ADVANCED CLIENT SIDE PROGRAMMING

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

Why learn ReactJS?

Today, many JavaScript frameworks are available in the market(like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into the DOM.

What is ReactDOM?

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

- render()
- findDOMNode()
- unmountComponentAtNode()
- hydrate()
- createPortal()

Pre-requisite: To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

```
import ReactDOM from 'react-dom'
```

render() Function

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of

React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

Syntax:

```
ReactDOM.render(element, container, callback)
```

Parameters: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Element to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function returns a reference to the component or null if a stateless component was rendered.

findDOMNode() Function

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

Syntax:

```
ReactDOM.findDOMNode(component)
```

Parameters: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.
Return Type: This function returns the DOM node where the component was rendered on success otherwise null.

unmountComponentAtNode() Function

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

Syntax:

```
ReactDOM.unmountComponentAtNode(container)
```

Parameters: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.
Return Type: This function returns true on success otherwise false.

hydrate() Function

This method is equivalent to the render() method but is implemented while using server-side rendering.

Syntax:

```
ReactDOM.hydrate(element, container, callback)
```

Parameters: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

createPortal() Function

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

Syntax:
```
ReactDOM.createPortal(child, container)
```

Parameters: This method takes two parameters as described below.

- **child:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.

Return Type: This function returns nothing.

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.

JSX converts HTML tags into react elements.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1>KVCET</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Expressions in JSX

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a `div` element.

```
import React from 'react';
import ReactDOM from 'react-do/client';

const myElement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: `<></>`.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
    <>
```

```
      <p>I am a paragraph.</p>
      <p>I am a paragraph too.</p>
    </>
  );

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <input type="text" />;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Attribute class = className

The `class` attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the `class` keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

JSX solved this by using `className` instead. When JSX is rendered, it translates `className` attributes into `class` attributes.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1 className="myclass">Hello World</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Conditions - if statements

React supports `if` statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the `if` statements outside of the JSX, or you could use a ternary expression instead:

Option 1:

Write `if` statements outside of the JSX code:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

React component

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

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

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

React component accomplish these feature using three concepts −

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

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

Creating a React component

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

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

The core difference between function and class component are −

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

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

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

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

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

Creating a class component

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

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

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

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

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

```
import React from 'react';
import './ExpenseEntryItem.css';
class ExpenseEntryItem extends React.Component {
}
```

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

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

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

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

Next, specify the component as default export class.

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

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

```
   }
}
export default ExpenseEntryItem;
```

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

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

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

Example

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

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

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

Next, serve the application using npm command.

```
npm start
```

Output

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

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

Creating a function component

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

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

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

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

ReactJS - Properties (props)

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

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

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

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

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

Props are arguments passed into React components.
Props are passed to components via HTML attributes.

React Props

React Props are like function arguments in JavaScript *and* attributes in HTML.

To send props into a component, use the same syntax as HTML attributes:

const myElement = <Car brand="Ford" />;

The component receives the argument as a props object:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

const myElement = <Car brand="Ford" />;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Pass Data

Props are also how you pass data from one component to another, as parameters.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
           <h1>Who lives in my garage?</h1>
           <Car brand="Ford" />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

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

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

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

Expense Rest Api Server

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

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

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

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

Initialize a new node application using the below command −

```
npm init
```

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

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

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

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

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

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

```
npm install express nedb cors
```

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

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

```
Pizza,80,2020-10-10,Food
Grape Juice,30,2020-10-12,Food
Cinema,210,2020-10-16,Entertainment
```

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

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

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

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

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

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

```
var express = require("express")
var cors = require('cors')
var expenseStore = require("./expensedb.js")
var app = express()
app.use(cors());
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
var HTTP_PORT = 8000
app.listen(HTTP_PORT, () => {
  console.log("Server running on port %PORT%".replace("%PORT%", HTTP_PORT))
});
app.get("/", (req, res, next) => {
  res.json({ "message": "Ok" })
});
app.get("/api/expenses", (req, res, next) => {
  expenseStore.find({}, function (err, docs) {
    res.json(docs);
  });
});
app.get("/api/expense/:id", (req, res, next) => {
  var id = req.params.id;
  expenseStore.find({ _id: id }, function (err, docs) {
    res.json(docs);
  })
});
app.post("/api/expense/", (req, res, next) => {
  var errors = []
  if (!req.body.item) {
    errors.push("No item specified");
  }
  var data = {
    name: req.body.name,
    amount: req.body.amount,
    category: req.body.category,
    spend_date: req.body.spend_date,
  }
  expenseStore.insert(data, function (err, docs) {
    return res.json(docs);
  });
})
app.put("/api/expense/:id", (req, res, next) => {
  var id = req.params.id;
  var errors = []
  if (!req.body.item) {
    errors.push("No item specified");
  }
  var data = {
    _id: id,
    name: req.body.name,
        amount: req.body.amount,
        category: req.body.category,
        spend_date: req.body.spend_date,
    }
    expenseStore.update( { _id: id }, data, function (err, docs) {
        return res.json(data);
    });
})
app.delete("/api/expense/:id", (req, res, next) => {
    var id = req.params.id;
    expenseStore.remove({ _id: id }, function (err, numDeleted) {
        res.json({ "message": "deleted" })
    });
})
app.use(function (req, res) {
    res.status(404);
});
```

Now, it is time to run the application.

