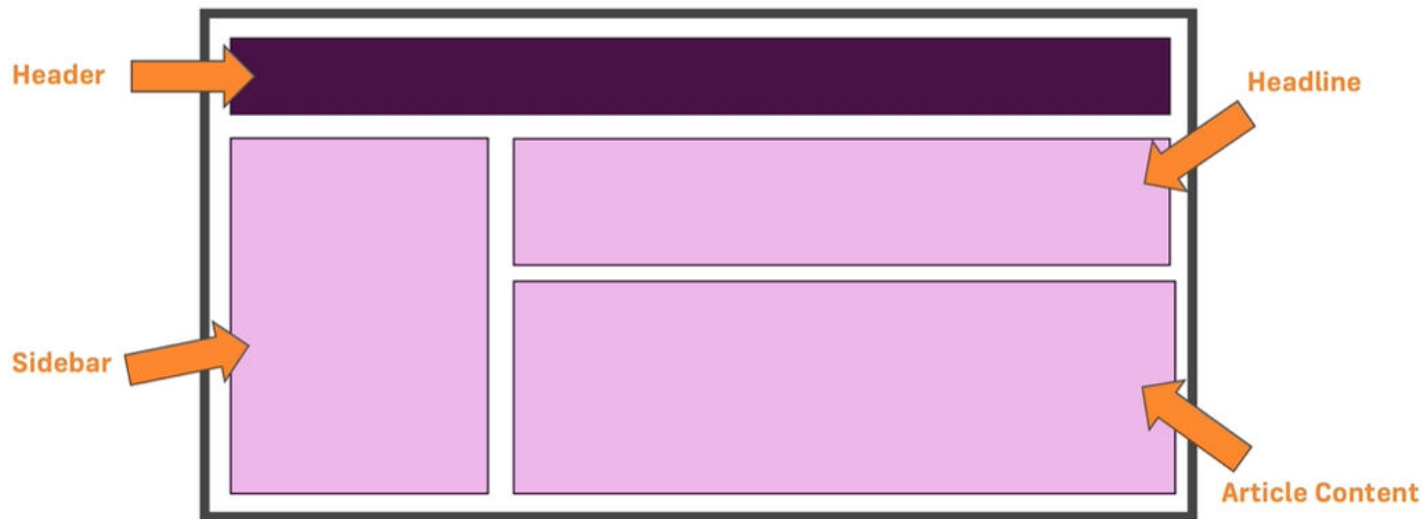


REACT.JS

JavaScript library for building user interfaces

What is React?

- From the official React page : A JavaScript library for building user interfaces
- Its not a framework; React does only one thing – create awesome UI!
- React is used to build single page applications.
- React.js is a JavaScript library. It was developed by engineers at Facebook.
- React is a declarative, efficient, and flexible JavaScript library for building user interfaces.
- It lets you compose complex UIs from small and isolated pieces of code called “components”.

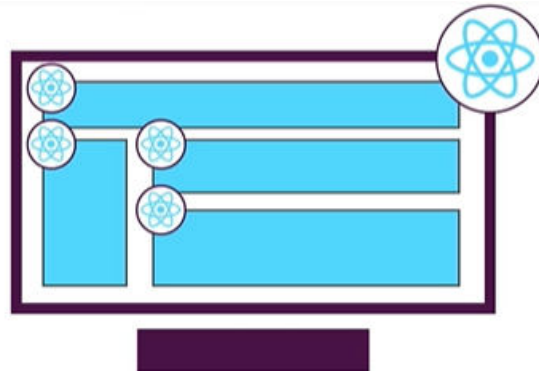


Client-side Javascript frameworks

- [Ember](#) : was initially released in December 2011. It is an older framework that has less users than more modern alternatives such as React and Vue
- [Angular](#) is an open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations.
- [Vue](#) : first released in 2014; is the youngest of the big four, but has enjoyed a recent uptick in popularity.
- [React](#) : released by Facebook in 2013. By this point, FB had already been using React to solve many of its problems internally.
 - React itself is not technically a framework; it's a library for rendering UI components.
 - React is used in combination with other libraries to make applications — React and [React Native](#) enable developers to make mobile applications; React and [ReactDOM](#) enable them to make web applications, etc.

Components

- A **Component** is one of the core building blocks of React.
- Its just a **custom HTML element!**
- Every application you will develop in React will be made up of pieces called components.
 - Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.



Why react

- Created and maintained by facebook
- Has a huge community on Github
- Component based architecture
- React is *fast*. Apps made in React can handle complex updates and still feel quick and responsive.
- React is *modular*. Instead of writing large, dense files of code, you can write many smaller, reusable files. React's modularity can be a beautiful solution to JavaScript's maintainability problems.
- React is *scalable*. Large programs that display a lot of changing data are where React performs best.
- React is *popular*.
- UI state becomes difficult to manage with vanilla Javascript

Requirements

- Ensure that NodeJS and typescript are installed
 - Install TypeScript as follows:
 - `npm install -g typescript`

```
C:\Users\Shrilata>node --version  
v14.16.0
```

```
C:\Users\Shrilata>npm --version  
6.14.11
```

```
C:\Users\Shrilata>tsc --version  
Version 4.4.3
```

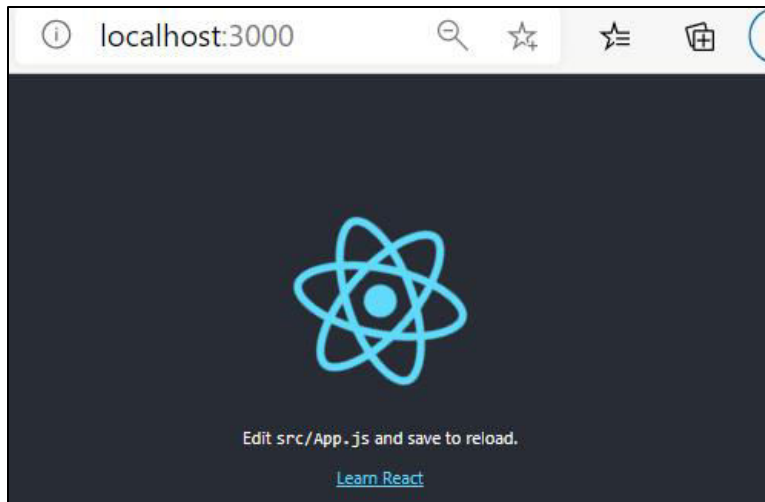
```
C:\Users\Shrilata>npx --version  
6.14.11
```

Using create-react app

- `npx create-react-app my-app`
- `cd my-app`
- `npm start`

Create React App is a comfortable environment for **learning React**, and is the best way to start building **a new single-page application** in React.

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You'll need to have Node `>= 14.0.0` and npm `>= 5.6` on your machine. To create a project, run:



Understanding the folder structure

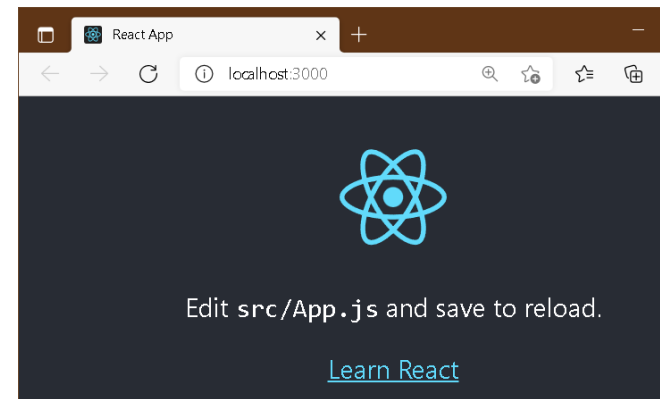
✓ MY-APP

- > node_modules
- > public
- > src
- 🔍 .gitignore
- { } package-lock.json
- { } package.json
- 📖 README.md

✓ public

- ★ favicon.ico
- <> index.html
- 🖼️ logo192.png
- 🖼️ logo512.png
- { } manifest.json
- ☰ robots.txt
- > src

```
"dependencies": {
  "@testing-library/jest-dom": "^5.11.6",
  "@testing-library/react": "^11.2.2",
  "@testing-library/user-event": "^12.2.2",
  "react": "^17.0.1",
  "react-dom": "^17.0.1",
  "react-scripts": "4.0.0",
  "web-vitals": "^0.2.4"
},
  > Debug
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```



```
E:\FreelanceTrg\ReactJS\Demo\my-app>npm start

> my-app@0.1.0 start E:\FreelanceTrg\ReactJS\Demo\my-app
> react-scripts start

i [wds]: Project is running at http://192.168.1.18/
i [wds]: webpack output is served from
i [wds]: Content not from webpack is served from E:\FreelanceTrg\ReactJS\Demo\my-app
i [wds]: 404s will fallback to /
starting the development server...
Compiled successfully!

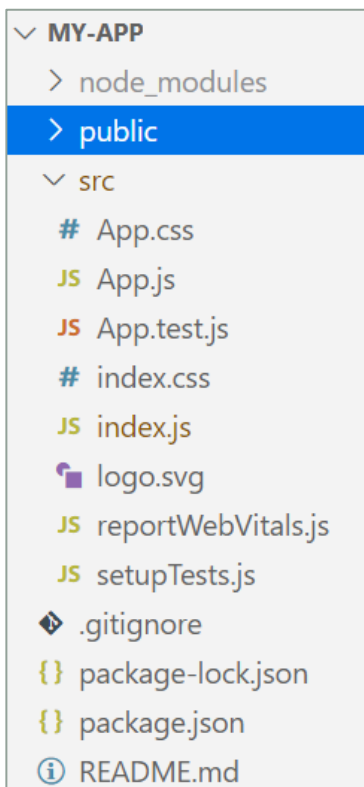
You can now view my-app in the browser.
```


Index.html

```
<> index.html > ...
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!-- ...
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!-- ...
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!-- ...
  </body>
</html>
```

- The root node is the HTML element where you want to display the result.
- It is like a container for content managed by React.
- It does NOT have to be a <div> element and it does NOT have to have the id='root'

Understanding the folder structure

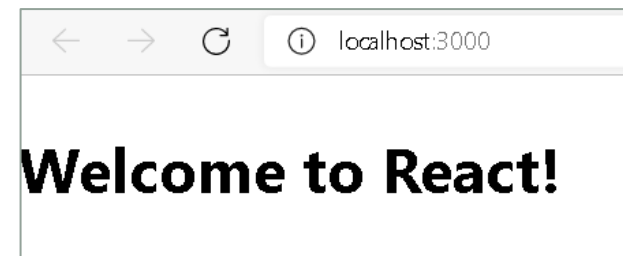


```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit <code>src/App.js</code> and save to reload.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn React  
        </a>  
      </header>  
    </div>  
  );  
}  
export default App;
```

```
function App() {  
  return (  
    <div>  
      <h2>Welcome to React!</h2>  
    </div>  
  );  
}  
export default App;
```

```
index.js  
import ReactDOM from 'react-dom';  
import './index.css';  
import App from './App';  
  
ReactDOM.render(<App />, document.getElementById('root'));
```

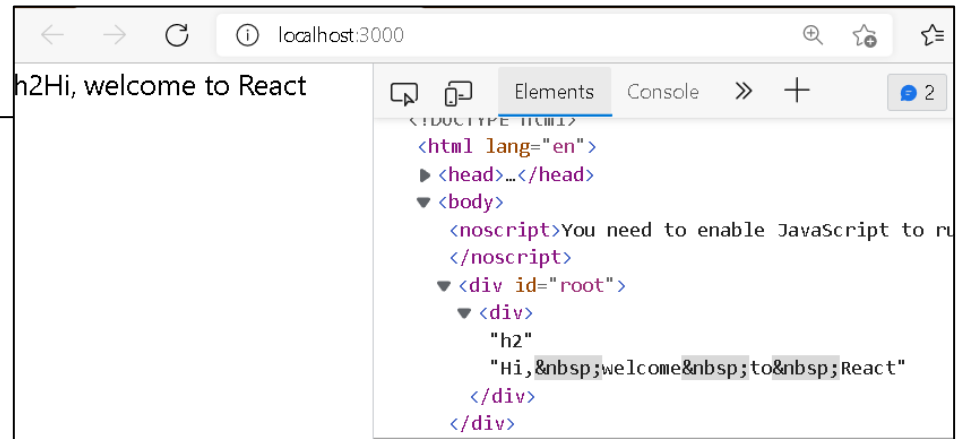
```
App.css > .App  
.App {  
  text-align: center;  
}
```



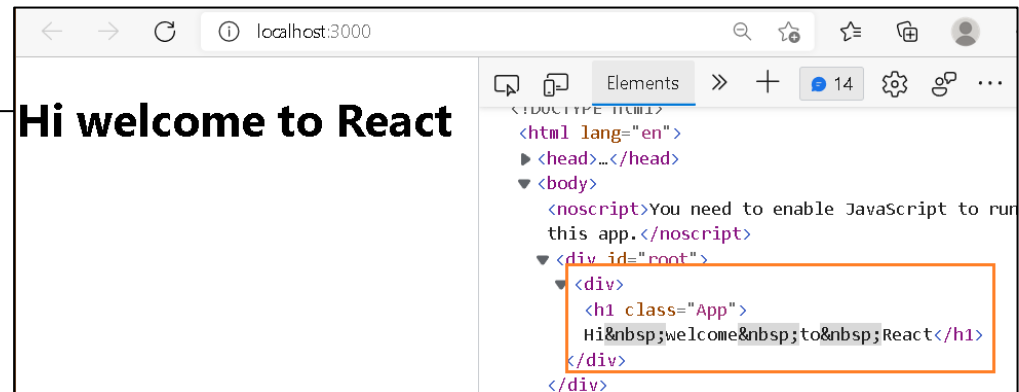
Understanding JSX

```
import React from 'react';

function App() {
  /*return (
    <div>
      <h2>Welcome to React!</h2>
    </div>
  );*/
  return React.createElement('div', null, 'h2', 'Hi, welcome to React');
}
export default App;
```



```
function App() {
  /*return (
    <div>
      <h2>Welcome to React!</h2>
    </div>
  );*/
  return React.createElement('div', null,
    React.createElement('h1', {className: 'App'}, 'Hi welcome to React'));
}
export default App;
```



Understanding JSX

- Eg : `const mytag = <h1>Hello React!</h1>;`

```
const myelement = <h1>Understanding JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

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

```
ReactDOM.render(myelement, document.getElementById('root'));
```

```
function App() {
  return (JSX attribute) React.HTMLAttributes<HTMLDivElement>.className?: string
  <div className="App">
    <h2>Welcome to React!</h2>
  </div>
);
}
export default App;
```

```
▼ <div class="App">
  <h1> Hi, welcome to React</h1>
</div>
```

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

```
ReactDOM.render(myelement, document.getElementById('root'));
```

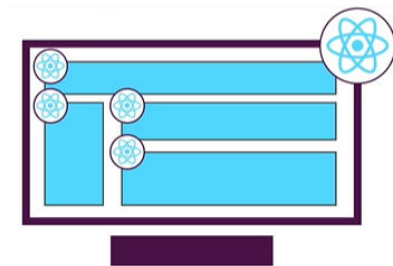
If we want to return more elements, we need to wrap it with one container element. Notice how we are using div as a wrapper for the two h1 elements.

