

Table of Contents

About the Tutorial	ii
Audience.....	ii
Prerequisites.....	ii
Copyright & Disclaimer	ii
Table of Contents	iii
1. ReactJS — Introduction.....	1
React versions.....	1
Features.....	1
Benefits.....	1
Applications	1
2. ReactJS — Installation.....	3
Toolchain	3
The <i>serve</i> static server	4
Babel compiler.....	5
Create React App toolchain	5
3. ReactJS — Architecture	7
Workflow of a React application	7
Architecture of the React Application	11
4. React — Creating a React Application	13
Using CDN	13
Using Create React App tool.....	15
Files and folders.....	16
Source code of the application	18
Customize the code	19
Run the application	20
Using custom solution	21

Using Rollup bundler	21
Using Parcel bundler.....	26
5. React — JSX	30
Expressions	30
Functions	31
Attributes.....	31
Expression in attributes	32
6. ReactJS — Component	33
Creating a React component	33
Creating a class component.....	34
Creating a function component.....	36
7. React — Styling	38
CSS stylesheet.....	38
Inline Styling	39
CSS Modules	40
8. React — Properties (<i>props</i>)	43
Create a component using properties.....	43
Nested components	48
Use components.....	51
Component collection	53
9. React — Event management	59
Introduce events in Expense manager app	64
10. React — State Management	69
What is state?.....	69
State management API	69
Stateless component	70
Create a stateful component.....	71
Introduce state in expense manager app.....	74

State management using React Hooks	79
Create a stateful component.....	80
Introducing state in expense manager app	82
Component Life cycle	86
Working example of life cycle API	89
Life cycle api in Expense manager app	91
Component life cycle using React Hooks.....	92
React <i>children</i> property aka Containment	95
Layout in component.....	98
Sharing logic in component aka Render props.....	100
Pagination.....	101
Material UI.....	111
11. React — Http client programming.....	118
Expense Rest Api Server	118
The fetch() api.....	122
12. React — Form programming	129
Controlled component	129
Uncontrolled Component.....	137
Formik.....	143
13. React — Routing	152
Install React Router	152
Nested routing.....	154
Creating navigation.....	154
14. React — Redux.....	161
Concepts.....	161
Redux API.....	162
Provider component.....	163
15. React — Animation	175

<i>React Transition Group</i>	175
<i>Transition</i>	175
<i>CSSTransition</i>	179
<i>TransitionGroup</i>	183
16. React — Testing	184
<i>Create React app</i>	184
Testing in a custom application	185
17. React — CLI Commands	187
Creating a new application	187
Selecting a template	187
Installing a dependency	187
Running the application	188
18. React — Building and Deployment	189
Building	189
Deployment	190
19. React — Example	191
Expense manager API	191
Install necessary modules	193
State management	199
List expenses	205
Add expense	209

1. ReactJS — Introduction

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

React versions

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

Features

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

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

Benefits

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

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

Applications

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

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

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

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

2. ReactJS — Installation

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

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

```
node --version
```

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

```
v14.2.0
```

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

Toolchain

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

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

Some of the popular toolchain to develop React applications are:

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

Tools required to develop a React application are:

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

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

The *serve* static server

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

```
npm install serve -g
```

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

Open a command prompt and go to your workspace.

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

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

```
mkdir static_site  
cd static_site
```

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

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

Next, run the *serve* command.

```
serve .
```

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

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

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



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

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

Babel compiler

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

To install *Babel* and its React companion, run the below command:

```
npm install babel-cli@6 babel-preset-react-app@3 -g
...
...
+ babel-cli@6.26.0
+ babel-preset-react-app@3.1.2
updated 2 packages in 8.685s
```

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

Create React App toolchain

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

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

Updating the toolchain

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

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

Advantages of using React toolchain

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

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

3. ReactJS — Architecture

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

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

React elements

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

JSX

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

React component

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

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

Workflow of a React application

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

Open a command prompt and go to your workspace.

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

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

```
mkdir static_site  
cd static_site
```

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

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

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

Next, serve the application using serve web server.

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

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



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

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

React.createElement

Used to create React elements. It expects three parameters:

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

ReactDOM.render

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

- React Element OR JSX
- Root element of the webpage

Nested React element

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

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

It will generate the below content:

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

Use JSX

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

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

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

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

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

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

Hello JSX!

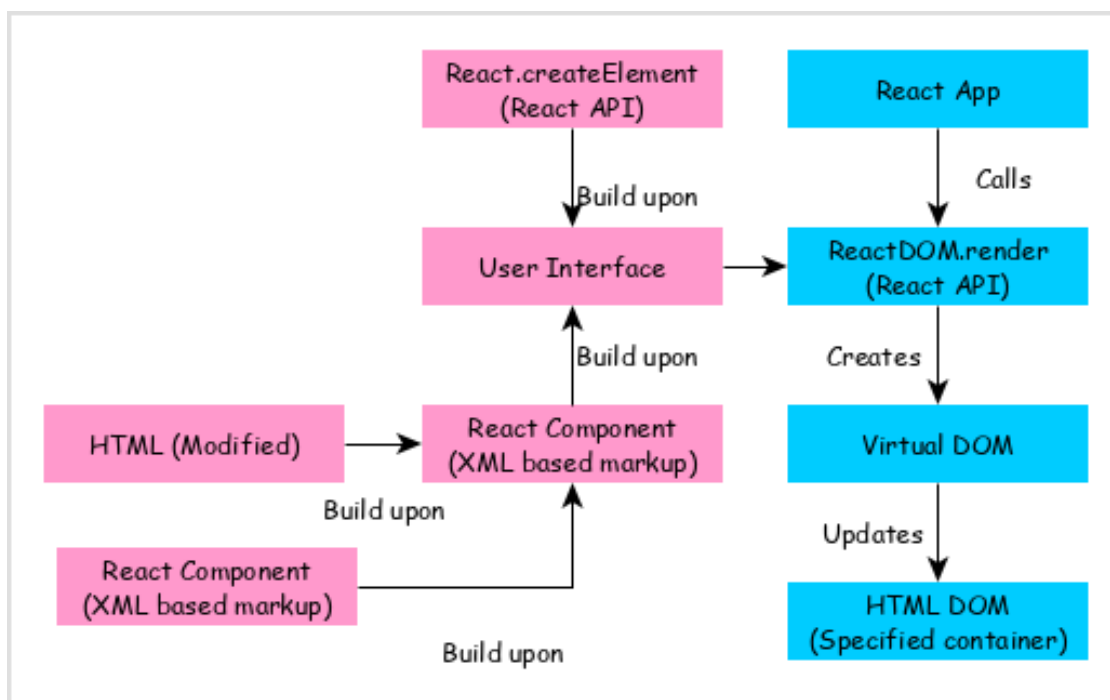
Next, let us create a new React component, *Greeting* and then try to use it in the webpage.

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

The result is same and as shown below:

Hello JSX!

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



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

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

Virtual DOM will be merged and rendered into the container.

Architecture of the React Application

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



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

4. React — Creating a React Application

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

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

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

Using CDN

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

Open a terminal and go to your workspace.

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

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

```
mkdir static_site  
cd static_site
```

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

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

Next, include *React library*.

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

```

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

```

Here,

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

```

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

```

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

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

```

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

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

```

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

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

```

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

```

```

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

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

```

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

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

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

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

Hello React!

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

Using Create React App tool

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

Open a terminal and go to your workspace.

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

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

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

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

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

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

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

Next, start the application.

```
npm start

Compiled successfully!

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

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

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

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



Let us analyse the structure of our React application.

Files and folders

The content of the React application is as follows:

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

```

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

```

Here,

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

```

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

```

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

- *react* and *react-dom* are core react libraries used to develop web application.
- *web-vitals* are general library to support application in different browser.
- *react-scripts* are core react scripts used to build and run application.
- *@testing-library/jest-dom*, *@testing-library/react* and *@testing-library/user-event* are testing library used to test the application after development.
- The **public folder** - Contains the core file, *index.html* and other web resources like images, logos, robots, etc., *index.html* loads our react application and render it in user's browser.
- The *src* folder - Contains the actual code of the application. We will check it next section.

Source code of the application

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

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

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

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

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

Here,

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

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

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

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

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

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

export default App;
```

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

Customize the code

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

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

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

The final structure of the application will be as follows:

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

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

```
import React from "react";

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



```

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

```

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

```

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

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

```

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

```

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

```

Run the application

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

```
> npm start
```

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

```

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

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

```

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

You can now view expense-manager in the browser.

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

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

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



Using custom solution

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

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

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

Using Rollup bundler

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

Open a terminal and go to your workspace.

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

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

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

Next, create and initialize the project.

```
npm init -y
```

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

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

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

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

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

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

Next, install corejs and regenerator runtime for async programming.

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

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

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

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

```

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

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

import postcss from 'rollup-plugin-postcss'

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

```

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

```

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

```

```

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

```

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

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

The final structure of the application will be as follows:

```

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

```

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

```

import React from "react";

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

```

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

```

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

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

```

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

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

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

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

Next, build and run the application.

```
npm start
```

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

```
> expense-manager-rollup@1.0.0 build /path/to/your/workspace/expense-manager-rollup  
> rollup -c  
  
rollup v2.36.1  
bundles src/index.js → dist\index.js...  
LiveReload enabled  
http://localhost:10001 -> /path/to/your/workspace/expense-manager-rollup/dist  
created dist\index.js in 4.7s  
  
waiting for changes...
```

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

Hello World!

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.

Hello World!

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

```
npm run build

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

√ Built in 6.42s.

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

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

5. React — JSX

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

- Assign to a variable.

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

- Assign to a variable based on a condition.

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

- Can be used as return value of a function.

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

- Can be used as argument of a function.

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

Expressions

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

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

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

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.

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

Here, *getCurrentTime()* is used to 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 have to be specified using *camelCase* convention (and it follows JavaScript DOM API) instead of normal HTML attribute name. For example, *class* attribute in HTML has to be defined as *className*. The following are few other examples:

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

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

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

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

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().toString();
  }

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

6. ReactJS — Component

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

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

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

React component accomplish these feature using three concepts:

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

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

Creating a React component

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

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

The core difference between function and class component are:

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

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

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

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

```
}
}
```

- Class components supports state management out of the box whereas function components does not support state management. But, React provides a hook, *useState()* for the function components to maintain its state.
- Class component have a life cycle and access to each life cycle events through dedicated callback apis. Function component does not have life cycle. Again, React provides a hook, *useEffect()* for the function component to access different stages of the component.

Creating a class component

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

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

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

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

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

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

class ExpenseEntryItem extends React.Component {
}
```

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

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

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

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

```

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

```

Next, specify the component as default export class.

```

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

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

export default ExpenseEntryItem;

```

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

```

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

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

```

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

```

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

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

```



```

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

```

Next, serve the application using npm command.

```
npm start
```

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

```

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

```

Creating a function component

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

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

```

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

```

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

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

7. React — Styling

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

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

Let us 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.

```

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

```

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

Inline Styling

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

Let us add inline styling in our component.

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

```

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

```

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

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

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

Also, style can be directly set inside the component:

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

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

Next, serve the application using npm command.

```
npm start
```

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

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

CSS Modules

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

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

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

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

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

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

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

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

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

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

The final and complete code is:

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

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

export default ExpenseEntryItem;
```

Next, serve the application using npm command.

```
npm start
```

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

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

8. React — Properties (*props*)

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

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

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

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

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

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

Let us learn one by one in this chapter.

Create a component using properties

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

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

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

Introduce construction function with argument props.

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

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

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



```

        </div>
    );
}

```

Here,

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

Now, we have successfully updated the component using properties.

```

import React from 'react'

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

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

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

export default ExpenseEntryItem;

```

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

```

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

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

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem

```

```

      name={name}
      amount={amount}
      spendDate={spendDate}
      category={category} />
    </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.

Item: *Grape Juice*
Amount: *30*
Spend Date: *Sat Oct 10 2020 05:30:00 GMT+0530 (India Standard Time)*
Category: *Food*

The complete code to do it using CDN in a webpage is as follows:

```

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

    <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js" crossorigin></script>
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    <script type="text/babel">
      class ExpenseEntryItem extends React.Component {
        constructor(props) {
          super(props);
        }

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

```

```

    <em>{this.props.category}</em></div>
      </div>
    );
  }
}

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

ReactDOM.render(
  <ExpenseEntryItem
    name={name}
    amount={amount}
    spendDate={spendDate}
    category={category} />,
  document.getElementById('react-app') );
</script>
</body>
</html>

```

Objects as properties

Let us learn how to use JavaScript object as attributes in this chapter.

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

Next, open *ExpenseEntryItem.js* file.

Next, change the *render()* method and access the input object item through *this.props.item* property.

```

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

```

Next, open *index.js* and represent the expense entry item in JavaScript object.

```

const item = {
  id: 1,
  name : "Grape Juice",
  amount : 30.5,
  spendDate: new Date("2020-10-10"),
  category: "Food"
}

```

Next, pass the object to the component using curly brace (`{}`) syntax in the component attributes.

```
<ExpenseEntryItem item={item} />
```

The complete code of `index.js` is as follows:

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

const item = {
  id: 1,
  name : "Grape Juice",
  amount : 30.5,
  spendDate: new Date("2020-10-10"),
  category: "Food"
}

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem item={item} />
  </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

```
Item: Grape Juice
Amount: 30
Spend Date:Sat Oct 10 2020 05:30:00 GMT+0530 (India Standard Time)
Category: Food
```

The complete code to do it using CDN in a webpage is as follows:

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

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

```

dom.development.js" crossorigin></script>
  <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  <script type="text/babel">
    class ExpenseEntryItem extends React.Component {
      constructor(props) {
        super(props);
      }

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

    const item = {
      id: 1,
      name : "Grape Juice",
      amount : 30.5,
      spendDate: new Date("2020-10-10"),
      category: "Food"
    }

    ReactDOM.render(
      <ExpenseEntryItem item={item} />,
      document.getElementById('react-app') );
  </script>
</body>
</html>

```

Nested components

As we learned earlier, React component is the building block of a React application. A React component is made up of the multiple individual components. React allows multiple components to be combined to create larger components. Also, React components can be nested to any arbitrary level. Let us see how React components can be composed in this chapter.

FormattedMoney component

Let us create a component, *FormattedMoney* to format the amount to two decimal places before rendering.

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

Next, Create a *FormattedMoney.js* file in the *src/components* folder and, Import *React* library.

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

Next, create a class, *FormattedMoney* by extending *React.Component*.

```
class FormattedMoney extends React.Component {  
}
```

Next, introduce construction function with argument props as shown below:

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

Next, create a method *format* to format the amount.

```
format(amount) {  
  return parseFloat(amount).toFixed(2)  
}
```

Next, create a method *render* to emit the formatted amount.

```
render() {  
  return (  
    <span>{this.format(this.props.value)}</span>  
  );  
}
```

Here, we have used the *format* method by passing *value* attribute through *this.props*.

Next, specify the component as default export class.

```
export default FormattedMoney;
```

Now, we have successfully created our *FormattedMoney* React component.

```
import React from 'react';  
  
class FormattedMoney extends React.Component {  
  constructor(props) {  
    super(props)  
  }  
  
  format(amount) {  
    return parseFloat(amount).toFixed(2)  
  }  
}
```

```

    render() {
      return (
        <span>{this.format(this.props.value)}</span>
      );
    }
  }
}

export default FormattedMoney;

```

FormattedDate component

Let us create another component, *FormattedDate* to format and show the date and time of the expense.

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

Next, create a file, *FormattedDate.js* in the *src/components* folder.

Next, import *React* library.

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

Next, create a class by extending *React.Component*.

```
class FormattedDate extends React.Component {
}
```

Next, introduce construction function with argument props as shown below:

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

Next, create a method *format* to format the date.

```
format(val) {
  const months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN", "JUL", "AUG",
    "SEP", "OCT",
    "NOV", "DEC"];
  let parsed_date = new Date(Date.parse(val));
  let formatted_date = parsed_date.getDate() +
    "-" + months[parsed_date.getMonth()] +
    "-" + parsed_date.getFullYear()

  return formatted_date;
}
```

Create a method *render* to emit the formatted date.

```
render() {
  return (
    <span>{this.format(this.props.value)}</span>
  );
}
```

Here, we have used the `format` method by passing value attribute through *this.props*.

Next, specify the component as default export class.

```
export default FormattedDate;
```

Now, we have successfully created our *FormattedDate* React component. The complete code is as follows:

```
import React from 'react';

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

  format(val) {
    const months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN", "JUL", "AUG",
"SEP", "OCT",
                    "NOV", "DEC"];
    let parsed_date = new Date(Date.parse(val));
    let formatted_date = parsed_date.getDate() +
                        "-" + months[parsed_date.getMonth()] +
                        "-" + parsed_date.getFullYear()

    return formatted_date;
  }

  render() {
    return (
      <span>{this.format(this.props.value)}</span>
    );
  }
}

export default FormattedDate;
```

Use components

Let us use newly created components and enhance our *ExpenseEntryItem* component.

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

Next, open *ExpenseEntryItem.js* file.

Next, import *FormattedMoney* and *FormattedDate*.