```
npm run start
```

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

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

It confirms that our application is working fine.

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

```
[
    ...
    {
        "name": "Pizza",
        "amount": 80,
        "spend_date": "2020-10-10T00:00:00.000Z",
        "category": "Food",
        "_id": "5H8rK8lLGJPVZ3gD"
    },
    ...
]
```

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

The fetch() api

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

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

Next, open the application in your favorite editor.

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

Next, create *components* folder under src folder.

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

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

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

Next, import *React* library.

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

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

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

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

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

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

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

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

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

Here,

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

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

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

Here,

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

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

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

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

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

  this.deleteRemoteItem(id);
}
```

Next, write the render method.

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

Finally, export the component.

export default ExpenseEntryItemList;

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

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

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

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

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

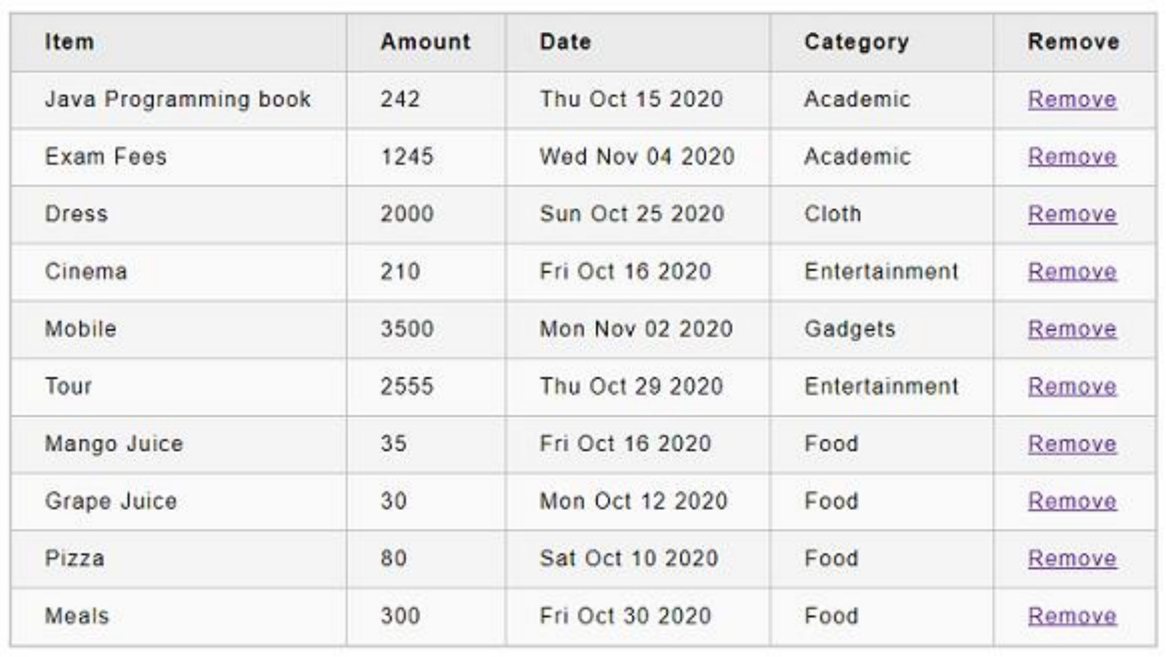
Next, open a new terminal window and start our server application.

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

Next, serve the client application using npm command.

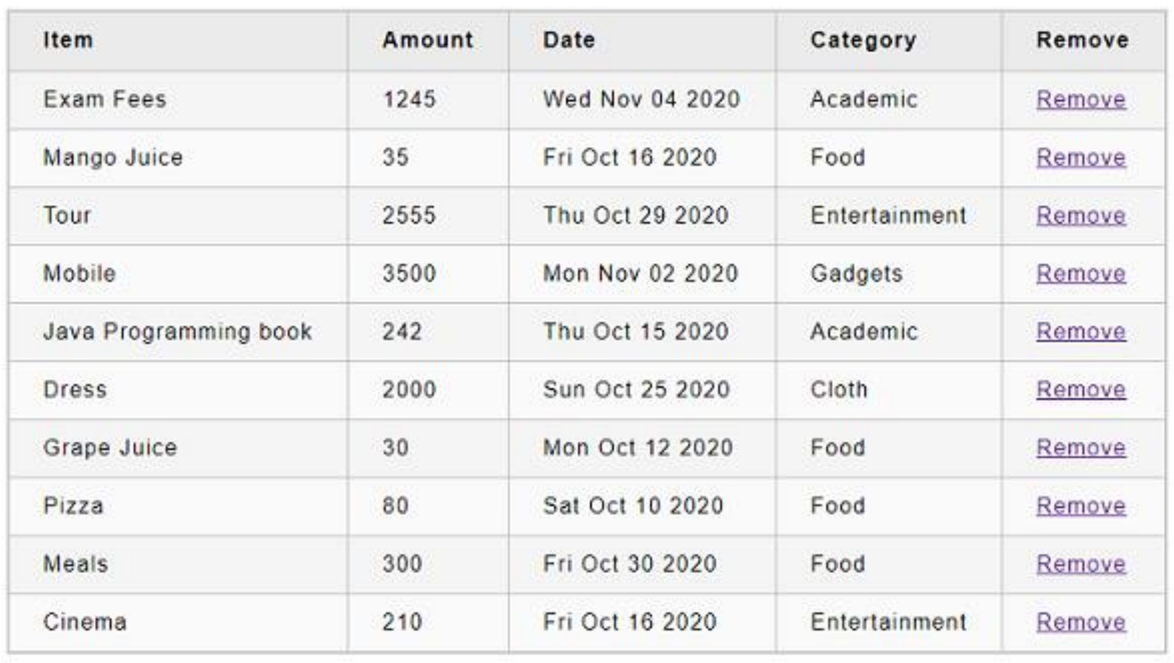
```
npm start
```

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



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

Try to remove the item by clicking the remove link.



