React is an open source, JavaScript library for developing user interface (UI) in web application. React is developed and released by **Facebook**. Facebook is continuously working on the *React* library and enhancing it by fixing bugs and introducing new features. This tutorial starts with the architecture of React, how-to guide to setup projects, creating components, JSX and then walks through advanced concepts like state management, form programming, routing and finally conclude with step by step working example.

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

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

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/.](https://nodejs.org/en/download/)

## Toolchain

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

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

Some of the popular toolchain to develop React applications are −

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

Tools required to develop a React application are −

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

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

## The *serve* static server

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

npm install serve -g

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

Open a command prompt and go to your workspace.

cd /go/to/your/workspace

Create a new folder, *static\_site* and change directory to newly created folder.

mkdir static\_site

cd static\_site

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>Static website</title>

</head>

<body>

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

</body>

</html>

Next, run the *serve* command.

serve .

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

serve ./index.html

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



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

│ Serving! │

│ │

│ - Local: http://localhost:57311 │

│ - On Your Network: http://192.168.56.1:57311 │

│ │

│ This port was picked because 5000 is in use. │

│ │

│ Copied local address to clipboard!

## Babel compiler

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

To install Babel and it’s React companion, run the below command −

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

...

...

+ babel-cli@6.26.0

+ babel-preset-react-app@3.1.2

updated 2 packages in 8.685s

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

## Create React App toolchain

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

> npm install -g create-react-app

+ create-react-app@4.0.1

added 6 packages from 4 contributors, removed 37 packages and updated 12 packages in 4.693s

### **Updating the toolchain**

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

npm install react-scripts@latest

### **Advantages of using React toolchain**

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

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

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

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

### **React elements**

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

### **JSX**

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

### **React component**

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

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

## Workflow of a React application

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

Open a command prompt and go to your workspace.

cd /go/to/your/workspace

Next, create a folder, *static\_site* and change directory to newly created folder.

mkdir static\_site

cd static\_site

### **Example**

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React Application</title>

</head>

<body>

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

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

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

<script language="JavaScript">

element = React.createElement('h1', {}, 'Hello React!')

ReactDOM.render(element, document.getElementById('react-app'));

</script>

</body>

</html>

Next, serve the application using serve web server.

serve ./hello.html

### **Output**

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



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

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

### **React.createElement**

Used to create React elements. It expects three parameters −

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

### **ReactDOM.render**

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

* React Element OR JSX
* Root element of the webpage

### **Nested React element**

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

**Example**

<script language="JavaScript">

element = React.createElement('div', {}, React.createElement('h1', {}, 'Hello React!'));

ReactDOM.render(element, document.getElementById('react-app'));

</script>

**Output**

It will generate the below content −

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

### **Use JSX**

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React Application</title>

</head>

<body>

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

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

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

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

<script type="text/babel">

ReactDOM.render(

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

document.getElementById('react-app')

);

</script>

</body>

</html>

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

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

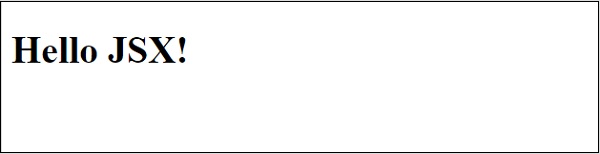
<script type="text/babel">

...

...

</script>

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



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

<script type="text/babel">

function Greeting() {

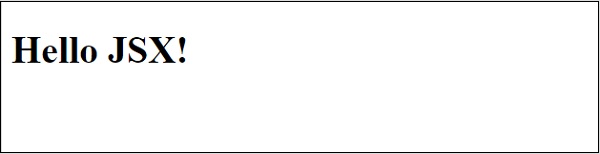
return <div><h1>Hello JSX!</h1></div>

}

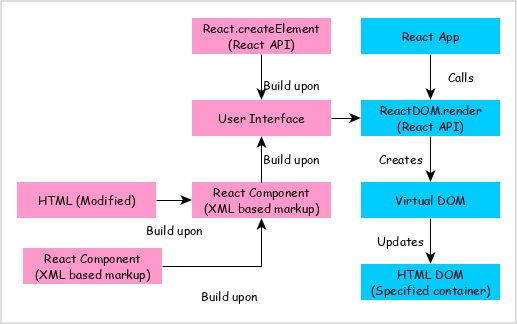
ReactDOM.render(<Greeting />, document.getElementById('react-app') );

</script>

The result is same and as shown below −



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



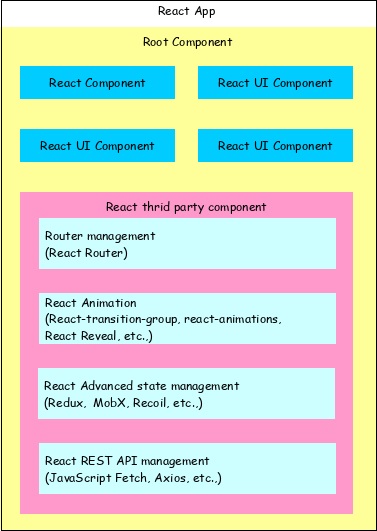
React app calls ***ReactDOM.render*** method by passing the user interface created using React component (coded in either JSX or React element format) and the container to render the user interface.