Creating a functional component

- Components are the essential building blocks of any app created with React
 - A single app most often consists of many components.
- A component is in essence, a piece of the UI - splitting the user interface into reusable and independent parts, each of which can be processed separately.
 - Components are independent and reusable bits of code.
 - It's an encapsulated piece of logic.
 - They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render function.

```
function ExpenseItem(){  
  return <h2>Expense Item</h2>  
}  
export default ExpenseItem;
```

```
const ExpenseItem = () => {  
  return <h2>Expense Item</h2>  
}  
export default ExpenseItem;
```

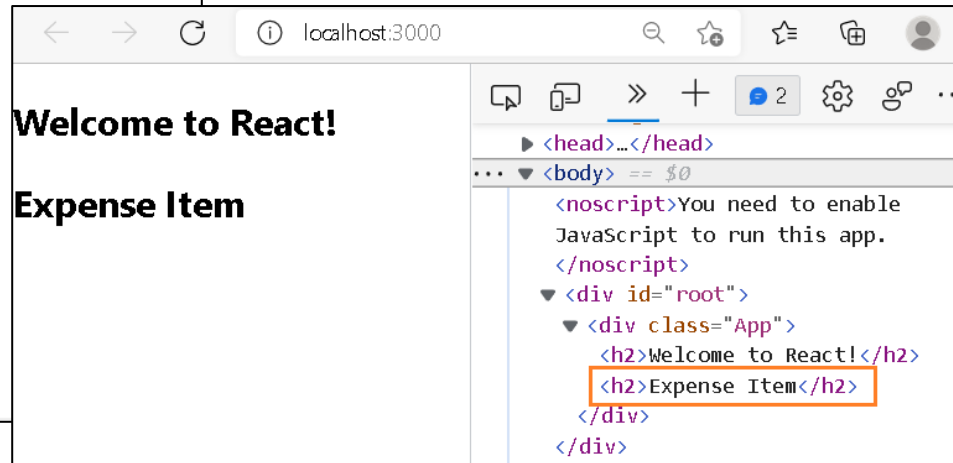


Creating a functional component

- Adding our component to main component
 - To use this component in your application, use similar syntax as normal HTML

```
App.js > ...
import React from 'react';
import ExpenseItem from '../components/ExpenseItem';

function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
    </div>
  );
}
export default App;
```



- Creating components makes them reusable and configurable.
- Reusing is simple. Eg, simply copy paste `<ExpenseItem />` multiple times in App.js.

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
      <ExpenseItem />
      <ExpenseItem />
    </div>
  );
}
```

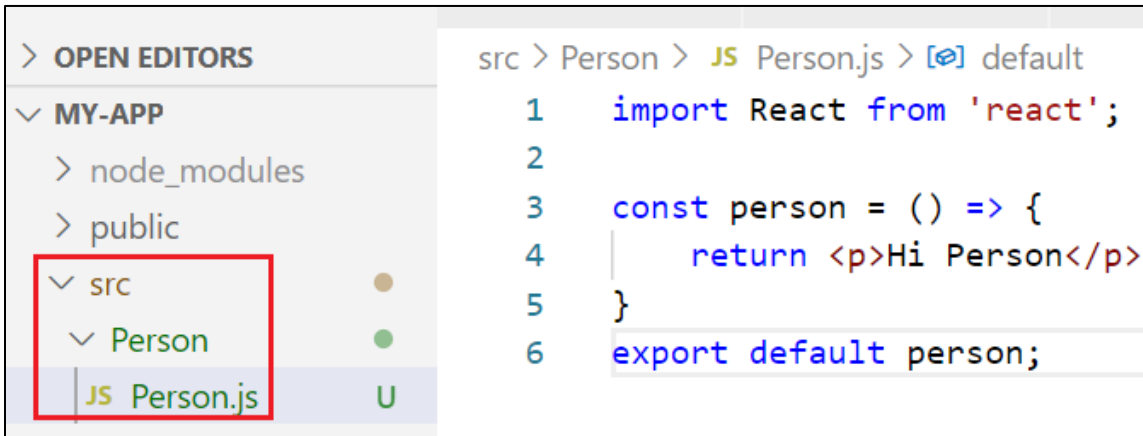
Welcome to React!

Expense Item

Expense Item

Expense Item

Another example



The image shows a VS Code interface. On the left, the 'EXPLORER' sidebar is open, showing a project structure with 'MY-APP' containing 'node_modules', 'public', 'src', and 'Person'. The 'Person' folder is expanded, showing 'JS Person.js'. On the right, the 'src > Person > JS Person.js' file is open in the editor, showing the following code:

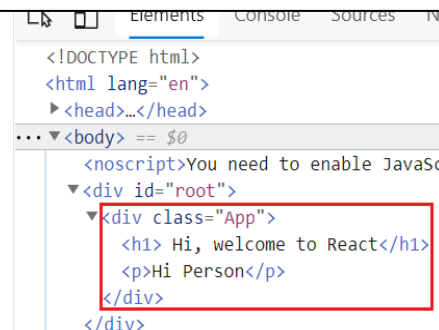
```
src > Person > JS Person.js > [🔗] default
1  import React from 'react';
2
3  const person = () => {
4    |    return <p>Hi Person</p>
5  }
6  export default person;
```

```
import React, {Component} from 'react';
import './App.css';
import Person from './Person/Person';
function App() {
  return (
    <div className="App">
      <h1> Hi, welcome to React</h1>
      <Person />
    </div>
  );
}
export default App;
```

```
return (
  <div className="App">
    <h1> Hi, welcome to React</h1>
    <Person />
    <Person />
    <Person />
  </div>
);
```

Hi, welcome to React

Hi Person



The image shows the 'Elements' panel in a browser's developer tools. It displays the rendered HTML structure of the application. The root element is a <div id="root">, which contains a <div class="App">. Inside the <div class="App">, there is an <h1> Hi, welcome to React</h1> and a <p>Hi Person</p>. The <div class="App"> and its children are highlighted with a red box.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body> == $0
    <noscript>You need to enable JavaSc
    <div id="root">
      <div class="App">
        <h1> Hi, welcome to React</h1>
        <p>Hi Person</p>
      </div>
    </div>
```

Making our functional component more complex

```
import "../ExpenseItem.css"

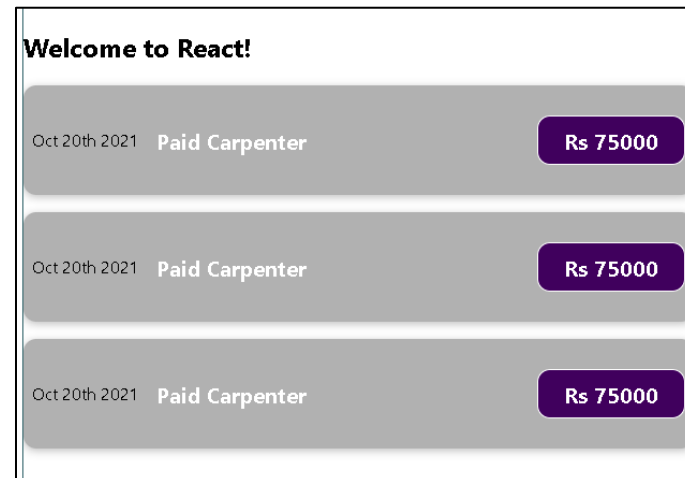
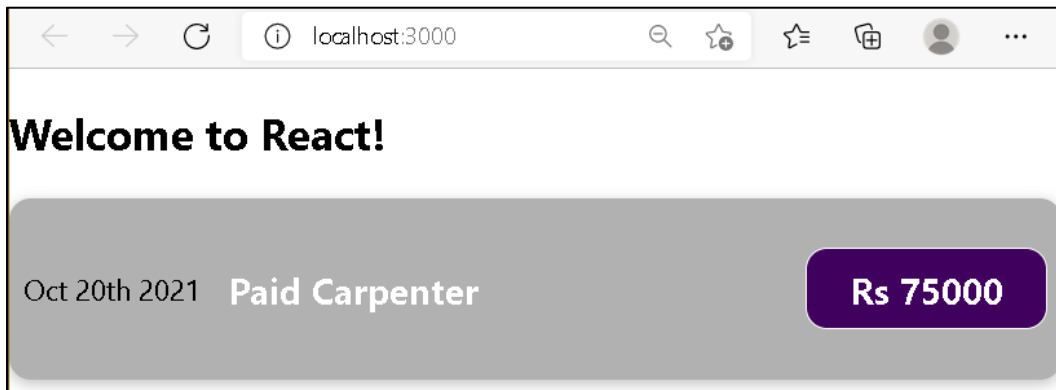
const ExpenseItem = () => {
  return (
    <div className="expense-item">
      <div>Oct 20th 2021</div>
      <div className="expense-item__description">
        <h2>Paid Carpenter</h2>
        <p className="expense-item__price">Rs 75000</p>
      </div>
    </div>
  )
}
export default ExpenseItem;
```

```
.expense-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
  padding: 0.5rem;
  margin: 1rem 0;
  border-radius: 12px;
  background-color: #4b4b4b6e;
}

.expense-item__description { ... }

.expense-item h2 { ... }

.expense-item__price { ... }
```



Components & JSX

- When creating components, you have the choice between two different ways:
- **Functional components** (also referred to as "presentational", "dumb" or "stateless" components)

```
const cmp = () => {  
  return <div>some JSX</div>  
}
```

using ES6 arrow functions as shown here is recommended but optional

- **class-based components** (also referred to as "containers", "smart" or "stateful" components)

```
class Cmp extends Component {  
  render () {  
    return <div>some JSX</div>  
  }  
}
```

Outputting dynamic content

- If we have some dynamic content in our jsx part which we want to run as javaScript code and not interpret as text, we have to wrap it in single curly braces.

```
import "../ExpenseItem.css"

const ExpenseItem = () => {

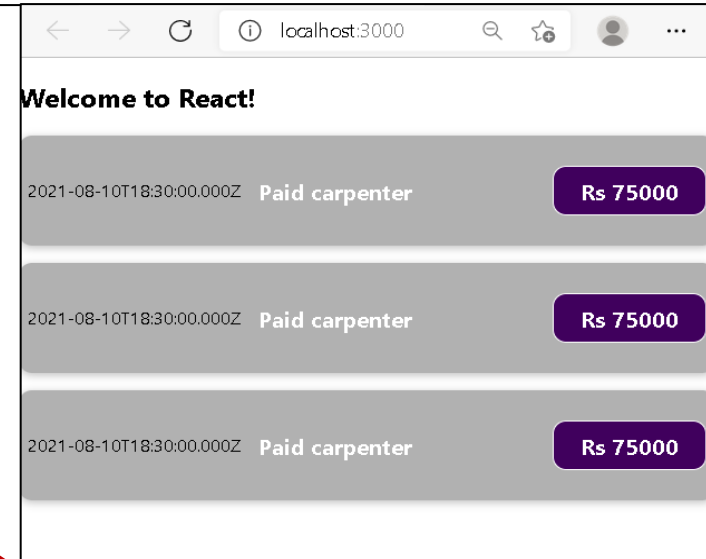
  const expDate = new Date(2021, 7, 11);
  const expTitle = "Paid carpenter";
  const expAmount = 75000

  return (
    <div className="expense-item">

      {/* single and multiline comments in JSX */}

      <div>{expDate.toISOString()}</div>
      <div className="expense-item__description">
        <h2>{expTitle}</h2>
        <p className="expense-item__price">Rs {expAmount}</p>
      </div>
    </div>
  )
}

export default ExpenseItem;
```



Comments in JSX

Outputting dynamic content – another example

```
//Person.js
import React from 'react';

const person = () => {
  return <p>Hi Person i am {Math.floor(Math.random() * 30)} years old</p>
}
export default person;
```

Wrap dynamic
content in JSX in {...}

```
function App() {
  return (
    <div className="App">
      <h1> Hi, welcome to React</h1>
      <Person />
      <Person />
      <Person />
    </div>
  );
}
```

Hi, welcome to React

Hi Person i am 27 years old

Hi Person i am 29 years old

Hi Person i am 12 years old

PROPS

Passing data via 'Props'

- “Props” stands for properties.
 - It is a special keyword in React used for passing data from one component to another.
 - Props are arguments passed into React components.
 - props are read-only. So, the data coming from a parent component can't be changed by the child component.
 - Props are passed to components via HTML attributes.
 - Props can be used to pass any kind of data such as: String, Array, Integer, Boolean, Objects or, Functions

```
import HelloComponent from "./HelloComponent";

const MessageComponent = () => {
  return(
    <div>
      <HelloComponent name="Shrilata" />
    </div>
  );
}
export default MessageComponent;
```

```
const HelloComponent = (props) => {
  return (<h3>Hello, welcome {props.name}</h3>)
}
export default HelloComponent;
```



Passing data via 'Props'

```
function App() {  
  return (  
    <div className="App">  
      <h2>Welcome to React!</h2>  
      <ExpenseItem expDate="20-12-2020" expTitle="Myntra shopping" expAmount="2500"/>  
      <ExpenseItem expDate="21-12-2020" expTitle="Microwave" expAmount="8000"/>  
    </div>  
  );  
}  
export default App;
```

you can use as many
props as you like

```
const ExpenseItem = (props) => {  
  return (  
    <div className="expense-item">  
      <div>{props.expDate}</div>  
      <div className="expense-item__description">  
        <h2>{props.expTitle}</h2>  
        <p className="expense-item__price">Rs {props.expAmount}</p>  
      </div>  
    </div>  
  )  
}  
export default ExpenseItem;
```

Welcome to React!

