</p>
         <p><Link to="/list">Click here</Link> to view expense list</p>
         <p><Link to="/add">Click here</Link> to add new expenses</p>
      </div>
   )
}
```

Finally, export the component.

```
export default Home;
```

The complete source code of the *Home* component is given below −

```
import React from 'react';
import { Link } from 'react-router-dom'

class Home extends React.Component {
   constructor(props) {
      super(props);
   }
   render() {
      return (
         <div>
            <p>Welcome to the React tutorial</p>
            <p><Link to="/list">Click here</Link> to view expense list</p>
            <p><Link to="/add">Click here</Link> to add new expenses</p>
         </div>
      )
   }
}
export default Home;
```

Next, create *ExpenseEntryItemList.js* file under *src/components* folder and create *ExpenseEntryItemList* component.

```
import React from 'react';
import { Link } from 'react-router-dom'

class ExpenseEntryItemList extends React.Component {
   constructor(props) {
      super(props);
   }
   render() {
      return (
         <div>
            <h1>Expenses</h1>
            <p><Link to="/add">Click here</Link> to add new expenses</p>
            <div>
               Expense list
            </div>
         </div>
      )
   }
}
export default ExpenseEntryItemList;
```

Next, create *ExpenseEntryItemForm.js* file under *src/components* folder and create *ExpenseEntryItemForm* component.

```
import React from 'react';
import { Link } from 'react-router-dom'
```

```
import { Link } from 'react-router-dom'

class ExpenseEntryItemForm extends React.Component {
   constructor(props) {
      super(props);
   }
   render() {
      return (
         <div>
            <h1>Add Expense item</h1>
            <p><Link to="/list">Click here</Link> to view new expense list</p>
            <div>
               Expense form
            </div>
         </div>
      )
   }
}
export default ExpenseEntryItemForm;
```

Next, create a file, *App.css* under *src/components* folder and add generic css styles.

```
html {
   font-family: sans-serif;
}
a{
   text-decoration: none;
}
p, li, a{
   font-size: 14px;
}
nav ul {
   width: 100%;
   list-style-type: none;
   margin: 0;
   padding: 0;
   overflow: hidden;
   background-color: rgb(235,235,235);
}
nav li {
   float: left;
}
nav li a {
   display: block;
   color: black;
   text-align: center;
   padding: 14px 16px;
   text-decoration: none;
   font-size: 16px;
}
nav li a:hover {
   background-color: rgb(187, 202, 211);
}
```

Next, create a file, *App.js* under *src/components* folder and start editing. The purpose of the *App* component is to handle all the screen in one component. It will configure routing and enable navigation to all other components.

Next, import React library and other components.

```
import React from 'react';

import Home from './Home'
import ExpenseEntryItemList from './ExpenseEntryItemList'
import ExpenseEntryItemForm from './ExpenseEntryItemForm'

import './App.css'
```

Next, import React router components.

```
import {
   BrowserRouter as Router,
   Link,
   Switch,
   Route
} from 'react-router-dom'
```

Next, write the *render()* method and configure routing.

```
function App() {
   return (
      <Router>
         <div>
            <nav>
               <ul>
                  <li>
                     <Link to="/">Home</Link>
                  </li>
                  <li>
                     <Link to="/list">List Expenses</Link>
                  </li>
                  <li>
                     <Link to="/add">Add Expense</Link>
                  </li>
               </ul>
            </nav>

            <Switch>
               <Route path="/list">
                  <ExpenseEntryItemList />
               </Route>
               <Route path="/add">
                  <ExpenseEntryItemForm />
               </Route>
               <Route path="/">
                  <Home />
               </Route>
            </Switch>
         </div>
      </Router>
   );
}
```

Next, create a file, *index.js* under the *src* folder and use *App* component.

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

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

Finally, create a *public* folder under the root folder and create *index.html* file.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <title>React router app</title>
   </head>
   <body>
      <div id="root"></div>
      <script type="text/JavaScript" src="./index.js"></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.

Try to navigate the links and confirm that the routing is working.

ReactJS - Redux

React redux is an advanced state management library for React. As we learned earlier, React only supports component level state management. In a big and complex application, large number of components are used. React recommends to move the state to the top level component and pass the state to the nested component using properties. It helps to some extent but it becomes complex when the components increases.

React redux chips in and helps to maintain state at the application level. React redux allows any component to access the state at any time. Also, it allows any component to change the state of the application at any time.

Let us learn about the how to write a React application using React redux in this chapter.

Concepts

React redux maintains the state of the application in a single place called Redux store. React component can get the latest state from the store as well as change the state at any time. Redux provides a simple process to get and set the current state of the application and involves below concepts.

Store − The central place to store the state of the application.

Actions − Action is an plain object with the type of the action to be done and the input (called payload) necessary to do the action. For example, action for adding an item in the store contains **ADD_ITEM** as type and an object with item's details as payload. The action can be represented as −