```
import FormattedMoney from './FormattedMoney'
import FormattedDate from './FormattedDate'
```

Next, update the *render* method by including *FormattedMoney* and *FormattedDate* component.

```
render() {
  return (
```



```

    <div>
      <div><b>Item:</b> <em>{this.props.item.name}</em></div>
      <div><b>Amount:</b>
        <em>
          <FormattedMoney
            value={this.props.item.amount} />
          </em>
        </div>
      <div><b>Spend Date:</b>
        <em>
          <FormattedDate value={this.props.item.spendDate} />
        </em>
      </div>
      <div><b>Category:</b>
        <em>{this.props.item.category}</em></div>
    </div>
  );
}

```

Here, we have passed the *amount* and *spendDate* through *value* attribute of the components.

The final updated source code of the *ExpenseEntryItem* component is given below:

```

import React from 'react'

import FormattedMoney from './FormattedMoney'
import FormattedDate from './FormattedDate'

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

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

```

```
}

export default ExpenseEntryItem;
```

Open *index.js* and call the *ExpenseEntryItem* component by passing the item object.

```
const item = {
  id: 1,
  name : "Grape Juice",
  amount : 30.5,
  spendDate: new Date("2020-10-10"),
  category: "Food"
}

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem item={item} />
  </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.

```
Item: Grape Juice
Amount:30.50
Spend Date:10-OCT-2020
Category:Food
```

Component collection

In modern application, developer encounters a lot of situation, where list of item (e.g. todos, orders, invoices, etc.,) has to be rendered in tabular format or gallery format. React provides clear, intuitive and easy technique to create list based user interface. React uses two existing features to accomplish this feature.

- JavaScript's built-in *map* method.
- React expression in *jsx*.

The *map* function accepts a collection and a mapping function. The map function will be applied to each and every item in the collection and the results are used to generate a new list.

For example, declare a JavaScript array with 5 random numbers as shown below:

```
let list = [10, 30, 45, 12, 24]
```

Now, apply an anonymous function, which double its input as shown below:

```
result = list.map((input) => input * 2);
```

Then, the resulting list is:

```
[20, 60, 90, 24, 48]
```

To refresh the React expression, let us create a new variable and assign a React element.

```
var hello = <h1>Hello!</h1>
var final = <div>{helloElement}</div>
```

Now, the React expression, *hello* will get merged with *final* and generate,

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

Let us apply the concept to create a component to show a collection of expense entry items in a tabular format.

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

Next, create a file, *ExpenseEntryItemList.css* in *src/components* folder to include styles for the component.

Next, create a file, *ExpenseEntryItemList.js* in *src/components* folder to create *ExpenseEntryItemList* component

Next, import *React* library and the stylesheet.

```
import React from 'react';

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

Next, create *ExpenseEntryItemList* class and call constructor function.

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

Next, create a *render* function.

```
render() {
}
```

Next, Use *map* method to generate a collection of HTML table rows each representing a single expense entry item in the list.

```
render() {
  const lists = this.props.items.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>
      </tr>
    );
  }

```

Here, *key* identifies each row and it has to be unique among the list.

Next, in the *render()* method, create a HTML table and include the *lists* expression in the rows section.

```

return (
  <table>
    <thead>
      <tr>
        <th>Item</th>
        <th>Amount</th>
        <th>Date</th>
        <th>Category</th>
      </tr>
    </thead>
    <tbody>
      {lists}
    </tbody>
  </table>
);

```

Finally, export the component.

```
export default ExpenseEntryItemList;
```

Now, we have successfully created the component to render the expense items into HTML table. The complete code is as follows:

```

import React from 'react';

import './ExpenseEntryItemList.css'

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

  render() {
    const lists = this.props.items.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>
      </tr>
    );
  }
}

```

```

    return (
      <table>
        <thead>
          <tr>
            <th>Item</th>
            <th>Amount</th>
            <th>Date</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          {lists}
        </tbody>
      </table>
    );
  }
}

export default ExpenseEntryItemList;

```

Next, open *index.js* and import our newly created *ExpenseEntryItemList* component.

```
import ExpenseEntryItemList from './components/ExpenseEntryItemList'
```

Next, declare a list (of expense entry item) and populate it with some random values in *index.js* file.

```

const items = [
  { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food" },
  { id: 1, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category: "Food" },
  { id: 1, name: "Cinema", amount: 210, spendDate: "2020-10-16", category: "Entertainment" },
  { id: 1, name: "Java Programming book", amount: 242, spendDate: "2020-10-15", category: "Academic" },
  { id: 1, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category: "Food" },
  { id: 1, name: "Dress", amount: 2000, spendDate: "2020-10-25", category: "Cloth" },
  { id: 1, name: "Tour", amount: 2555, spendDate: "2020-10-29", category: "Entertainment" },
  { id: 1, name: "Meals", amount: 300, spendDate: "2020-10-30", category: "Food" },
  { id: 1, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category: "Gadgets" },
  { id: 1, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category: "Academic" }
]

```

Next, use *ExpenseEntryItemList* component by passing the *items* through *items* attributes.

```

ReactDOM.render(
  <React.StrictMode>

```

```

    <ExpenseEntryItemList items={items} />
  </React.StrictMode>,
  document.getElementById('root')
);

```

The complete code of *index.js* is as follows:

```

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

const items = [
  { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food"
},
  { id: 1, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category:
"Food" },
  { id: 1, name: "Cinema", amount: 210, spendDate: "2020-10-16", category:
"Entertainment" },
  { id: 1, name: "Java Programming book", amount: 242, spendDate: "2020-10-15",
category: "Academic" },
  { id: 1, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category:
"Food" },
  { id: 1, name: "Dress", amount: 2000, spendDate: "2020-10-25", category:
"Cloth" },
  { id: 1, name: "Tour", amount: 2555, spendDate: "2020-10-29", category:
"Entertainment" },
  { id: 1, name: "Meals", amount: 300, spendDate: "2020-10-30", category:
"Food" },
  { id: 1, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category:
"Gadgets" },
  { id: 1, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category:
"Academic" }
]

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItem item={item} />
  </React.StrictMode>,
  document.getElementById('root')
);

```

Next, open *ExpenseEntryItemList.css* and add style for the table.

```

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;
}

```

Next, serve the application using npm command.

```
npm start
```

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

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

9. React — 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) => {
  console.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={(e) => this.log(extra, e)}> ... </div>
```

Create a event-aware component

Let us create a new component, *MessageWithEvent* and handle events in the component to better understand event management in React application.

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

Next, create a file, *MessageWithEvent.js* in *src/components* folder to create *MessageWithEvent* component.

Import *React* library.

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

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

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

Next, create an event handler method, *logEventToConsole*, which will log event details to the console.

```
logEventToConsole(e) {
  console.log(e.target.innerHTML);
}
```

Next, create a *render* function.

```
render() {
}
```

Next, create a greeting message and return it.

```
render() {
  return (
    <div>
      <p>Hello {this.props.name}!</p>
    </div>
  );
}
```

Next, set *logEventToConsole* method as the event handler for click event of the root container(*div*).

```
render() {
  return (
    <div onClick={this.logEventToConsole}>
      <p>Hello {this.props.name}!</p>
    </div>
  );
}
```

Next, update the constructor by binding the event handler.

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

    this.logEventToConsole = this.logEventToConsole.bind();
  }
}
```

Finally, export the component.

```
export default MessageWithEvent;
```

The complete code of the *MessageWithEvent* component is given below:

```
import React from 'react';

class MessageWithEvent extends React.Component {
  constructor(props) {
```

```

    super(props);

    this.logEventToConsole = this.logEventToConsole.bind();
  }

  logEventToConsole(e) {
    console.log(e.target.innerHTML);
  }

  render() {
    return (
      <div onClick={this.logEventToConsole}>
        <p>Hello {this.props.name}!</p>
      </div>
    );
  }
}

export default MessageWithEvent;

```

Next, open index.js and import *MessageWithEvent*.

```
import MessageWithEvent from './components/MessageWithEvent'
```

Next, build the user interface of the application by using *MessageWithEvent* component.

```

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

ReactDOM.render(
  <React.StrictMode>
    <div>
      <MessageWithEvent name="React" />
      <MessageWithEvent name="React developer" />
    </div>
  </React.StrictMode>,
  document.getElementById('root')
);

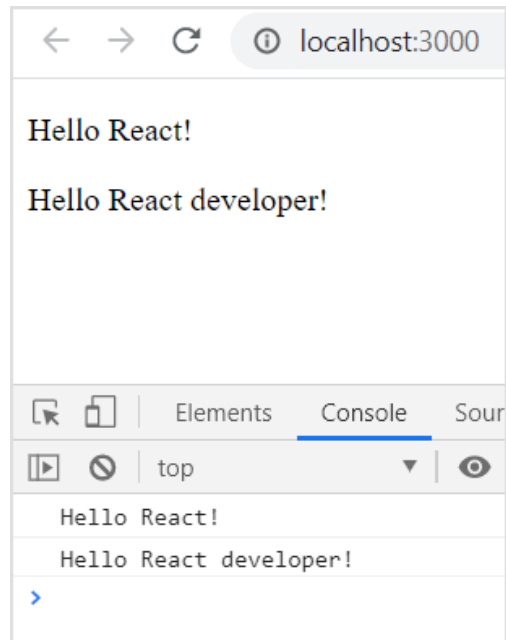
```

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.

Now, click both *MessageWithEvent* component and the application will emit messages in the console as shown below.



Let us try to pass and an extra information (for example, *msgid*) to event handler.

First, update the *logEventToConsole* to accept an extra argument, *msgid*.

```
logEventToConsole(msgid, e) {
  console.log(e.target.innerHTML);
  console.log(msgid);
}
```

Next, pass message id to the event handler by binding the message id in the *render* method.

```
render() {
  return (
    <div onClick={this.logEventToConsole.bind(this,
    Math.floor(Math.random() * 10))}>
      <p>Hello {this.props.name}!</p>
    </div>
  );
}
```

The complete and updated code is as follows:

```
import React from 'react';

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

    this.logEventToConsole = this.logEventToConsole.bind();
  }

  logEventToConsole(msgid, e) {
    console.log(e.target.innerHTML);
    console.log(msgid);
  }
}
```

```

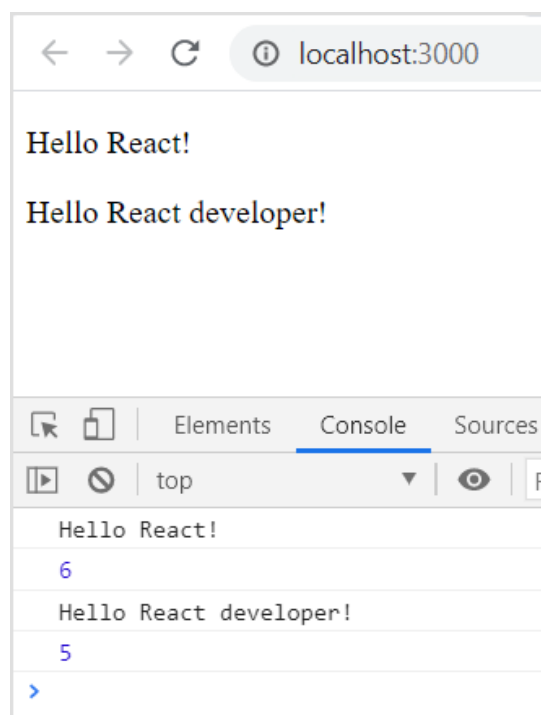
    }

    render() {
      return (
        <div onClick={this.logEventToConsole.bind(this,
Math.floor(Math.random() * 10))}>
          <p>Hello {this.props.name}!</p>
        </div>
      );
    }
  }

export default MessageWithEvent;

```

Run the application and you will find that the event emits message id in the console.



Introduce events in Expense manager app

Let us do some event management in our expense application. We can try to highlight the expense entry item in the table when the user moves the cursor over it.

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

Open *ExpenseEntryItemList.js* file and add a method *handleMouseEnter* to handle the event fired (*onMouseEnter*) when the user moves the mouse pointer into the expense items (td - table cell).

```

handleMouseEnter(e) {
  e.target.parentNode.classList.add("highlight");
}

```

Here,

- Event handler tries to find the parent node (tr) of the event target (td) node using *parentNode* method. The *parentNode* method is the standard DOM method to find the immediate parent the current node.
- Once the parent node is found, event handler access the list of the css class attached to the parent node and adds 'highlight' class using *add* method. *classList* is the standard DOM property to get list of class attached to the node and it can be used to add / remove class from a DOM node.

Next, add a method, *handleMouseLeave* to handle the event fired when the user moves out of the expense item.

```
handleMouseLeave(e) {
  e.target.parentNode.classList.remove("highlight");
}
```

Here, Event handler removes the *highlight* class from the DOM.

Next, add a method, *handleMouseOver* to check where the mouse is currently positioned. It is optional to find the where about of the mouse pointer in the DOM.

```
handleMouseOver(e) {
  console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
}
```

Next, bind all event handlers in the constructor of the component.

```
this.handleMouseEnter = this.handleMouseEnter.bind();
this.handleMouseLeave = this.handleMouseLeave.bind();
this.handleMouseOver = this.handleMouseOver.bind();
```

Next, attach the event handlers to the corresponding tag in the *render* method.

```
render() {
  const lists = this.props.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>
    </tr>
  );

  return (
    <table onMouseOver={this.handleMouseOver}>
      <thead>
        <tr>
          <th>Item</th>
          <th>Amount</th>
          <th>Date</th>
          <th>Category</th>
        </tr>
      </thead>
```

```

        <tbody>
          {lists}
        </tbody>
      </table>
    );
  }

```

The final and complete code of the *ExpenseEntryItemList* is as follows:

```

import React from 'react';

import './ExpenseEntryItemList.css';

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

    this.handleMouseEnter = this.handleMouseEnter.bind();
    this.handleMouseLeave = this.handleMouseLeave.bind();
    this.handleMouseOver = this.handleMouseOver.bind();
  }

  handleMouseEnter(e) {
    e.target.parentNode.classList.add("highlight");
  }

  handleMouseLeave(e) {
    e.target.parentNode.classList.remove("highlight");
  }

  handleMouseOver(e) {
    console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
  }

  render() {
    const lists = this.props.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>
      </tr>
    );

    return (
      <table onMouseOver={this.handleMouseOver}>
        <thead>
          <tr>
            <th>Item</th>
            <th>Amount</th>
            <th>Date</th>
            <th>Category</th>
          </tr>

```

```

                </thead>
                <tbody>
                    {lists}
                </tbody>
            </table>
        );
    }
}

export default ExpenseEntryItemList;

```

Next, open the css file, *ExpenseEntryItemList.css* and add a css class, *highlight*.

```

tr.highlight td {
    background-color: #a6a8bd;
}

```

Next, open *index.js* and use the *ExpenseEntryItemList* component.

```

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

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" }
]

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItemList items={items} />
  </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 application will respond to mouse events and highlight the currently selected row.

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

10. React — 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.

State management API

As we learned earlier, React component maintains and expose it's state through *this.state* of the component. React provides a single API to maintain state in the component. The API is *this.setState()*. It accepts either a JavaScript object or a function that returns a JavaScript object.

The signature of the *setState* API is as follows:

```
this.setState( { ... object ...} );
```

A simple example to set / update name is as follows:

```
this.setState( { name: 'John' } )
```

The signature of the *setState* with function is as follows:

```
this.setState( (state, props) =>
  ... function returning JavaScript object ... );
```

Here,

- *state* refers the current state of the React component
- *props* refers the current properties of the React component.

React recommends to use *setState* API with function as it works correctly in *async* environment. Instead of lambda function, normal JavaScript function can be used as well.

```
this.setState( function(state, props) {
    return ... JavaScript object ...
})
```

A simple example to update the amount using function is as follows:

```
this.setState( (state, props) => ({
    amount: this.state.amount + this.props.additionaAmount
}))
```

React state should not be modified directly through *this.state* member variable and updating the state through member variable does not re-render the component.

A special feature of React state API is that it will be merged with the existing state instead of replacing the state. For example, we can update any one of the state fields at a time instead of updating the whole object. This feature gives the developer the flexibility to easily handle the state data.

For example, let us consider that the internal state contains a student record.

```
{
  name: 'John',
  age: 16
}
```

We can update only the age using *setState* API, which will automatically merge the new object with the existing student record object.

```
this.setState( (state, props) => ({
  age: 18
}));
```

Stateless component

React component with internal state is called *Stateful component* and React component without any internal state management is called *Stateless component*. React recommends to create and use as many stateless component as possible and create stateful component only when it is absolutely necessary. Also, React does not share the state with child component. The data needs to be passed to the child component through child's properties.

An example to pass date to the *FormattedDate* component is as follows:

```
<FormattedDate value={this.state.item.spend_date} />
```

The general idea is not to overcomplicate the application logic and use advanced features only when necessary.