20-12-2020 Myntra shopping

Rs 2500

21-12-2020 Microwave

Rs 8000

React uses 'props' to pass attributes from
'parent' component to 'child' component.

Working with props

```
function App() {  
  return (  
    <div className="App">  
      <h1> Hi, welcome to React</h1>  
      <Person name="Shri" age="20"/>  
      <Person name="Soha" age="23">Hobbies : Coding</Person>  
      <Person name="sandeep" age="45"/>  
    </div>  
  );  
}
```

Hi, welcome to React

Hi i am Shri and i am 20 years old

Hi i am Soha and i am 23 years old

Hi i am sandeep and i am 45 years old

```
//Person.js  
import React from 'react';  
  
const person = (props) => {  
  return <p>Hi i am {props.name} and i am {props.age} years old</p>  
}  
export default person;
```

React uses 'props' to pass attributes from 'parent' component to 'child' component.

Working with props

```
function App() {
  const expenses = [
    {
      title: 'Groceries',
      amount: 900,
      date: new Date(2020, 7, 14),
    },
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'SofaSet', amount: 25000, date: new Date(2021, 2, 28),
    }
  ];

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem expDate={expenses[0].date} expTitle={expenses[0].title}
expAmount={expenses[0].amount}/>
      <ExpenseItem expDate={expenses[1].date} expTitle={expenses[1].title}
expAmount={expenses[1].amount}/>
      <ExpenseItem expDate={expenses[2].date} expTitle={expenses[2].title}
expAmount={expenses[2].amount}/>
    </div>
  );
}
export default App;
```

Welcome to React!

2020-08-13T18:30:00.000Z Groceries

Rs 900

2021-03-11T18:30:00.000Z New TV

Rs 34000

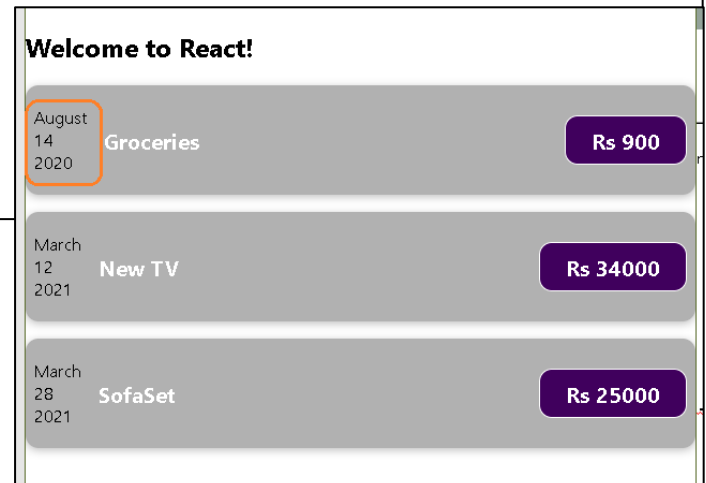
2021-03-27T18:30:00.000Z SofaSet

Rs 25000

“Javascript” in components

```
const ExpenseItem = (props) => {
  const month = props.expDate.toLocaleString('en-US', {month: 'long'});
  const day = props.expDate.toLocaleString('en-US', {day: '2-digit'});
  const year = props.expDate.getFullYear();

  return (
    <div className="expense-item">
      <div>
        <div>{month}</div>
        <div>{day}</div>
        <div>{year}</div>
      </div>
      <div className="expense-item__description">
        <h2>{props.expTitle}</h2>
        <p className="expense-item__price">Rs {props.expAmount}</p>
      </div>
    </div>
  )
}
export default ExpenseItem;
```



```
import ExpenseDate from "../ExpenseDate";
```

```
const ExpenseItem = (props) => {
```

```
  return (
```

```
    <div className="expense-item">
```

```
      <ExpenseDate date={props.expDate}/>
```

```
      <div className="expense-item__description">
```

```
        <h2>{props.expTitle}</h2>
```

```
        <p className="expense-item__price">Rs {props.expAmount}</p>
```

```
      </div>
```

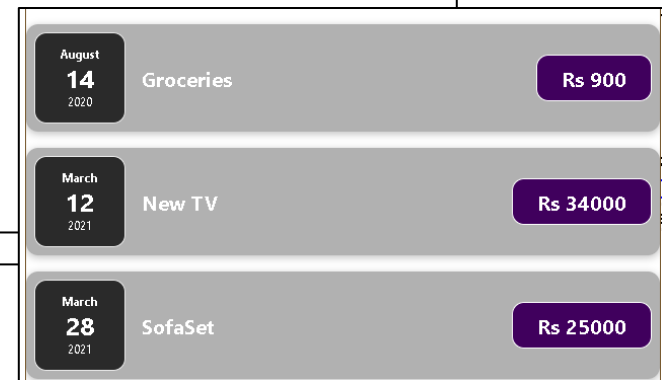
```
    </div>
```

```
  )
```

```
}
```

```
export default ExpenseItem;
```

Splitting components further



```
import "../ExpenseDate.css"
```

```
const ExpenseDate = (props) => {
```

```
  const month = props.date.toLocaleString('en-US', {month: 'long'});
```

```
  const day = props.date.toLocaleString('en-US', {day: '2-digit'});
```

```
  const year = props.date.getFullYear();
```

```
  return (
```

```
    <div className="expense-date">
```

```
      <div className="expense-date__month">{month}</div>
```

```
      <div className="expense-date__day">{day}</div>
```

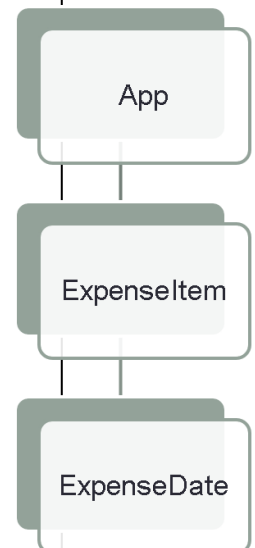
```
      <div className="expense-date__year">{year}</div>
```

```
    </div>
```

```
  );
```

```
}
```

```
export default ExpenseDate;
```

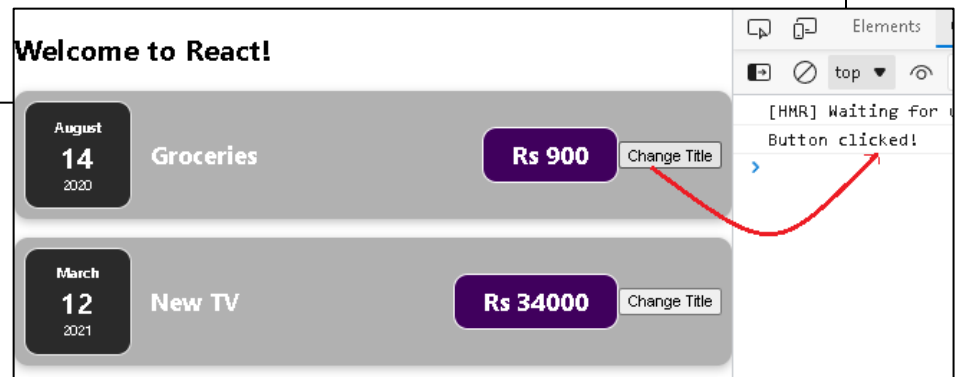


EVENTS AND EVENT HANDLING

Listening to events and working with event handlers

```
const ExpenseItem = (props) => {  
  
  let btnHandler = () => {  
    console.log("Button clicked!")  
  }  
  
  return (  
    <div className="expense-item">  
      <ExpenseDate date={props.expDate}/>  
      <div className="expense-item__description">  
        <h2>{props.expTitle}</h2>  
        <p className="expense-item__price">Rs {props.expAmount}</p>  
      </div>  
      <button onClick={btnHandler}>Change Title</button>  
    </div>  
  )  
}  
export default ExpenseItem;
```

No parenthesis ()



STATEFUL COMPONENTS

React State

- The state is a built-in React object that is used to contain data or information about the component.
- A component's state can change over time; whenever it changes, the component re-renders.
 - The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.
- A component with state is known as stateful component.
- State allows us to create components that are dynamic and interactive.
 - State is private, it must not be manipulated from the outside.
 - Also, it is important to know when to use 'state', it is generally used with data that is bound to change.

Component without state

```
const ExpenseItem = (props) => {  
  let title = props.expTitle;  
  
  let btnHandler = () => {  
    title = "updated expense"  
    console.log("Button clicked!")  
  }  
  
  return (  
    <div className="expense-item">  
      <ExpenseDate date={props.expDate}/>  
      <div className="expense-item__description">  
        <h2>{title}</h2>  
        <p className="expense-item__price">Rs {props.expAmount}</p>  
      </div>  
      <button onClick={btnHandler}>Change Title</button>  
    </div>  
  )  
}  
export default ExpenseItem;
```

React Hooks

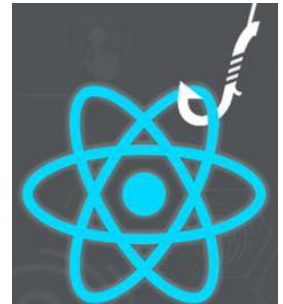
- Hooks allow us to "hook" into React features such as state and lifecycle methods
 - React Hooks are special functions provided by React to handle a specific functionality inside a React functional component.
 - Eg React provides *useState()* function to manage state in a functional component.
 - When a React functional component uses React Hooks, React Hooks attach itself into the component and provides additional functionality.
- You must import Hooks from react
 - Eg : `import React, { useState } from "react";` Here - `useState` is a Hook to keep track of the application state.
- There are some rules for hooks:
 - Hooks can only be called inside React function components.
 - Hooks can only be called at the top level of a component.
 - Hooks cannot be conditional
 - Hooks will not work in React class components.
 - If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks

Working with “state” in functional component

- The React useState Hook allows us to track state in a function component.
- To use the useState Hook, we first need to import it into our component.
 - `import { useState } from "react";`
 - We initialize our state by calling `useState` in our function component.

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

const UseStateComponent = () => {
  useState(); //hooks go here
}
```



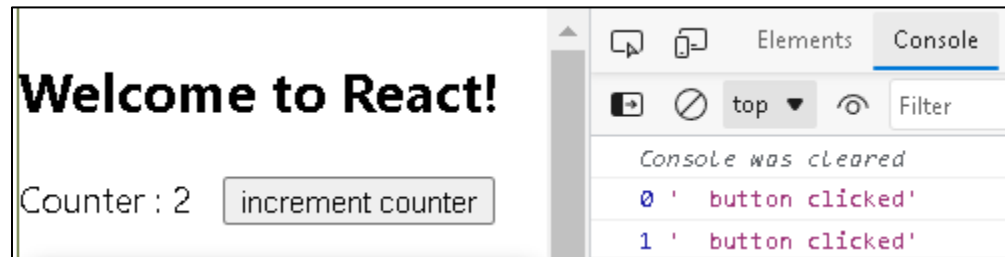
- `useState` accepts an initial state and returns two values:
 1. The current state.
 2. A function that updates the state.

▪ Eg:

```
function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

- The first value, `color`, is our current state.
- The second value, `setColor`, is the function that is used to update our state.
- Lastly, we set the initial state to an empty string: `useState("")`

Working with “state” in functional component

[illegible]

Working with “state” in functional component

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

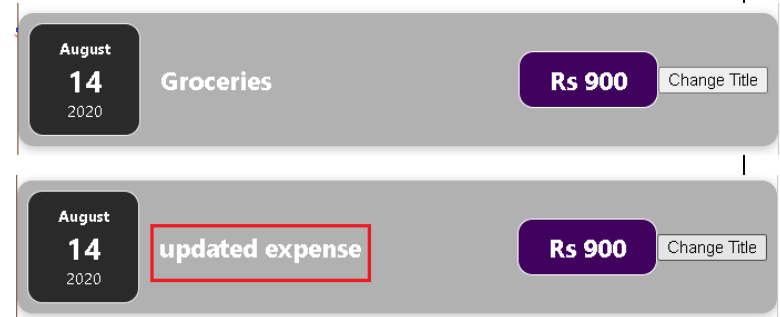
const ExpenseItem = (props) => {

  const [title, setTitle] = useState(props.expTitle);

  let btnHandler = () => {
    setTitle("updated expense")
    console.log("Button clicked!")
  }

  return (
    <div className="expense-item">
      <ExpenseDate date={props.expDate}/>
      <div className="expense-item__description">
        <h2>{title}</h2>
        <p className="expense-item__price">Rs {props.expAmount}</p>
      </div>
      <button onClick={btnHandler}>Change Title</button>
    </div>
  )
}

export default ExpenseItem;
```



props and state

- props and state are CORE concepts of React.
 - Actually, only changes in props and/ or state trigger React to re-render your components and potentially update the DOM in the browser
- Props : allow you to pass data from a parent (wrapping) component to a child component.
 - Eg : AllPosts Component : “title” is the custom property (prop) set up on the custom Post component.
 - Post Component: receives the props argument. React will pass one argument to your component function; an object, which contains all properties you set up on <Post ... /> .
 - {props.title} then dynamically outputs the title property of the props object - which is available since we set the title property inside AllPosts component

//AllPosts

```
const posts = () => {  
  return (  
    <div>  
      <Post title="My first Post" />  
      <Post title="My second Post" />  
    </div>  
  );  
}
```

//Post

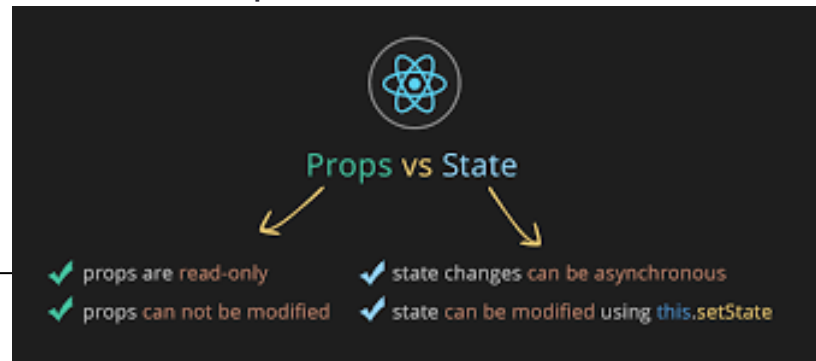
```
const post = (props) => {  
  return (  
    <div>  
      <h1>{props.title}</h1>  
    </div>  
  );  
}
```

props and state

- State : While props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component's, well, state from within.
 - Changes to state also trigger an UI update.
 - Example: NewPost Component: this component contains state . Only class-based components can define and use state . You can of course pass the state down to functional components, but these then can't directly edit it.