```
{
   type: 'ADD_ITEM',
   payload: { name: '..', ... }
}
```

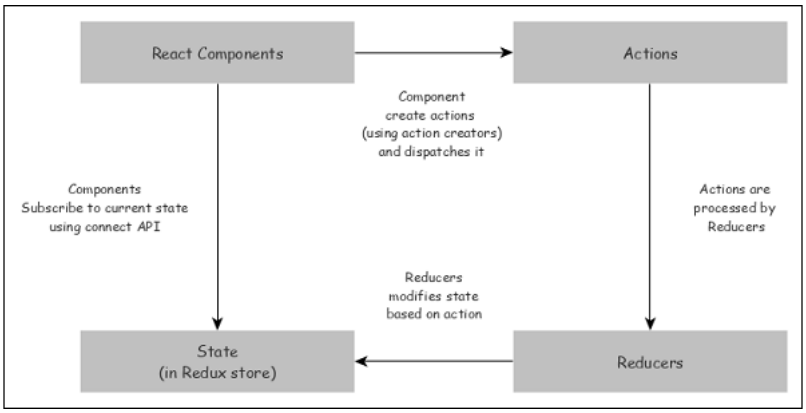
Reducers − Reducers are pure functions used to create a new state based on the existing state and the current action. It returns the newly created state. For example, in add item scenario, it creates a new item list and merges the item from the state and new item and returns the newly created list.

Action creators − *Action creator* creates an action with proper action type and data necessary for the action and returns the action. For example, ***addItem*** action creator returns below object −

```
{
   type: 'ADD_ITEM',
   payload: { name: '..', ... }
}
```

Component − Component can connect to the store to get the current state and dispatch action to the store so that the store executes the action and updates it's current state.

The workflow of a typical redux store can be represented as shown below.



- React component subscribes to the store and get the latest state during initialization of the application.
- To change the state, React component creates necessary action and dispatches the action.
- Reducer creates a new state based on the action and returns it. Store updates itself with the new state.
- Once the state changes, store sends the updated state to all its subscribed component.

Redux API

Redux provides a single api, *connect* which will connect a components to the store and allows the component to get and set the state of the store.

The signature of the connect API is −

```
function connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)
```

All parameters are optional and it returns a HOC (higher order component). A higher order component is a function which wraps a component and returns a new component.

```
let hoc = connect(mapStateToProps, mapDispatchToProps)
let connectedComponent = hoc(component)
```

Let us see the first two parameters which will be enough for most cases.

- **mapStateToProps** − Accepts a function with below signature.

```
(state, ownProps?) => Object
```

Here, **state** refers current state of the store and **Object** refers the new props of the component. It gets called whenever the state of the store is updated.

```
(state) => { prop1: this.state.anyvalue }
```

- ***mapDispatchToProps*** − Accepts a function with below signature.

```
Object | (dispatch, ownProps?) => Object
```

Here, **dispatch** refers the dispatch object used to dispatch action in the redux store and **Object** refers one or more dispatch functions as props of the component.

```
(dispatch) => {
   addDispatcher: (dispatch) => dispatch({ type: 'ADD_ITEM', payload: { } }),
   removeispatcher: (dispatch) => dispatch({ type: 'REMOVE_ITEM', payload: { } }),
}
```

Provider component

React Redux provides a Provider component and its sole purpose to make the Redux store available to its all nested components connected to store using connect API. The sample code is given below −

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { App } from './App'
import createStore from './createReduxStore'

const store = createStore()

ReactDOM.render(
   <Provider store={store}>
      <App />
   </Provider>,
   document.getElementById('root')
)
```

Now, all the component inside the App component can get access to the Redux store by using connect API.

Working example

Let us recreate our expense manager application and uses the React redux concept to maintain the state of the application.

First, create a new react application, *react-message-app* using Create *React App* or *Rollup* bundler by following instruction in Creating a React application chapter.

Next, install Redux and React redux library.

```
npm install redux react-redux --save
```

Next, install uuid library to generate unique identifier for new expenses.

```
npm install uuid --save
```

Next, open the application in your favorite editor.

Next, create *src* folder under the root directory of the application.

Next, create *actions* folder under *src* folder.

Next, create a file, *types.js* under *src/actions* folder and start editing.

Next, add two action type, one for add expense and one for remove expense.

```
export const ADD_EXPENSE = 'ADD_EXPENSE';
export const DELETE_EXPENSE = 'DELETE_EXPENSE';
```

Next, create a file, *index.js* under *src/actions* folder to add action and start editing.

Next, import **uuid** to create unique identifier.

```
import { v4 as uuidv4 } from 'uuid';
```

Next, import action types.

```
import { ADD_EXPENSE, DELETE_EXPENSE } from './types';
```

Next, add a new function to return action type for adding an expense and export it.

```
export const addExpense = ({ name, amount, spendDate, category }) => ({
   type: ADD_EXPENSE,
   payload: {
      id: uuidv4(),
      name,
      amount,
      spendDate,
      category
   }
});
```

Here, the function expects expense object and return action type of **ADD_EXPENSE** along with a payload of expense information.

Next, add a new function to return action type for deleting an expense and export it.

```
export const deleteExpense = id => ({
   type: DELETE_EXPENSE,
   payload: {
      id
   }
});
```

Here, the function expects id of the expense item to be deleted and return action type of 'DELETE_EXPENSE' along with a payload of expense id.