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

Virtual DOM will be merged and rendered into the container.

## Architecture of the React Application

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



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

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.

* [Simple application using CDN](https://www.tutorialspoint.com/reactjs/reactjs_using_cdn.htm)

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

Open a terminal and go to your workspace.

cd /go/to/your/workspace

Next, create a folder, *static\_site* and change directory to newly created folder.

mkdir static\_site

cd static\_site

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>Simple React app</title>

</head>

<body>

</body>

</html>

Next, include *React library*.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>Simple React app</title>

</head>

<body>

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

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

</body>

</html>

Here,

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

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

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

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

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React based application</title>

</head>

<body>

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

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

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

</body>

</html>

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

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React based application</title>

</head>

<body>

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

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

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

<script language="JavaScript">

element = React.createElement('h1', {}, 'Hello React!')

ReactDOM.render(element, document.getElementById('react-app'));

</script>

</body>

</html>

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

Next, serve the application using serve web server.

serve ./hello.html

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



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

* [Complex application using *React Create App* cli](https://www.tutorialspoint.com/reactjs/reactjs_using_create_react_app_tool.htm)

1) Install create-react-app

Command: npm install -g create-react-app

2)create the new project

Create-react-app project-name

4)npm start

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

Open a terminal and go to your workspace.

> cd /go/to/your/workspace

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

> create-react-app expense-manager

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

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

cd expense-manager

npm install

The *npm install* will install the necessary library under *node\_modules* folder.

Next, start the application.

npm start

Compiled successfully!

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

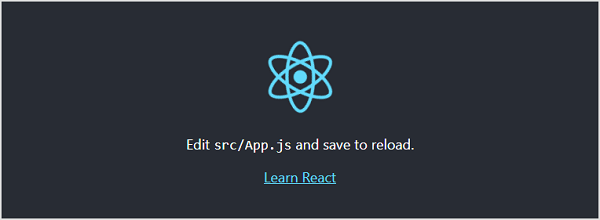
Local: http://localhost:3000

On Your Network: http://192.168.56.1:3000

Note that the development build is not optimized.

To create a production build, use npm run build.

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



Let us analyse the structure of our React application.

Files and folders

The content of the React application is as follows −

|-- README.md

|-- node\_modules

|-- package-lock.json

|-- package.json

|-- public

| |-- favicon.ico

| |-- index.html

| |-- logo192.png

| |-- logo512.png

| |-- manifest.json

| `-- robots.txt

`-- src

|-- App.css

|-- App.js

|-- App.test.js

|-- index.css

|-- index.js

|-- logo.svg

|-- reportWebVitals.js

`-- setupTests.js

Here,

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

{

"name": "expense-manager",

"version": "0.1.0",

"private": true,

"dependencies": {

"@testing-library/jest-dom": "^5.11.6",

"@testing-library/react": "^11.2.2",

"@testing-library/user-event": "^12.6.0",

"react": "^17.0.1",

"react-dom": "^17.0.1",

"react-scripts": "4.0.1",

"web-vitals": "^0.2.4"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test",

"eject": "react-scripts eject"

},

"eslintConfig": {

"extends": [

"react-app",

"react-app/jest"

]

},

"browserslist": {

"production": [

">0.2%",

"not dead",

"not op\_mini all"

],

"development": [

"last 1 chrome version",

"last 1 firefox version",

"last 1 safari version"

]

}

}

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

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

Source code of the application

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

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

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

// If you want to start measuring performance in your app, pass a function

// to log results (for example: reportWebVitals(console.log))

// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals

reportWebVitals();

Here,

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

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

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

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

import './App.css';

function App() {

return (

<h1>Hello React!</h1>

);

}

export default App;

* ***App.css*** − Used to style the *App* component. Let us remove all styles and start with fresh code.
* ***App.test.js*** − Used to write unit test function for our component.
* ***setupTests.js*** − Used to setup the testing framework for our application.
* ***reportWebVitals.js*** − Generic web application startup code to support all browsers.
* ***logo.svg*** − Logo in SVG format and can be loaded into our application using import keyword. Let us remove it from the project.
* [Complex application using customized method](https://www.tutorialspoint.com/reactjs/reactjs_customized_code.htm)

Customize the code

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

Delete all files under src and public folder.

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

The final structure of the application will be as follows −

|-- package-lock.json

|-- package.json

`-- public

|-- index.html

`-- src

|-- index.js

`-- components

| |-- mycom.js

| |-- mycom.css

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

import React from "react";

class HelloWorld extends React.Component {

render() {

return (

<div>

<h1>Hello World!</h1>

</div>

);

}

}

export default HelloWorld;

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

import React from 'react';

import ReactDOM from 'react-dom';

import HelloWorld from './components/HelloWorld';

ReactDOM.render(

<React.StrictMode>

<HelloWorld />

</React.StrictMode>,

document.getElementById('root')

);

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Expense Manager</title>

</head>

<body>

<div id="root"></div>

</body>

</html>

Run the application

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

> npm start

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

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

> react-scripts start

i ｢wds｣: Project is running at http://192.168.56.1/

i ｢wds｣: webpack output is served from

i ｢wds｣: Content not from webpack is served from D:\path\to\expense-manager\public

i ｢wds｣: 404s will fallback to /

Starting the development server...