Create a stateful component

Let us create a React application to show the current date and time.

First, create a new react application, *react-clock-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, *Clock.js* under *src/components* folder and start editing.

Next, import *React* library.

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

Next, create *Clock* component.

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

Next, initialize state with current date and time.

```
  constructor(props) {  
    super(props);  
  
    this.state = {  
      date: new Date()  
    }  
  }
```

Next, add a method, *setTime()* to update the current time:

```
  setTime() {  
    console.log(this.state.date);  
    this.setState((state, props) => {  
      {  
        date: new Date()  
      }  
    })  
  }
```

Next, use JavaScript method, *setInterval* and call *setTime()* method every second to ensure that the component's state is updated every second.

```

constructor(props) {
  super(props);

  this.state = {
    date: new Date()
  }

  setInterval( () => this.setTime(), 1000);
}

```

Next, create a *render* function.

```

render() {
}

```

Next, update the `render()` method to show the current time.

```

render() {
  return (
    <div><p>The current time is {this.state.date.toString()}</p></div>
  );
}

```

Finally, export the component.

```

export default Clock;

```

The complete source code of the *Clock* component is as follows:

```

import React from 'react';

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

    this.state = {
      date: new Date()
    }

    setInterval( () => this.setTime(), 1000);
  }

  setTime() {
    console.log(this.state.date);
    this.setState((state, props) => (
      {
        date: new Date()
      }
    ))
  }

  render() {
    return (
      <div><p>The current time is {this.state.date.toString()}</p></div>
    );
  }
}

```

```

    });
  }
}

export default Clock;

```

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

```

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

ReactDOM.render(
  <React.StrictMode>
    <Clock />
  </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>Clock</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. The application will show the time and update it every second.

```
The current time is Wed Nov 11 2020 10:10:18 GMT+0530 (India Standard Time)
```

The above application works fine but throws an error in the console.

```
Can't call setState on a component that is not yet mounted.
```

The error message indicates that the *setState* has to be called only after the component is mounted.

What is mounting?

React component has a life-cycle and *mounting* is one of the stages in the life cycle. Let us learn more about the life-cycle in the upcoming chapters.

Introduce state in expense manager app

Let us introduce state management in the expense manager application by adding a simple feature to remove an expenses item.

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

Next, open *ExpenseEntryItemList.js* file.

Next, initialize the state of the component with the expense items passed into the components through properties.

```
this.state = {
  items: this.props.items
}
```

Next, add the *Remove* label in the *render()* method.

```
<thead>
  <tr>
    <th>Item</th>
    <th>Amount</th>
    <th>Date</th>
    <th>Category</th>
    <th>Remove</th>
  </tr>
</thead>
```

Next, update the lists in the *render()* method to include the remove link. Also, use items in the state (*this.state.items*) instead of items from the properties (*this.props.items*).

```
const 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>
);
```

Next, implement *handleDelete* method, which will remove the relevant expense item from the state.

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

  this.setState((state, props) => {
    let items = [];

    state.items.forEach((item, idx) => {
```

```

        if(item.id !== id)
            items.push(item)
    })

    let newState = {
        items: items
    }
    return newState;
  })
}

```

Here,

- Expense items are fetched from the current state of the component.
- Current expense items are looped over to find the item referred by the user using id of the item.
- Create a new item list with all the expense item except the one referred by the user

Next, add a new row to show the total expense amount.

```

<tr>
  <td colSpan="1" style={{ textAlign: "right" }}>Total Amount</td>
  <td colSpan="4" style={{ textAlign: "left" }}>
    {this.getTotal()}
  </td>
</tr>

```

Next, implement the *getTotal()* method to calculate the total expense amount.

```

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

```

The complete code of the *render()* method is as follows:

```

render() {
  const 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).toLocaleDateString()}</td>
      <td>{item.category}</td>
      <td><a href="#"
        onClick={(e) => this.handleDelete(item.id, e)}>Remove</a></td>
    </tr>
  );

  return (

```



```

    <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}
        <tr>
          <td colspan="1" style={{ textAlign: "right" }}>Total
Amount</td>
          <td colspan="4" style={{ textAlign: "left" }}>
            {this.getTotal()}
          </td>
        </tr>
      </tbody>
    </table>
  );
}

```

Finally, the updated code of the *ExpenseEntryItemList* is as follows:

```

import React from 'react';

import './ExpenseEntryItemList.css';

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

    this.state = {
      items: this.props.items
    }

    this.handleMouseEnter = this.handleMouseEnter.bind();
    this.handleMouseLeave = this.handleMouseLeave.bind();
    this.handleMouseOver = this.handleMouseOver.bind();
  }

  handleMouseEnter(e) {
    e.target.parentNode.classList.add("highlight");
  }

  handleMouseLeave(e) {
    e.target.parentNode.classList.remove("highlight");
  }

  handleMouseOver(e) {
    console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
  }
}

```

```

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

  this.setState((state, props) => {
    let items = [];

    state.items.forEach((item, idx) => {
      if(item.id !== id)
        items.push(item)
    })

    let newState = {
      items: items
    }
    return newState;
  })
}

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

render() {
  const 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).toLocaleDateString()}</td>
      <td>{item.category}</td>
      <td><a href="#"
        onClick={(e) => this.handleDelete(item.id,
e)}>Remove</a></td>
    </tr>
  );

  return (
    <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}
        <tr>
            <td colSpan="1" style={{ textAlign: "right" }}>Total
Amount</td>
            <td colSpan="4" style={{ textAlign: "left" }}>
                {this.getTotal()}
            </td>
        </tr>
    </tbody>
</table>
    );
}
}

export default ExpenseEntryItemList;

```

Next, Update the *index.js* and include the *ExpenseEntryItemList* component.

```

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

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" }
]

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItemList items={items} />
  </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.

Finally, to remove an expense item, click the corresponding remove link. It will remove the corresponding item and refresh the user interface as shown in animated gif.

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			

State management using React Hooks

React introduces an entirely new concepts called *React Hooks* from React 16.8. Even though, it is a relatively new concept, it enables React functional component to have its own state and life-cycle. Also, React Hooks enables functional component to use many of the feature not available earlier. Let us see how to do state management in a functional component using React Hooks in this chapter.

What is React Hooks?

React Hooks are special functions provided by React to handle a specific functionality inside a React functional component. React provides a Hook function for every supported feature. For example, React provides `useState()` function to manage state in a functional component. When a React functional component uses React Hooks, React Hooks attach itself into the component and provides additional functionality.

The general signature of `useState()` Hook is as follows:

```
const [<state variable>, <state update function>] = useState(<initial value>);
```

For example, state management in clock component using Hooks can be done as specified below:

```
const [currentDateTime, setCurrentDateTime] = useState(new Date());

setInterval(() => setCurrentDateTime(new Date()), 1000);
```

Here,

- *currentDateTime* - Variable used to hold current date and time (returned by *useState()*)
- *setCurrentDate()* - Function used to set current date and time (returned by *useState()*)

Create a stateful component

Let us recreate our clock component using Hooks in this chapter.

First, create a new react application, *react-clock-hook-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, *Clock.js* under *src/components* folder and start editing.

Next, import *React* library and React state Hook, *useState*.

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

Next, create *Clock* component.

```
function Clock() {

}
```

Next, create state Hooks to maintain date and time.

```
const [currentDateTime, setCurrentDateTime] = useState(new Date());
```

Next, set date and time for every second.

```
setInterval(() => setCurrentDateTime(new Date()), 1000);
```

Next, create the user interface to show the current date and time using *currentDateTime* and return it.

```
return (
  <div><p>The current time is {currentDateTime.toString()}</p></div>
);
```

Finally, export the component using the code snippet:

```
export default Clock;
```

The complete source code of the *Clock* component is as follows:

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

function Clock(props) {
  const [currentDateTime, setCurrentDateTime] = useState(new Date());

  setInterval(() => setCurrentDateTime(new Date()), 1000);

  return (
    <div><p>The current time is {currentDateTime.toString()}</p></div>
  );
}

export default Clock;
```

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

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

ReactDOM.render(
  <React.StrictMode>
    <Clock />
  </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>Clock</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. The application will show the time and update it every second.

The current time is Wed Nov 11 2020 10:10:18 GMT+0530 (India Standard Time)

The above application works fine. But, *setCurrentDateTime()* set to execute every second has to be removed when the application ends. We can do this using another Hook,

useEffect provided by React. We will learn it in the upcoming chapter (*Component life cycle*).

Introducing state in expense manager app

Let us introduce state management in the expense manager application by adding a simple feature to remove an expenses item using Hooks in this chapter.

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

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

Next, import *React* library and React state Hook, *useState*.

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

Next, import the css, *ExpenseEntryItem.css*.

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

Next, create *ExpenseEntryItemListFn* component.

```
function ExpenseEntryItemListFn(props) {  
  }  
}
```

Next, initialize the state Hooks of the component with the expense items passed into the components through properties.

```
const [items, setItems] = useState(props.items);
```

Next, create event handlers to highlight the rows.

```
function handleMouseEnter(e) {  
  e.target.parentNode.classList.add("highlight");  
}  
  
function handleMouseLeave(e) {  
  e.target.parentNode.classList.remove("highlight");  
}  
  
function handleMouseOver(e) {  
  console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");  
}
```

Next, create event handler to remove the selected items using *items* and *setItems()*.

```
function handleDelete(id, e) {  
  e.preventDefault();  
  console.log(id);  
  
  let newItems = [];
```

```

    items.forEach((item, idx) => {
      if (item.id !== id)
        newItems.push(item)
    })

    setItems(newItems);
  }

```

Next, create *getTotal()* method to calculate the total amount.

```

function getTotal() {
  let total = 0;
  for (var i = 0; i < items.length; i++) {
    total += items[i].amount
  }
  return total;
}

```

Next, create user interface to show the expenses by looping over the *items*.

```

const lists = items.map((item) =>
  <tr key={item.id} onMouseEnter={handleMouseEnter}
onMouseLeave={handleMouseLeave}>
    <td>{item.name}</td>
    <td>{item.amount}</td>
    <td>{new Date(item.spendDate).toLocaleDateString()}</td>
    <td>{item.category}</td>
    <td><a href="#"
      onClick={(e) => handleDelete(item.id, e)}>Remove</a></td>
  </tr>
);

```

Next, create the complete UI to show the expenses and return it.

```

return (
  <table onMouseOver={handleMouseOver}>
    <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" }}>
          {getTotal()}
        </td>
      </tr>
    </tbody>
  </table>
)

```



```

        </tbody>
    </table>
);

```

Finally, export the function as shown below:

```
export default ExpenseEntryItemListFn;
```

The complete code of the *ExpenseEntryItemListFn* is as follows:

```

import React, { useState } from 'react';
import './ExpenseEntryItemList.css'

function ExpenseEntryItemListFn(props) {
    const [items, setItems] = useState(props.items);

    function handleMouseEnter(e) {
        e.target.parentNode.classList.add("highlight");
    }

    function handleMouseLeave(e) {
        e.target.parentNode.classList.remove("highlight");
    }

    function handleMouseOver(e) {
        console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
    }

    function handleDelete(id, e) {
        e.preventDefault();
        console.log(id);

        let newItems = [];

        items.forEach((item, idx) => {
            if (item.id !== id)
                newItems.push(item)
        })

        setItems(newItems);
    }

    function getTotal() {
        let total = 0;
        for (var i = 0; i < items.length; i++) {
            total += items[i].amount
        }
        return total;
    }

    const lists = items.map((item) =>
        <tr key={item.id} onMouseEnter={handleMouseEnter}
onMouseLeave={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) => handleDelete(item.id, e)}>Remove</a></td>
    </tr>
);

return (
    <table onMouseOver={handleMouseOver}>
        <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" }}>
                    {getTotal()}
                </td>
            </tr>
        </tbody>
    </table>
);
}

export default ExpenseEntryItemListFn;

```

Next, update the *index.js* and include the *ExpenseEntryItemListFn* component:

```

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

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" }
  ]

  ReactDOM.render(
    <React.StrictMode>
      <ExpenseEntryItemListFn items={items} />
    </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.

Finally, to remove an expense item, click the corresponding remove link. It will remove the corresponding item and refresh the user interface as shown in animated gif.

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			

Component Life cycle

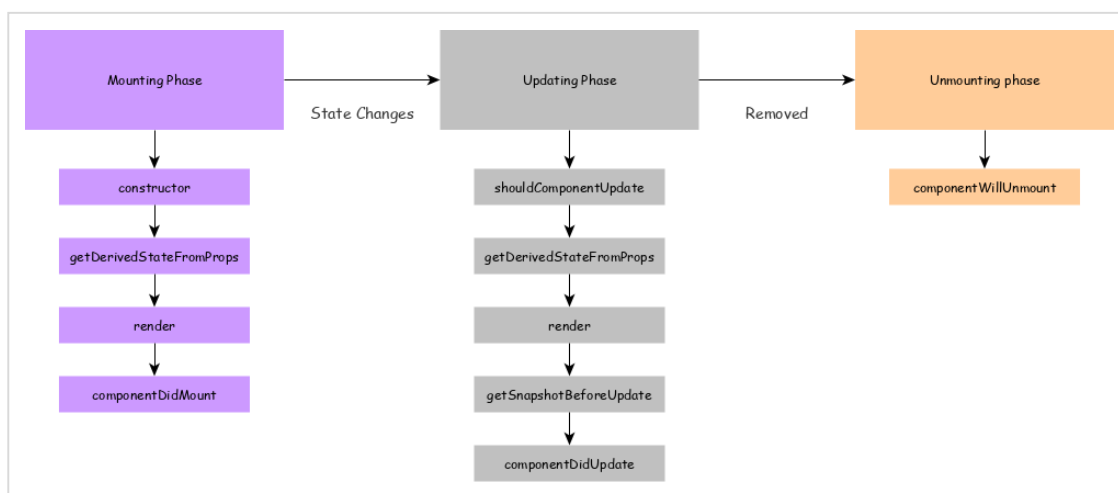
In React, Life cycle of a component represents the different stages of the component during its existence. React provides callback function to attach functionality in each and every stages of the React life cycle. Let us learn the life cycle (and the related API) of a React component in this chapter.

Life cycle API

Each React component has three distinct stages.

- **Mounting** - Mounting represents the rendering of the React component in the given DOM node.
- **Updating** - Updating represents the re-rendering of the React component in the given DOM node during state changes / updates.
- **Unmounting** - Unmounting represents the removal of the React component.

React provides a collection of life cycle events (or callback API) to attach functionality, which will to be executed during the various stages of the component. The visualization of life cycle and the sequence in which the life cycle events (APIs) are invoked as shown below.



constructor() - Invoked during the initial construction phase of the React component. Used to set initial state and properties of the component.

render() - Invoked after the construction of the component is completed. It renders the component in the virtual DOM instance. This is specified as mounting of the component in the DOM tree.

componentDidMount() - Invoked after the initial mounting of the component in the DOM tree. It is the good place to call API endpoints and to do network requests. In our clock component, *setInterval* function can be set here to update the state (current date and time) for every second.

```
componentDidMount() {
  this.timeFn = setInterval( () => this.setTime(), 1000);
}
```

componentDidUpdate() - Similar to *ComponentDidMount()* but invoked during the update phase. Network request can be done during this phase but only when there is difference in component's current and previous properties.

The signature of the API is as follows:

```
componentDidUpdate(prevProps, prevState, snapshot)
```

- **prevProps** - Previous properties of the component.
- **prevState** - Previous state of the component.
- **snapshot** - Current rendered content.

componentWillUnmount() - Invoked after the component is unmounted from the DOM. This is the good place to clean up the object. In our clock example, we can stop updating the date and time in this phase.