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

ReactJS - State Management

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

What is state?

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

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

The simple representation of the state is as follows −

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

ReactJS - State Management API

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

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

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

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

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

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

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

Here,

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

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

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

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

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

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

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

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
});
```

ReactJS - Stateless Component

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

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

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

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

Create a stateful component

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

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

Next, open the application in your favorite editor.

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

Next, create *components* folder under src folder.

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

Next, import *React* library.

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

Next, create *Clock* component.

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

Next, initialize state with current date and time.

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

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

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

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

```
constructor(props) {
  super(props);
  this.state = {
    date: new Date()
  }
  setInterval( () => this.setTime(), 1000);
}
```

Next, create a *render* function.

```
render() {
}
Next, update the render() method to show the current time.
render() {
  return (
    <div><p>The current time is {this.state.date.toString()}</p></div>
  );
}
```

Finally, export the component.

export default Clock;

The complete source code of the Clock component is as follows −

```
import React from 'react';

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    }
    setInterval( () => this.setTime(), 1000);
  }
  setTime() {
    console.log(this.state.date);
    this.setState((state, props) => (
      {
        date: new Date()
      }
    ))
  }
  render() {
    return (
      <div>
        <p>The current time is {this.state.date.toString()}</p>
      </div>
    );
  }
}
export default Clock;
```

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

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

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

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

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

Next, serve the application using npm command.

```
npm start
```

Next, open the browser and enter *http://localhost:3000* in the address bar and press enter. The application will show the time and update it every second.

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

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

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

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

What is mounting?

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

Introduce state in expense manager app

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

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

Next, open *ExpenseEntryItemList.js* file.

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

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

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

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

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

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

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

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

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

    state.items.forEach((item, idx) => {
      if(item.id != id)
        items.push(item)
    })
    let newState = {
      items: items
    }
    return newState;
  })
}
```

Here,

Expense items are fetched from the current state of the component.

Current expense items are looped over to find the item referred by the user using id of the item.

Create a new item list with all the expense item except the one referred by the user

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

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

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

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

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

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

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

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

class ExpenseEntryItemList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      items: this.props.items
    }
    this.handleMouseEnter = this.handleMouseEnter.bind();
    this.handleMouseLeave = this.handleMouseLeave.bind();
    this.handleMouseOver = this.handleMouseOver.bind();
  }
  handleMouseEnter(e) {
    e.target.parentNode.classList.add("highlight");
  }
  handleMouseLeave(e) {
    e.target.parentNode.classList.remove("highlight");
  }
  handleMouseOver(e) {
    console.log("The mouse is at (" + e.clientX + ", " + e.clientY + ")");
  }
  handleDelete = (id, e) => {
    e.preventDefault();
    console.log(id);
    this.setState((state, props) => {
      let items = [];
      state.items.forEach((item, idx) => {
        if(item.id != id)
          items.push(item)
      })
```