Compiled successfully!

You can now view expense-manager in the browser.

Local: http://localhost:3000

On Your Network: http://192.168.56.1:3000

Note that the development build is not optimized.

To create a production build, use npm run build.

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



Using custom solution

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

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

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

## Using Rollup bundler

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

Open a terminal and go to your workspace.

cd /go/to/your/workspace

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

mkdir expense-manager-rollup

cd expense-manager-rollup

Next, create and initialize the project.

npm init -y

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

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

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

npm install @babel/preset-env @babel/preset-react

@babel/core @babel/plugin-proposal-class-properties -D

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

npm i -D rollup postcss@8.1 @rollup/plugin-babel

@rollup/plugin-commonjs @rollup/plugin-node-resolve

@rollup/plugin-replace rollup-plugin-livereload

rollup-plugin-postcss rollup-plugin-serve postcss@8.1

postcss-modules@4 rollup-plugin-postcss

Next, install corejs and regenerator runtime for async programming.

npm i regenerator-runtime core-js

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

{

"presets": [

[

"@babel/preset-env",

{

"useBuiltIns": "usage",

"corejs": 3,

"targets": "> 0.25%, not dead"

}

],

"@babel/preset-react"

],

"plugins": [

"@babel/plugin-proposal-class-properties"

]

}

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

import babel from '@rollup/plugin-babel';

import resolve from '@rollup/plugin-node-resolve';

import commonjs from '@rollup/plugin-commonjs';

import replace from '@rollup/plugin-replace';

import serve from 'rollup-plugin-serve';

import livereload from 'rollup-plugin-livereload';

import postcss from 'rollup-plugin-postcss'

export default {

input: 'src/index.js',

output: {

file: 'public/index.js',

format: 'iife',

},

plugins: [

commonjs({

include: [

'node\_modules/\*\*',

],

exclude: [

'node\_modules/process-es6/\*\*',

],

}),

resolve(),

babel({

exclude: 'node\_modules/\*\*'

}),

replace({

'process.env.NODE\_ENV': JSON.stringify('production'),

}),

postcss({

autoModules: true

}),

livereload('public'),

serve({

contentBase: 'public',

port: 3000,

open: true,

}), // index.html should be in root of project

]

}

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

...

"main": "public/index.js",

"style": "public/styles.css",

"files": [

"public"

],

"scripts": {

"start": "rollup -c -w",

"build": "rollup"

},

...

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

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

The final structure of the application will be as follows −

|-- package-lock.json

|-- package.json

|-- rollup.config.js

|-- .babelrc

`-- public

|-- index.html

`-- src

|-- index.js

`-- components

| |-- mycom.js

| |-- mycom.css

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

import React from "react";

class HelloWorld extends React.Component {

render() {

return (

<div>

<h1>Hello World!</h1>

</div>

);

}

}

export default HelloWorld;

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

import React from 'react';

import ReactDOM from 'react-dom';

import HelloWorld from './components/HelloWorld';

ReactDOM.render(

<React.StrictMode>

<HelloWorld />

</React.StrictMode>,

document.getElementById('root')

);

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

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Expense Manager :: Rollup version</title>

</head>

<body>

<div id="root"></div>

<script type="text/JavaScript" src="./index.js"></script>

</body>

</html>

Next, build and run the application.

npm start

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

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

> rollup -c

rollup v2.36.1

bundles src/index.js → dist\index.js...

LiveReload enabled

http://localhost:10001 -> /path/to/your/workspace/expense-manager-rollup/dist

created dist\index.js in 4.7s

waiting for changes...

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



## Using Parcel bundler

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

First, install the parcel bundler.

npm install -g parcel-bundler

Open a terminal and go to your workspace.

cd /go/to/your/workspace

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

mkdir expense-manager-parcel

cd expense-manager-parcel

Next, create and initialize the project.

npm init -y

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

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

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

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

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

{

"presets": [

"@babel/preset-env",

"@babel/preset-react"

],

"plugins": [

"@babel/plugin-proposal-class-properties"

]

}

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

...

"main": "src/index.js",

"scripts": {

"start": "parcel public/index.html",

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

},

...

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

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

The final structure of the application will be as follows −

|-- package-lock.json

|-- package.json

|-- .babelrc

`-- public

|-- index.html

`-- src

|-- index.js

`-- components

| |-- mycom.js

| |-- mycom.css

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

import React from "react";

class HelloWorld extends React.Component {

render() {

return (

<div>

<h1>Hello World!</h1>

</div>

);

}

}

export default HelloWorld;

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

import React from 'react';

import ReactDOM from 'react-dom';

import HelloWorld from './components/HelloWorld';

ReactDOM.render(

<React.StrictMode>

<HelloWorld />

</React.StrictMode>,

document.getElementById('root')

);

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

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Expense Manager :: Parcel version</title>

</head>

<body>

<div id="root"></div>

<script type="text/JavaScript" src="../src/index.js"></script>

</body>

</html>

Next, build and run the application.

npm start

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

> expense-manager-parcel@1.0.0 dev /go/to/your/workspace/expense-manager-parcel

> parcel index.html Server running at http://localhost:1234

√ Built in 10.41s.