```
class NewPost extends Component { // state can only be accessed in class-based components!
  state = {
    counter: 1
  };

  render () { // Needs to be implemented in class-based components! Return some JSX!
    return (
      <div>{this.state.counter}</div>
    );
  }
}
```



props and state

- Props are immutable.
 - They should not be updated by the component to which they are passed.
 - They are owned by the component which passes them to some other component.
-
- State is something internal and private to the component.
 - State can and will change depending on the interactions with the outer world.
 - State should store as simple data as possible, such as whether an input checkbox is checked or not or a CSS class that hides or displays the component

Simple example : props + state

```
import React, {useState} from 'react'
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const [uname, setUname] = useState('Shrilata')
  const [email, setEmail] = useState('shrilata@gmail.com')

  return(
    <ChildComponent uname={uname} email={email} />
  );
}
export default ParentComponent;
```

```
const ChildComponent = (props) => {
  return(
    <div>
      <div>Name : {props.uname}</div>
      <div>Email : {props.email}</div>
    </div>
  );
}
export default ChildComponent;
```

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ParentComponent />
    </div>
  );
}
```

Welcome to React!

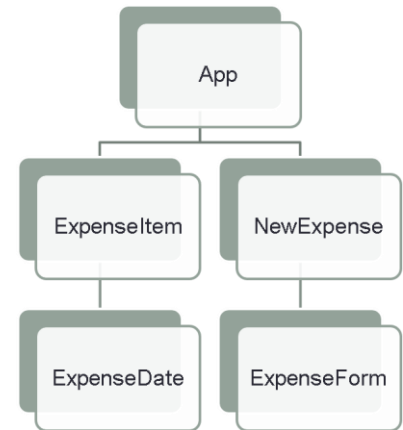
Name : Shrilata
Email : shrilata@gmail.com

USING FORM FOR INPUT

Adding form inputs

```
import "../ExpenseForm.css"
const ExpenseForm = () => {
  return(
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__control">
          <label>Title</label>
          <input />
        </div>
        <div className="new-expense__control">
          <label>Amount</label>
          <input type="number"/>
        </div>
        <div className="new-expense__control">
          <label>Date</label>
          <input type="date" min="2019-01-01"
            max="2022-12-31" />
        </div>
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
}
export default ExpenseForm;
```

```
const NewExpense = () => {
  return(
    <div className="new-expense">
      <ExpenseForm />
    </div>
  );
}
export default NewExpense;
```



The screenshot shows the application's user interface. At the top is a purple 'Add Expense' form with three input fields: 'Title', 'Amount', and 'Date' (with a date picker). Below the form is a list of expenses, each with a date, title, and amount.

Date	Title	Amount	Action
August 14 2020	Groceries	Rs 900	Change Title
March 12 2021	New TV	Rs 34000	Change Title
March 28 2021	SofaSet	Rs 25000	Change Title

Listening to user input

```
const ExpenseForm = () => {  
  
  const titleChangeHandler = (event) => {  
    console.log(event.target.value)  
  }  
  
  return(  
    <form>  
      <div className="new-expense__controls">  
        <div className="new-expense__control">  
          <label>Title</label>  
          <input onChange={titleChangeHandler}/>  
        </div>  
        ...  
      </div>  
      <div className="new-expense__actions">  
        <button type="submit">Add Expense</button>  
      </div>  
    </form>  
  );  
}  
export default ExpenseForm;
```

The image shows a web application interface for adding an expense. The form is titled 'ExpenseForm' and is rendered with a purple background. It contains three input fields: 'Title', 'Amount', and 'Date'. The 'Title' field is highlighted with a red box and contains the text 'table'. The 'Amount' field is empty. The 'Date' field is empty and has a date picker icon. A red box also highlights a dropdown menu on the right side of the form, which shows a list of suggestions: 't', 'ta', 'tab', 'tabl', and 'table'. The 'table' suggestion is highlighted. At the bottom right of the form is a purple button labeled 'Add Expense'.

Storing input into state – working with multiple states

```
import React, {useState} from 'react';
const ExpenseForm = () => {

  const [inputTitle, setInputTitle] = useState('')
  const [inputAmount, setInputAmount] = useState('')
  const [inputDate, setInputDate] = useState('')

  const titleChangeHandler = (event) => {
    setInputTitle(event.target.value)
  }
  const amountChangeHandler = (event) => {
    setInputAmount(event.target.value)
  }
  const dateChangeHandler = (event) => {
    setInputDate(event.target.value)
  }
  return(
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__control">
          <label>Title</label> <input onChange={titleChangeHandler}/>
        </div>
        <div className="new-expense__control">
          <label>Amount</label>
          <input type="number" onChange={amountChangeHandler}/>
        </div> ...
      </div>
    </form>
  )
}
```

Form submission

```
const ExpenseForm = () => {  
  
  const [inputTitle, setInputTitle] = useState('')  
  const [inputAmount, setInputAmount] = useState('')  
  const [inputDate, setInputDate] = useState('')  
  
  const titleChangeHandler = (event) => { ...  
  }  
  const amountChangeHandler = (event) => { ...  
  }  
  const dateChangeHandler = (event) => { ...  
  }  
  const submitHandler = (event) => {  
    event.preventDefault();  
  }  
  
  return(  
    <form onSubmit={submitHandler}>  
      <div className="new-expense__controls"> ...  
      </div>  
      <div className="new-expense__actions">  
        <button type="submit">Add Expense</button>  
      </div>  
    </form>  
  );  
}
```

Form submission – extracting data, 2-way binding

```
const ExpenseForm = () => {
```

```
  const [inputTitle, setInputTitle] = useState('')
  const [inputAmount, setInputAmount] = useState('')
  const [inputDate, setInputDate] = useState('')
```

```
  const titleChangeHandler = (event) => {...}
  const amountChangeHandler = (event) => {...}
  const dateChangeHandler = (event) => {...}
```

```
  const submitHandler = (event) => {
    event.preventDefault();
    const expenseData = {
      title:inputTitle,
      amount:inputAmount,
      date:inputDate
    }
    console.log(expenseData)
    setInputAmount('')
    setInputDate('')
    setInputTitle('')
  }
```

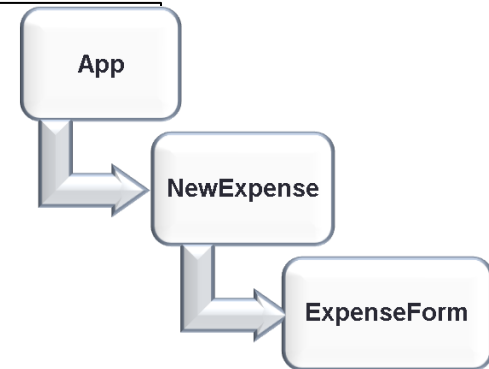
The form has a purple header. Below it are three input fields: 'Title' with the value 'Mouse', 'Amount' with the value '900', and 'Date' with the value '01-12-2021'. A purple button labeled 'Add Expense' is at the bottom right.

The form is identical to the one above. To its right, a console log window is open, showing the message: '[HMR] Waiting for update signal from WDS...' and a log entry: '{title: 'Mouse', amount: '900', date: '2021-12-01'}'.

```
    return(
      <form onSubmit={submitHandler}>
        <div className="new-expense__controls">
          <div className="new-expense__control">
            <label>Title</label>
            <input value={inputTitle}
              onChange={titleChangeHandler}/>
          </div>
          ...
        </form>
      );
    )
  }
```

Passing data from child to parent component

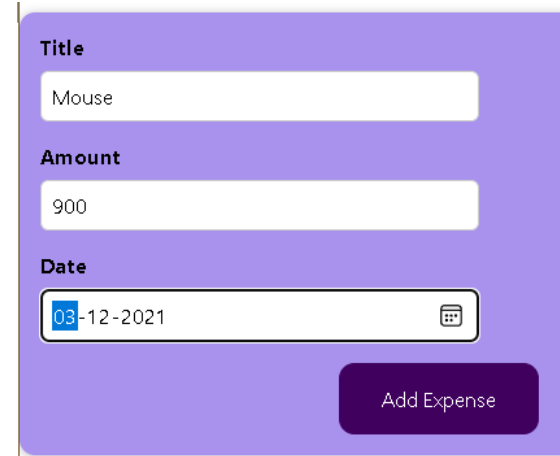
```
const NewExpense = () => {  
  
  const saveExpenseDataHandler = (inputExpenseData) => {  
    const expenseData = {  
      ...inputExpenseData,  
      id: Math.random().toString()  
    }  
    console.log("In NewExpense ", expenseData)  
  }  
  
  return(  
    <div className="new-expense">  
      <ExpenseForm onSaveExpenseData={saveExpenseDataHandler}/>  
    </div>  
  );  
}  
  
export default NewExpense;
```



```
const ExpenseForm = (props) => {  
  
  ...  
  const submitHandler = (event) => {  
    event.preventDefault();  
    const expenseData = {  
      title: inputTitle,  
      amount: inputAmount,  
      date: new Date(inputDate)  
    }  
    //console.log(expenseData)  
    props.onSaveExpenseData(expenseData);  
    ...  
  }  
}
```

Passing data from child to parent

```
function App() {  
  const expenses = [...];  
  
  const addExpenseHandler = expense => {  
    console.log("In App component ", expense)  
  }  
  
  return (  
    <div className="App">  
      <h2>Welcome to React!</h2>  
      <NewExpense onAddExpense={addExpenseHandler} />  
      ...  
    );  
  }  
  export default App;
```



The form is titled "Add Expense" and is set against a purple background. It contains three input fields: "Title" with the value "Mouse", "Amount" with the value "900", and "Date" with the value "03-12-2021". A calendar icon is visible next to the date field. A dark purple button labeled "Add Expense" is at the bottom right.

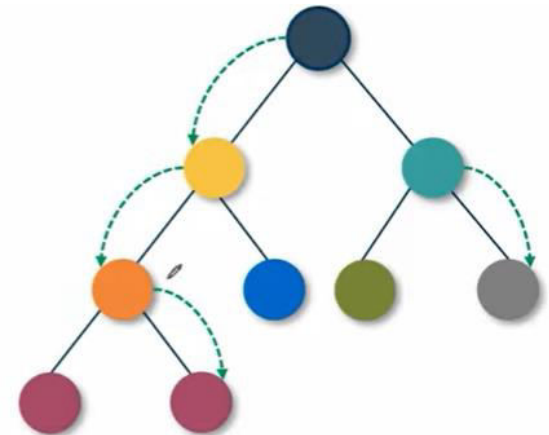
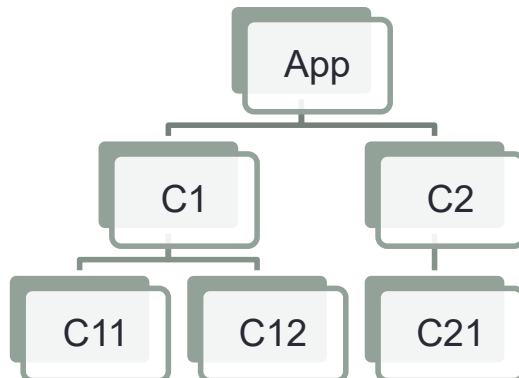
In App component [App.js:20](#)
▶ {title: 'Mouse', amount: '900', date: '2021-12-03', id: '0.25759054649698765'}

```
const NewExpense = (props) => {  
  
  const saveExpenseDataHandler = (inputExpenseData) => {  
    const expenseData = {  
      ...inputExpenseData,  
      id: Math.random().toString()  
    }  
    //console.log("In NewExpense ", expenseData)  
    props.onAddExpense(expenseData)  
  }  
  return(...);  
}  
export default NewExpense;
```

LIFTING STATE UP

Lifting state up in React.js

- In a typical application, you pass a lot of state down to child components as props.
 - These props are often passed down multiple component levels.
 - That's how state is shared vertically in your application.
- Often there will be a need to **share state between different components**.
 - The common approach to share state between two components is to move the state to common parent of the two components.
 - This approach is called **as lifting state up** in React.js
 - React components can manage their own state
 - Often components need to communicate state to others
 - Siblings do not pass state to each other directly
 - State should pass through a parent, then trickle down



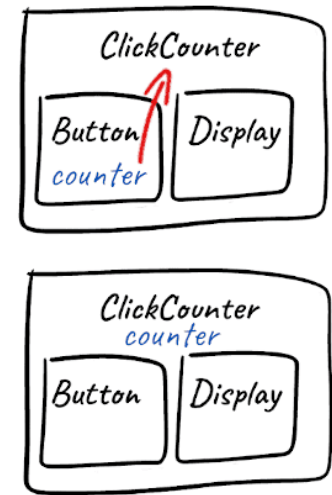
Lifting state up – simple example

```
import React, {useState} from 'react'
import Display from './Display'
import Button from './Button'

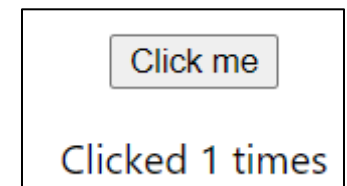
const ClickCounter = () => {
  const [counter, setCounter] = useState(0)

  const incrCounter = () => {
    setCounter(counter + 1)
  }
  return(
    <div>
      <Button onClick={incrCounter}>Button</Button>
      <Display message={`Clicked ${counter} times`} />
    </div>
  );
}
export default ClickCounter;
```

```
const Button = (props) => {
  return <button onClick={props.onClick}>Click me</button>;
}
export default Button;
```