The complete source code of the action is given below −

```
import { v4 as uuidv4 } from 'uuid';
import { ADD_EXPENSE, DELETE_EXPENSE } from './types';

export const addExpense = ({ name, amount, spendDate, category }) => ({
   type: ADD_EXPENSE,
   payload: {
      id: uuidv4(),
      name,
      amount,
      spendDate,
      category
   }
});
export const deleteExpense = id => ({
   type: DELETE_EXPENSE,
   payload: {
      id
   }
});
```

Next, create a new folder, reducers under *src* folder.

Next, create a file, *index.js* under *src/reducers* to write reducer function and start editing.

Next, import the action types.

```
import { ADD_EXPENSE, DELETE_EXPENSE } from '../actions/types';
```

Next, add a function, *expensesReducer* to do the actual feature of adding and updating expenses in the redux store.

```
export default function expensesReducer(state = [], action) {
   switch (action.type) {
      case ADD_EXPENSE:
         return [...state, action.payload];
```

```
      case DELETE_EXPENSE:
         return state.filter(expense => expense.id !== action.payload.id);
      default:
         return state;
   }
}
```

The complete source code of the reducer is given below −

```
import { ADD_EXPENSE, DELETE_EXPENSE } from '../actions/types';

export default function expensesReducer(state = [], action) {
   switch (action.type) {
      case ADD_EXPENSE:
         return [...state, action.payload];
      case DELETE_EXPENSE:
         return state.filter(expense => expense.id !== action.payload.id);
      default:
         return state;
   }
}
```

Here, the reducer checks the action type and execute the relevant code.

Next, create *components* folder under *src* folder.

Next, create a file, *ExpenseEntryItemList.css* under *src/components* folder and add generic style for the html tables.

```
html {
   font-family: sans-serif;
}
table {
   border-collapse: collapse;
   border: 2px solid rgb(200,200,200);
   letter-spacing: 1px;
   font-size: 0.8rem;
}
td, th {
   border: 1px solid rgb(190,190,190);
   padding: 10px 20px;
}
th {
   background-color: rgb(235,235,235);
}
td, th {
   text-align: left;
}
tr:nth-child(even) td {
   background-color: rgb(250,250,250);
}
tr:nth-child(odd) td {
   background-color: rgb(245,245,245);
}
caption {
   padding: 10px;
}
tr.highlight td {
   background-color: #a6a8bd;
}
```

Next, create a file, *ExpenseEntryItemList.js* under *src/components* folder and start editing.

Next, import React and React redux library.

```
import React from 'react';
import { connect } from 'react-redux';
```

Next, import ExpenseEntryItemList.css file.

```
import './ExpenseEntryItemList.css';
```

Next, import action creators.

```
import { deleteExpense } from '../actions';
import { addExpense } from '../actions';
```

Next, create a class, ExpenseEntryItemList and call constructor with **props**.

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

Next, create **mapStateToProps** function.

```
const mapStateToProps = state => {
   return {
      expenses: state
   };
};
```

Here, we copied the input state to **expenses** props of the component.

Next, create **mapDispatchToProps** function.

```
const mapDispatchToProps = dispatch => {
   return {
      onAddExpense: expense => {
         dispatch(addExpense(expense));
      },
      onDelete: id => {
         dispatch(deleteExpense(id));
      }
   };
};
```

Here, we created two function, one to dispatch add expense **(addExpense)** function and another to dispatch delete expense **(deleteExpense)** function and mapped those function to props of the component.

Next, export the component using **connect** api.

```
export default connect(
   mapStateToProps,
   mapDispatchToProps
)(ExpenseEntryItemList);
```

Now, the component gets three new properties given below −

- expenses − list of expense
- onAddExpense − function to dispatch **addExpense** function
- onDelete − function to dispatch **deleteExpense** function

Next, add few expense into the redux store in the constructor using **onAddExpense** property.

```
if (this.props.expenses.length == 0)
{
   const items = [
      { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food" },
      { id: 2, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category: "Food" },
      { id: 3, name: "Cinema", amount: 210, spendDate: "2020-10-16", category: "Entertainment" },
      { id: 4, name: "Java Programming book", amount: 242, spendDate: "2020-10-15", category: "Academic" },
      { id: 5, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category: "Food" },
      { id: 6, name: "Dress", amount: 2000, spendDate: "2020-10-25", category: "Cloth" },
      { id: 7, name: "Tour", amount: 2555, spendDate: "2020-10-29", category: "Entertainment" },
      { id: 8, name: "Meals", amount: 300, spendDate: "2020-10-30", category: "Food" },
      { id: 9, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category: "Gadgets" },
      { id: 10, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category: "Academic" }
   ]
   items.forEach((item) => {
      this.props.onAddExpense(
         {
            name: item.name,
            amount: item.amount,
            spendDate: item.spendDate,
            category: item.category
         }
      );
```

```
   })
}
```

Next, add an event handler to delete the expense item using expense id.

```
handleDelete = (id,e) => {
   e.preventDefault();
   this.props.onDelete(id);
}
```

Here, the event handler calls the **onDelete** dispatcher, which call **deleteExpense** along with the expense id.