```
        let newState = {
          items: items
        }
        return newState;
      })
    }
    getTotal() {
      let total = 0;
      for(var i = 0; i < this.state.items.length; i++) {
        total += this.state.items[i].amount
      }
      return total;
    }
    render() {
      const lists = this.state.items.map((item) =>
        <tr key={item.id} onMouseEnter={this.handleMouseEnter}
onMouseLeave={this.handleMouseLeave}>
          <td>{item.name}</td>
          <td>{item.amount}</td>
          <td>{new Date(item.spendDate).toDateString()}</td>
          <td>{item.category}</td>
          <td><a href="#"
            onClick={(e) =>  this.handleDelete(item.id, e)}>Remove</a></td>
        </tr>
      );
      return (
        <table onMouseOver={this.handleMouseOver}>
          <thead>
            <tr>
              <th>Item</th>
              <th>Amount</th>
              <th>Date</th>
              <th>Category</th>
              <th>Remove</th>
            </tr>
          </thead>
          <tbody>
            {lists}
            <tr>
              <td colSpan="1" style={{ textAlign: "right" }}>Total Amount</td>
              <td colSpan="4" style={{ textAlign: "left" }}>
                {this.getTotal()}
              </td>
            </tr>
          </tbody>
        </table>
      );
    }
}
export default ExpenseEntryItemList;
```

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

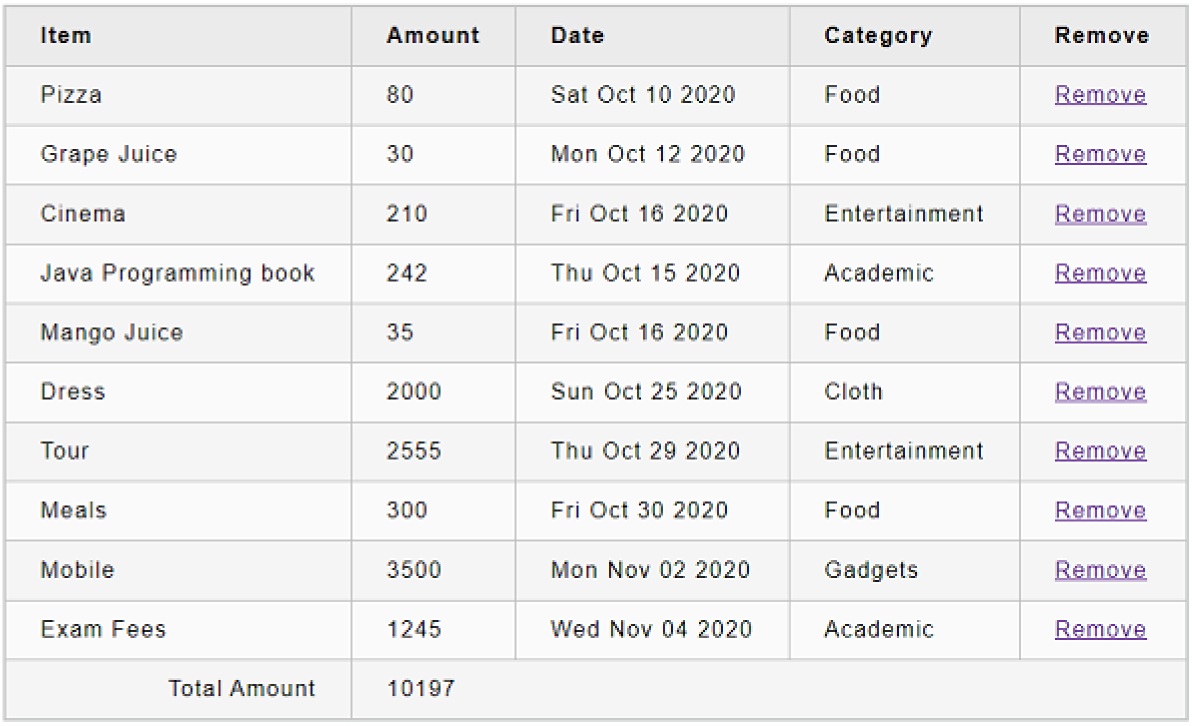
```
import React from 'react';
import ReactDOM from 'react-dom';
import ExpenseEntryItemList from './components/ExpenseEntryItemList'

const items = [
   { id: 1, name: "Pizza", amount: 80, spendDate: "2020-10-10", category: "Food" },
   { id: 2, name: "Grape Juice", amount: 30, spendDate: "2020-10-12", category: "Food" },
   { id: 3, name: "Cinema", amount: 210, spendDate: "2020-10-16", category: "Entertainment" },
   { id: 4, name: "Java Programming book", amount: 242, spendDate: "2020-10-15", category:
"Academic" },
   { id: 5, name: "Mango Juice", amount: 35, spendDate: "2020-10-16", category: "Food" },
   { id: 6, name: "Dress", amount: 2000, spendDate: "2020-10-25", category: "Cloth" },
   { id: 7, name: "Tour", amount: 2555, spendDate: "2020-10-29", category: "Entertainment" },
   { id: 8, name: "Meals", amount: 300, spendDate: "2020-10-30", category: "Food" },
   { id: 9, name: "Mobile", amount: 3500, spendDate: "2020-11-02", category: "Gadgets" },
   { id: 10, name: "Exam Fees", amount: 1245, spendDate: "2020-11-04", category: "Academic" }
]
ReactDOM.render(
   <React.StrictMode>
      <ExpenseEntryItemList items={items} />
   </React.StrictMode>,
   document.getElementById('root')
);
```

Next, serve the application using npm command.

```
npm start
```

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

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



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

ReactJS - Component Life Cycle

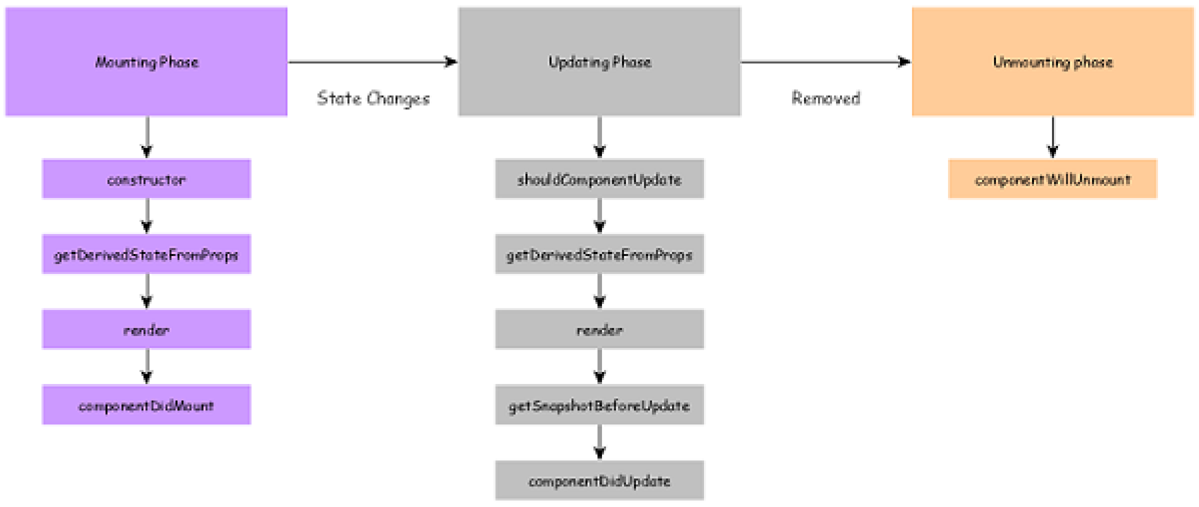
In React, Life cycle of a component represents the different stages of the component during its existence. React provides callback function to attach functionality in each and every stages of the React life cycle. Let us learn the life cycle (and the related API) of a React component in this chapter.

Life cycle API

Each React component has three distinct stages.

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

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



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

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

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

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

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

The signature of the API is as follows −

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

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

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

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

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

The signature is as follows −

shouldComponentUpdate(nextProps, nextState)

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

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

The signature of the API is as follows −

static getDerivedStateFromProps(props, state)

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

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

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

The signature of the API is as follows −

getSnapshotBeforeUpdate(prevProps, prevState)

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

Working example of life cycle API

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

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

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

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

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

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

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

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

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

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