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



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

* Assign to a variable.

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

* Assign to a variable based on a condition.

var canGreet = true;

if(canGreet) {

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

}

* Can be used as return value of a function.

function Greeting() {

return <h1>Hello React!</h1>

}

greeting = Greeting()

* Can be used as argument of a function.

function Greet(message) {

ReactDOM.render(message, document.getElementById('react-app')

}

Greet(<h1>Hello React!</h1>)

## Expressions

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

### **Example**

<script type="text/babel">

var cTime = new Date().toTimeString();

ReactDOM.render(

<div><p>The current time is {cTime}</p></div>,

document.getElementById('react-app') );

</script>

### **Output**

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

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

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

## Functions

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

### **Example**

<script type="text/babel">

var cTime = new Date().toTimeString();

ReactDOM.render(

<div><p>The current time is {cTime}</p></div>,

document.getElementById('react-app')

);

</script>

### **Output**

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

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

## Attributes

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

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

### **Example**

<style>

.red { color: red }

</style>

<script type="text/babel">

function getCurrentTime() {

return new Date().toTimeString();

}

ReactDOM.render(

<div>

<p>The current time is <span className="red">{getCurrentTime()}</span></p>

</div>,

document.getElementById('react-app')

);

</script>

### **Output**

The output is as follows −

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

## Expression in attributes

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

<style>

.red { color: red }

</style>

<script type="text/babel">

function getCurrentTime() {

return new Date().toTimeString();

}

var class\_name = "red";

ReactDOM.render(

<div>

<p>The current time is <span className={class\_name}>{getCurrentTime()}</span></p>

</div>,

document.getElementById('react-app')

);

</script>

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

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

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

React component accomplish these feature using three concepts −

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

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

## Creating a React component

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

* Function component − Uses plain JavaScript function.
* ES6 class component − Uses ES6 class.

The core difference between function and class component are −

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

function Hello() {

return '<div>Hello</div>'

}

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

class ExpenseEntryItem extends React.Component {

render() {

return (

<div>Hello</div>

);

}

}

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

## Creating a class component

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

{

'name': 'Mango juice',

'amount': 30.00,

'spend\_date': '2020-10-10'

'category': 'Food',

}

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

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

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

import React from 'react';

import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {

}

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

class ExpenseEntryItem extends React.Component {

render() {

}

}

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

class ExpenseEntryItem extends React.Component {

render() {

return (

<div>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

}

Next, specify the component as default export class.

import React from 'react';

import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {

render() {

return (

<div>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

}

export default ExpenseEntryItem;

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

import React from 'react';

import ReactDOM from 'react-dom';

import ExpenseEntryItem from './components/ExpenseEntryItem'

ReactDOM.render(

<React.StrictMode>

<ExpenseEntryItem />

</React.StrictMode>,

document.getElementById('root')

);

### **Example**

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React application :: ExpenseEntryItem component</title>

</head>

<body>

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

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

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

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

<script type="text/babel">

class ExpenseEntryItem extends React.Component {

render() {

return (

<div>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

}

ReactDOM.render(

<ExpenseEntryItem />,

document.getElementById('react-app') );

</script>

</body>

</html>

Next, serve the application using npm command.

npm start

### **Output**

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

Item: Mango Juice

Amount: 30.00

Spend Date: 2020-10-10

Category: Food

## Creating a function component

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

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

function ExpenseEntryItem() {

return (

<div>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

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

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

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

* CSS Stylesheet
* Inline Styling
* CSS Modules

## CSS Stylesheet

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

Let us style our *ExpenseEntryItem* component.

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

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

div.itemStyle {

color: brown;

font-size: 14px;

}

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

import React from 'react';

import './ExpenseEntryItem.css';

class ExpenseEntryItem extends React.Component {

render() {

return (

<div className="itemStyle">

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

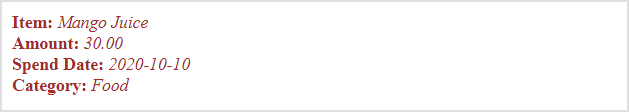
}

export default ExpenseEntryItem;

Next, serve the application using npm command.

npm start

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



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

## Inline Styling

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

Let us add inline styling in our component.

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

itemStyle = {

color: 'brown',

fontSize: '14px'

}

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

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

render() {

return (

<div style={ this.itemStyle }>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

Also, style can be directly set inside the component −

render() {

return (

<div style={

{

color: 'brown',

fontSize: '14px'

}

}>

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

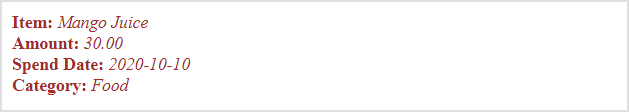
}

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

Next, serve the application using npm command.

npm start

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



## CSS Modules

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

Open expense-manager application in your favorite editor.