```
componentDidMount() {
  this.timeFn = setInterval( () => this.setTime(), 1000);
}
```

shouldComponentUpdate() - Invoked during the update phase. Used to specify whether the component should update or not. If it returns false, then the update will not happen.

The signature is as follows:

```
shouldComponentUpdate(nextProps, nextState)
```

- **nextProps** - Upcoming properties of the component
- **nextState** - Upcoming state of the component

getDerivedStateFromProps - Invoked during both initial and update phase and just before the **render()** method. It returns the new state object. It is rarely used where the changes in properties results in state change. It is mostly used in animation context where the various state of the component is needed to do smooth animation.

The signature of the API is as follows:

```
static getDerivedStateFromProps(props, state)
```

- **props** - current properties of the component
- **state** - Current state of the component

This is a static method and does not have access to **this** object.

getSnapshotBeforeUpdate - Invoked just before the rendered content is committed to DOM tree. It is mainly used to get some information about the new content. The data returned by this method will be passed to **ComponentDidUpdate()** method. For example, it is used to maintain the user's scroll position in the newly generated content. It returns user's scroll position. This scroll position is used by **componentDidUpdate()** to set the scroll position of the output in the actual DOM.

The signature of the API is as follows:

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

- **prevProps** - Previous properties of the component.
- **prevState** - Previous state of the component.

Working example of life cycle API

Let us use life cycle api in our *react-clock-app* application.

Open *react-clock-hook-app* in your favorite editor.

Next, open *src/components/Clock.js* file and start editing.

Next, remove the *setInterval()* method from the constructor.

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

  this.state = {
    date: new Date()
  }
}
```

Next, add *componentDidMount()* method and call *setInterval()* to update the date and time every second. Also, store the reference to stop updating the date and time later.

```
componentDidMount() {
  this.setTimeRef = setInterval(() => this.setTime(), 1000);
}
```

Next, add *componentWillUnmount()* method and call *clearInterval()* to stop the date and time update calls.

```
componentWillUnmount() {
  clearInterval(this.setTimeRef)
}
```

Now, we have updated the Clock component and the complete source code of the component is given below:

```
import React from 'react';

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

    this.state = {
      date: new Date()
    }
  }

  componentDidMount() {
    this.setTimeRef = setInterval(() => this.setTime(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.setTimeRef)
  }
}
```

```

    setTime() {
      this.setState((state, props) => {
        console.log(state.date);

        return {
          date: new Date()
        }
      })
    }

    render() {
      return (
        <div><p>The current time is {this.state.date.toString()}</p></div>
      );
    }
  }

export default Clock;

```

Next, open `index.js` and use `setTimeout` to remove the clock from the DOM after 5 seconds.

```

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

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

setTimeout(() => {
  ReactDOM.render(
    <React.StrictMode>
      <div><p>Clock is removed from the DOM.</p></div>
    </React.StrictMode>,
    document.getElementById('root')
  );
}, 5000);

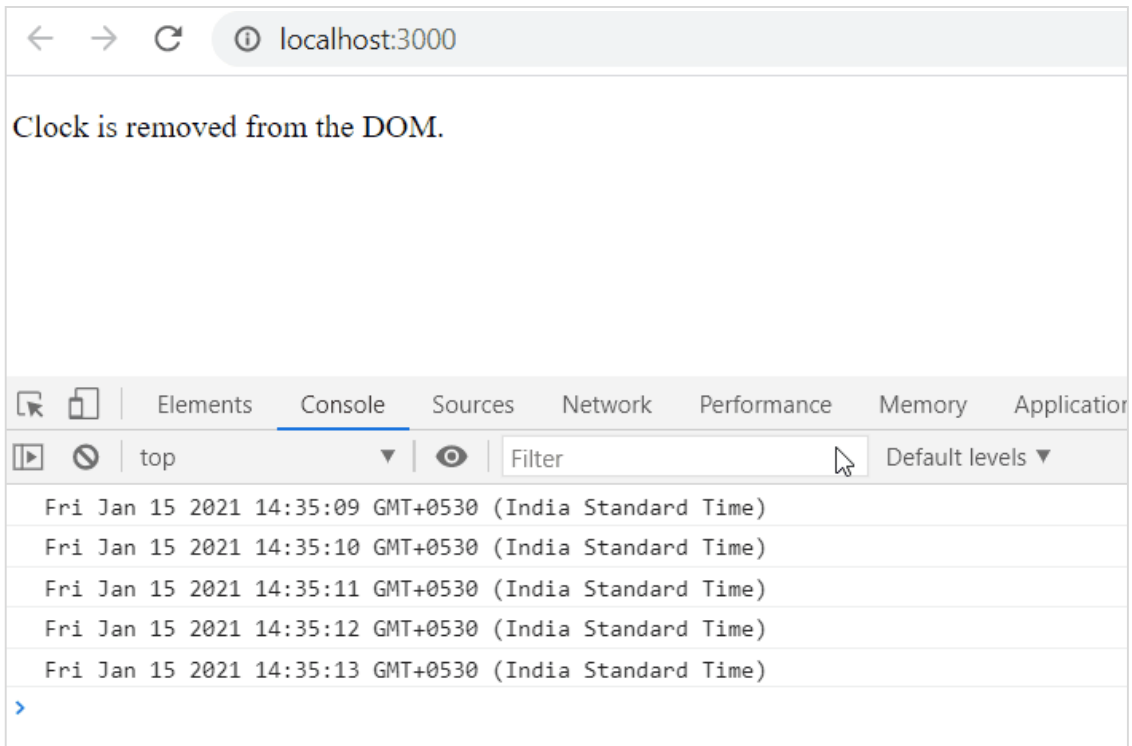
```

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 clock will be shown for 5 second and then, it will be removed from the DOM. By checking the console log, we can found that the cleanup code is properly executed.



Life cycle api in Expense manager app

Let us add life cycle api in the expense manager and log it whenever the api is called. This will give insight about the life cycle of the component.

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

Next, update *ExpenseEntryItemList* component with below methods.

```
componentDidMount() {
  console.log("ExpenseEntryItemList :: Initialize :: componentDidMount :: Component mounted");
}

shouldComponentUpdate(nextProps, nextState) {
  console.log("ExpenseEntryItemList :: Update :: shouldComponentUpdate invoked :: Before update");

  return true;
}

static getDerivedStateFromProps(props, state) {
  console.log("ExpenseEntryItemList :: Initialize / Update :: getDerivedStateFromProps :: Before update");

  return null;
}

getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log("ExpenseEntryItemList :: Update :: getSnapshotBeforeUpdate :: Before update");
}
```



```

    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log("ExpenseEntryItemList :: Update :: componentDidUpdate :: Component updated");
  }

  componentWillUnmount() {
    console.log("ExpenseEntryItemList :: Remove :: componentWillUnmount :: Component unmounted");
  }
}

```

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.

Next, check the console log. It will show the life cycle api during initialization phase as shown below.

```

ExpenseEntryItemList :: Initialize / Update :: getDerivedStateFromProps :: Before update
ExpenseEntryItemList :: Initialize :: componentDidMount :: Component mounted

```

Next, remove an item and then, check the console log. It will show the life cycle api during the update phase as shown below.

```

ExpenseEntryItemList :: Initialize / Update :: getDerivedStateFromProps :: Before update
ExpenseEntryItemList.js:109 ExpenseEntryItemList :: Update :: shouldComponentUpdate invoked :: Before update
ExpenseEntryItemList.js:121 ExpenseEntryItemList :: Update :: getSnapshotBeforeUpdate :: Before update
ExpenseEntryItemList.js:127 ExpenseEntryItemList :: Update :: componentDidUpdate :: Component updated

```

Finally, remove all the life cycle api as it may hinder the application performance. Life cycle api should be used only if the situation demands.

Component life cycle using React Hooks

React Hooks provides a special Hook, `useEffect()` to execute certain functionality during the life cycle of the component. `useEffect()` combines `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` life cycle into a single api.

The signature of the `useEffect()` api is as follows:

```

useEffect(
  <executeFn>,

```

```
    <values>
  );
```

Here,

- **executeFn** - Function to execute when an effect occurs with an optional return function. The return function will be execute when a clean up is required (similar to `componentWillUnmount`).
- **values** - array of values the effect depends on. React Hooks execute the `executeFn` only when the values are changed. This will reduce unnecessary calling of the `executeFn`.

Let us add `useEffect()` Hooks in our `react-clock-hook-app` application.

Open `react-clock-hook-app` in your favorite editor.

Next, open `src/components/Clock.js` file and start editing.

Next, import `useEffect` api.

```
import React, { useState, useEffect } from 'react';
```

Next, call `useEffect` with function to set date and time every second using `setInterval` and return a function to stop updating the date and time using `clearInterval`.

```
useEffect(
  () => {
    let setTime = () => {
      console.log("setTime is called");
      setCurrentDateTime(new Date());
    }

    let interval = setInterval(setTime, 1000);

    return () => {
      clearInterval(interval);
    }
  },
  []
);
```

Here,

- Created a function, `setTime` to set the current time into the state of the component.
- Called the `setInterval` JavaScript api to execute `setTime` every second and stored the reference of the `setInterval` in the `interval` variable.
- Created a return function, which calls the `clearInterval` api to stop executing `setTime` every second by passing the `interval` reference.

Now, we have updated the Clock component and the complete source code of the component is as follows:

```

import React, { useState, useEffect } from 'react';

function Clock() {
  const [currentDateTime, setCurrentDateTime] = useState(new Date());

  useEffect(
    () => {
      let setTime = () => {
        console.log("setTime is called");
        setCurrentDateTime(new Date());
      }

      let interval = setInterval(setTime, 1000);

      return () => {
        clearInterval(interval);
      }
    },
    []
  );

  return (
    <div><p>The current time is {currentDateTime.toString()}</p></div>
  );
}

export default Clock;

```

Next, open `index.js` and use `setTimeout` to remove the clock from the DOM after 5 seconds.

```

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

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

setTimeout(() => {
  ReactDOM.render(
    <React.StrictMode>
      <div><p>Clock is removed from the DOM.</p></div>
    </React.StrictMode>,
    document.getElementById('root')
  );
}, 5000);

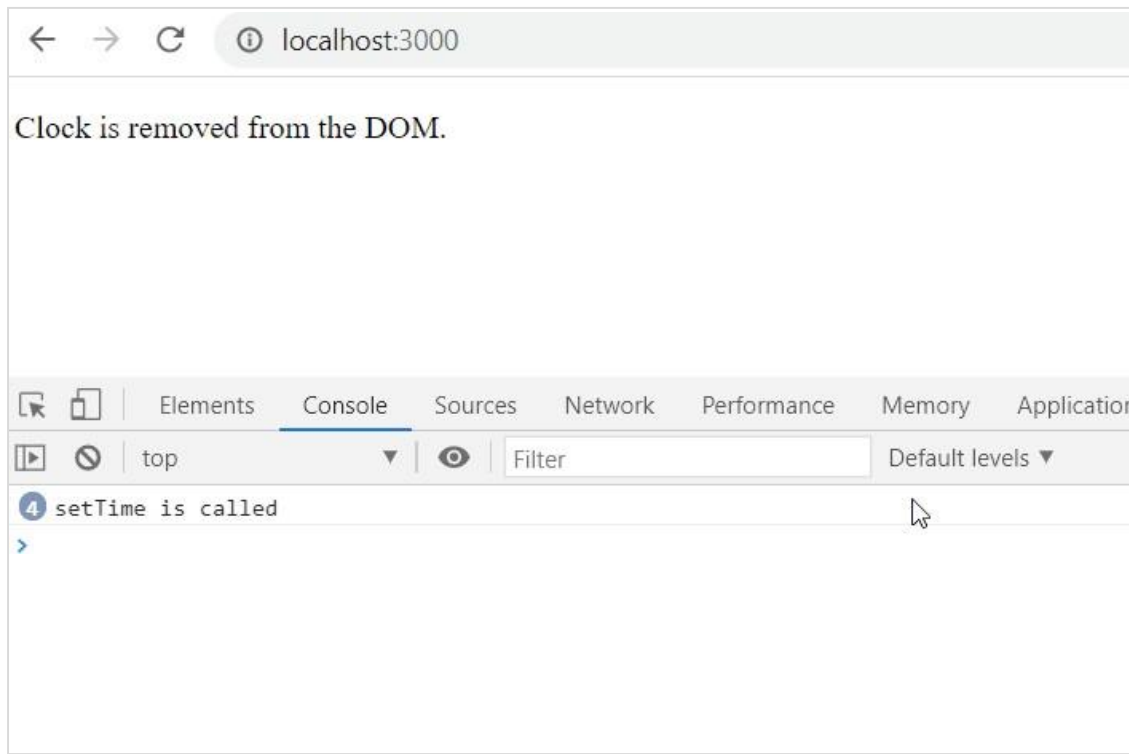
```

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 clock will be shown for 5 seconds and then, it will be removed from the DOM. By checking the console log, we can find that the cleanup code is properly executed.



React children property aka Containment

React allows arbitrary children user interface content to be included inside the component. The children of a component can be accessed through `this.props.children`. Adding children inside the component is called *containment*. *Containment* is used in situation where certain section of the component is dynamic in nature.

For example, a rich text message box may not know its content until it is called. Let us create *RichTextMessage* component to showcase the feature of React children property in this chapter.

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, 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, *RichTextMessage.js* under *src/components* folder and start editing.

Next, import *React* library.

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

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

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

Next, add *render()* method and show the user interface of the component along with it's children

```
render() {
  return (
    <div>{this.props.children}</div>
  )
}
```

Here,

- *props.children* returns the children of the component.
- Wraped the children inside a *div* tag.

Finally, export the component.

```
export default RichTextMessage;
```

The complete source code of the *RichTextMessage* component is given below:

```
import React from 'react';

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

  render() {
    return (
      <div>{this.props.children}</div>
    )
  }
}

export default RichTextMessage;
```

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

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

ReactDOM.render(
  <React.StrictMode>
    <RichTextMessage>
      <h1>Containment is really a cool feature.</h1>
    </RichTextMessage>
  </React.StrictMode>
, document.getElementById('root'));
```

```

    </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, serve the application using npm command.

```
npm start
```

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

Containment is really a cool feature.

Browser emits component's children wrapped in *div* tag as shown below:

```

<div id="root">
  <div>
    <div>
      <h1>Containment is really a cool feature.</h1>
    </div>
  </div>
</div>

```

Next, change the child property of *RichTextMessage* component in *index.js*.

```

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

ReactDOM.render(
  <React.StrictMode>
    <RichTextMessage>
      <h1>Containment is really an excellent feature.</h1>
    </RichTextMessage>
  </React.StrictMode>,

```

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

Now, browser updates the component's children content and emits as shown below:

```
<div id="root">
  <div>
    <div>
      <h1>Containment is really an excellent feature.</h1>
    </div>
  </div>
</div>
```

In short, containment is an excellent feature to pass arbitrary user interface content to the component.

Layout in component

One of the advanced features of React is that it allows arbitrary user interface (UI) content to be passed into the component using properties. As compared to React's special *children* property, which allows only a single user interface content to be passed into the component, this option enables multiple UI content to be passed into the component. This option can be seen as an extension of *children* property. One of the use cases of this option is to layout the user interface of a component.

For example, a component with customizable header and footer can use this option to get the custom header and footer through properties and layout the content.

A quick and simple example with two properties, *header* and *footer* is given below:

```
<Layout header={<h1>Header</h1>} footer={<p>footer</p>} />
```

And the layout render logic is as follows:

```
return (<div>
  <div>
    {props.header}
  </div>
  <div>
    Component user interface
  </div>
  <div>
    {props.footer}
  </div>
</div>)
```

Let us add a simple header and footer to our expense entry list (*ExpenseEntryItemList*) component.

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

Next, open the file, *ExpenseEntryItemList.js* in *src/components* folder.

Next, use *header* and *footer* props in the *render()* method.

```

return (
  <div>
    <div>{this.props.header}</div>
    ... existing code ...
    <div>{this.props.footer}</div>
  </div>
);

```

Next, open *index.js* and include *header* and *footer* property while using the *ExpenseEntryItemList* component.

```

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItemList items={items}
      header={
        <div><h1>Expense manager</h1></div>
      }

      footer={
        <div style={{ textAlign: "left" }}><p style={{ fontSize: 12 }}>Sample
application</p></div>
      } />
    </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.

Expense manager

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			

Sample application

Sharing logic in component aka Render props

Render props is an advanced concept used to share logic between React components. As we learned earlier, a component can receive arbitrary UI content or React elements (objects) through properties. Usually, the component render the React elements it receives as is along with its own user interface as we have seen in children and layout concept. They do not share any logic between them.

Going one step further, React allows a component to take a function which returns user interface instead of plain user interface object through properties. The sole purpose of the function is to render the UI. Then, the component will do advanced computation and will call the passed in function along with computed value to render the UI.

In short, component's property, which accepts a JavaScript function that renders user interface is called *Render Props*. Component receiving *Render props* will do advanced logic and share it with *Render props*, which will render the user interface using the shared logic.

Lot of advanced third-party library are based on *Render Props*. Some of the libraries using *Render Props* are:

- React Router
- Formik
- Downshift

For example, **Formik** library component will do the form validation and submission and pass the form design to the calling function aka *Render Props*. Similarly, *React Router* do the routing logic while delegating the UI design to other components using *Render Props*.

Pagination

Pagination is one of the distinct features requiring special logic and at the same time, the UI rendering may vary drastically. For example, the pagination can be done for a tabular content as well as a gallery content. Pagination component will do the pagination logic and delegates the rendering logic to other component. Let us try to create a pagination component (Pager) for our Expense manager application using *Render props*

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

Next, create a file, *Pager.css* under *src/components* to add styles for pagination component.

```
a{
  text-decoration: none;
}

p, li, a{
  font-size: 12px;
}

.container{
  width: 720px;
  max-width: 340px;
  margin: 0 auto;
  position: relative;
  text-align: center;
}

.pagination{
  padding: 14px 0;
}

.pagination ul{
  margin: 0 auto;
  padding: 0;
  list-style-type: none;
  text-align: left;
}

.pagination a{
  display: inline-block;
  padding: 10px 18px;
  color: #222;
}

.p1 a{
  width: 30px;
  height: 30px;
  line-height: 30px;
}
```

```
padding: 0;
text-align: center;
}

.p1 a.is-active{
  background-color: #887cb8;
  border-radius: 25%;
  color: #fff;
}
```

Next, create a file, *Pager.js* in *src/components* folder to create *Pager* component and start editing.

Next, import *React* library.

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

Next, import pagination stylesheet.