```
import React from 'react';

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    }
  }
  componentDidMount() {
    this.setTimeRef = setInterval(() => this.setTime(), 1000);
  }
  componentWillUnmount() {
    clearInterval(this.setTimeRef)
  }
  setTime() {
    this.setState((state, props) => {
      console.log(state.date);
      return {
        date: new Date()
      }
    })
  }
  render() {
    return (
      <div>
        <p>The current time is {this.state.date.toString()}</p>
      </div>
    );
  }
}
export default Clock;
```

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

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

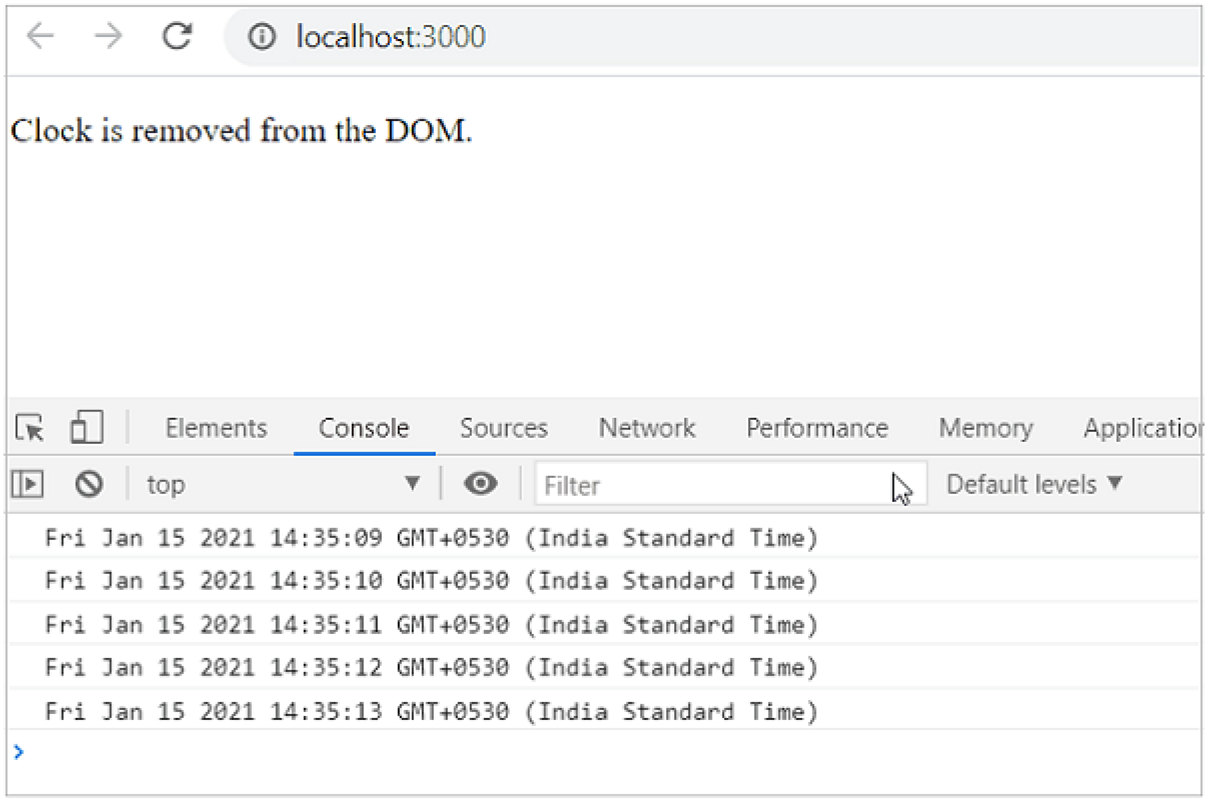
```
ReactDOM.render(
  <React.StrictMode>
    <Clock />
  </React.StrictMode>,
  document.getElementById('root')
);
setTimeout(() => {
  ReactDOM.render(
    <React.StrictMode>
      <div><p>Clock is removed from the DOM.</p></div>
    </React.StrictMode>,
    document.getElementById('root')
  );
},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