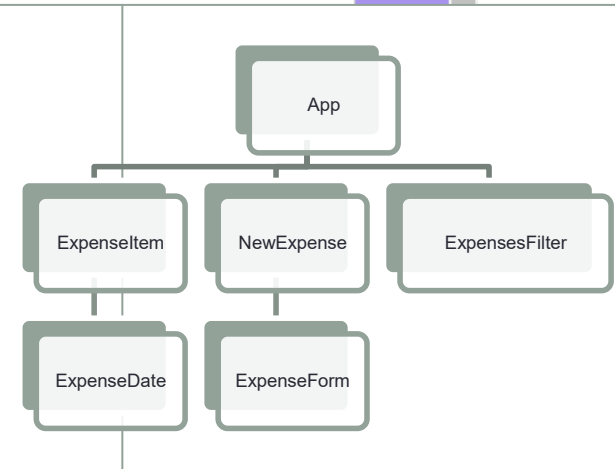
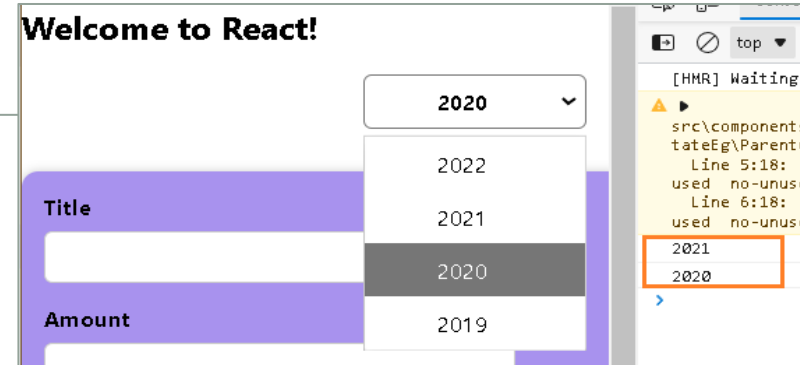


```
const Display = (props) => {
  return <p>{props.message}</p>;
}
export default Display;
```



A small aside – add filter to date

```
const ExpensesFilter = (props) => {  
  
  const selectChangeHandler = (event) => {  
    console.log(event.target.value)  
    props.onSelectYear(event.target.value)  
  }  
  
  return (  
    <div className='expenses-filter'>  
      <div className='expenses-filter__control'>  
        <label>Filter by year</label>  
        <select onChange={selectChangeHandler}>  
          <option value='2022'>2022</option>  
          <option value='2021'>2021</option>  
          <option value='2020'>2020</option>  
          <option value='2019'>2019</option>  
        </select>  
      </div>  
    </div>  
  );  
};  
export default ExpensesFilter;
```



```
function App() {  
  const [filteredYear, setFilteredYear] = useState(2020);  
  
  const selectYearHandler = filteredValue => {  
    setFilteredYear(filteredValue)  
  }  
  
  return (  
    <div className="App">  
      <h2>Welcome to React!</h2>  
      <ExpensesFilter onSelectYear={selectYearHandler}/>  
    </div>  
  );  
}
```

WORKING WITH LISTS AND CONDITIONALS

```

const SimpleListComponent = () => {
  const nums = [1,2,3,4,5]
  const updatedNums = nums.map((num)=>{
    return <li>{num*num}</li>;
  });

  const items =[{name:'Item-1'}, {name:'Item-2'}]
  const updatedItems = items.map(item => (
    <p>{item.name}</p>
  ))

  const students = [{name:"Anita",rollno:101},
                    {name:"Sunita",rollno:102},
                    {name:"Kavita",rollno:103}]
  const updatedStudents = students.map(student => (
    <tr>
      <td>Name {student.name} </td>
      <td>Rollno {student.rollno} </td>
    </tr>
  ))
  return(
    <div>
      <h2> Numbers List</h2> {updatedNums}
      <h2> Items List</h2> {updatedItems}
      <h2> Students List</h2>
      <table border="1">{updatedStudents}</table>
    </div>
  );}
export default SimpleListComponent

```

Working with lists

Numbers List

- 1
- 4
- 9
- 16
- 25

Items List

Item-1

Item-2

Students List

Name Anita	Rollno 101
Name Sunita	Rollno 102
Name Kavita	Rollno 103

Working with lists in child component

```
import SimpleListChildComponent from "./SimpleListChildComponent";

const SimpleListComponent = () => {
  const nums = [1,2,3,4,5]

  const items =[{name:'Item-1'}, {name:'Item-2'}]

  const students = [{name:"Anita",rollno:101},
                    {name:"Sunita",rollno:102},
                    {name:"Kavita",rollno:103}]

  return(
    <div>
      <SimpleListChildComponent
        nums={nums}
        items={items}
        students={students} />
    </div>
  );
}
export default SimpleListComponent
```

```
const SimpleListChildComponent = (props) => {

  const nums = props.nums;
  const updatedNums = nums.map(...);
  const items = props.items
  const updatedItems = items.map(...)

  const students = props.students;
  const updatedStudents = students.map(...)

  return(...);
}
export default SimpleListChildComponent;
```

Working with the expenses list

```
const expenses = [
  { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
  { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { title: 'SofaSet', amount: 25000, date: new Date(2021, 2, 28)}
];
return (
  <div className="App">
    <h2>Welcome to React!</h2>
    <ExpenseItem
      expDate={expenses[0].date}
      expTitle={expenses[0].title}
      expAmount={expenses[0].amount}
    />
    <ExpenseItem
      expDate={expenses[1].date}
      expTitle={expenses[1].title}
      expAmount={expenses[1].amount}
    />
    <ExpenseItem
      expDate={expenses[2].date}
      expTitle={expenses[2].title}
      expAmount={expenses[2].amount}
    />
  </div>
);
}
```

Working with the expenses list

```
function App() {
  const expenses = [
    { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
  ];
  return (
    <div className="App">
      <h2>Welcome to React!</h2>

      {expenses.map(expense => {
        return <ExpenseItem
          expDate={expense.date}
          expTitle={expense.title}
          expAmount={expense.amount}
        />
      })}
      {/* <ExpenseItem
        expDate={expenses[0].date}
        expTitle={expenses[0].title}
        expAmount={expenses[0].amount}
      /> ... */}
    </div>
  );
}
export default App;
```

```
{expenses.map(expense =>
  (<ExpenseItem
    expDate={expense.date}
    expTitle={expense.title}
    expAmount={expense.amount}
  />))
}
```

August 14 2020	Groceries	Rs 900	Change Title
March 12 2021	New TV	Rs 34000	Change Title
March 28 2021	Sofa Set	Rs 25000	Change Title

Using stateful lists

```
const DUMMY_EXP = [
  { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
  { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { title: 'New Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
];

function App() {
  const [expenses, setExpenses] = useState(DUMMY_EXP)

  const addExpenseHandler = expense => {
    setExpenses(prevArr => {return [expense, ...prevArr]})
  }

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />

      {expenses.map(expense => (<ExpenseItem
        expDate={expense.date}
        expTitle={expense.title}
        expAmount={expense.amount}
        />))}

    </div>
  );
}

export default App;
```

The top part of the image shows the 'NewExpense' form. It has a purple header with 'Title' and 'Amount' labels. The 'Title' input contains 'Keyboard' and the 'Amount' input contains '1500'. Below these is a 'Date' input showing '04-12-2021' with a calendar icon. An 'Add Expense' button is at the bottom right. Below the form is a list of three expenses: 'Groceries' (August 14, 2020, Rs 900), 'New TV' (March 12, 2021, Rs 34000), and 'New Sofa Set' (March 28, 2021, Rs 25000). Each item has a 'Change Title' button.

The bottom part of the image shows the expense list after adding a new expense. The 'Keyboard' expense (December 04, 2021, Rs 1500) is now at the top of the list, highlighted with an orange border. Below it are the 'Groceries', 'New TV', and 'New Sofa Set' expenses. Each item has a 'Change Title' button.

Lists and keys

✖ Warning: Each child in a list should have a unique "key" prop. [index.js:1](#) ⓘ

Check the render method of `App`. See <https://reactjs.org/link/warning-keys> for more information.

at ExpenseItem (<http://localhost:3000/static/js/main.chunk.js:685:19>)

at App (<http://localhost:3000/static/js/main.chunk.js:184:89>)

```
const DUMMY_EXP = [
  { id:101, title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
  { id:102, title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { id:103, title: 'New Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
];

function App() {

  const [expenses, setExpenses] = useState(DUMMY_EXP)
  ...
  return (
    <div className="App">
      ...
      {expenses.map(expense => (
        <ExpenseItem
          key={expense.id}
          expDate={expense.date}
          expTitle={expense.title}
          expAmount={expense.amount}
        />))
      }
    </div>
  )
  ...
}
```

Lists and keys

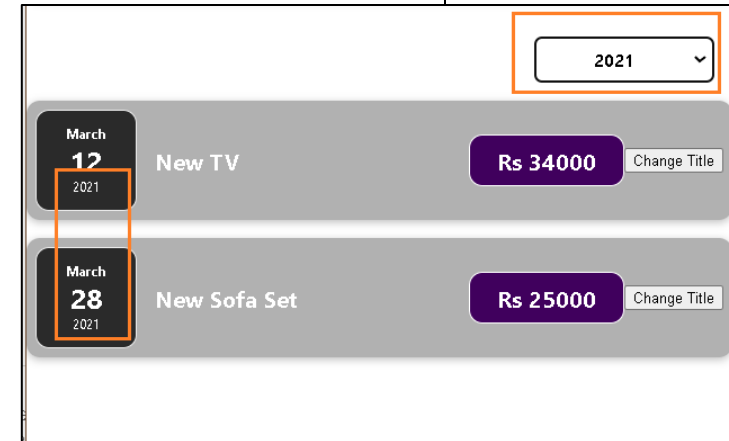
- When creating a list in JSX, React may show you an error and ask for a key.
 - Keys are unique identifiers that must be attached to the top-level element inside a map.
 - Keys are used by React to know how to update a list whether adding, updating, or deleting items.
 - This is part of how React is so fast with large lists.
 - Keys are a way to help React know how to efficiently update a list.
 - We can add a key using the key prop like so:

```
<div>
  {people.map(person => (
    <p key={person.name}>{person.name}</p>
  ))}
</div>
```

Implementing filters

```
function App() {  
  const [expenses, setExpenses] = useState(DUMMY_EXP)  
  const [filteredYear, setFilteredYear] = useState(2020);  
  
  const filteredExpensesArr = expenses.filter(expense => {  
    return expense.date.getFullYear().toString() === filteredYear;  
  })  
}
```

```
...  
const selectYearHandler = filteredValue => {  
  setFilteredYear(filteredValue)  
}  
  
return (  
  <div className="App">  
    <h2>Welcome to React!</h2>  
    <NewExpense onAddExpense={addExpenseHandler} />  
    <ExpensesFilter onSelectYear={selectYearHandler}/>  
    {filteredExpensesArr.map(expense => (  
      <ExpenseItem  
        key={expense.id}  
        expDate={expense.date}  
        expTitle={expense.title}  
        expAmount={expense.amount}  
      />))  
  )  
}
```



Rendering content conditionally

- Conditional rendering means to render a specific HTML element or React component depending on a prop or state value.
 - In a conditional render, a React component decides based on one or several conditions which DOM elements it will return.
 - For instance, based on some logic it can either return a list of items or a text that says "Sorry, the list is empty".

```
if(condition_is_met) {  
    renderSectionOfUI();  
}
```

Rendering content conditionally : example

```
/*const users = [  
  { id: '1', firstName: 'Shrilata', lastName: 'T' },  
  { id: '2', firstName: 'Anita', lastName: 'Patil' },  
];*/  
const users = []
```

```
function ListUsers() {  
  return (  
    <div>  
      <List list={users} />  
    </div>  
  );  
}
```

```
function List({ list }) {  
  if (!list) {  
    return null;  
  }  
  return (  
    <ul>  
      {list.map(item => (  
        <Item key={item.id} item={item} />  
      ))}  
    </ul>  
  );  
}
```

```
if (!list.length) {  
  return <p>Sorry, the list is empty.</p>;  
}
```

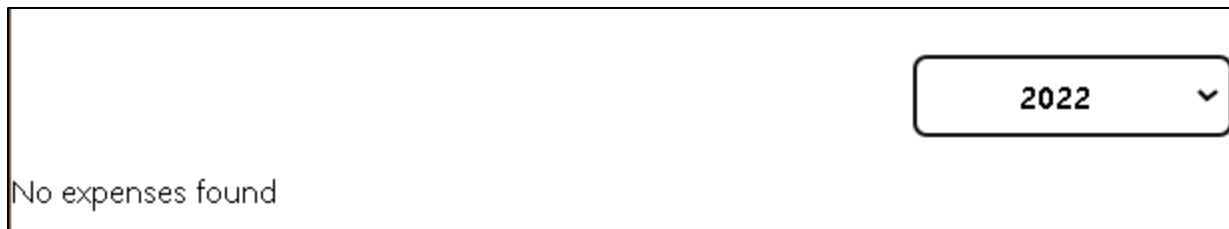
```
function Item({ item }) {  
  return (  
    <li>  
      {item.firstName} {item.lastName}  
    </li>  
  );  
}  
export default ListUsers;
```

Hello Conditional Rendering

- Shrilata T
- Anita Patil

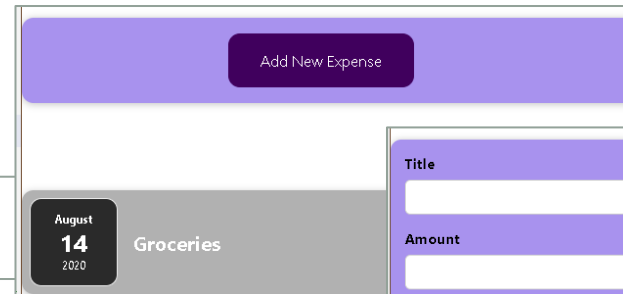
Rendering content conditionally : Expense tracker example

```
{filteredExpensesArr.length ==0 ? <p>No expenses found</p> :  
  filteredExpensesArr.map(expense => (  
    <ExpenseItem  
      key={expense.id}  
      expDate={expense.date}  
      expTitle={expense.title}  
      expAmount={expense.amount}  
    />))  
}
```