```
import './Pager.css'
```

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

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

Next, initialize the state of the component with expense items (items) and number of items to show in a page (pageCount).

```
this.state = {
  items: this.props.items,
  pageCount: this.props.pageCount
}
```

Write a function, *calculate()*, which accepts current state and the current page number. It calculates the total of available pages based on the page count and the items to show in current page (filteredItems). Finally, it returns the computed values.

```
calculate(state, pageNo) {
  let currentPage = pageNo;

  let totalPages = Math.ceil(state.items.length / state.pageCount);

  if(currentPage > totalPages)
    currentPage = totalPages;

  let hasPreviousPage = currentPage == 1 ? false : true;
  let hasNextPage = currentPage == totalPages ? false : true;
  let first = (currentPage - 1) * state.pageCount
  let last = first + state.pageCount;
  let filteredItems = state.items.slice(first, last)
```

```

    let newState = {
      items: state.items,
      filteredItems: filteredItems,
      currentPage: currentPage,
      totalPages: totalPages,
      pageCount: state.pageCount
    }

    return newState;
  }

```

Next, call the *calculate()* method in the constructor to get the initial details about the pagination.

```
this.state = this.calculate(this.state, 1);
```

Next, write an event handler method, *handleClick* to handle the page navigation event of the pagination component.

```

handleClick(pageNo, e) {
  e.preventDefault();

  this.setState((state, props) => {
    return this.calculate(state, pageNo);
  })
}

```

Next, write an event handler to remove the expense item. Since, the pager component handles all the expense items, the remove feature should be moved here.

```

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

  this.setState((state, props) => {
    let items = [];

    state.items.forEach((item, idx) => {
      if (item.id !== id)
        items.push(item)
    })

    let newState = {
      items: items
    }
    return newState;
  })

  this.setState((state, props) => {
    return this.calculate(state, this.state.currentPage);
  })
}

```

Next, create a render method to create the pagination user interface and then call the *Render Props* by passing the items needed to render for the current page.

```
render() {
  let pageArray = new Array();
  let i = 1;
  for (i = 1; i <= this.state.totalPages; i++)
    pageArray.push(i)

  const pages = pageArray.map((idx) =>
    <a href="#" key={idx} onClick={this.handleClick.bind(this, idx)}
    className={idx == this.state.currentPage ? "is-active" : ""}><li>{idx}</li></a>
  );

  let propsToPass = {
    items: this.state.filteredItems,
    deleteHandler: this.handleDelete
  }

  return (
    <div>
      {this.props.render(propsToPass)}

      <div style={{ width: 720, margin: 0 }}>
        <div className="container">
          <div className="pagination p1">
            <ul>
              {this.state.currentPage != 1 ? <a href="#"
onClick={this.handleClick.bind(this, this.state.currentPage -
1)}><li>< </li></a> : <span>&nbsp;</span>}
              {pages}
              {this.state.currentPage != this.state.totalPages ?
                <a href="#" onClick={this.handleClick.bind(this,
this.state.currentPage + 1)}><li>> </li></a> : <span>&nbsp;</span>}
            </ul>
          </div>
        </div>
      </div>
    </div>
  )
}
```

Here,

- Used *map* to create the pagination buttons.
- Used conditional rendering to show/hide first and last page.
- Used *this.props.render* to call and render the passed in component.

Finally, Export the component.

```
export default Pager
```

The complete code of the *Pager* component is as follows:

```

import React from 'react';
import './Pager.css'

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

    this.state = {
      items: this.props.items,
      pageCount: this.props.pageCount,
    }

    this.state = this.calculate(this.state, 1);
  }

  calculate(state, pageNo) {
    let currentPage = pageNo;

    let totalPages = Math.ceil(state.items.length / state.pageCount);

    if(currentPage > totalPages)
      currentPage = totalPages;

    let hasPreviousPage = currentPage == 1 ? false : true;
    let hasNextPage = currentPage == totalPages ? false : true;
    let first = (currentPage - 1) * state.pageCount
    let last = first + state.pageCount;
    let filteredItems = state.items.slice(first, last)

    let newState = {
      items: state.items,
      filteredItems: filteredItems,
      currentPage: currentPage,
      totalPages: totalPages,
      pageCount: state.pageCount
    }

    return newState;
  }

  handleClick(pageNo, e) {
    e.preventDefault();

    this.setState((state, props) => {
      return this.calculate(state, pageNo);
    })
  }

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

    this.setState((state, props) => {
      let items = [];

```

```

        state.items.forEach((item, idx) => {
            if (item.id !== id)
                items.push(item)
        })

        let newState = {
            items: items
        }
        return newState;
    })

    this.setState((state, props) => {
        return this.calculate(state, this.state.currentPage);
    })
}

render() {
    let pageArray = new Array();
    let i = 1;
    for (i = 1; i <= this.state.totalPages; i++)
        pageArray.push(i)

    const pages = pageArray.map((idx) =>
        <a href="#" key={idx} onClick={this.handleClick.bind(this, idx)}
className={idx == this.state.currentPage ? "is-active" : ""}><li>{idx}</li></a>
    );

    let propsToPass = {
        items: this.state.filteredItems,
        deleteHandler: this.handleDelete
    }

    return (
        <div>
            {this.props.render(propsToPass)}

            <div style={{ width: 720, margin: 0 }}>
                <div className="container">
                    <div className="pagination p1">
                        <ul>
                            {this.state.currentPage !== 1 ? <a href="#"
onClick={this.handleClick.bind(this, this.state.currentPage -
1)}><li>&lt;</li></a> : <span>&nbsp;</span>}
                            {pages}
                            {this.state.currentPage !==
this.state.totalPages ?
                                <a href="#"
onClick={this.handleClick.bind(this, this.state.currentPage +
1)}><li>&gt;>/li></a> : <span>&nbsp;</span>}
                        </ul>
                    </div>
                </div>
            </div>
        </div>
    )
}

```

```

        </div>
    )
}
}

export default Pager

```

Next, open *ExpenseEntryItemList.js* file and update the *getDerivedStateFromProps* function to get the current expense item list by copying the passed-in props into the new state.

```

static getDerivedStateFromProps(props, state) {
    let newState = {
        items: props.items
    }
    return newState;
}

```

Next, update the *handleDelete* method and call the event handler method passed into the component from from pager component through *onDelete* property.

```

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

    if(this.props.onDelete !== null)
        this.props.onDelete(id, e);
}

```

Next, remove rendering of header and footer, if any available in the *render* method.

```

return (
    <div>
        <div>{this.props.header}</div>
        ... existing code ...
        <div>{this.props.footer}</div>
    </div>
);

```

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

```

import React from 'react';

import './ExpenseEntryItemList.css';

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

        this.state = {
            items: this.props.items
        }

        this.handleMouseEnter = this.handleMouseEnter.bind();
    }
}

```



```

    this.handleMouseLeave = this.handleMouseLeave.bind();
    this.handleMouseOver = this.handleMouseOver.bind();
  }

  handleMouseEnter(e) {
    e.target.parentNode.classList.add("highlight");
  }

  handleMouseLeave(e) {
    e.target.parentNode.classList.remove("highlight");
  }

  handleMouseOver(e) {
    console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
  }

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

    if(this.props.onDelete != null)
      this.props.onDelete(id, e);

    /*console.log(id);

    this.setState((state, props) => {
      let items = [];

      state.items.forEach((item, idx) => {
        if (item.id != id)
          items.push(item)
      })

      let newState = {
        items: items
      }
      return newState;
    })*//
  }

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

  static getDerivedStateFromProps(props, state) {
    let newState = {
      items: props.items
    }
    return newState;
  }
}

```

```

    render() {
      const 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}
              <tr>
                <td colspan="1" style={{ textAlign: "right"
}}>Total Amount</td>
                <td colspan="4" style={{ textAlign: "left" }}>
                  {this.getTotal()}
                </td>
              </tr>
            </tbody>
          </table>
        </div>
      );
    }
  }

  export default ExpenseEntryItemList;

```

Next, open *index.js* and call the *Pager* component and pass the *ExpenseEntryItemList* as render props.

```

import React from 'react';
import ReactDOM from 'react-dom';

import Pager from './components/Pager'
import ExpenseEntryItemList from './components/ExpenseEntryItemList'

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" }
  ]

  const pageCount = 3;

  ReactDOM.render(
    <React.StrictMode>
      <Pager
        items={items}
        pageCount={pageCount}
        render={
          pagerState => (
            <div>
              <ExpenseEntryItemList items={pagerState.items}
                onDelete={pagerState.deleteHandler} />
            </div>
          )
        }
      />
    </React.StrictMode>,
    document.getElementById('root')
  );

```

As we see, the *Pager* component accept any render props and only calculate the pagination logic.

Next, serve the application using npm command.

```
npm start
```

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

Item	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
Total Amount	320			

1
2
3
4
>

Material UI

React community provides a huge collection of advanced UI component framework. Material UI is one of the popular React UI frameworks. Let us learn how to use material UI library in this chapter.

Installation

Material UI can be installed using npm package.

```
npm install @material-ui/core
```

Material UI recommends roboto fonts for UI. To use Roboto font, include it using Googleapis links.

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />
```

To use font icons, use icon link from googleapis:

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/icon?family=Material+Icons" />
```

To use SVG icons, install **@material-ui/icons** package:

```
npm install @material-ui/icons
```

Working example

Let us recreate the expense list application and use material ui components instead of html tables.

First, create a new react application, *react-materialui-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 @material-ui/core @material-ui/icons --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, *ExpenseEntryItemList.js* in *src/components* folder to create *ExpenseEntryItemList* component

Next, import *React* library and the stylesheet.

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

Next, import *Material-UI* library.

```
import { withStyles } from '@material-ui/core/styles';
import Table from '@material-ui/core/Table';
import TableBody from '@material-ui/core/TableBody';
import TableCell from '@material-ui/core/TableCell';
import TableContainer from '@material-ui/core/TableContainer';
import TableHead from '@material-ui/core/TableHead';
import TableRow from '@material-ui/core/TableRow';
import Paper from '@material-ui/core/Paper';
```

Next, create *ExpenseEntryItemList* class and call constructor function.

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

Next, create a *render* function.

```
render() {
}
```

Next, apply styles for table rows and table cells in the *render* method.

```
const StyledTableCell = withStyles((theme) => ({
  head: {
    backgroundColor: theme.palette.common.black,
    color: theme.palette.common.white,
  },
  body: {
    fontSize: 14,
  },
}))(TableCell);

const StyledTableRow = withStyles((theme) => ({
  root: {
```

```

    '&:nth-of-type(odd)': {
      backgroundColor: theme.palette.action.hover,
    },
  },
}))(TableRow);

```

Next, Use map method to generate a collection of Material UI `StyledTableRow` each representing a single expense entry item in the list.

```

const lists = this.props.items.map((item) =>
  <StyledTableRow key={item.id}>
    <StyledTableCell component="th" scope="row">
      {item.name}
    </StyledTableCell>
    <StyledTableCell align="right">{item.amount}</StyledTableCell>
    <StyledTableCell align="right">
      {new Date(item.spendDate).toDateString()}
    </StyledTableCell>
    <StyledTableCell align="right">{item.category}</StyledTableCell>
  </StyledTableRow>
);

```

Here, *key* identifies each row and it has to be unique among the list.

Next, in the *render()* method, create a Material UI table and include the *lists* expression in the rows section and return it.

```

return (
  <TableContainer component={Paper}>
    <Table aria-label="customized table">
      <TableHead>
        <TableRow>
          <StyledTableCell>Title</StyledTableCell>
          <StyledTableCell align="right">Amount</StyledTableCell>
          <StyledTableCell align="right">Spend date</StyledTableCell>
          <StyledTableCell align="right">Category</StyledTableCell>
        </TableRow>
      </TableHead>
      <TableBody>
        {lists}
      </TableBody>
    </Table>
  </TableContainer> );

```

Finally, export the component.

```

export default ExpenseEntryItemList;

```

Now, we have successfully created the component to render the expense items using material ui components.

The complete source code of the component is as given below:

```

import React from 'react';

import { withStyles } from '@material-ui/core/styles';
import Table from '@material-ui/core/Table';
import TableBody from '@material-ui/core/TableBody';
import TableCell from '@material-ui/core/TableCell';
import TableContainer from '@material-ui/core/TableContainer';
import TableHead from '@material-ui/core/TableHead';
import TableRow from '@material-ui/core/TableRow';
import Paper from '@material-ui/core/Paper';

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

  render() {
    const StyledTableCell = withStyles((theme) => ({
      head: {
        backgroundColor: theme.palette.common.black,
        color: theme.palette.common.white,
      },
      body: {
        fontSize: 14,
      },
    }))(TableCell);

    const StyledTableRow = withStyles((theme) => ({
      root: {
        '&:nth-of-type(odd)': {
          backgroundColor: theme.palette.action.hover,
        },
      },
    }))(TableRow);

    const lists = this.props.items.map((item) =>
      <StyledTableRow key={item.id}>
        <StyledTableCell component="th" scope="row">
          {item.name}
        </StyledTableCell>
        <StyledTableCell align="right">{item.amount}</StyledTableCell>
        <StyledTableCell align="right">new
Date(item.spendDate).toDateString())</StyledTableCell>
        <StyledTableCell
align="right">{item.category}</StyledTableCell>
      </StyledTableRow>
    );

    return (
      <TableContainer component={Paper}>
        <Table aria-label="customized table">
          <TableHead>
            <TableRow>
              <StyledTableCell>Title</StyledTableCell>

```

```

                <StyledTableCell align="right">Amount</StyledTableCell>
                <StyledTableCell align="right">Spend
date</StyledTableCell>
                <StyledTableCell
align="right">Category</StyledTableCell>
            </TableRow>
        </TableHead>
        <TableBody>
            {lists}
        </TableBody>
    </Table>
</TableContainer> );
    }
}

export default ExpenseEntryItemList;

```

Next, open *index.js* and import react library and our newly created *ExpenseEntryItemList* component.

```

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

```

Next, declare a list (of expense entry item) and populate it with some random values in *index.js* file.

```

const items = [
  { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category:
"Food" },
  { id: 1, name: "Grape Juice", amount: 30, spendDate: "2020-10-12",
category: "Food" },
  { id: 1, name: "Cinema", amount: 210, spendDate: "2020-10-16", category:
"Entertainment" },
  { id: 1, name: "Java Programming book", amount: 242, spendDate: "2020-10-
15", category: "Academic" },
  { id: 1, name: "Mango Juice", amount: 35, spendDate: "2020-10-16",
category: "Food" },
  { id: 1, name: "Dress", amount: 2000, spendDate: "2020-10-25", category:
"Cloth" },
  { id: 1, name: "Tour", amount: 2555, spendDate: "2020-10-29", category:
"Entertainment" },
  { id: 1, name: "Meals", amount: 300, spendDate: "2020-10-30", category:
"Food" },
  { id: 1, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category:
"Gadgets" },
  { id: 1, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04",
category: "Academic" }
]

```

Next, use *ExpenseEntryItemList* component by passing the *items* through *items* attributes.

```

ReactDOM.render(
  <React.StrictMode>

```



```

    <ExpenseEntryItemList items={items} />
  </React.StrictMode>,
  document.getElementById('root')
);

```

The complete code of *index.js* is as follows:

```

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

const items = [
  { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food" },
  { id: 1, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category: "Food" },
  { id: 1, name: "Cinema", amount: 210, spendDate: "2020-10-16", category: "Entertainment" },
  { id: 1, name: "Java Programming book", amount: 242, spendDate: "2020-10-15", category: "Academic" },
  { id: 1, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category: "Food" },
  { id: 1, name: "Dress", amount: 2000, spendDate: "2020-10-25", category: "Cloth" },
  { id: 1, name: "Tour", amount: 2555, spendDate: "2020-10-29", category: "Entertainment" },
  { id: 1, name: "Meals", amount: 300, spendDate: "2020-10-30", category: "Food" },
  { id: 1, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category: "Gadgets" },
  { id: 1, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category: "Academic" }
]

ReactDOM.render(
  <React.StrictMode>
    <ExpenseEntryItemList items={items} />
  </React.StrictMode>,
  document.getElementById('root')
);

```

Next, serve the application using npm command.

```
npm start
```

Next, open *index.html* file in the public folder and include the material UI font and icons.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Material UI App</title>
    <link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=sw

```

```

ap" />
  <link rel="stylesheet"
href="https://fonts.googleapis.com/icon?family=Material+Icons" />
</head>
<body>
  <div id="root"></div>
  <script type="text/JavaScript" src="./index.js"></script>
</body>
</html>

```

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

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

11. React — 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 RestApi 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": "5H8rK81LGJPVZ3gD"
  },
  ...
]

```

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 = {
  isLoading: 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({
    isLoading: 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({
          isLoading: 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.isLoading) {
    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

12. React — 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

Controlled component has to follow a specific process to do form programming. Let us check the step by step process to be followed for a single input element.

Create a form element.