Next, serve the application using npm command.

```
npm start
```

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

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



Clock is removed from the DOM.

```
Fri Jan 15 2021 14:35:09 GMT+0530 (India Standard Time)
Fri Jan 15 2021 14:35:10 GMT+0530 (India Standard Time)
Fri Jan 15 2021 14:35:11 GMT+0530 (India Standard Time)
Fri Jan 15 2021 14:35:12 GMT+0530 (India Standard Time)
Fri Jan 15 2021 14:35:13 GMT+0530 (India Standard Time)
```

Life cycle api in Expense manager app

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

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

Next, update ExpenseEntryItemList component with below methods.

```
componentDidMount() {
   console.log("ExpenseEntryItemList :: Initialize :: componentDidMount :: Component mounted");
}
shouldComponentUpdate(nextProps, nextState) {
   console.log("ExpenseEntryItemList :: Update :: shouldComponentUpdate invoked :: Before
update");
   return true;
}
static getDerivedStateFromProps(props, state) {
   console.log("ExpenseEntryItemList :: Initialize / Update :: getDerivedStateFromProps :: Before
update");
   return null;
}
getSnapshotBeforeUpdate(prevProps, prevState) {
   console.log("ExpenseEntryItemList :: Update :: getSnapshotBeforeUpdate :: Before update");
   return null;
}
componentDidUpdate(prevProps, prevState, snapshot) {
   console.log("ExpenseEntryItemList :: Update :: componentDidUpdate :: Component updated");
}
componentWillUnmount() {
   console.log("ExpenseEntryItemList :: Remove :: componentWillUnmount :: Component
unmounted");
}
```

Next, serve the application using npm command.

```
npm start
```

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

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

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

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

```
ExpenseEntryItemList :: Initialize / Update :: getDerivedStateFromProps :: Before update
ExpenseEntryItemList.js:109 ExpenseEntryItemList :: Update :: shouldComponentUpdate invoked
:: Before update
ExpenseEntryItemList.js:121 ExpenseEntryItemList :: Update :: getSnapshotBeforeUpdate ::
Before update
```

```
ExpenseEntryItemList.js:127 ExpenseEntryItemList :: Update :: componentDidUpdate ::
Component updated
```

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

LocalStorage in ReactJS

In this article, we are going to see how to set and retrieve data in the **localStorage** memory of the user's browser in a React application.

LocalStorage is a web storage object to store the data on the user's computer locally, which means the stored data is saved across browser sessions and the data stored has no expiration time.

Syntax

```
// To store data
localStorage.setItem('Name', 'Rahul');


// To retrieve data
localStorage.getItem('Name');


// To clear a specific item
localStorage.removeItem('Name');


// To clear the whole data stored in localStorage
localStorage.clear();
```

Set, retrieve and remove data in localStorage

In this example, we will build a React application which takes the username and password from the user and stores it as an item in the localStorage of the user's computer.

Example

App.jsx

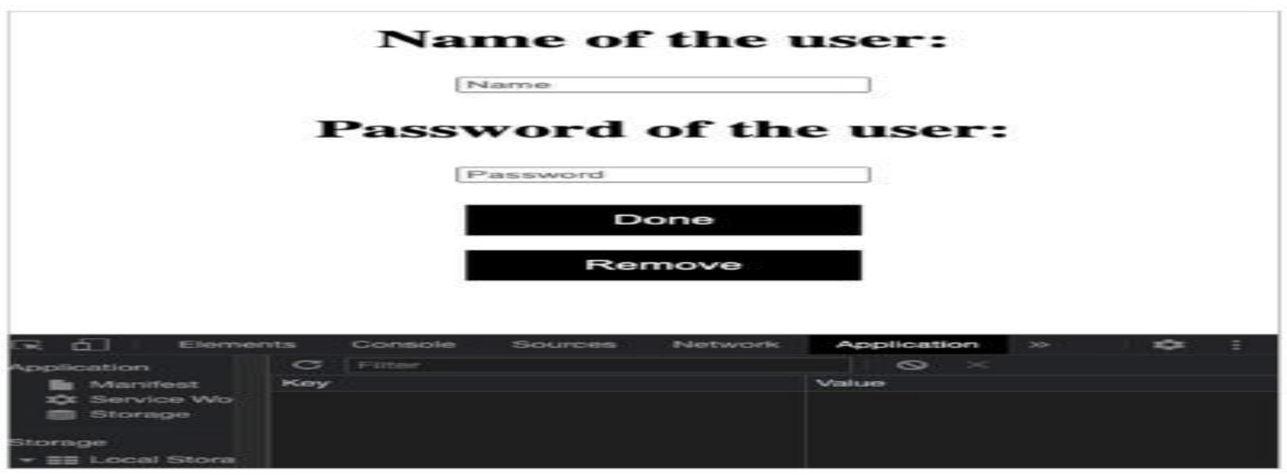
```
import React, { useState } from 'react';
```

```
const App = () => {

  const [name, setName] = useState('');
  const [pwd, setPwd] = useState('');

  const handle = () => {
    localStorage.setItem('Name', name);
    localStorage.setItem('Password', pwd);
  };
  const remove = () => {
    localStorage.removeItem('Name');
    localStorage.removeItem('Password');
  };
  return (
    <div className="App">
      <h1>Name of the user:</h1>
      <input
        placeholder="Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <h1>Password of the user:</h1>
      <input
        type="password"
        placeholder="Password"
        value={pwd}
        onChange={(e) => setPwd(e.target.value)}
      />
      <div>
        <button onClick={handle}>Done</button>
      </div>
      {localStorage.getItem('Name') && (
        <div>
          Name: <p>{localStorage.getItem('Name')}</p>
        </div>
      )}
      {localStorage.getItem('Password') && (
        <div>
          Password: <p>{localStorage.getItem('Password')}</p>
        </div>
      )}
      <div>
        <button onClick={remove}>Remove</button>
      </div>
    </div>
  );
};
export default App;
```

In the above example, when the **Done** button is clicked, the handle function is executed which will set the items in the localStorage of the user and display it. But when the **Remove** button is clicked, the **remove** function is executed which will remove the items from the localStorage.