Next, add a method to calculate the total amount of all expenses.

```
getTotal() {
   let total = 0;
   for (var i = 0; i < this.props.expenses.length; i++) {
      total += this.props.expenses[i].amount
   }
   return total;
}
```

Next, add *render()* method and list the expense item in the tabular format.

```
render() {
   const lists = this.props.expenses.map(
      (item) =>
      <tr key={item.id}>
         <td>{item.name}</td>
         <td>{item.amount}</td>
         <td>{new Date(item.spendDate).toDateString()}</td>
         <td>{item.category}</td>
         <td><a href="#"
            onClick={(e) => this.handleDelete(item.id, e)}>Remove</a></td>
      </tr>
   );
   return (
      <div>
         <table>
            <thead>
               <tr>
                  <th>Item</th>
                  <th>Amount</th>
                  <th>Date</th>
                  <th>Category</th>
                  <th>Remove</th>
               </tr>
            </thead>
            <tbody>
               {lists}
               <tr>
                  <td colSpan="1" style={{ textAlign: "right" }}>Total Amount</td>
                  <td colSpan="4" style={{ textAlign: "left" }}>
                     {this.getTotal()}
                  </td>
               </tr>
            </tbody>
         </table>
      </div>
   );
}
```

Here, we set the event handler *handleDelete* to remove the expense from the store.

The complete source code of the *ExpenseEntryItemList* component is given below −

```
import React from 'react';
import { connect } from 'react-redux';
import './ExpenseEntryItemList.css';
import { deleteExpense } from '../actions';
import { addExpense } from '../actions';

class ExpenseEntryItemList extends React.Component {
   constructor(props) {
      super(props);
```

```
      if (this.props.expenses.length == 0){
         const items = [
            { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food" },
            { id: 2, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category: "Food" },
            { id: 3, name: "Cinema", amount: 210, spendDate: "2020-10-16", category: "Entertainment" },
            { id: 4, name: "Java Programming book", amount: 242, spendDate: "2020-10-15", category: "Academic" },
            { id: 5, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category: "Food" },
            { id: 6, name: "Dress", amount: 2000, spendDate: "2020-10-25", category: "Cloth" },
            { id: 7, name: "Tour", amount: 2555, spendDate: "2020-10-29", category: "Entertainment" },
            { id: 8, name: "Meals", amount: 300, spendDate: "2020-10-30", category: "Food" },
            { id: 9, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category: "Gadgets" },
            { id: 10, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category: "Academic" }
         ]
         items.forEach((item) => {
            this.props.onAddExpense(
               {
                  name: item.name,
                  amount: item.amount,
                  spendDate: item.spendDate,
                  category: item.category
               }
            );
         })
      }
   }
   handleDelete = (id,e) => {
      e.preventDefault();
      this.props.onDelete(id);
   }
   getTotal() {
      let total = 0;
      for (var i = 0; i < this.props.expenses.length; i++) {
         total += this.props.expenses[i].amount
      }
      return total;
   }
   render() {
      const lists = this.props.expenses.map((item) =>
         <tr key={item.id}>
            <td>{item.name}</td>
            <td>{item.amount}</td>
            <td>{new Date(item.spendDate).toDateString()}</td>
            <td>{item.category}</td>
            <td><a href="#"
               onClick={(e) => this.handleDelete(item.id, e)}>Remove</a></td>
         </tr>
      );
      return (
         <div>
            <table>
               <thead>
                  <tr>
                     <th>Item</th>
                     <th>Amount</th>
                     <th>Date</th>
                     <th>Category</th>
                     <th>Remove</th>
                  </tr>
               </thead>
               <tbody>
                  {lists}
                  <tr>
                     <td colSpan="1" style={{ textAlign: "right" }}>Total Amount</td>
                     <td colSpan="4" style={{ textAlign: "left" }}>
                        {this.getTotal()}
                     </td>
                  </tr>
               </tbody>
            </table>
         </div>
      );
   }
}
const mapStateToProps = state => {
```

```
   return {
      expenses: state
   };
};
const mapDispatchToProps = dispatch => {
   return {
      onAddExpense: expense => {
         dispatch(addExpense(expense));
      },
      onDelete: id => {
         dispatch(deleteExpense(id));
      }
   };
};
export default connect(
   mapStateToProps,
   mapDispatchToProps
)(ExpenseEntryItemList);
```

Next, create a file, *App.js* under the *src/components* folder and use *ExpenseEntryItemList* component.

```
import React, { Component } from 'react';
import ExpenseEntryItemList from './ExpenseEntryItemList';

class App extends Component {
   render() {
      return (
         <div>
            <ExpenseEntryItemList />
         </div>
      );
   }
}
export default App;
```

Next, create a file, *index.js* under src folder.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers';
import App from './components/App';

const store = createStore(rootReducer);