```
<input type="text" name="username" />
```

Create a state for input element.

```
this.state = {  
  username: ''  
}
```

Add a value attribute and assign the value from state.

```
<input type="text" name="username" value={this.state.username} />
```

Add a *onChange* attribute and assign a handler method.

```
<input type="text" name="username" value={this.state.username}  
  onChange={this.handleUsernameChange} />
```

Write the handler method and update the state whenever the event is fired.

```
handleUsernameChange(e) {  
  this.setState({  
    username = e.target.value
```

```
});
}
```

Bind the event handler in the constructor of the component.

```
this.handleChange = this.handleChange.bind(this)
```

Finally, get the input value using **username** from **this.state** during validation and submission.

```
handleSubmit(e) {
  e.preventDefault();

  alert(this.state.username);
}
```

Let us create a simple form to add expense entry using controller component in this chapter.

First, create a new react application, *react-form-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.

In the next step, create *src* folder under the root directory of the application.

Further to the above process, create *components* folder under *src* folder.

Next, create a file, *ExpenseForm.css* under *src* folder to style the component.

```
input[type=text], input[type=number], input[type=date], select {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 4px;
  box-sizing: border-box;
}

input[type=submit] {
  width: 100%;
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

input[type=submit]:hover {
  background-color: #45a049;
}
```

```
input:focus {
border: 1px solid #d9d5e0;
}

#expenseForm div {
border-radius: 5px;
background-color: #f2f2f2;
padding: 20px;
}
```

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

Next, import *React* library.

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

Next, import *ExpenseForm.css* file.

```
import './ExpenseForm.css'
```

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

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

Next, initialize the state of the component.

```
this.state = {
  item: {}
}
```

Next, create *render()* method and add a form with input fields to add expense items.

```
render() {
  return (
    <div id="expenseForm">
      <form>
        <label for="name">Title</label>
        <input type="text" id="name" name="name" placeholder="Enter expense title" />

        <label for="amount">Amount</label>
        <input type="number" id="amount" name="amount" placeholder="Enter expense amount" />

        <label for="date">Spend Date</label>
        <input type="date" id="date" name="date" placeholder="Enter date" />

        <label for="category">Category</label>
        <select id="category" name="category">
          <option value="">Select</option>
          <option value="Food">Food</option>
        </select>
      </form>
    </div>
  )
}
```



```

    <option value="Entertainment">Entertainment</option>
    <option value="Academic">Academic</option>
  </select>

  <input type="submit" value="Submit" />
</form>
</div>
)
}

```

Next, create event handler for all the input fields to update the expense detail in the state.

```

handleNameChange(e) {
  this.setState( (state, props) => {
    let item = state.item

    item.name = e.target.value;

    return { item: item }
  });
}

handleAmountChange(e) {
  this.setState( (state, props) => {
    let item = state.item

    item.amount = e.target.value;

    return { item: item }
  });
}

handleDateChange(e) {
  this.setState( (state, props) => {
    let item = state.item

    item.date = e.target.value;

    return { item: item }
  });
}

handleCategoryChange(e) {
  this.setState( (state, props) => {
    let item = state.item

    item.category = e.target.value;

    return { item: item }
  });
}

```

Next, bind the event handler in the constructor.

```

this.handleNameChange = this.handleNameChange.bind(this);
this.handleAmountChange = this.handleAmountChange.bind(this);
this.handleDateChange = this.handleDateChange.bind(this);
this.handleCategoryChange = this.handleCategoryChange.bind(this);

```

Next, add an event handler for the submit action.

```

onSubmit = (e) => {
  e.preventDefault();

  alert(JSON.stringify(this.state.item));
}

```

Next, attach the event handlers to the form.

```

render() {
  return (
    <div id="expenseForm">
      <form onSubmit={this.onSubmit}>
        <label for="name">Title</label>
        <input type="text" id="name" name="name" placeholder="Enter expense title"
          value={this.state.item.name}
          onChange={this.handleChange} />

        <label for="amount">Amount</label>
        <input type="number" id="amount" name="amount" placeholder="Enter expense amount"
          value={this.state.item.amount}
          onChange={this.handleAmountChange} />

        <label for="date">Spend Date</label>
        <input type="date" id="date" name="date" placeholder="Enter date"
          value={this.state.item.date}
          onChange={this.handleDateChange} />

        <label for="category">Category</label>
        <select id="category" name="category"
          value={this.state.item.category}
          onChange={this.handleCategoryChange} >
          <option value="">Select</option>
          <option value="Food">Food</option>
          <option value="Entertainment">Entertainment</option>
          <option value="Academic">Academic</option>
        </select>

        <input type="submit" value="Submit" />
      </form>
    </div>
  )
}

```

Finally, export the component.

```
export default ExpenseForm
```

The complete code of the *ExpenseForm* component is as follows:

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

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

    this.state = {
      item: {}
    }

    this.handleNameChange = this.handleNameChange.bind(this);
    this.handleAmountChange = this.handleAmountChange.bind(this);
    this.handleChange = this.handleChange.bind(this);
    this.handleCategoryChange = this.handleCategoryChange.bind(this);
  }

  handleNameChange(e) {
    this.setState( (state, props) => {
      let item = state.item

      item.name = e.target.value;

      return { item: item }
    });
  }

  handleAmountChange(e) {
    this.setState( (state, props) => {
      let item = state.item

      item.amount = e.target.value;

      return { item: item }
    });
  }

  handleChange(e) {
    this.setState( (state, props) => {
      let item = state.item

      item.date = e.target.value;

      return { item: item }
    });
  }

  handleCategoryChange(e) {
    this.setState( (state, props) => {
      let item = state.item
```

```

        item.category = e.target.value;

        return { item: item }
    });
}

onSubmit = (e) => {
    e.preventDefault();

    alert(JSON.stringify(this.state.item));
}

render() {
    return (
<div id="expenseForm">
    <form onSubmit={this.onSubmit}>
        <label for="name">Title</label>
        <input type="text" id="name" name="name" placeholder="Enter expense title"
            value={this.state.item.name}
            onChange={this.handleChange} />

        <label for="amount">Amount</label>
        <input type="number" id="amount" name="amount" placeholder="Enter expense
amount"
            value={this.state.item.amount}
            onChange={this.handleAmountChange} />

        <label for="date">Spend Date</label>
        <input type="date" id="date" name="date" placeholder="Enter date"
            value={this.state.item.date}
            onChange={this.handleChange} />

        <label for="category">Category</label>
        <select id="category" name="category"
            value={this.state.item.category}
            onChange={this.handleChange} >
            <option value="">Select</option>
            <option value="Food">Food</option>
            <option value="Entertainment">Entertainment</option>
            <option value="Academic">Academic</option>
        </select>

        <input type="submit" value="Submit" />
    </form>
</div>
    )
}
}

export default ExpenseForm;

```

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

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

ReactDOM.render(
  <React.StrictMode>
    <ExpenseForm />
  </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, serve the application using npm command.


```
npm start
```

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


Title

Amount

Spend Date



Category



Submit

Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a pop-up message box.

The screenshot shows a web browser at localhost:3000. A form is displayed with the following fields:

- Title:** Grape juice
- Amount:** 34
- Spend Date:** 01/13/2021
- Category:** Food

 A green 'Submit' button is at the bottom. A pop-up message box is open, showing the text 'localhost:3000 says' and a JSON object: `{\"name\": \"Grape juice\", \"amount\": \"34\", \"date\": \"2021-01-13\", \"category\": \"Food\"}`. An 'OK' button is in the bottom right of the pop-up.

Uncontrolled Component

As we learned earlier, uncontrolled component does not support React based form programming. Getting a value of a React DOM element (form element) is not possible without using React api. One way to get the content of the react component is using React *ref* feature.

React provides a *ref* attribute for all its DOM element and a corresponding api, `React.createRef()` to create a new reference (**this.ref**). The newly created reference can be attached to the form element and the attached form element's value can be accessed using **this.ref.current.value** whenever necessary (during validation and submission).

Let us see the step by step process to do form programming in uncontrolled component.

Create a reference.

```
this.inputRef = React.createRef();
```

Create a form element.

```
<input type="text" name="username" />
```

Attach the already created reference in the form element.

```
<input type="text" name="username" ref={this.inputRef} />
```

To set default value of an input element, use *defaultValue* attribute instead of *value* attribute. If *value* is used, it will get updated during rendering phase of the component.

```
<input type="text" name="username" ref={this.inputRef} defaultValue="default value" />
```

Finally, get the input value using **this.inputRef.current.value** during validation and submission.

```
handleSubmit(e) {
  e.preventDefault();

  alert(this.inputRef.current.value);
}
```

Let us create a simple form to add expense entry using uncontrolled component in this chapter.

First, create a new react application, *react-form-uncontrolled-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.

Create *src* folder under the root directory of the application.

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

Next, create a file, *ExpenseForm.css* under *src* folder to style the component.

```
input[type=text], input[type=number], input[type=date], select {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 4px;
  box-sizing: border-box;
}

input[type=submit] {
  width: 100%;
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

input[type=submit]:hover {
  background-color: #45a049;
}

input:focus {
```

```
border: 1px solid #d9d5e0;
}

#expenseForm div {
border-radius: 5px;
background-color: #f2f2f2;
padding: 20px;
}
```

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

Next, import *React* library.

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

Next, import *ExpenseForm.css* file.

```
import './ExpenseForm.css'
```

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

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

Next, create React reference for all input fields.

```
this.nameInputRef = React.createRef();
this.amountInputRef = React.createRef();
this.dateInputRef = React.createRef();
this.categoryInputRef = React.createRef();
```

Next, create *render()* method and add a form with input fields to add expense items.

```
render() {
  return (
    <div id="expenseForm">
      <form>
        <label for="name">Title</label>
        <input type="text" id="name" name="name" placeholder="Enter expense title" />

        <label for="amount">Amount</label>
        <input type="number" id="amount" name="amount" placeholder="Enter expense amount" />

        <label for="date">Spend Date</label>
        <input type="date" id="date" name="date" placeholder="Enter date" />

        <label for="category">Category</label>
        <select id="category" name="category" >
          <option value="">Select</option>
          <option value="Food">Food</option>
        </select>
      </form>
    </div>
  );
}
```



```

    <option value="Entertainment">Entertainment</option>
    <option value="Academic">Academic</option>
  </select>

  <input type="submit" value="Submit" />
</form>
</div>
)
}

```

Next, add an event handler for the submit action.

```

onSubmit = (e) => {
  e.preventDefault();

  let item = {};

  item.name = this.nameInputRef.current.value;
  item.amount = this.amountInputRef.current.value;
  item.date = this.dateInputRef.current.value;
  item.category = this.categoryInputRef.current.value;

  alert(JSON.stringify(item));
}

```

Next, attach the event handlers to the form.

```

render() {
  return (
    <div id="expenseForm">
      <form onSubmit={(e) => this.onSubmit(e)}>
        <label for="name">Title</label>
        <input type="text" id="name" name="name" placeholder="Enter expense title"
          ref={this.nameInputRef} />

        <label for="amount">Amount</label>
        <input type="number" id="amount" name="amount" placeholder="Enter expense
amount"
          ref={this.amountInputRef} />

        <label for="date">Spend Date</label>
        <input type="date" id="date" name="date" placeholder="Enter date"
          ref={this.dateInputRef} />

        <label for="category">Category</label>
        <select id="category" name="category"
          ref={this.categoryInputRef} >
          <option value="">Select</option>
          <option value="Food">Food</option>
          <option value="Entertainment">Entertainment</option>
          <option value="Academic">Academic</option>
        </select>

        <input type="submit" value="Submit" />
      </form>
    </div>
  );
}

```

```

    </form>
  </div>
)
}

```

Finally, export the component.

```
export default ExpenseForm
```

The complete code of the *ExpenseForm* component is given below:

```

import React from 'react';
import './ExpenseForm.css'

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

    this.nameInputRef = React.createRef();
    this.amountInputRef = React.createRef();
    this.dateInputRef = React.createRef();
    this.categoryInputRef = React.createRef();
  }

  onSubmit = (e) => {
    e.preventDefault();

    let item = {};

    item.name = this.nameInputRef.current.value;
    item.amount = this.amountInputRef.current.value;
    item.date = this.dateInputRef.current.value;
    item.category = this.categoryInputRef.current.value;

    alert(JSON.stringify(item));
  }

  render() {
    return (
      <div id="expenseForm">
        <form onSubmit={e => this.onSubmit(e)}>
          <label for="name">Title</label>
          <input type="text" id="name" name="name" placeholder="Enter expense title"
            ref={this.nameInputRef} />

          <label for="amount">Amount</label>
          <input type="number" id="amount" name="amount" placeholder="Enter expense
amount"
            ref={this.amountInputRef} />

          <label for="date">Spend Date</label>
          <input type="date" id="date" name="date" placeholder="Enter date"
            ref={this.dateInputRef} />

```

```

    <label for="category">Category</label>
    <select id="category" name="category"
      ref={this.categoryInputRef} >
      <option value="">Select</option>
      <option value="Food">Food</option>
      <option value="Entertainment">Entertainment</option>
      <option value="Academic">Academic</option>
    </select>

    <input type="submit" value="Submit" />
  </form>
</div>
  )
}
}

export default ExpenseForm;

```

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

```

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

ReactDOM.render(
  <React.StrictMode>
    <ExpenseForm />
  </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, serve the application using npm command.

```
npm start
```

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

Title

Enter expense title

Amount

Enter expense amount

Spend Date

mm/dd/yyyy

Category

Select

Submit

Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a pop-up message box.

localhost:3000

Title

Grape juice

Amount

34

Spend Date

01/13/2021

Category

Food

Submit

localhost:3000 says

{"name":"Grape juice","amount":"34","date":"2021-01-13","category":"Food"}

OK

Formik

Formik is third party React form library. It provides basic form programming and validation. It is based on controlled component and greatly reduces the time to do form programming. Let us recreate the expense form using *Formik* library.

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

Next, install the *Formik* library.

```
cd /go/to/workspace  
npm install formik --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, *ExpenseForm.css* under *src* folder to style the component.

```
input[type=text], input[type=number], input[type=date], select {  
  width: 100%;  
  padding: 12px 20px;  
  margin: 8px 0;  
  display: inline-block;  
  border: 1px solid #ccc;  
  border-radius: 4px;  
  box-sizing: border-box;  
}  
  
input[type=submit] {  
  width: 100%;  
  background-color: #4CAF50;  
  color: white;  
  padding: 14px 20px;  
  margin: 8px 0;  
  border: none;  
  border-radius: 4px;  
  cursor: pointer;  
}  
  
input[type=submit]:hover {  
  background-color: #45a049;  
}  
  
input:focus {  
  border: 1px solid #d9d5e0;  
}  
  
#expenseForm div {  
  border-radius: 5px;  
  background-color: #f2f2f2;  
  padding: 20px;  
}  
  
#expenseForm span {  
  color: red;  
}
```

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

Next, import *React* and *Formik* library.

```
import React from 'react';
import { Formik } from 'formik';
```

Next, import *ExpenseForm.css* file.

```
import './ExpenseForm.css'
```

Next, create *ExpenseForm* class.