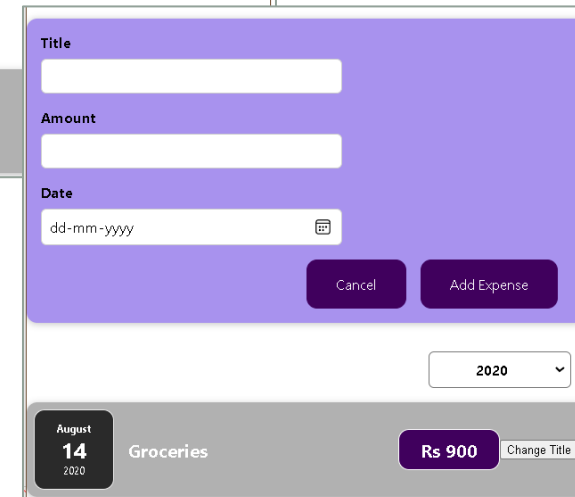


Rendering content conditionally : one more example

```
const NewExpense = (props) => {  
  const [showForm, setShowForm] = useState(false)  
  
  const showFormHandler = () => {  
    setShowForm(true)  
  }  
  
  const saveExpenseDataHandler = (inputExpenseData) => {...}  
  return(  
    <div className="new-expense">  
      {!showForm && <button onClick={showFormHandler}>Add New Expense </button>}  
      {showForm && <ExpenseForm onCancel={stopShowForm}  
        onSaveExpenseData={saveExpenseDataHandler}/>}  
    </div>  
  );  
}  
export default NewExpense;
```



```
//ExpenseForm  
<div className="new-expense__actions">  
  <button type="button"  
    onClick={props.onCancel}>Cancel</button>  
  <button type="submit">Add Expense</button>  
</div>
```



CLASS-BASED COMPONENTS

```
const HelloComponent = (props) => {  
  return (<h3>Hello, welcome user!!</h3>)  
}  
export default HelloComponent;
```

Functional components are regular javascript functions which return renderable results (typically JSX)

```
class HelloComponent extends Component{  
  render(){  
    return (<h3>Hello, welcome user!!</h3>)  
  }  
}  
export default HelloComponent;
```

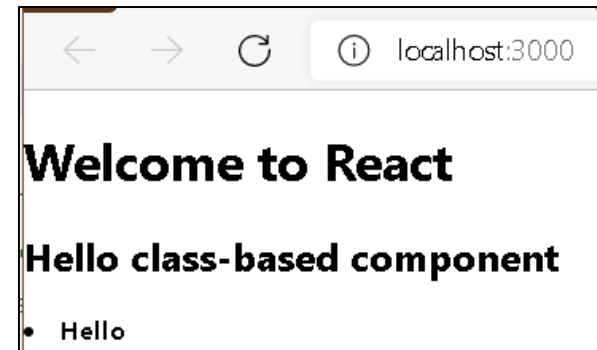
Class based components are defined using Javascript classes where a render method defines the to-be-rendered output

Class-based components : Examples

```
import React, {Component} from 'react';
class HelloComponent extends Component{
  render(){
    return (<h2>Hello class-based component</h2>)
  }
}
export default HelloComponent;
```

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

class User extends Component{
  render(){
    return <li className='user'>Hello User</li>
  }
};
export default User;
```



```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
      <HelloComponent />
      <User />
    </div>
  );
}
```

Class-based components : passing into props

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

```
const DUMMY_USERS = [
  { id: 'u1', name: 'Shrilata' },
  { id: 'u2', name: 'Soha' },
  { id: 'u3', name: 'Sia' },
];
```

```
class Users extends Component{
  render(){
    return(
      <div>
        <User name={DUMMY_USERS[0].name} />
        <User name={DUMMY_USERS[1].name} />
        <User name={DUMMY_USERS[2].name} />
      </div>
    );
  }
}
```

```
export default Users;
```

```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
      <Users />
    </div>
  );
}
```

Welcome to React

- Hello Max
- Hello Manuel
- Hello Julie

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

class User extends Component{
  render(){
    return (<li className='user'>Hello {this.props.name}</li>);
  }
};

export default User;
```

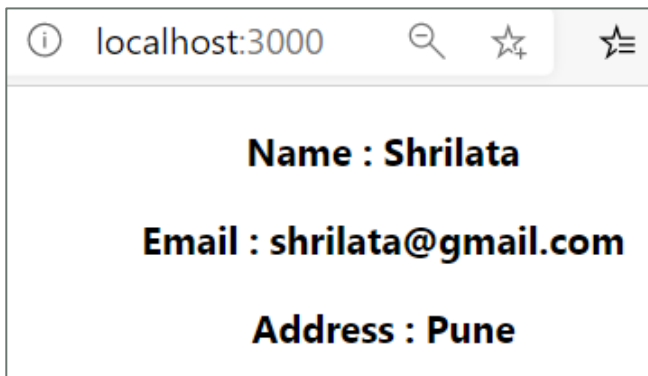
React State : Example

```
import React, {Component} from 'react';
import './App.css';
import StatefulComponent from "../StatefulComponent/StatefulComponent";
```

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

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

class statefulComponent extends Component{
  state = {
    name: "Shrilata",
    email: "shrilata@gmail.com",
    address:"Pune"
  }
  render(){
    return(
      <div>
        <h3>Name : {this.state.name}</h3>
        <h3>Email : {this.state.email}</h3>
        <h3>Address : {this.state.address}</h3>
      </div>
    );
  }
}
export default statefulComponent;
```



State and props

```
class App extends Component {
  state = {
    persons: [
      {name: "Shri", age: 20},
      {name: "Soha", age: 23},
      {name: "Sandeep", age: 30},
    ]
  }
  render() {
    return (
      <div className="App">
        <h1> Hi, welcome to React</h1>
        <button>Switch name</button>
        <Person name={this.state.persons[0].name} age={this.state.persons[0].age}/>
        <Person name={this.state.persons[1].name} age={this.state.persons[1].age}>
          Hobbies : Coding
        </Person>
        <Person name={this.state.persons[2].name} age={this.state.persons[2].age}/>
      </div>
    );
  }
}
```

```
const person = (props) => {
  return (
    <div>
      <p>Hi i am {props.name}
        and i am {props.age} years old</p>
    </div>
  )
}
export default person;
```

Hi, welcome to React

Switch name

Hi i am Shri and i am 20 years old

Hi i am Soha and i am 23 years old

Hobbies : Coding

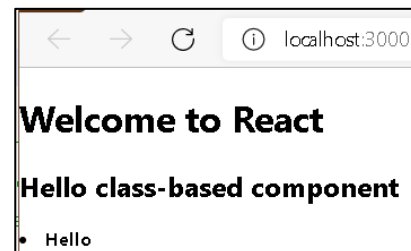
Hi i am Sandeep and i am 30 years old

LIFECYCLE EVENTS

Lifecycle Events

- React class components can have hooks for several lifecycle events
 - During the lifetime of a component, there's a series of events that gets called, and to each event you can hook and provide custom functionality.
 - React lifecycle methods are a series of events that happen from the birth of a React component to its death.
 - There are 4 phases in a React component lifecycle:
 - initial
 - Mounting
 - Updating
 - Unmounting

```
class HelloComponent extends Component{
  constructor(){
    super();
    this.state = {message: "hello" }
  }
  render(){
    return (<h2>Hello World</h2>)
  }
}
```



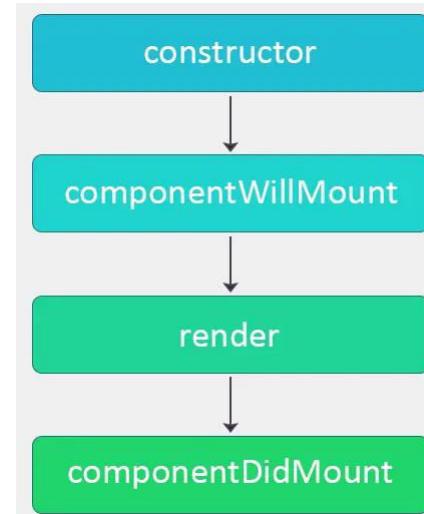
Lifecycle methods

- React lifecycle methods:
 - Each React lifecycle phase has a number of lifecycle methods that you can override to run code at specified times during the process.
 - These are popularly known as component lifecycle methods.
- Initialisation phase:
 - `Constructor(props)`: This is a special function that is called when new components are created. In this, we initialize the state or props of the component.

Lifecycle methods

- The mounting phase refers to the phase during which a component is created and inserted to the DOM.
- The following methods are called in order.
 - **ComponentWillMount()**: This function is called immediately before mounting occurs. It is called right before the first rendering is performed.
 - **render()**: You have this method for all the components created. It returns the Html node.
 - **componentDidMount()**: This method is called right after the react component is mounted on DOM or you can say that it is called right after render method executed for the first time
 - Here we can make API call, foreg, interact with dom or perform any ajax call to load data.

```
class HelloComponent extends Component{  
  componentDidMount(){  
    console.log("Mounted component")  
  }  
  render(){  
    return <h1>Hello</h1>  
  }  
}
```



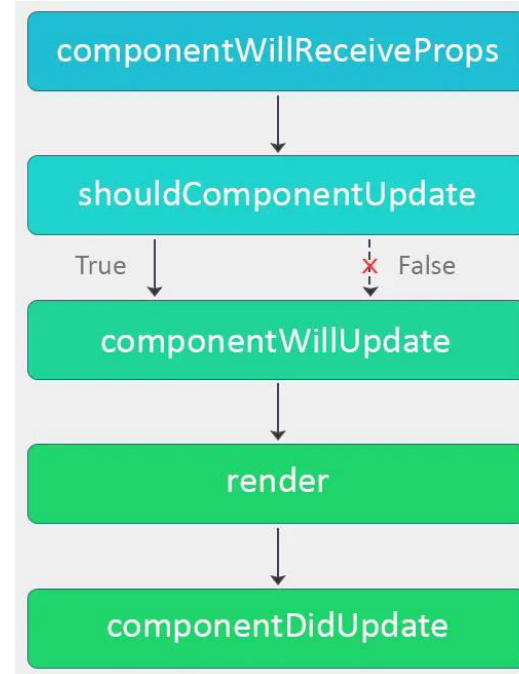
Hello

CONSOLE WAS CLEARED
Mounted component



Lifecycle methods

- Update: In this state, the dom is interacted by a user and updated. For example, you enter something in the textbox, so the state properties are updated.
 - The component is re-rendered whenever a change is made to react component's state or props, you can simply say that the component is updated.
- Following are the methods available in update state:
 - `shouldComponentUpdate()` : called when the component is updated.
 - `componentDidUpdate()` : after the component is updated.
- UnMounting: this state comes into the picture when the Component is not required or removed.
- Following are the methods available in unmount state:
 - `ComponentWillUnmount()`: called when the Component is removed or destroyed.



`componentWillUnmount`

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

class LifecycleComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: ''};

    this.UpdateName = this.UpdateName.bind(this);
    this.testclick = this.testclick.bind(this);
  }

  UpdateName(event) {
    this.setState({name: event.target.value});
  }

  testclick(event) {
    alert("The name entered is: "+ this.state.name);
  }

  componentDidMount() {
    console.log('Mounting State : calling method componentDidMount');
  }

  shouldComponentUpdate() {
    console.log('Update State : calling method shouldComponentUpdate');
    return true;
  }
}
```

```

componentDidUpdate() {
  console.log('Update State : calling method
componentDidUpdate')
}
componentWillUnmount() {
  console.log('UnMounting State : calling method
componentWillUnmount');
}

render() {
  return (
    <div>
      Enter Your Name:<input type="text"
value={this.state.name} onChange={this.UpdateName} /><br/>
      <h2>{this.state.name}</h2>
      <input type="button"
        value="Click Here"
        onClick={this.testclick} />
    </div>
  );
}
}

export default LifecycleComponent;

```

Welcome to React

Enter Your Name:

Shrilata

Mounting State : calling method componentDidMount
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate
Update State : calling method shouldComponentUpdate
Update State : calling method componentDidUpdate

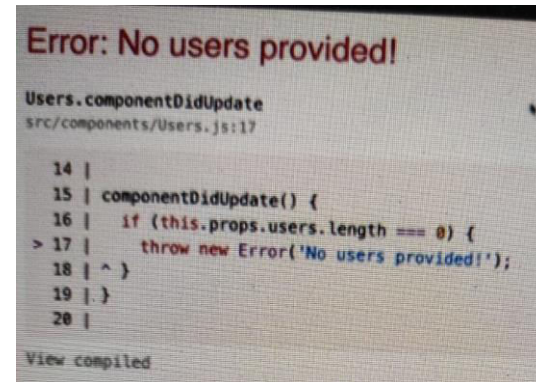
[Lifecycle Methods \(kirupa.com\)](http://kirupa.com)

ERROR BOUNDARIES

Using Error Boundaries

```
const person = (props) => {  
  const rnd = Math.random();  
  if(rnd > 0.7){  
    throw new Error("Something went wrong");  
  }  
  return (  
    ...  
  )  
};  
export default person;
```

```
class Users extends Component{  
  
  componentDidUpdate(){  
    if(this.props.users.length == 0)  
      throw new Error("No users in list!")  
  }  
  ...  
}
```



- A JavaScript error in a part of the UI shouldn't break the whole app.
- To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Using Error Boundaries

```
import React, {Component} from 'react';
class ErrorBoundary extends Component{
  state = {
    hasError:false,
    errorMessage:''
  }
  componentDidCatch = (error, info) => {
    this.setState({hasError:true, errorMessage:error});
  }
  render(){
    if(this.state.hasError)
      return <h1>{this.state.errorMessage}</h1>
    else
      return this.props.children
  }
}
export default ErrorBoundary;
```

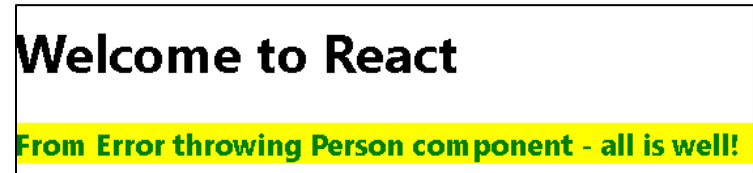
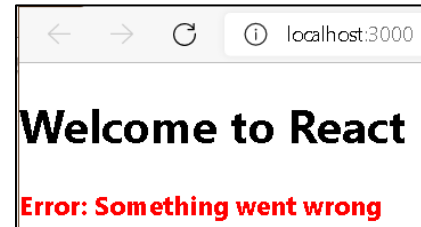
```
import React from 'react'
function Artists({artistName}) {
  if (artistName === 'peruzzi') {
    throw new Error ('not performing tonight!')
  }
  return (
    <div>
      {artistName}
    </div>
  )
}
export default Artists
```

Use it like this:
<ErrorBoundary>
 <Artists />
</ErrorBoundary>

- Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering
- <https://reactjs.org/docs/error-boundaries.html>
- [React.Component – React \(reactjs.org\)](#)

Using Error Boundaries

```
function App() {  
  return (  
    <div className="App">  
      <h1> Welcome to React</h1>  
      <ErrorBoundary >  
        <Person />  
      </ErrorBoundary>  
    </div>  
  );  
}
```



FORMS AND FORMS VALIDATION

Handling user input the right way

Controlled Components and Uncontrolled components

- React forms present a unique challenge because you can either allow the browser to handle most of the form elements and collect data through React change events, or you can use React to fully control the element by setting and updating the input value directly.
 - The first approach is called an **uncontrolled** component because React is not setting the value.
 - The second approach is called a **controlled** component because React is actively updating the input.
- In HTML, form data is usually handled by the DOM.
- In React, form data is usually handled by the components.
 - When the data is handled by the components, all the data is stored in the component state.