ReactDOM.render(
   <Provider store={store}>
      <App />
   </Provider>,
   document.getElementById('root')
);
```

Here,

- Create a store using **createStore** by attaching the our reducer.
- Used Provider component from React redux library and set the store as props, which enables all the nested component to ***connect*** to store using connect api.

Finally, create a *public* folder under the root folder and create *index.html* file.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <title>React Containment App</title>
   </head>
   <body>
      <div id="root"></div>
      <script type="text/JavaScript" src="./index.js"></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.

Clicking the remove link will remove the item from redux store.

Item	Amount	Date	Category	Remove
Pizza	80	Sat Oct 10 2020	Food	Remove
Grape Juice	30	Mon Oct 12 2020	Food	Remove
Cinema	210	Fri Oct 16 2020	Entertainment	Remove
Java Programming book	242	Thu Oct 15 2020	Academic	Remove
Mango Juice	35	Fri Oct 16 2020	Food	Remove
Dress	2000	Sun Oct 25 2020	Cloth	Remove
Tour	2555	Thu Oct 29 2020	Entertainment	Remove
Meals	300	Fri Oct 30 2020	Food	Remove
Mobile	3500	Mon Nov 02 2020	Gadgets	Remove
Exam Fees	1245	Wed Nov 04 2020	Academic	Remove
Total Amount	10197			

ReactJS - Animation

Animation is an exciting feature of modern web application. It gives a refreshing feel to the application. React community provides many excellent react based animation library like React Motion, React Reveal, react-animations, etc., React itself provides an animation library, *React Transition Group* as an add-on option earlier. It is an independent library enhancing the earlier version of the library. Let us learn React Transition Group animation library in this chapter.

React Transition Group

React Transition Group library is a simple implementation of animation. It does not do any animation out of the box. Instead, it exposes the core animation related information. Every animation is basically transition of an element from one state to another. The library exposes minimum possible state of every element and they are given below −

- Entering
- Entered
- Exiting
- Exited

The library provides options to set CSS style for each state and animate the element based on the style when the element moves from one state to another. The library provides in props to set the current state of the element. If *in* props value is true, then it means the element is moving from *entering* state to *exiting* state. If in props value is false, then it means the element is moving from exiting to exited.

Transition

Transition is the basic component provided by the *React Transition Group* to animate an element. Let us create a simple application and try to fade in / fade out an element using *Transition element*.

First, create a new react application, *react-animation-app* using *Create React App* or *Rollup* bundler by following instruction in *Creating a React application* chapter.

Next, install *React Transition Group* library.

```
cd /go/to/project
npm install react-transition-group --save
```

Next, open the application in your favorite editor.

Next, create src folder under the root directory of the application.

Next, create *components* folder under *src* folder.

Next, create a file, *HelloWorld.js* under *src/components* folder and start editing.

Next, import *React* and animation library.

```
import React from 'react';
import { Transition } from 'react-transition-group'
```

Next, create the *HelloWorld* component.

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

Next, define transition related styles as JavaScript objects in the constructor.

```
this.duration = 2000;
this.defaultStyle = {
   transition: `opacity ${this.duration}ms ease-in-out`,
   opacity: 0,
}
this.transitionStyles = {
   entering: { opacity: 1 },
   entered: { opacity: 1 },
   exiting: { opacity: 0 },
   exited: { opacity: 0 },
};
```

Here,

- *defaultStyles* sets the transition animation
- *transitionStyles* set the styles for various states

Next, set the initial state for the element in the constructor.

```
this.state = {
   inProp: true
}
```

Next, simulate the animation by changing the *inProp* values every 3 seconds.

```
setInterval(() => {
   this.setState((state, props) => {
      let newState = {
         inProp: !state.inProp
      };
      return newState;
   })
}, 3000);
```

Next, create a *render* function.

```
render() {
   return (
   );
}
```

Next, add *Transition* component. Use *this.state.inProp* for *in* prop and *this.duration* for *timeout prop*. Transition component expects a function, which returns the user interface. It is basically a *Render props*.

```
render() {
   return (
      <Transition in={this.state.inProp} timeout={this.duration}>
         {state => ({
            ... component's user interface.
         })
      </Transition>
   );
}
```

Next, write the components user interface inside a container and set the *defaultStyle* and *transitionStyles* for the container.

```
render() {
   return (
      <Transition in={this.state.inProp} timeout={this.duration}>
         {state => (
            <div style={{
               ...this.defaultStyle,
               ...this.transitionStyles[state]
            }}>
               <h1>Hello World!</h1>
            </div>
```

```
         )}
      </Transition>
   );
}
```

Finally, expose the component.

```
export default HelloWorld
```

The complete source code of the component is as follows −

```
import React from "react";
import { Transition } from 'react-transition-group';

class HelloWorld extends React.Component {
   constructor(props) {
      super(props);
      this.duration = 2000;
      this.defaultStyle = {
         transition: `opacity ${this.duration}ms ease-in-out`,
         opacity: 0,
      }
      this.transitionStyles = {
         entering: { opacity: 1 },
         entered: { opacity: 1 },
         exiting: { opacity: 0 },
         exited: { opacity: 0 },
      };
      this.state = {
         inProp: true
      }
      setInterval(() => {
         this.setState((state, props) => {
            let newState = {
               inProp: !state.inProp
            };
            return newState;
         })
      }, 3000);
   }
   render() {
      return (
         <Transition in={this.state.inProp} timeout={this.duration}>
            {state => (
               <div style={{
                  ...this.defaultStyle,
                  ...this.transitionStyles[state]
               }}>
                  <h1>Hello World!</h1>
               </div>
            )}
         </Transition>
      );
   }
}
export default HelloWorld;
```

Next, create a file, *index.js* under the *src* folder and use *HelloWorld* 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')
);
```