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

div.itemStyle {

color: 'brown';

font-size: 14px;

}

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

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

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

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

<div className={styles.itemStyle}>

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

The final and complete code is −

import React from 'react';

import './ExpenseEntryItem.css';

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

class ExpenseEntryItem extends React.Component {

render() {

return (

<div className={styles.itemStyle} >

<div><b>Item:</b> <em>Mango Juice</em></div>

<div><b>Amount:</b> <em>30.00</em></div>

<div><b>Spend Date:</b> <em>2020-10-10</em></div>

<div><b>Category:</b> <em>Food</em></div>

</div>

);

}

}

export default ExpenseEntryItem;

Next, serve the application using npm command.

npm start

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

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](https://www.tutorialspoint.com/reactjs/reactjs_create_component_using_properties.htm)
* [Nested Components](https://www.tutorialspoint.com/reactjs/reactjs_nested_components.htm)
* [Use Component](https://www.tutorialspoint.com/reactjs/reactjs_use_components.htm)
* [Component Collection](https://www.tutorialspoint.com/reactjs/reactjs_component_collection.htm)

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

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

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

Introduce construction function with argument props.

constructor(props) {

super(props);

}

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

render() {

return (

<div>

<div><b>Item:</b> <em>{this.props.name}</em></div>

<div><b>Amount:</b> <em>{this.props.amount}</em></div>

<div><b>Spend date:</b>

<em>{this.props.spenddate.tostring()}</em></div>

<div><b>Category:</b> <em>{this.props.category}</em></div>

</div>

);

}

Here,

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

Now, we have successfully updated the component using properties.

import React from 'react'

import './ExpenseEntryItem.css';

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

class ExpenseEntryItem extends React.Component {

constructor(props) {

super(props);

}

render() {

return (

<div>

<div><b>Item:</b> <em>{this.props.name}</em></div>

<div><b>Amount:</b> <em>{this.props.amount}</em></div>

<div><b>Spend Date:</b>

<em>{this.props.spendDate.toString()}</em></div>

<div><b>Category:</b> <em>{this.props.category}</em></div>

</div>

);

}

}

export default ExpenseEntryItem;

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

import React from 'react';

import ReactDOM from 'react-dom';

import ExpenseEntryItem from './components/ExpenseEntryItem'

const name = "Grape Juice"

const amount = 30.00

const spendDate = new Date("2020-10-10")

const category = "Food"

ReactDOM.render(

<React.StrictMode>

<ExpenseEntryItem

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.



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.



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>

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

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

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

log() {

cosole.log("Event is fired");

}

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

log = () =;> {

cosole.log("Event is fired");

}

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

log(e) {

cosole.log("Event is fired");

console.log(e.target);

}

The alternative lambda syntax is −

log = (e) => {

cosole.log("Event is fired");

console.log(e.target);

}

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

log(extra, e) {

cosole.log("Event is fired");

console.log(e.target);

console.log(extra);

console.log(this);

}

The alternative lambda syntax is as follows −

log = (extra, e) => {

cosole.log("Event is fired");

console.log(e.target);

console.log(extra);

console.log(this);

}

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

constructor(props) {

super(props);

this.logContent = this.logContent.bind(this);

}

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

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

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

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

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

The alternate lambda syntax is as follows −

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

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')

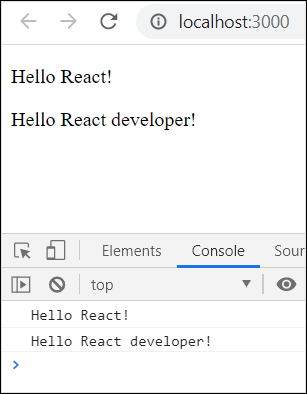
);

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>

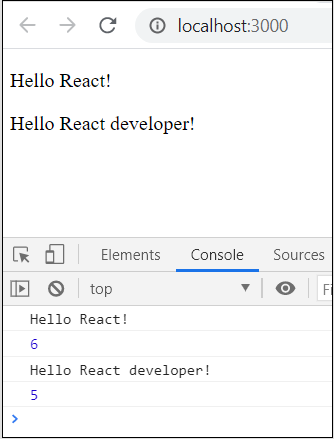
);

}

}

export default MessageWithEvent;

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



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

## What is state?

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

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

The simple representation of the state is as follows −

{

date: '2020-10-10 10:10:10'

}

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

Here,

* [State management API](https://www.tutorialspoint.com/reactjs/reactjs_state_management_api.htm)
* [Stateless component](https://www.tutorialspoint.com/reactjs/reactjs_stateless_component.htm)
* [State management using React Hooks](https://www.tutorialspoint.com/reactjs/reactjs_state_management_react_hooks.htm)
* [Component Life cycle](https://www.tutorialspoint.com/reactjs/reactjs_component_life_cycle.htm)
* [Component life cycle using React Hooks](https://www.tutorialspoint.com/reactjs/reactjs_component_life_cycle_react_hooks.htm)
* [Layout in component](https://www.tutorialspoint.com/reactjs/reactjs_layout_component.htm)
* [Pagination](https://www.tutorialspoint.com/reactjs/reactjs_pagination.htm)
* [Material UI](https://www.tutorialspoint.com/reactjs/reactjs_material_ui.htm)

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.

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

});

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

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

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

## Expense Rest Api Server

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

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

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

cd /go/to/workspace

mkdir apiserver

cd apiserver

Initialize a new node application using the below command −

npm init

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

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.