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

Next, set initial values of the expense item in the constructor.

```
this.initialValues = { name: '', amount: '', date: '', category: '' }
```

Next, create a validation method. Formik will send the current values entered by the user.

```
validate = (values) => {
  const errors = {};
  if (!values.name) {
    errors.name = 'Required';
  }
  if (!values.amount) {
    errors.amount = 'Required';
  }
  if (!values.date) {
    errors.date = 'Required';
  }
  if (!values.category) {
    errors.category = 'Required';
  }
  return errors;
}
```

Next, create a method to submit the form. Formik will send the current values entered by the user.

```
handleSubmit = (values, setSubmitting) => {
  setTimeout(() => {
    alert(JSON.stringify(values, null, 2));
    setSubmitting(false);
  }, 400);
}
```

Next, create *render()* method. Use *handleChange*, *handleBlur* and *handleSubmit* method provided by Formik as input elements event handler.

```
render() {
  return (
```

```

    <div id="expenseForm">
      <Formik
        initialValues={this.initialValues}
        validate={values => this.validate(values)}
        onSubmit={(values, { setSubmitting }) =>
this.handleSubmit(values, setSubmitting)}
      >
        {({
          values,
          errors,
          touched,
          handleChange,
          handleBlur,
          handleSubmit,
          isSubmitting,
          /* and other goodies */
        }) => (
          <form onSubmit={handleSubmit}>
            <label for="name">Title <span>{errors.name &&
touched.name && errors.name}</span></label>
            <input type="text" id="name" name="name"
placeholder="Enter expense title"
              onChange={handleChange}
              onBlur={handleBlur}
              value={values.name} />

            <label for="amount">Amount <span>{errors.amount &&
touched.amount && errors.amount}</span></label>
            <input type="number" id="amount" name="amount"
placeholder="Enter expense amount"
              onChange={handleChange}
              onBlur={handleBlur}
              value={values.amount} />

            <label for="date">Spend Date <span>{errors.date &&
touched.date && errors.date}</span></label>
            <input type="date" id="date" name="date"
placeholder="Enter date"
              onChange={handleChange}
              onBlur={handleBlur}
              value={values.date} />

            <label for="category">Category
            <span>{errors.category && touched.category && errors.category}</span></label>
            <select id="category" name="category"
              onChange={handleChange}
              onBlur={handleBlur}
              value={values.category}>
              <option value="">Select</option>
              <option value="Food">Food</option>
              <option
value="Entertainment">Entertainment</option>
              <option value="Academic">Academic</option>
            </select>
          </form>
        )}
      </Formik>
    </div>

```

```

                                <input type="submit" value="Submit"
disabled={isSubmitting} />
                                </form>
                                )}
                                </Formik>
                                </div>
                                )
                                }

```

Finally, export the component.

```
export default ExpenseForm
```

The complete code of the *ExpenseForm* component is given below:

```

import React from 'react';
import './ExpenseForm.css'

import { Formik } from 'formik';

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

    this.initialValues = { name: '', amount: '', date: '', category: '' }
  }

  validate = (values) => {
    const errors = {};
    if (!values.name) {
      errors.name = 'Required';
    }
    if (!values.amount) {
      errors.amount = 'Required';
    }
    if (!values.date) {
      errors.date = 'Required';
    }
    if (!values.category) {
      errors.category = 'Required';
    }
    return errors;
  }

  handleSubmit = (values, setSubmitting) => {
    setTimeout(() => {
      alert(JSON.stringify(values, null, 2));
      setSubmitting(false);
    }, 400);
  }

  render() {

```



```

    return (
      <div id="expenseForm">
        <Formik
          initialValues={this.initialValues}
          validate={values => this.validate(values)}
          onSubmit={(values, { setSubmitting }) =>
            this.handleSubmit(values, setSubmitting)}
        >
          {({
            values,
            errors,
            touched,
            handleChange,
            handleBlur,
            handleSubmit,
            isSubmitting,
            /* and other goodies */
          }) => (
            <form onSubmit={handleSubmit}>
              <label for="name">Title <span>{errors.name &&
                touched.name && errors.name}</span></label>
              <input type="text" id="name" name="name"
                placeholder="Enter expense title"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.name} />

              <label for="amount">Amount <span>{errors.amount
                && touched.amount && errors.amount}</span></label>
              <input type="number" id="amount" name="amount"
                placeholder="Enter expense amount"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.amount} />

              <label for="date">Spend Date <span>{errors.date
                && touched.date && errors.date}</span></label>
              <input type="date" id="date" name="date"
                placeholder="Enter date"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.date} />

              <label for="category">Category
                <span>{errors.category && touched.category && errors.category}</span></label>
              <select id="category" name="category"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.category}>
                <option value="">Select</option>
                <option value="Food">Food</option>
                <option
                  value="Entertainment">Entertainment</option>
                <option value="Academic">Academic</option>
              </select>
            </form>
          )}
        </Formik>
      </div>
    )
  }
}

```

```

                                </select>

                                <input type="submit" value="Submit"
disabled={isSubmitting} />
                                </form>
                                )}
                                </Formik>
                                </div>
                                )
                                }
                                }

export default ExpenseForm;

```

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

```

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

ReactDOM.render(
  <React.StrictMode>
    <ExpenseForm />
  </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, serve the application using npm command.


```
npm start
```

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

Title

Amount

Spend Date



Category

Select

▼

Submit

Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a popup message box.

← → ↻ ⓘ localhost:3000 ☆ 📷 📄 ⚙️ 👤 ⋮


Title

Grape juice

Amount

34

Spend Date

01/13/2021 

Category

Food ▼



Submit

localhost:3000 says

```
{"name": "Grape juice", "amount": "34", "date": "2021-01-13", "category": "Food"}
```

OK

The interactive version of the form is as follows:

Title
<input type="text" value="Enter expense title"/>
Amount
<input type="text" value="Enter expense amount"/>
Spend Date
<input type="text" value="mm/dd/yyyy"/> 
Category
<input type="text" value="Select"/> 
<input type="submit" value="Submit"/>

13. React — 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'

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.

[Home](#)[List Expenses](#)[Add Expense](#)

Welcome to the React tutorial

[Click here](#) to view expense list

[Click here](#) to add new expenses

14. React — 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: '..', ... }
}
```

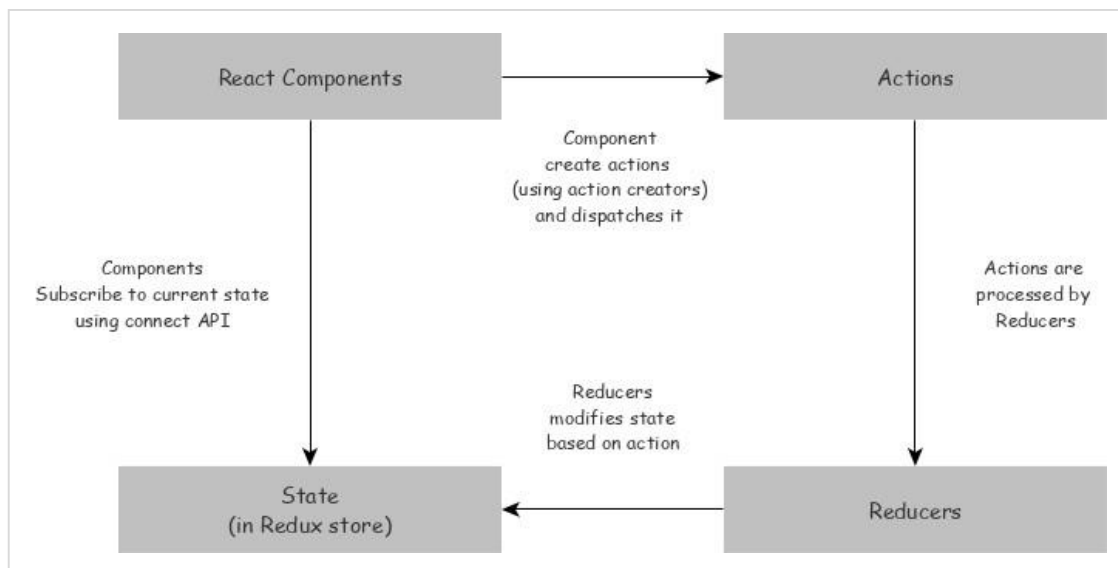
Reducers: Reducers are pure functions used to create a new state based on the existing state and the current action. It returns the newly created state. For example, in add item scenario, it creates a new item list and merges the item from the state and new item and returns the newly created list.

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.

ReduxAPI

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: { } }),
  removeDispatcher: (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="#"
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			

15. React — 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.

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



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.

Hello World!

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>
```

16. React — 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
```

It will run all the available test in our project and report the result.

```
■ 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.
```

17. React — 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 in HTTPS mode.

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

18. React — Building and 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",  
  ...  
}
```

19. React — 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

Expense manager API

First, create a new expense Rest API application by following instruction from *Http Client Programming -> Expense Rest API Server* and start the server. The expense server will be running at `http://localhost:8000`.

Create a skeleton application

Open a terminal and go to your workspace.

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

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

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

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

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

```
cd react-expense-app
npm install
```

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

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

Next, create a folder, *components* under *src* to include our React components. The final structure of the application will be as follows:

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

Let us create our root component, *App*, which will render the entire application.

Create a file, *App.js* under *components* folder and write a simple component to emit *Hello World* message.

```
import React from "react";

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

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 App from './components/App'

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

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

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Expense App</title>
  </head>
```

```
<body>
  <div id="root"></div>
</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.

Hello World!

Install necessary modules

The application uses below third party react libraries given below:

- Redux
- React Redux
- React Router
- Formik
- Redux Thunk (for async fetch api)

Install all the libraries using *npm* package manager using the below command:

```
npm install --save redux react-redux react-router-dom formik redux-thunk
```

Configure Router

Next, create a new file, *Home.js* under *src/components* folder and write a basic *Home* component.

```
import React from "react";

class Home extends React.Component {
  render() {
    return (
      <div>
        <h1>Home</h1>
      </div>
    );
  }
}

export default Home;
```

Next, create a new file, *ExpenseEntryItemForm.js* under *src/components* folder and write a basic *ExpenseEntryItemForm* component.

```
import React from "react";

class ExpenseEntryItemForm extends React.Component {
  render() {
    return (
      <div>
        <h1>Expense list</h1>
      </div>
    );
  }
}
export default ExpenseEntryItemForm;
```

Next, create a new file, *ExpenseEntryItemList.js* under *src/components* folder and write a basic *ExpenseEntryItemList* component.

```
import React from "react";

class ExpenseEntryItemList extends React.Component {
  render() {
    return (
      <div>
        <h1>Expense form</h1>
      </div>
    );
  }
}
export default ExpenseEntryItemList;
```

Create a new file, *App.css* under *src/components* folder and add generic styles for the application.

```
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);
}

input[type=text], input[type=number], input[type=date], select {
width: 100%;
padding: 12px 20px;
margin: 8px 0;
display: inline-block;
border: 1px solid #ccc;
border-radius: 4px;
box-sizing: border-box;
}

input[type=submit] {
width: 100%;
background-color: #4CAF50;
color: white;
padding: 14px 20px;
margin: 8px 0;
border: none;
border-radius: 4px;
cursor: pointer;
}

input[type=submit]:hover {
background-color: #45a049;
}

input:focus {
border: 1px solid #d9d5e0;
}

#expenseForm div {
border-radius: 5px;
background-color: #f2f2f2;
padding: 20px;
}
```



```
#expenseForm span {
  color: red;
}

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, open *App.js* and import router dependencies.

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

Next, import *App.css*.

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

Next, import newly created components.

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

Next, configure Router in the *App* component.

```
class App extends React.Component {
  render() {
    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">
              <div style={ { padding: "10px 0px" } }>
                <ExpenseEntryItemList />
              </div>
            </Route>
            <Route path="/add">
              <div style={ { padding: "10px 0px" } }>
                <ExpenseEntryItemForm />
              </div>
            </Route>
            <Route path="/">
              <div>
                <Home />
              </div>
            </Route>
          </Switch>
        </div>
      </Router>
    );
  }
}
```

The complete source code of the *App* component is given below:

```

import React from "react";

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

import './App.css';

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

class App extends React.Component {
  render() {
    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">
              <div style={{ padding: "10px 0px" }}>
                <ExpenseEntryItemList />
              </div>
            </Route>
            <Route path="/add">
              <div style={{ padding: "10px 0px" }}>
                <ExpenseEntryItemForm />
              </div>
            </Route>
            <Route path="/">
              <div>
                <Home />
              </div>
            </Route>
          </Switch>
        </div>
      </Router>
    );
  }
}

```

```

    }
  }
}

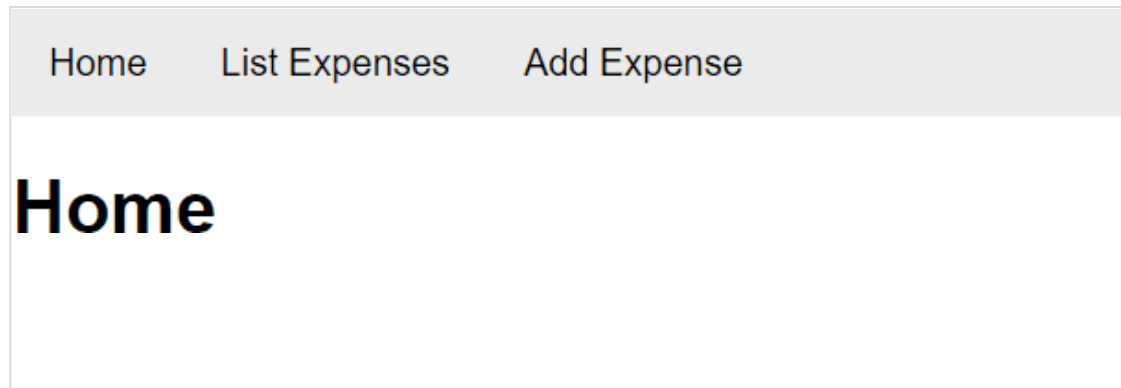
export default App;

```

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.



State management

We are going to do below action to manage our redux store.

- Fetching the expenses from server through async fetch api and set it in Redux store.
- Add new expense to the server through async fetch programming and set add the new expense in the Redux store.
- Delete existing expense from the server through async fetch api and update the Redux store.

Let us create action types, action creator, actions and reducers for managing the Redux state.

Create a folder *actions* under *src* folder.

Next, create a file, *types.js* to create action types.

```

export const LIST_EXPENSE_STARTED = 'LIST_EXPENSE_STARTED';
export const LIST_EXPENSE_SUCCESS = 'LIST_EXPENSE_SUCCESS';
export const LIST_EXPENSE_FAILURE = 'LIST_EXPENSE_FAILURE';

export const ADD_EXPENSE_STARTED = 'ADD_EXPENSE_STARTED';
export const ADD_EXPENSE_SUCCESS = 'ADD_EXPENSE_SUCCESS';
export const ADD_EXPENSE_FAILURE = 'ADD_EXPENSE_FAILURE';

export const DELETE_EXPENSE_STARTED = 'DELETE_EXPENSE_STARTED';

```

```
export const DELETE_EXPENSE_SUCCESS = 'DELETE_EXPENSE_SUCCESS';
export const DELETE_EXPENSE_FAILURE = 'DELETE_EXPENSE_FAILURE';
```

Next, create a file, *index.js* under *actions* folder to create action creators.

```
import {
  LIST_EXPENSE_STARTED, LIST_EXPENSE_SUCCESS, LIST_EXPENSE_FAILURE,
  ADD_EXPENSE_STARTED, ADD_EXPENSE_SUCCESS, ADD_EXPENSE_FAILURE,
  DELETE_EXPENSE_STARTED, DELETE_EXPENSE_SUCCESS, DELETE_EXPENSE_FAILURE,
} from "../types";

export const getExpenseListStarted = () => {
  return {
    type: LIST_EXPENSE_STARTED
  }
}

export const getExpenseListSuccess = data => {
  return {
    type: LIST_EXPENSE_SUCCESS,
    payload: {
      data
    }
  }
}

export const getExpenseListFailure = error => {
  return {
    type: LIST_EXPENSE_FAILURE,
    payload: {
      error
    }
  }
}

export const addExpenseStarted = () => {
  return {
    type: ADD_EXPENSE_STARTED
  }
}

export const addExpenseSuccess = data => {
  return {
    type: ADD_EXPENSE_SUCCESS,
    payload: {
      data
    }
  }
}

export const addExpenseFailure = error => {
  return {
    type: ADD_EXPENSE_FAILURE,
    payload: {
```

```

        error
      }
    }
  }

export const deleteExpenseStarted = () => {
  return {
    type: DELETE_EXPENSE_STARTED
  }
}

export const deleteExpenseSuccess = data => {
  return {
    type: DELETE_EXPENSE_SUCCESS,
    payload: {
      data
    }
  }
}

export const deleteExpenseFailure = error => {
  return {
    type: DELETE_EXPENSE_FAILURE,
    payload: {
      error
    }
  }
}

```

Here, we created one action creator for every possible outcome (success, failure and error) of fetch api. Since we are going to use three web api calls and each call will have three possible outcomes, we use 9 action creators.

Next, create a file, *expenseActions.js* under *actions* folder and create three functions to fetch, add and delete expenses and to dispatch state changes.

```

import {
  getExpenseListStarted, getExpenseListSuccess, getExpenseListFailure,
  addExpenseStarted, addExpenseSuccess, addExpenseFailure,
  deleteExpenseStarted, deleteExpenseSuccess, deleteExpenseFailure
} from "../index";

export const getExpenseList = () => async dispatch => {
  dispatch(getExpenseListStarted());
  try {
    const res = await fetch('http://localhost:8000/api/expenses');
    const data = await res.json();
    var items = [];
    data.forEach((item) => {
      let newItem = {
        id: item._id,
        name: item.name,
        amount: item.amount,
        spendDate: item.spend_date,

```

```

        category: item.category
      }
      items.push(newItem)
    });
    dispatch(getExpenseListSuccess(items));
  } catch (err) {
    dispatch(getExpenseListFailure(err.message));
  }
}

export const addExpense = (data) => async dispatch => {
  dispatch(addExpenseStarted());

  let newItem = {
    name: data.name,
    amount: data.amount,
    spend_date: data.spendDate,
    category: data.category
  }
  console.log(newItem);

  try {
    const res = await fetch('http://localhost:8000/api/expense', {
      method: 'POST',
      body: JSON.stringify(newItem),
      headers: {
        "Content-type": "application/json; charset=UTF-8"
      }
    });
    const data = await res.json();
    newItem.id = data._id;
    dispatch(addExpenseSuccess(newItem));
  } catch (err) {
    console.log(err);
    dispatch(addExpenseFailure(err.message));
  }
}

export const deleteExpense = (id) => async dispatch => {
  dispatch(deleteExpenseStarted());
  try {
    const res = await fetch('http://localhost:8000/api/expense/' + id, {
      method: 'DELETE'
    });
    const data = await res.json();
    dispatch(deleteExpenseSuccess(id));
  } catch (err) {
    dispatch(deleteExpenseFailure(err.message));
  }
}

```

Here,

- Used async fetch api to do web api calls.

- Used dispatch function to dispatch proper action during success, failure and error events.

Create a folder, *reducers* under *src* folder and create a file, *index.js* under *reducers* folder to create Redux reducers.