Finally, create a *public* folder under the root folder and create *index.html* file.

```
<!DOCTYPE html>
<html lang="en">
```

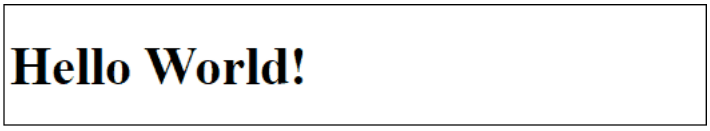
```
   <head>
      <meta charset="utf-8">
      <title>React Containment App</title>
   </head>
   <body>
      <div id="root"></div>
      <script type="text/JavaScript" src="./index.js"></script>
   </body>
</html>
```

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.

Clicking the remove link will remove the item from redux store.



CSSTransition

CSSTransition is built on top of *Transition* component and it improves *Transition* component by introducing *classNames* prop. *classNames* prop refers the css class name used for various state of the element.

For example, *classNames=hello* prop refers below css classes.

```
.hello-enter {
   opacity: 0;
}
.hello-enter-active {
   opacity: 1;
   transition: opacity 200ms;
}
.hello-exit {
   opacity: 1;
}
.hello-exit-active {
   opacity: 0;
   transition: opacity 200ms;
}
```

Let us create a new component *HelloWorldCSSTransition* using *CSSTransition* component.

First, open our *react-animation-app application* in your favorite editor.

Next, create a new file, *HelloWorldCSSTransition.css* under *src/components* folder and enter transition classes.

```
.hello-enter {
   opacity: 1;
   transition: opacity 2000ms ease-in-out;
}
.hello-enter-active {
   opacity: 1;
   transition: opacity 2000ms ease-in-out;
}
.hello-exit {
   opacity: 0;
   transition: opacity 2000ms ease-in-out;
}
.hello-exit-active {
   opacity: 0;
   transition: opacity 2000ms ease-in-out;
}
```

Next, create a new file, *HelloWorldCSSTransition.js* under *src/components* folder and start editing.

Next, import *React* and animation library.

```
import React from 'react';
import { CSSTransition } from 'react-transition-group'
```

Next, import *HelloWorldCSSTransition.css*.

```
import './HelloWorldCSSTransition.css'
```

Next, create the *HelloWorld* component.

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

Next, define duration of the transition in the constructor.

```
this.duration = 2000;
```

Next, set the initial state for the element in the constructor.

```
this.state = {
   inProp: true
}
```

Next, simulate the animation by changing the inProp values every 3 seconds.

```
setInterval(() => {
   this.setState((state, props) => {
      let newState = {
         inProp: !state.inProp
      };
      return newState;
   })
}, 3000);
```

Next, create a *render* function.

```
render() {
   return (
   );
}
```

Next, add *CSSTransition* component. Use *this.state.inProp* for in prop, *this.duration* for *timeout* prop and *hello* for *classNames* prop. *CSSTransition* component expects user interface as child prop.

```
render() {
   return (
      <CSSTransition in={this.state.inProp} timeout={this.duration}
         classNames="hello">
         // ... user interface code ...
      </CSSTransition>
   );
}
```

Next, write the components user interface.

```
render() {
   return (
       <CSSTransition in={this.state.inProp} timeout={this.duration}
      classNames="hello">
      <div>
          <h1>Hello World!</h1>
      </div>
       </CSSTransition>
   );
}
```

Finally, expose the component.

```
export default HelloWorldCSSTransition;
```

The complete source code of the component is given below −

```
import React from 'react';
import { CSSTransition } from 'react-transition-group'
import './HelloWorldCSSTransition.css'
```

```
class HelloWorldCSSTransition extends React.Component {
   constructor(props) {
      super(props);
      this.duration = 2000;
      this.state = {
         inProp: true
      }
      setInterval(() => {
         this.setState((state, props) => {
            let newState = {
               inProp: !state.inProp
            };
            return newState;
         })
      }, 3000);
   }
   render() {
      return (
         <CSSTransition in={this.state.inProp} timeout={this.duration}
            classNames="hello">
            <div>
               <h1>Hello World!</h1>
            </div>
         </CSSTransition>
      );
   }
}
export default HelloWorldCSSTransition;
```

Next, create a file, *index.js* under the *src* folder and use *HelloWorld* component.

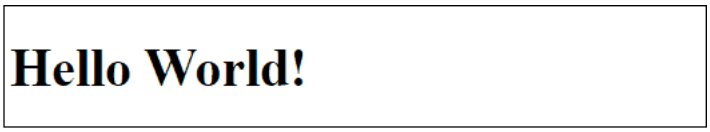
```
import React from 'react';
import ReactDOM from 'react-dom';
import HelloWorldCSSTransition from './components/HelloWorldCSSTransition';

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

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.

The message will fade in and out for every 3 seconds.



TransitionGroup

TransitionGroup is a container component, which manages multiple transition component in a list. For example, while each item in a list use *CSSTransition, TransitionGroup* can be used to group all the item for proper animation.