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

Use `npm install <pkg>` afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

package name: (apiserver)

version: (1.0.0)

description: Rest api for Expense Application

entry point: (index.js) server.js

test command:

git repository:

keywords:

author:

license: (ISC)

About to write to \path\to\workspace\expense-rest-api\package.json:

{

"name": "expense-rest-api",

"version": "1.0.0",

"description": "Rest api for Expense Application",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

Is this OK? (yes) yes

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

npm install express nedb cors

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

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

Pizza,80,2020-10-10,Food

Grape Juice,30,2020-10-12,Food

Cinema,210,2020-10-16,Entertainment

Java Programming book,242,2020-10-15,Academic

Mango Juice,35,2020-10-16,Food

Dress,2000,2020-10-25,Cloth

Tour,2555,2020-10-29,Entertainment

Meals,300,2020-10-30,Food

Mobile,3500,2020-11-02,Gadgets

Exam Fees,1245,2020-11-04,Academic

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

var store = require("nedb")

var fs = require('fs');

var expenses = new store({ filename: "expense.db", autoload: true })

expenses.find({}, function (err, docs) {

if (docs.length == 0) {

loadExpenses();

}

})

function loadExpenses() {

readCsv("data.csv", function (data) {

console.log(data);

data.forEach(function (rec, idx) {

item = {}

item.name = rec[0];

item.amount = parseFloat(rec[1]);

item.spend\_date = new Date(rec[2]);

item.category = rec[3];

expenses.insert(item, function (err, doc) {

console.log('Inserted', doc.item\_name, 'with ID', doc.\_id);

})

})

})

}

function readCsv(file, callback) {

fs.readFile(file, 'utf-8', function (err, data) {

if (err) throw err;

var lines = data.split('\r\n');

var result = lines.map(function (line) {

return line.split(',');

});

callback(result);

});

}

module.exports = expenses

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

var express = require("express")

var cors = require('cors')

var expenseStore = require("./expensedb.js")

var app = express()

app.use(cors());

var bodyParser = require("body-parser");

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

var HTTP\_PORT = 8000

app.listen(HTTP\_PORT, () => {

console.log("Server running on port %PORT%".replace("%PORT%", HTTP\_PORT))

});

app.get("/", (req, res, next) => {

res.json({ "message": "Ok" })

});

app.get("/api/expenses", (req, res, next) => {

expenseStore.find({}, function (err, docs) {

res.json(docs);

});

});

app.get("/api/expense/:id", (req, res, next) => {

var id = req.params.id;

expenseStore.find({ \_id: id }, function (err, docs) {

res.json(docs);

})

});

app.post("/api/expense/", (req, res, next) => {

var errors = []

if (!req.body.item) {

errors.push("No item specified");

}

var data = {

name: req.body.name,

amount: req.body.amount,

category: req.body.category,

spend\_date: req.body.spend\_date,

}

expenseStore.insert(data, function (err, docs) {

return res.json(docs);

});

})

app.put("/api/expense/:id", (req, res, next) => {

var id = req.params.id;

var errors = []

if (!req.body.item) {

errors.push("No item specified");

}

var data = {

\_id: id,

name: req.body.name,

amount: req.body.amount,

category: req.body.category,

spend\_date: req.body.spend\_date,

}

expenseStore.update( { \_id: id }, data, function (err, docs) {

return res.json(data);

});

})

app.delete("/api/expense/:id", (req, res, next) => {

var id = req.params.id;

expenseStore.remove({ \_id: id }, function (err, numDeleted) {

res.json({ "message": "deleted" })

});

})

app.use(function (req, res) {

res.status(404);

});

Now, it is time to run the application.

npm run start

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

{

"message": "Ok"

}

It confirms that our application is working fine.

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

[

...

{

"name": "Pizza",

"amount": 80,

"spend\_date": "2020-10-10T00:00:00.000Z",

"category": "Food",

"\_id": "5H8rK8lLGJPVZ3gD"

},

...

]

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

## The fetch() api

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

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

Next, open the application in your favorite editor.

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

Next, create *components* folder under src folder.

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

html {

font-family: sans-serif;

}

table {

border-collapse: collapse;

border: 2px solid rgb(200,200,200);

letter-spacing: 1px;

font-size: 0.8rem;

}

td, th {

border: 1px solid rgb(190,190,190);

padding: 10px 20px;

}

th {

background-color: rgb(235,235,235);

}

td, th {

text-align: left;

}

tr:nth-child(even) td {

background-color: rgb(250,250,250);

}

tr:nth-child(odd) td {

background-color: rgb(245,245,245);

}

caption {

padding: 10px;

}

tr.highlight td {

background-color: #a6a8bd;

}

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

Next, import *React* library.

import React from 'react';

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

class ExpenseEntryItemList extends React.Component {

constructor(props) {

super(props);

}

}

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

this.state = {

isLoaded: false,

items: []

}

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

setItems(remoteItems) {

var items = [];

remoteItems.forEach((item) => {

let newItem = {

id: item.\_id,

name: item.name,

amount: item.amount,

spendDate: item.spend\_date,

category: item.category

}

items.push(newItem)

});

this.setState({

isLoaded: true,

items: items

});

}

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

fetchRemoteItems() {

fetch("http://localhost:8000/api/expenses")