Output

This will produce the following result.



React Events

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.

Handling events with react have some syntactic differences from handling events on DOM. These are:

1. React events are named as **camelCase** instead of **lowercase**.
2. With JSX, a function is passed as the **event handler** instead of a **string**. For example:

Event declaration in plain HTML:

```
<button onclick="showMessage()">
       Hello JavaTpoint
</button>
```

Event declaration in React:

```
<button onClick={showMessage}>
```

```
    Hello JavaTpoint
</button>
```

3. In react, we cannot return **false** to prevent the **default** behavior. We must call **preventDefault** event explicitly to prevent the default behavior. For example:

In plain HTML, to prevent the default link behavior of opening a new page, we can write:

```
<a href="#" onclick="console.log('You had clicked a Link.'); return false">
    Click_Me
</a>
```

In React, we can write it as:

```
function ActionLink() {
    function handleClick(e) {
        e.preventDefault();
        console.log('You had clicked a Link.');
    }
    return (
        <a href="#" onClick={handleClick}>
              Click_Me
        </a>
    );
}
```

In the above example, e is a **Synthetic Event** which defines according to the **W3C** spec.

Now let us see how to use Event in React.

Example

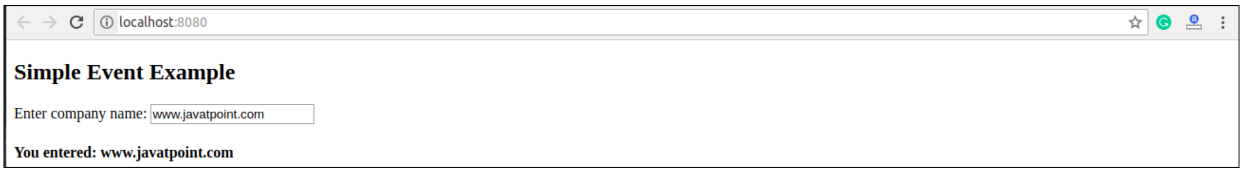
In the below example, we have used only one component and adding an onChange event. This event will trigger the **changeText** function, which returns the company name.

```
import React, { Component } from 'react';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            companyName: ''
        };
    }
    changeText(event) {
        this.setState({
            companyName: event.target.value
        });
    }
    render() {
        return (
            <div>
                <h2>Simple Event Example</h2>
                <label htmlFor="name">Enter company name: </label>
                <input type="text" id="companyName" onChange={this.changeText.bind(this)}/>
                <h4>You entered: { this.state.companyName }</h4>
            </div>
        );
    }
}
export default App;
```

Output

When you execute the above code, you will get the following output.

After entering the name in the textbox, you will get the output as like below screen.



localhost:8080

Simple Event Example

Enter company name: www.javatpoint.com

You entered: www.javatpoint.com

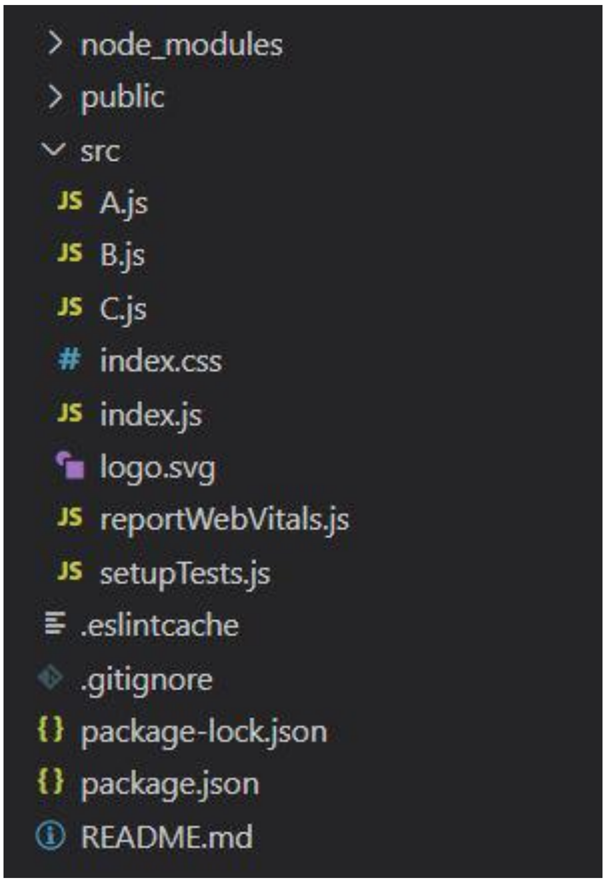
Lifting State up in ReactJS

Lifting up the State: As we know, every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.
Time to use Lift up the State: If the data in "parent and children components" or in "cousin components" is Not in Sync.
Example 1: If we have 2 components in our App. **A -> B** where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.
Example 2: If we have 3 components in our App.

```
     A

    / \

   B   C
```

Where A is the parent of B and C. In this case, If there is some Data only in component B but, component C also wants that data. We know Component C cannot access the data because a component can talk only to its parent or child (Not cousins).