```
<TransitionGroup>
   {items.map(({ id, text }) => (
      <CSSTransition key={id} timeout={500} classNames="item" >
         <Button
            onClick={() =>
               setItems(items =>
                  items.filter(item => item.id !== id)
               )
            }
            >
            &times;
         </Button>
         {text}
```

```
      </CSSTransition>
   ))}
</TransitionGroup>
```

ReactJS - Testing

Testing is one of the processes to make sure that the functionality created in any application is working in accordance with the business logic and coding specification. React recommends *React testing library* to test React components and *jest* test runner to run the test. The *react-testing-library* allows the components to be checked in isolation.

It can be installed in the application using below command −

```
npm install --save @testing-library/react @testing-library/jest-dom
```

Create React app

Create React app configures *React testing library* and *jest* test runner by default. So, testing a React application created using *Create React App* is just a command away.

```
cd /go/to/react/application
npm test
```

The *npm* test command is similar to *npm build* command. Both re-compiles as and when the developer changes the code. Once the command is executed in the command prompt, it emits below questions.

```
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
   › Press a to run all tests.
   › Press f to run only failed tests.
   › Press q to quit watch mode.
   › Press p to filter by a filename regex pattern.
   › Press t to filter by a test name regex pattern.
   › Press Enter to trigger a test run.
```

Pressing a will try to run all the test script and finally summaries the result as shown below −

```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.312 s, estimated 12 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Testing in a custom application

Let us write a custom React application using *Rollup bundler* and test it using *React testing library* and *jest* test runner in this chapter.

First, create a new react application, *react-test-app* using *Rollup* bundler by following instruction in Creating a *React application* chapter.

Next, install the testing library.

```
cd /go/to/react-test-app
npm install --save @testing-library/react @testing-library/jest-dom
```

Next, open the application in your favorite editor.

Next, create a file, *HelloWorld.test.js* under *src/components* folder to write test for *HelloWorld* component and start editing.

Next, import react library.

```
import React from 'react';
```

Next, import the testing library.

```
import { render, screen } from '@testing-library/react'; import '@testing-library/jest-dom';
```

Next, import our *HelloWorld* component.

```
import HelloWorld from './HelloWorld';
```

Next, write a test to check the existence of *Hello World text* in the document.

```
test('test scenario 1', () => {
   render(<HelloWorld />);
   const element = screen.getByText(/Hello World/i);
   expect(element).toBeInTheDocument();
});
```

The complete source code of the test code is given below −

```
import React from 'react';
import { render, screen } from '@testing-library/react';
import '@testing-library/jest-dom';
import HelloWorld from './HelloWorld';

test('test scenario 1', () => {
   render(<HelloWorld />);
   const element = screen.getByText(/Hello World/i);
   expect(element).toBeInTheDocument();
});
```

Next, install jest test runner, if it is not installed already in the system.

```
npm install jest -g
```

Next, run jest command in the root folder of the application.

```
jest
```

Next, run jest command in the root folder of the application.

```
PASS  src/components/HelloWorld.test.js
   √ test scenario 1 (29 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        5.148 s
Ran all test suites.
```

ReactJS - CLI Commands

Let us learn the basic command available in Create React App command line application in this chapter.

Creating a new application

Create React App provides multiple ways to create React application.

Using *npx* script.

```
npx create-react-app <react-app-name>
npx create-react-app hello-react-app
```

Using *npm* package manager.

```
npm init react-app <react-app-name>
npm init react-app hello-react-app
```

Using *yarn* package manager.

```
yarn init react-app <react-app-name>
yarn init react-app hello-react-app
```

Selecting a template

Create React App creates React application using default template. Template refers the initial code with certain build-in functionality. There are hundreds of template with many advanced features are available in npm package server. *Create React App* allows the users to select the template

through *–template* command line switch.

```
create-react-app my-app --template typescript
```

Above command will create react app using *cra-template-typescript* package from npm server.

Installing a dependency

React dependency package can be installed using normal *npm* or *yarn* package command as React uses the project structure recommended by *npm* and *yarn*.

Using *npm* package manager.

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

Using *yarn* package manager.

```
yarn add react-router-dom
```

Running the application

React application can be started using *npm* or *yarn* command depending on the package manager used in the project.

Using *npm* package manager.

```
npm start
```

Using *yarn* package manager.

```
yarn start
```

To run the application in secure mode (HTTPS), set an environment variable, *HTTPS* and set it to true before starting the application. For example, in windows command prompt (cmd.exe), the below command set *HTTPS* and starts the application is HTTPS mode.

```
set HTTPS=true && npm start
```

ReactJS - Building & Deployment

Let us learn how to do production build and deployment of React application in this chapter.

Building

Once a React application development is done, application needs to be bundled and deployed to a production server. Let us learn the command available to build and deploy the application in this chapter.