.then(res => res.json())

.then(

(result) => {

this.setItems(result);

},

(error) => {

this.setState({

isLoaded: false,

error

});

}

)

}

Here,

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

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

deleteRemoteItem(id) {

fetch('http://localhost:8000/api/expense/' + id, { method: 'DELETE' })

.then(res => res.json())

.then(

() => {

this.fetchRemoteItems()

}

)

}

Here,

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

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

componentDidMount() {

this.fetchRemoteItems();

}

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

handleDelete = (id, e) => {

e.preventDefault();

console.log(id);

this.deleteRemoteItem(id);

}

Next, write the render method.

render() {

let lists = [];

if (this.state.isLoaded) {

lists = this.state.items.map((item) =>

<tr key={item.id} onMouseEnter={this.handleMouseEnter} onMouseLeave={this.handleMouseLeave}>

<td>{item.name}</td>

<td>{item.amount}</td>

<td>{new Date(item.spendDate).toDateString()}</td>

<td>{item.category}</td>

<td><a href="#" onClick={(e) => this.handleDelete(item.id, e)}>Remove</a></td>

</tr>

);

}

return (

<div>

<table onMouseOver={this.handleMouseOver}>

<thead>

<tr>

<th>Item</th>

<th>Amount</th>

<th>Date</th>

<th>Category</th>

<th>Remove</th>

</tr>

</thead>

<tbody>

{lists}

</tbody>

</table>

</div>

);

}

Finally, export the component.

export default ExpenseEntryItemList;

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

import React from 'react';

import ReactDOM from 'react-dom';

import ExpenseEntryItemList from './components/ExpenseEntryItemList';

ReactDOM.render(

<React.StrictMode>

<ExpenseEntryItemList />

</React.StrictMode>,

document.getElementById('root')

);

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>React App</title>

</head>

<body>

<div id="root"></div>

<script type="text/JavaScript" src="./index.js"></script>

</body>

</html>

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

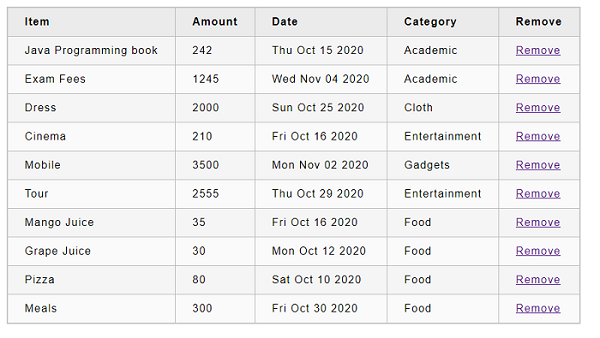
cd /go/to/server/application

npm start

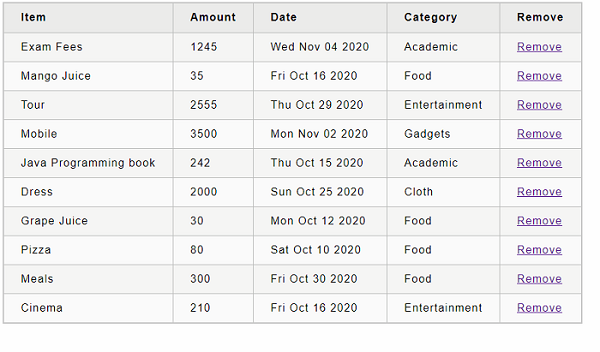
Next, serve the client application using npm command.

npm start

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



Try to remove the item by clicking the remove link.



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](https://www.tutorialspoint.com/reactjs/reactjs_controlled_component.htm)
* [Uncontrolled Component](https://www.tutorialspoint.com/reactjs/reactjs_uncontrolled_component.htm)
* [Formik](https://www.tutorialspoint.com/reactjs/reactjs_formik.htm)

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.handleUsernameChange = this.handleUsernameChange.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>

<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={(e) => this.onSubmit(e)}>

<label for="name">Title</label>

<input type="text" id="name" name="name" placeholder="Enter expense title"

value={this.state.item.name}

onChange={this.handleNameChange} />

<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.handleDateChange = this.handleDateChange.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 }

});

}

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 }

});

}

onSubmit = (e) => {

e.preventDefault();

alert(JSON.stringify(this.state.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"

value={this.state.item.name}

onChange={this.handleNameChange} />

<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>

)