Problem: Let's Implement this with a simple but general example. We are considering the second example.
Complete File Structure:



```
> node_modules
> public
v src
  A.js
  B.js
  C.js
  index.css
  index.js
  logo.svg
  reportWebVitals.js
  setupTests.js
.eslintcache
.gitignore
package-lock.json
package.json
README.md
```

Approach: To solve this, we will Lift the state of component B and component C to component A. Make A.js as our Main Parent by changing the path of App in the index.js file

Before:

```
import App from './App';
```

After:

```
import App from './A';
```

Filename- A.js:

- Javascript

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

import B from './B'

import C from './C'
```

```
class A extends Component {


  constructor(props) {

    super(props);

    this.handleTextChange = this.handleTextChange.bind(this);

    this.state = {text: ''};

  }


  handleTextChange(newText) {

    this.setState({text: newText});

  }


  render() {

    return (

        <React.Fragment>

          <B text={this.state.text}

             handleTextChange={this.handleTextChange}/>

          <C text={this.state.text} />

        </React.Fragment>

    );

  }
}



export default A;
```

Filename- B.js:

- Javascript

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



class B extends Component {



constructor(props) {

    super(props);

    this.handleTextChange = this.handleTextChange.bind(this);

}



handleTextChange(e){

    this.props.handleTextChange(e.target.value);

}



render() {

    return (
        <input value={this.props.text}

               onChange={this.handleTextChange} />

    );

}

}



export default B;
```

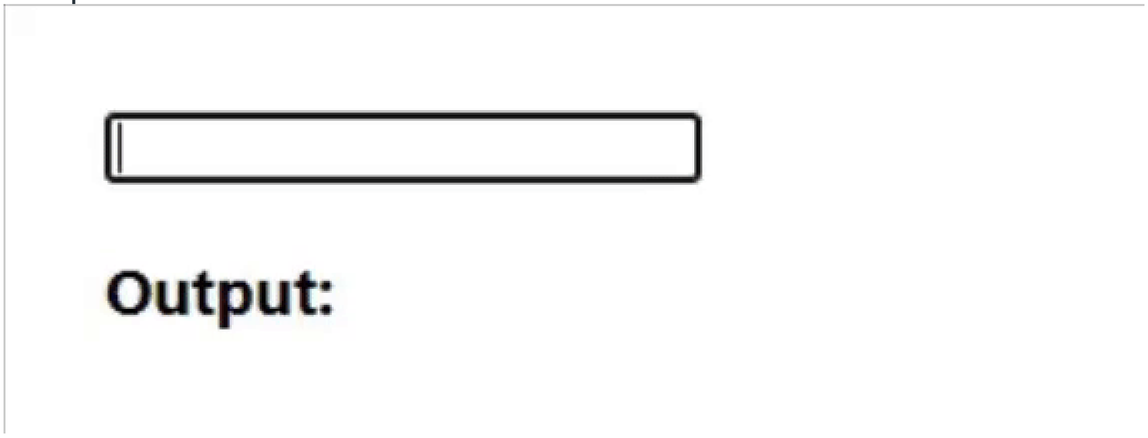
Filename- C.js:

- Javascript

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



class C extends Component {



render() {

    return (

        <h3>Output: {this.props.text}</h3>

    );

}

}
```

```
export default C;
```

Output: Now, component C can Access text in component B through component A.



Output:

Composition vs Inheritance

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

In this section, we will consider a few problems where developers new to React often reach for inheritance, and show how we can solve them with composition.

Containment

Some components don't know their children ahead of time. This is especially common for components like `Sidebar` or `Dialog` that represent generic "boxes". We recommend that such components use the special `children` prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}    </div>
  );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">        Welcome      </h1>      <p
className="Dialog-message">        Thank you for visiting our spacecraft!
</p>    </FancyBorder>
  );
}
```

Try it on CodePen

Anything inside the `<FancyBorder>` JSX tag gets passed into the `FancyBorder` component as a `children` prop.
Since `FancyBorder` renders `{props.children}` inside a `<div>`, the passed elements appear in the final output.
While this is less common, sometimes you might need multiple "holes" in a component. In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}      </div>
      <div className="SplitPane-right">
        {props.right}      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />      }
      right={
        <Chat />      } />
  );
}
```

Try it on CodePen

React elements like `<Contacts />` and `<Chat />` are just objects, so you can pass them as props like any other data. This approach may remind you of "slots" in other libraries but there are no limitations on what you can pass as props in React.

Specialization

Sometimes we think about components as being "special cases" of other components. For example, we might say that a `WelcomeDialog` is a special case of `Dialog`.

In React, this is also achieved by composition, where a more "specific" component renders a more "generic" one and configures it with props:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}      </h1>
      <p className="Dialog-message">
        {props.message}      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog      title="Welcome"      message="Thank you for visiting our
spacecraft!" />  );
}
```

Try it on CodePen

Composition works equally well for components defined as classes:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
              message="How should we refer to you?">
        <input value={this.state.login}
onChange={this.handleChange} />        <button onClick={this.handleSignUp}>
Sign Me Up!        </button>      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

Try it on CodePen

So What About Inheritance?

At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.