A single command is enough to create a production build of the application.

```
npm run build
> expense-manager@0.1.0 build path\to\expense-manager
> react-scripts build

Creating an optimized production build...
Compiled with warnings.

File sizes after gzip:

   41.69 KB   build\static\js\2.a164da11.chunk.js
    2.24 KB   build\static\js\main.de70a883.chunk.js
    1.4  KB   build\static\js\3.d8a9fc85.chunk.js
    1.17 KB   build\static\js\runtime-main.560bee6e.js
  493     B   build\static\css\main.e75e7bbe.chunk.css

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
You may serve it with a static server:

   npm install -g serve
   serve -s build
```

```
Find out more about deployment here:

   https://cra.link/deployment
```

Once the application is build, the application is available under *build/static* folder.

By default, *profiling* option is disable and can be enabled through *–profile* command line option. *–profile* will include profiling information in the code. The profiling information can be used along with React DevTools to analyse the application.

```
npm run build -- --profile
```

Deployment

Once the application is build, it can be deployed to any web server. Let us learn how to deploy a React application in this chapter.

Local deployment

Local deployment can be done using serve package. Let us first install serve package using below command −

```
npm install -g server
```

To start the application using serve, use the below command −

```
cd /go/to/app/root/folder
serve -s build
```

By default, serve serve the application using port 5000. The application can be viewed @ *http://localhost:5000*.

Production deployment

Production deployment can be easily done by copying the files under build/static folder to the production application's root directory. It will work in all web server including Apache, IIS, Nginx, etc.

Relative path

By default, the production build is created assuming that the application will be hosted in the root folder of a web application. If the application needs to be hosted in a subfolder, then use below configuration in the package.json and then build the application.

```
{ ... "homepage": "http://domainname.com/path/to/subfolder", ... }
```

ReactJS - Example

Let us create a sample expense manager application by applying the concepts that we have learned in this tutorial. Some of the concepts are listed below −

- React basics (component, jsx, props and state)
- Router using **react-router**
- Http client programming (Web API)
- Form programming using Formik
- Advanced state management using Redux
- Async / await programming

Features

Some of the features of our sample expense manager application are −

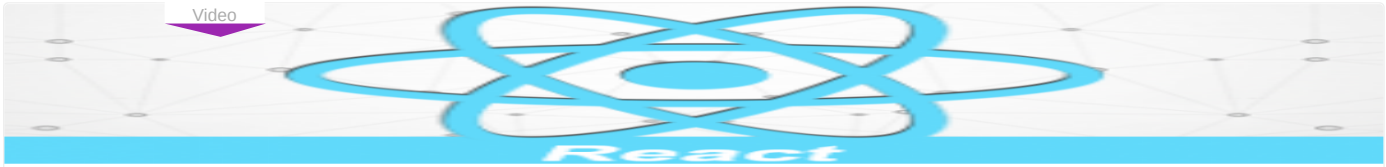
- Listing all the expenses from the server
- Add an expense item
- Delete an expense item

Here,

- Expense manager API
- Install necessary modules
- State management
- List expenses

- Add expense

Useful Video Courses

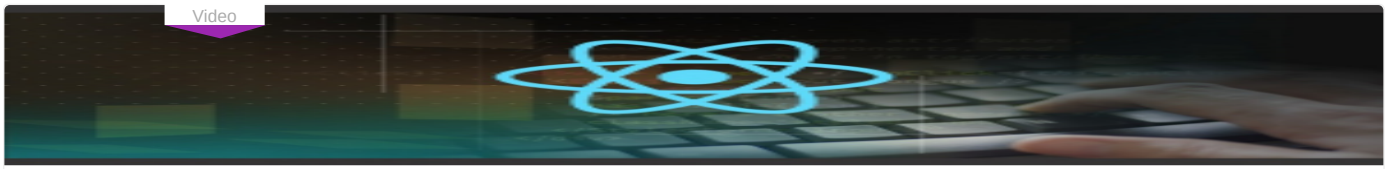


ReactJS Online Training

20 Lectures 1.5 hours

Anadi Sharma

More Detail



Master ReactJS: Learn React JS From Scratch

60 Lectures 4.5 hours

Skillbakerystudios

More Detail

Finest Laravel Course - Learn From 0 To Ninja With ReactJS

165 Lectures 13 hours

Paul Carlo Tordecilla

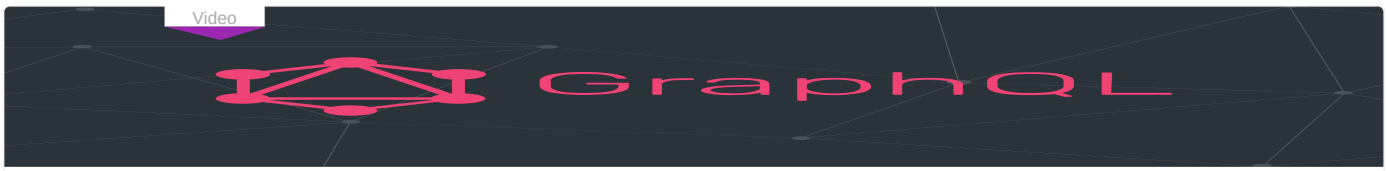
More Detail

React: Web Apps With ReactJS And Redux - The Complete Course

63 Lectures 9.5 hours

TELCOMA Global

More Detail



The GraphQL Apollo (With ReactJS And MongoDB)

17 Lectures 2 hours

Mohd Raqif Warsi

More Detail