}

}

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>

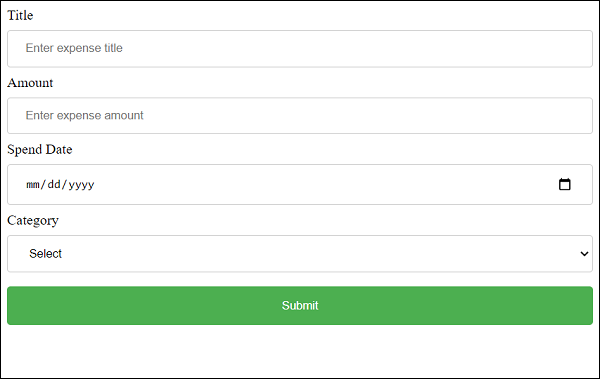
</body>

</html>

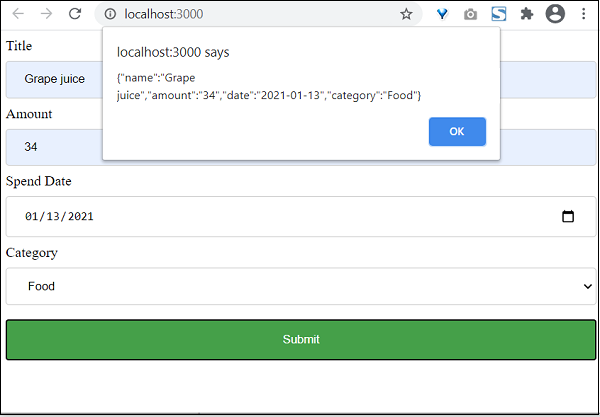
Next, serve the application using npm command.

npm start

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



Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a pop-up message box.



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 defalut 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>

<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>

)

}

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>

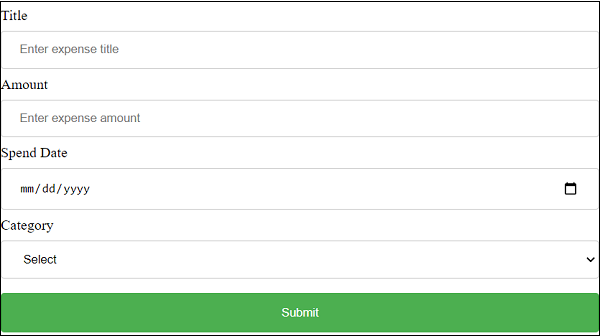
</body>

</html>

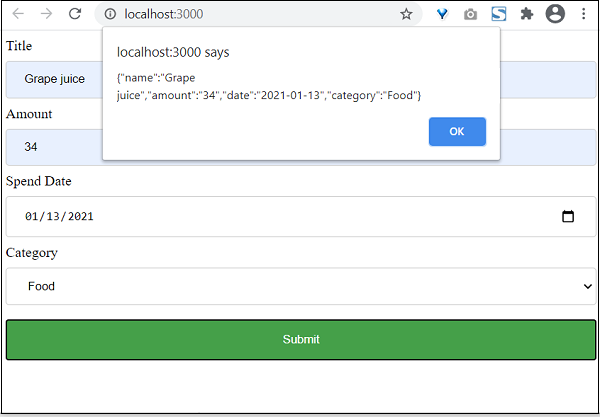
Next, serve the application using npm command.

npm start

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



Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a pop-up message box.



*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>

<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>

<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>

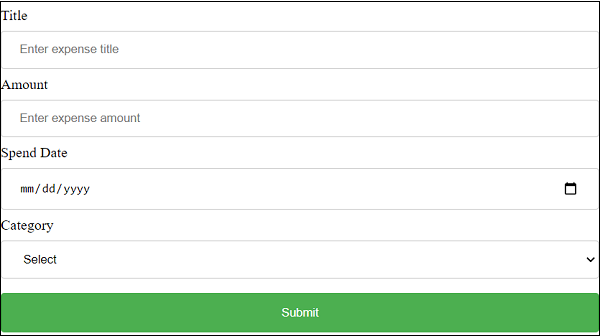
</body>

</html>

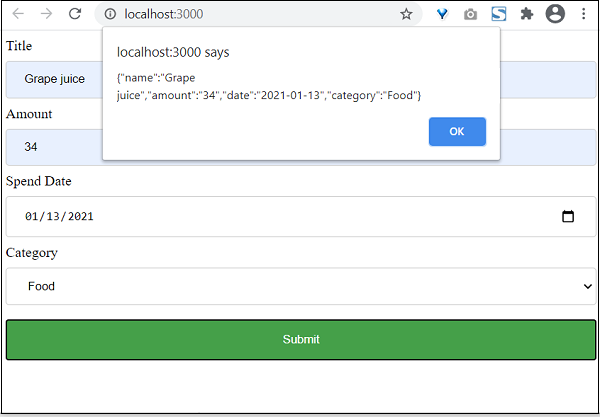
Next, serve the application using npm command.

npm start

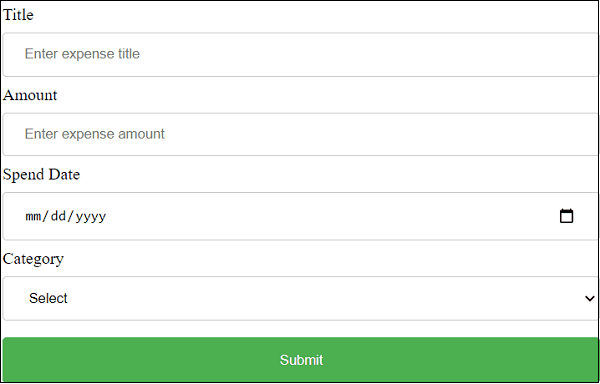
Next, open the browser and enter *http://localhost:3000* in the address bar and press enter.



Finally, enter a sample expense detail and click submit. The submitted data will be collected and showed in a popup message box.



The interactive version of the form is as follows −



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](https://reactrouter.com/) 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.