```
import {
  LIST_EXPENSE_STARTED, LIST_EXPENSE_SUCCESS, LIST_EXPENSE_FAILURE,
  ADD_EXPENSE_STARTED, ADD_EXPENSE_SUCCESS, ADD_EXPENSE_FAILURE,
  DELETE_EXPENSE_STARTED, DELETE_EXPENSE_SUCCESS, DELETE_EXPENSE_FAILURE
} from "../actions/types";

// define initial state of user
const initialState = {
  data: null,
  loading: false,
  error: null
}

export default function expenseReducer(state = initialState, action) {
  switch (action.type) {
    case LIST_EXPENSE_STARTED:
      return {
        ...state,
        loading: true
      }
    case LIST_EXPENSE_SUCCESS:
      const { data } = action.payload;
      return {
        ...state,
        data,
        loading: false
      }
    case LIST_EXPENSE_FAILURE:
      const { error } = action.payload;
      return {
        ...state,
        error
      }
    case ADD_EXPENSE_STARTED:
      return {
        ...state,
        loading: true
      }
    case ADD_EXPENSE_SUCCESS:
      return {
        ...state,
        loading: false
      }
    case ADD_EXPENSE_FAILURE:
      const { expenseError } = action.payload;
      return {
        ...state,
        expenseError
      }
  }
}
```



```

    }
    case DELETE_EXPENSE_STARTED:
      return {
        ...state,
        loading: true
      }
    case DELETE_EXPENSE_SUCCESS:
      return {
        ...state,
        data: state.data.filter(expense => expense.id !==
action.payload.data),
        loading: false
      }
    case DELETE_EXPENSE_FAILURE:
      const { deleteError } = action.payload;
      return {
        ...state,
        deleteError
      }
    default:
      return state
  }
}

```

Here, we have updated the redux store state for each action type.

Next, open *index.js* file under *src* folder and include *Provider* component so that all the components can connect and work with the redux store.

```

import React from 'react';
import ReactDOM from 'react-dom';

import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

import { Provider } from 'react-redux';

import rootReducer from './reducers';
import App from './components/App';

const store = createStore(rootReducer, applyMiddleware(thunk));

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

```

Here,

- Imported *createStore* and *applyMiddleware*
- Imported *thunk* from *redux-thunk* library (for async fetch api)
- Imported *Provider* from *redux* library

- Created newstore using createStore by configuring reducer and thunk middleware
- Attached the Provider component as top level component with redux store

Listexpenses

Open *ExpenseEntryItemList.js* and import *connect* from redux library.

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

Next, import addExpenseList and deleteExpense actions.

```
import { getExpenseList, deleteExpense } from '../actions/expenseActions';
```

Next, add constructor with props.

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

Next, call *getExpenseList* in *componentDidMount()* life cycle.

```
componentDidMount() {
  this.props.getExpenseList();
}
```

Next, write a method to handle the remove expense option.

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

  this.props.deleteExpense(id);
}
```

Now, let us write a function, *getTotal* to calculate the total expenses.

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

Next, update the *render* method and list the expense items.

```
render() {
  let lists = [];

  if (this.props.expenses != null) {
```

```

        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>
      );
    }
  }

```

Next, write *mapStateToProps* and *mapDispatchToProps* methods.

```

const mapStateToProps = state => {
  return {
    expenses: state.data
  };
};

const mapDispatchToProps = {
  getExpenseList,
  deleteExpense
};

```

Here, we have mapped the expenses item from redux store to *expenses* property and attach dispatcher, *getExpenseList* and *deleteExpense* to component properties.

Finally, connect component to Redux store using *connect* api.

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

The complete source code of the application is given below:

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

import { getExpenseList, deleteExpense } from '../actions/expenseActions';

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

  componentDidMount() {
    this.props.getExpenseList();
  }

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

    this.props.deleteExpense(id);
  }

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

  render() {
    let lists = [];

    if (this.props.expenses != null) {
      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.data
  };
};

const mapDispatchToProps = {
  getExpenseList,
  deleteExpense
};

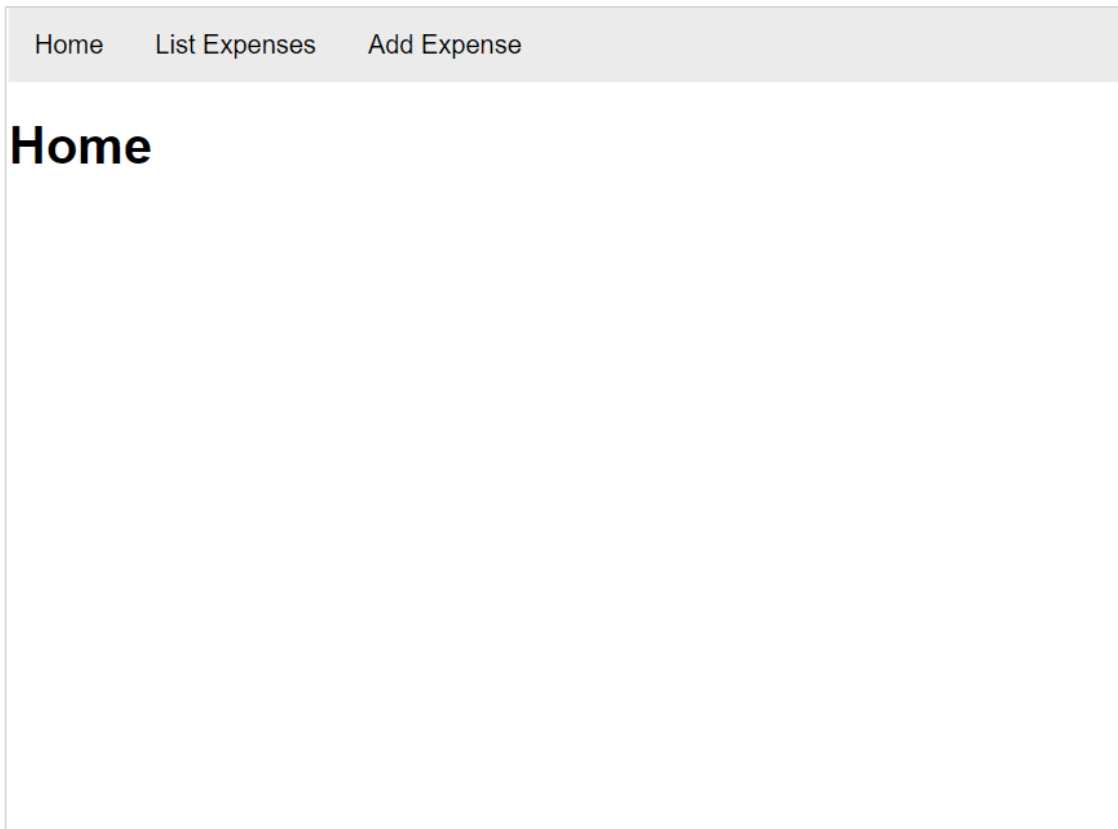
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(ExpenseEntryItemList);

```

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.



Add expense

Open *ExpenseEntryItemForm.js* and import *connect* from *redux* library.

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

Next, import *Formik* library.

```
import { Formik } from 'formik';
```

Next, import *withRouter* method from *router* library.

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

Next, import *addExpense* from our *action* library.

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

Next, create constructor with initial values for expenses.

```
constructor(props) {  
  super(props);  
  
  this.initialValues = { name: '', amount: '', spend_date: '', category: '' }  
}
```

Next, write the *validate* method.

```

validate = (values) => {
  const errors = {};
  if (!values.name) {
    errors.name = 'Required';
  }
  if (!values.amount) {
    errors.amount = 'Required';
  }
  if (!values.spend_date) {
    errors.spend_date = 'Required';
  }
  if (!values.category) {
    errors.category = 'Required';
  }
  return errors;
}

```

Next, add event handler method.

```

handleSubmit = (values, setSubmitting) => {
  setTimeout(() => {
    let newItem = {
      name: values.name,
      amount: values.amount,
      spendDate: values.spend_date,
      category: values.category
    }
    this.props.addExpense(newItem);
    setSubmitting(false);
    this.props.history.push("/list");
  }, 400);
}

```

Here,

- Used *addExpense* method to add expense item
- Use router history method to move to expense list page.

Next, update render method with form created using Formik library.

```

render() {
  return (
    <div id="expenseForm">
      <Formik
        initialValues={this.initialValues}
        validate={values => this.validate(values)}
        onSubmit={(values, { setSubmitting }) =>
          this.handleSubmit(values, setSubmitting)}
      >
        {({
          values,
          errors,
          touched,

```

```

        handleChange,
        handleBlur,
        handleSubmit,
        isSubmitting,
        /* and other goodies */
      }) => (
        <form onSubmit={handleSubmit}>
          <label for="name">Title <span>{errors.name &&
touched.name && errors.name}</span></label>
          <input type="text" id="name" name="name"
placeholder="Enter expense title"
            onChange={handleChange}
            onBlur={handleBlur}
            value={values.name} />

          <label for="amount">Amount <span>{errors.amount &&
touched.amount && errors.amount}</span></label>
          <input type="number" id="amount" name="amount"
placeholder="Enter expense amount"
            onChange={handleChange}
            onBlur={handleBlur}
            value={values.amount} />

          <label for="spend_date">Spend Date
<span>{errors.spend_date && touched.spend_date &&
errors.spend_date}</span></label>
          <input type="date" id="spend_date"
name="spend_date" placeholder="Enter date"
            onChange={handleChange}
            onBlur={handleBlur}
            value={values.spend_date} />

          <label for="category">Category
<span>{errors.category && touched.category && errors.category}</span></label>
          <select id="category" name="category"
            onChange={handleChange}
            onBlur={handleBlur}
            value={values.category}>
            <option value="">Select</option>
            <option value="Food">Food</option>
            <option
value="Entertainment">Entertainment</option>
            <option value="Academic">Academic</option>
          </select>

          <input type="submit" value="Submit"
disabled={isSubmitting} />
        </form>
      )}
    </Formik>
  </div>
)
}

```


Next, map dispatch method to component properties.

```
const mapDispatchToProps = {
  addExpense,
};
```

Finally, connect the component to store and also wrap the component with *WithRouter* to get programmatic access to router links.

```
export default withRouter(connect(
  null,
  mapDispatchToProps
)(ExpenseEntryItemForm));
```

The complete source code of the component is as follows:

```
import React from "react";

import { connect } from 'react-redux';
import { Formik } from 'formik';
import { withRouter } from "react-router-dom";
import { addExpense } from '../actions/expenseActions';

class ExpenseEntryItemForm extends React.Component {

  constructor(props) {
    super(props);

    this.initialValues = { name: '', amount: '', spend_date: '', category: '' }
  }

  validate = (values) => {
    const errors = {};
    if (!values.name) {
      errors.name = 'Required';
    }
    if (!values.amount) {
      errors.amount = 'Required';
    }
    if (!values.spend_date) {
      errors.spend_date = 'Required';
    }
    if (!values.category) {
      errors.category = 'Required';
    }
    return errors;
  }

  handleSubmit = (values, setSubmitting) => {
    setTimeout(() => {
      let newItem = {
        name: values.name,
        amount: values.amount,
        spendDate: values.spend_date,
```

```

        category: values.category
      }
      this.props.addExpense(newItem);
      setSubmitting(false);
      this.props.history.push("/list");
    }, 400);
  }

  render() {
    return (
      <div id="expenseForm">
        <Formik
          initialValues={this.initialValues}
          validate={values => this.validate(values)}
          onSubmit={(values, { setSubmitting }) => this.handleSubmit(values,
setSubmitting)}
        >
          {{{
            values,
            errors,
            touched,
            handleChange,
            handleBlur,
            handleSubmit,
            isSubmitting,
            /* and other goodies */
          }} => (
            <form onSubmit={handleSubmit}>
              <label for="name">Title <span>{errors.name && touched.name &&
errors.name}</span></label>
              <input type="text" id="name" name="name" placeholder="Enter
expense title"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.name} />

              <label for="amount">Amount <span>{errors.amount &&
touched.amount && errors.amount}</span></label>
              <input type="number" id="amount" name="amount"
placeholder="Enter expense amount"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.amount} />

              <label for="spend_date">Spend Date <span>{errors.spend_date &&
touched.spend_date && errors.spend_date}</span></label>
              <input type="date" id="spend_date" name="spend_date"
placeholder="Enter date"
                onChange={handleChange}
                onBlur={handleBlur}
                value={values.spend_date} />

              <label for="category">Category <span>{errors.category &&
touched.category && errors.category}</span></label>

```

```

        <select id="category" name="category"
          onChange={handleChange}
          onBlur={handleBlur}
          value={values.category}>
          <option value="">Select</option>
          <option value="Food">Food</option>
          <option value="Entertainment">Entertainment</option>
          <option value="Academic">Academic</option>
        </select>

        <input type="submit" value="Submit" disabled={isSubmitting} />
      </form>
    )}
  </Formik>
</div>
)
}
}

const mapDispatchToProps = {
  addExpense,
};

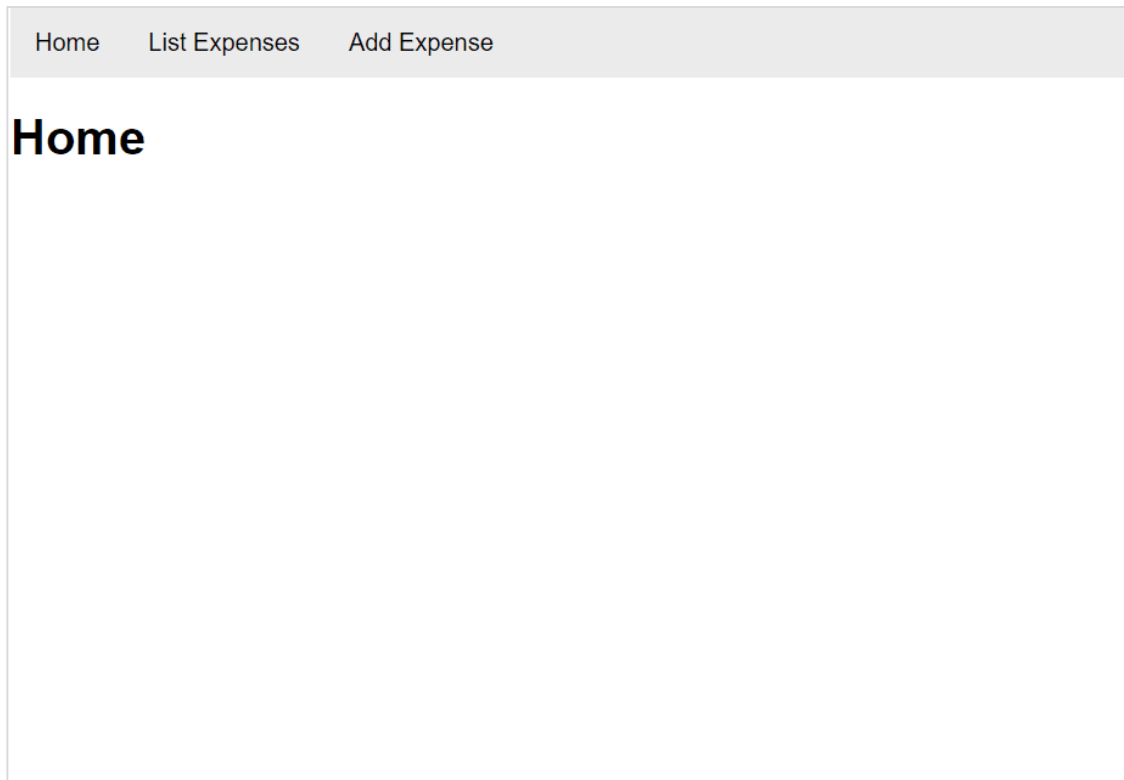
export default withRouter(connect(
  null,
  mapDispatchToProps
)(ExpenseEntryItemForm));

```

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.



Finally, we have successfully created a simple react application with basic features.

Conclusion

React is one of the most popular and highly recommended UI frameworks. True to its popularity, it is being developed for a very long time and actively maintained. Learning react framework is a good starting point for the front end developers and will surely help them to improve their professional career.