Controlled Inputs in class components

- An input is said to be “controlled” when React is responsible for maintaining and setting its state.
 - The state is kept in sync with the input’s value, meaning that changing the input will update the state, and updating the state will change the input.

```
class ControlledInput extends React.Component {
  state = { name: '' };

  handleInput = (event) => {
    this.setState({name: event.target.value });
  }
  handleSubmit = (event) => {
    alert('A name was submitted: ' + this.state.name);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        Name : <input value={this.state.name}
                      onChange={this.handleInput} />
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```



Controlled Inputs using React hooks (functional components)

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

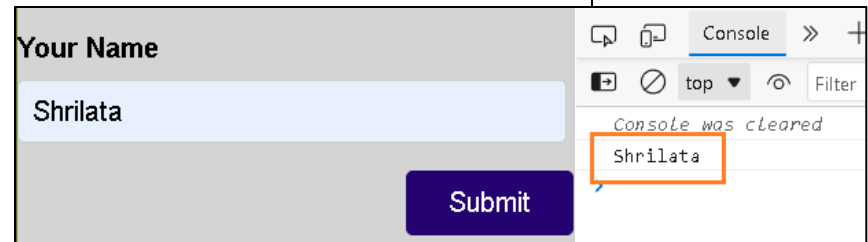
const SimpleInput = (props) => {
  const [inputName, setInputName] = useState('')

  const inputNameHandler = (event) => {
    setInputName(event.target.value)
  }

  const formSubmitHandler = event => {
    event.preventDefault();
    console.log(inputName)
  }

  return (
    <form onSubmit={formSubmitHandler}>
      <div className='form-control'>
        <label htmlFor='name'>Your Name</label>
        <input type='text' id='name' onChange={inputNameHandler}/>
      </div>
      <div className="form-actions">
        <button>Submit</button>
      </div>
    </form>
  );
};

export default SimpleInput;
```



Controlled Inputs

- Controlled inputs open up the following possibilities:
 - **instant input validation**: we can give the user instant feedback without having to wait for them to submit the form (e.g. if their password is not complex enough)
 - **instant input formatting**: we can add proper separators to currency inputs, or grouping to phone numbers on the fly
 - **conditionally disable form submission**: we can enable the submit button after certain criteria are met (e.g. the user consented to the terms and conditions)
 - **dynamically generate new inputs**: we can add additional inputs to a form based on the user's previous input (e.g. adding details of additional people on a hotel booking)

Handling Multiple Form Inputs

```
class ControlledLoginForm extends React.Component {
```

```
  state = {  
    username: '',  
    email: ''  
  };
```

```
  handleInput = (event) => {  
    let name = event.target.name;  
    let val = event.target.value;  
    this.setState({[name]: val});  
    console.log(this.state)  
  }  
  handleSubmit = (event) => {  
    alert('A name was submitted: '  
      + this.state.username);  
    event.preventDefault();  
  }
```

```
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <h3>Hello {this.state.username} {this.state.email}</h3>  
        Name : <input name="username" onChange={this.handleInput} /><br />  
        Email : <input name="email" onChange={this.handleInput} />  
        <input type="submit" value="Submit" />  
      </form>  
    );  
  }  
}
```

- {username: 'shrilata'} //where username is event.target.name which is field name ie username
- {email: 'Shrilata@gmail.com'} //where email is event.target.name which is field name ie email

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/'. The page content includes a 'Welcome' message, a form with 'Name' and 'Email' fields, and a 'Submit' button. A modal dialog box is open, displaying the message 'localhost:3000 says A name was submitted: shrilata'. The form fields are filled with 'shrilata' for Name and 'shrilata@gmail.com' for Email.

One more example

Name:

Shrilata

Observation:

Good fabric

Desired color:

Green

T-shirt Size: Small ☐ Medium ☐ Large ☒ XL ☐ XXL ☐ 3XL ☐

Submit

```
class MultipleInputFields extends Component {  
  
  state = {  
    name:'',  
    onservaion:'',  
    color:'',  
    size:'',  
  }  
  handleChanges = (event) => {  
    let name = event.target.name;  
    let val = event.target.value;  
    this.setState({[name]: val});  
    console.log(this.state)  
  };  
  submitFormHandler = (event) => {  
    event.preventDefault();  
    console.log("from submit ", this.state);  
  };  
  render(){  
    const colors = ['Blue', 'Red', 'Green', 'Yellow'];  
    const sizes = ['Small', 'Medium', 'Large', 'XL', 'XXL', '3XL'];
```

```
MultipleInputFields.js:16  
  {name: 'Shrilata', onservaion: '', color: '', size: '', observation: 'Good fabric'}  
MultipleInputFields.js:16  
  {name: 'Shrilata', onservaion: '', color: '', size: '', observation: 'Good fabric'}  
MultipleInputFields.js:16  
  {name: 'Shrilata', onservaion: '', color: '', size: '', observation: 'Good fabric'}  
MultipleInputFields.js:16  
  {name: 'Shrilata', onservaion: '', color: '', size: '', observation: 'Good fabric'}  
MultipleInputFields.js:16  
  {name: 'Shrilata', onservaion: '', color: '', size: '', observation: 'Good fabric'}  
MultipleInputFields.js:21  
from submit {name: 'Shrilata', onservaion: '', color: 'green', size: 'LARGE', observation: 'Good fabric'}
```

```

return (
  <form onSubmit={this.submitFormHandler}>
    <div className='form-control'>
      <label>Name:</label>
      <input name="name" type="text" value={this.state.name} onChange={this.handleChange} />
    </div>
    <div className='form-control'>
      <label>Observation:</label>
      <textarea name="observation" value={this.state.observation} onChange={this.handleChange} />
    </div>
    <div className='form-control'>
      <label>Desired color:</label>
      <select name="color" value={this.state.color} onChange={this.handleChange}>
        {colors.map((color, i) => <option key={i} value={color.toLowerCase()}>{color}</option>)}
      </select>
    </div>
    <div >
      <label>T-shirt Size:</label>
      {sizes.map((size, i) =>
        <label key={i}> {size}
          <input
            name="size" value={size.toUpperCase()} checked={this.state.size === size.toUpperCase()}
            onChange={this.handleChange} type="radio" />
          </label>
        )}
    </div>
    <div className="form-actions">
      <button type="submit">Submit</button>
    </div>
  </form>
)
}
export default MultipleInputFields;

```


Controlled components : Summary

- A controlled component is bound to a value, and its adjustments will be handled in code by using event-based callbacks.
 - Here, the input form variable is handled by the react itself rather than the DOM.
 - In this case, the mutable state is maintained in the state property and modified using `setState()`.
- Controlled components have functions which regulate the data that occurs at each on Change event.
 - This data is subsequently saved in the `setState()` method and updated. It helps components manage the elements and data of the form easier.
-
- You can use the controlled component when you create:
 - Forms validation so that when you type, you always have to know the input value to verify whether it is true or not!
 - Disable submission icon, except for valid data in all fields
 - If you have a format such as the input for a credit card

Validation

```
class ControlledInputValidation1 extends React.Component {
  state = { age: '' };

  handleInput = (event) => {
    let nam = event.target.name;
    let val = event.target.value;
    if (nam === "age") {
      if (!Number(val))
        alert("Age must be a number");
    }
    this.setState({[nam]: val});
  }
  handleSubmit = (event) => {
    alert('A age was submitted: ' + this.state.age);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        Age : <input name="age" value={this.state.age} onChange={this.handleInput}
        />

        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Age :

Uncontrolled Inputs

- “uncontrolled” form inputs: React doesn’t track the input’s state.
 - HTML input elements naturally keep track of their own state as part of the DOM, and so when the form is submitted we have to read the values from the DOM elements themselves.
- “uncontrolled” form inputs: React doesn’t track the input’s state.
 - If the DOM handles the data, then the form is **uncontrolled**, and if the state of the form component manages the data, then the form is said to be **controlled**
 - Uncontrolled components are inputs that do not have a value property. In opposite to controlled components, it is the application's responsibility to keep the component state and the input value in sync.
 - In order to do this, React allows us to create a “**ref**” (reference) to associate with an element, giving access to the underlying DOM node

Uncontrolled Inputs

- In **uncontrolled** components form data is being handled by DOM itself.
- For example here we can reference form values by name
- This is quick and dirty way of handling forms. It is mostly useful for simple forms or when you are *just learning React*.
- HTML input elements keep track of their own state
 - When the form is submitted we typically read the values from the DOM elements ourselves

First Name:

Last Name:

```
class ProfileForm extends Component {
  handleSubmit = (event) => {
    event.preventDefault();

    const firstName = event.target.firstName.value;
    const lastName = event.target.lastName.value;

    // Here we do something with form data
    console.log(firstName, lastName)
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input name="firstName" type="text" />
        </label>
        <label>
          Surname:
          <input name="lastName" type="text" />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Uncontrolled Inputs

- “**ref**” is used to receive the form value from DOM.
 - To enable this, React allows us to create a “ref” (reference) to associate with an element, giving access to the underlying DOM node.
 - Refs provide a way to access DOM nodes or React elements created in the render method.
 - Refs are created using `React.createRef()` and attached to React elements via the **ref** attribute.
 - Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.myRef = React.createRef();  
  }  
  render() {  
    return <div ref={this.myRef} />;  
  }  
}
```

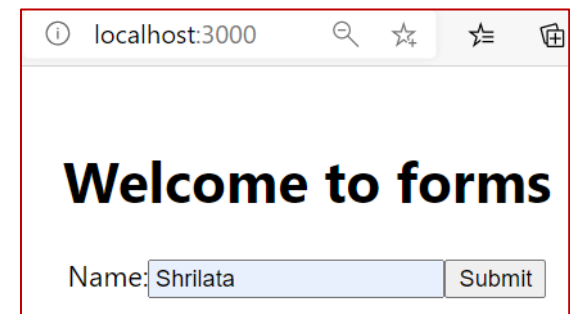
Uncontrolled Inputs

```
import React, {Component} from 'react';
class SimpleForm extends Component {
  constructor(props) {
    super(props);
    // create a ref to store the DOM element
    this.nameEl = React.createRef();
  }

  handleSubmit = (e) => {
    e.preventDefault();
    alert(this.nameEl.current.value);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Name:
          <input type="text" ref={this.nameEl} />
        </label>
        <input type="submit" name="Submit" />
      </form>
    )
  }
}
export default SimpleForm;
```

- You initialize a new ref in the constructor by calling `React.createRef()`, assigning it to an instance property so it's available for the lifetime of the component.
- In order to associate the ref with an input, it's passed to the element as the special ref attribute.
- Once this is done, the input's underlying DOM node can be accessed via **`this.nameEl.current`**.



localhost:3000

Welcome to forms

Name:

Another example : Login form

```
class LoginForm extends Component {  
  constructor(props) {  
    super(props);  
    this.nameEl = React.createRef();  
    this.passwordEl = React.createRef();  
    this.rememberMeEl = React.createRef();  
  }
```

```
  handleSubmit = (e) => {  
    e.preventDefault();  
    const data = {  
      username: this.nameEl.current.value,  
      password: this.passwordEl.current.value,  
      rememberMe: this.rememberMeEl.current.checked,  
    }  
    console.log(data)  
  }
```

```
  render(){  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <fieldset><legend>Login Form</legend>  
        <input type="text" placeholder="username" ref={this.nameEl} /><br></br>  
        <input type="password" placeholder="password" ref={this.passwordEl} /><br></br>  
        <label><input type="checkbox" ref={this.rememberMeEl} />Remember me  
        </label><br></br>  
        <button type="submit" className="myButton">Login</button>  
      </fieldset>  
    </form>  
  );  
}
```

Login Form

username

password

☐ Remember me

Login

```
{username: 'aaa', password: 'bbb', rememberMe: true}
```

COMPOSITION VS. INHERITANCE

Composition over Inheritance

- Composition and inheritance are the approaches to use multiple components together in React.js .
- This helps in code reuse.
- React recommend using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.
- Composition works with functions as well as classes both.

Inheritance in JS

```
class Automobile {
  constructor() {
    this.vehicleName = automobile;
    this.numWheels = null;
  }
  printNumWheels() {
    console.log(`This ${this.vehicleName} has ${this.numWheels} wheels`);
  }
}

class Car extends Automobile {
  constructor() {
    super(this);
    this.vehicleName = 'car';
    this.numWheels = 4;
  }
}

class Bicycle extends Automobile {
  constructor() {
    super(this);
    this.vehicleName = 'bike';
    this.numWheels = 2;
  }
}

const car = new Car();
const bike = new Bicycle();
car.printNumWheels() // This car has 4 wheels
bike.printNumWheels() // This bike has 2 wheels
```

```
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.methodA = this.methodA.bind(this);
  }

  methodA() {
    console.log("methodA in parent class");
  }

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

Console output

In child class, calling parent [child.js:9](#)
method...

methodA in parent class [parent.js:10](#)

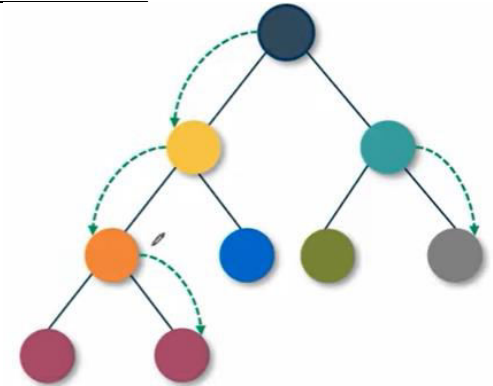
```
import Parent from "../parent";

class Child extends Parent {
  constructor() {
    super();
  }
  render() {
    console.log("In child class, calling parent method...");
    this.methodA();
    return false;
  }
}
```

Composition

- Composition is a code reuse technique where a larger object is created by combining multiple smaller objects.

```
class App extends React.Component {  
  render() {  
    return <Toolbar theme="dark" />;  
  }  
}  
  
function Toolbar(props) {  
  return (  
    <div>  
      <ThemedButton theme={props.theme} />  
    </div>  
  );  
}  
  
class ThemedButton extends React.Component {  
  render() {  
    return <Button theme={this.props.theme} />;  
  }  
}
```



```
class App extends Component {
  state = {
    date: new Date()
    ...
  }
  ...
  return (
    <div className="container">
      <NewsHeader className="news" subject="Sports"
        date={this.state.date.toString()} />
    </div>
  )
}
```

```
const newsHeader = (props) => {
  return(
    <div>
      <h1>News for {props.date}</h1>
      <h2>News Heading : {props.subject}</h2>
      <NewsContent title="Content Title-1" content="Lots of Content-1" />
      <NewsContent title="Content Title-2" content="Lots of Content-2" />
      <NewsContent title="Content Title-3" content="Lots of Content-3" />
    </div>
  )
}
```

```
const newsContent = (props) => {
  return(
    <div>
      {/*complex code that filters out news based on subject*/}
      <h4><b><i>News Title {props.title}</i></b></h4>
      <h4>News Content : {props.content}</h4>
      <Author title={props.title} name="Shrilata" />
    </div>
  )
}
```

```
const author = (props) => {
  return(
    <h6>Author for {props.title} - {props.name}</h6>
  )
}
```

News for Sun Jun 1

News Heading : Sports

News Title Content Title-1

News Content : Lots of Content-1

Author for Content Title-1 - Shrilata

News Title Content Title-2

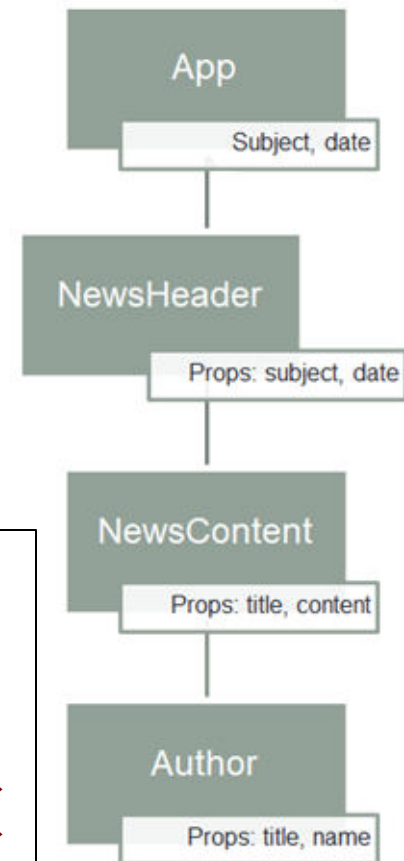
News Content : Lots of Content-2

Author for Content Title-2 - Shrilata

News Title Content Title-3

News Content : Lots of Content-3

Author for Content Title-3 - Shrilata

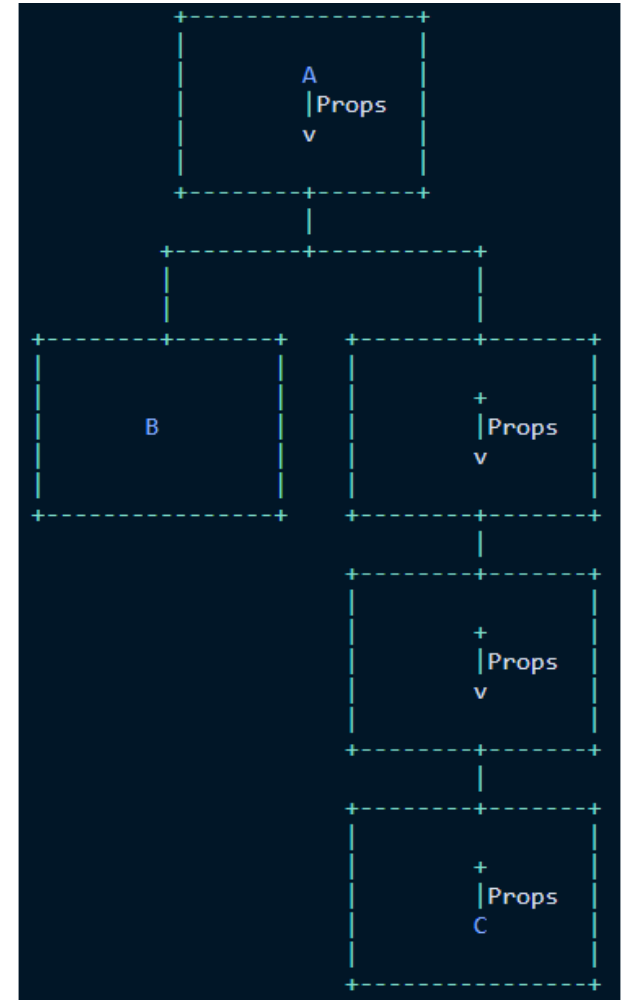


CONTEXT

Context

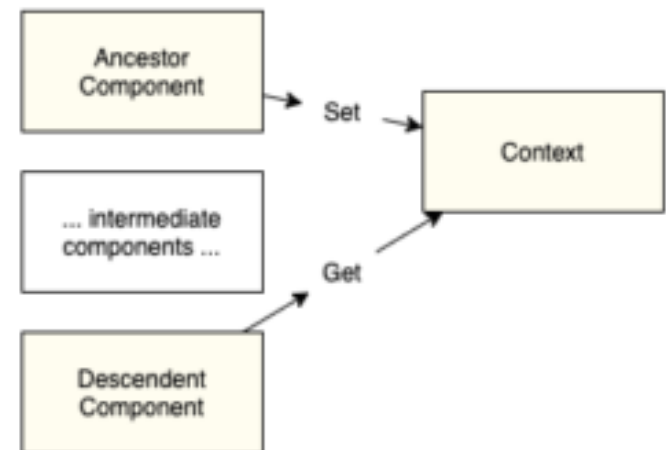
- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language

```
class App extends React.Component {  
  render() {  
    return <Toolbar theme="dark" />;  
  }  
}  
  
function Toolbar(props) {  
  return (  
    <div>  
      <ThemedButton theme={props.theme} />  
    </div>  
  );  
}  
  
class ThemedButton extends React.Component {  
  render() {  
    return <Button theme={this.props.theme} />;  
  }  
}
```



React Context API

- Store the state in a Context value in the common ancestor component (called the Provider Component), and access it from as many components as needed (called Consumer Components), which can be nested at any depth under this ancestor.
- This solution has the same benefits as the Props solution, but because of what could be called “hierarchical scoping”, it has the added benefit that any component can access the state in any Context that is rooted above itself in React’s hierarchy, without this state needing to be passed down to it as props.
- React.js takes care of all the magic behind the scenes to make this work.
- Primary situations where the React Context API really shines are:
 - When your state needs to be accessed or set from deeply nested components.
 - When your state needs to be accessed or set from many child components.



Three aspects to using React Contexts

- 1) Defining the Context object so we can use it.
 - If we wanted to store data about the current user of a web app, we could create a `UserContext` that can be used in the next two steps:

```
// Here we provide the initial value of the context
const UserContext = React.createContext({
  currentUser: null,
});
```

Note: It doesn't matter where this Context lives, as long as it can be accessed by all components that need to use it in the next two steps.

- 2) Providing a value for a Context in the hierarchy.
 - Assuming you had an `AccountView` component, you might provide a value like this

```
const AccountView = (props) => {
  const [currentUser, setCurrentUser] = React.useState(null);
  return (
    /* Here we provides the actual value for its descendents */
    <UserContext.Provider value={{ currentUser: currentUser }}>
      <AccountSummary/>
      <AccountProfile/>
    </UserContext.Provider>
  );
};
```

Three aspects to using React Contexts

- 3) Accessing the current Context value lower in the hierarchy.
 - If the AccountSummary component needed the user, we could have just passed it as a prop. But let's assume that it doesn't directly access the user data, but rather contains another component that does:

```
// Here we don't use the Context directly, but render children that do.
const AccountSummary = (props) => {
  return (
    <AccountSummaryHeader/>
    <AccountSummaryDashboard/>
    <AccountSummaryFooter/>
  );
};
```

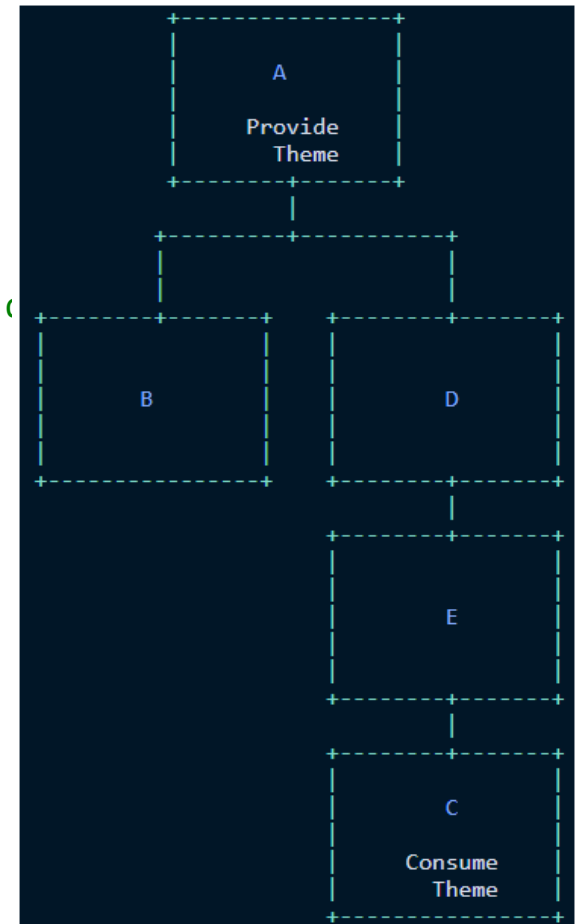
- All three of these child components may not want to access the current user's data. But as an example, let's just look at the AccountSummaryHeader component:

```
const AccountSummaryHeader = (props) => {
  // Here we retrieve the current value of the context
  const context = React.useContext(UserContext);
  return (
    <section><h2>{context.currentUser.name}</h2> </section>
  );
};
```

Context

- Using context, we can avoid passing props through intermediate elements

```
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
// A component in the middle doesn't have to pass the theme
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```



Bringing Bootstrap into your React App

- After creating your app project:
- `..Demo\second-app> npm install --save react-bootstrap bootstrap@3`
- Then start the app : `npm start`
- In `Index.js`, put these as the 1st 2 lines:
- `import 'bootstrap/dist/css/bootstrap.css';`
- `import 'bootstrap/dist/css/bootstrap-theme.css';`
- Now use Bootstrap classes in `App.js` or any other component:
- Eg :

```
render(){  
  return(  
    <div className="container">  
      <h1 className="text-danger">TODO LIST </h1>  
    </div>  
  );  
}
```

REDUX

Because state management can be hard

What is state

- Eg:

```
const state = {  
  posts: [],  
  signUpModal: {  
    open: false  
  }  
}
```

```
<div className={this.state.signUpModal.open ? 'hidden' : ''}>  
  Sign Up Modal  
</div>
```
- state references the condition of something at a particular point in time, such as whether a modal is open or not.
- In a React component the state holds data which can be rendered to the user.
- The state in React could also change in response to actions and events: in fact you can update the local component's state with `this.setState()`.
- So, in general a typical JavaScript application is full of state. For example, state is:
 - what the user sees (data)
 - the data we fetch from an API
 - the URL
 - the items selected inside a page
 - eventual errors to show to the user

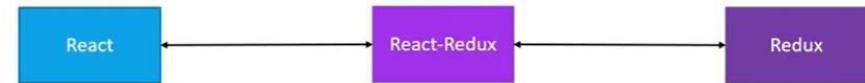
State can be complex

- Even an innocent single page app could grow out of control without clear boundaries between every layer of the application. This holds particularly true in React.
 - You can get by with keeping the state within a parent React component (or in context) as long as the application remains small.
 - Then things will become tricky especially when you add more behaviours to the app. At some point you may want to reach for a consistent way to keep track of state changes.

Create a Redux application

```
npx create-react-app redux-app  
cd redux-app  
npm install redux react-redux
```

React-Redux is the official Redux UI binding library for React



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

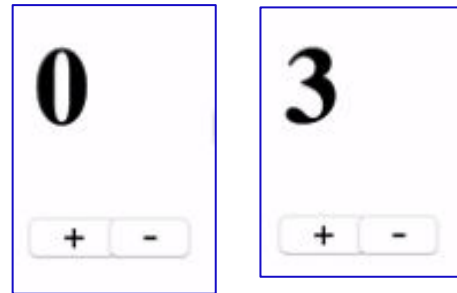
//STORE -> GLOBALISED STATE

//ACTION -> INCREMENT

//REDUCER

//DISPATCH

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



Create a Redux application

- Step-1 : Create the store:

```
import {createStore} from 'redux';  
const myStore = createStore(reducer-name)
```

- Step-2 : Create action

```
const increment = () => {  
  return {  
    type : 'INCREMENT' //name of the action  
  }  
}  
const decrement = () => {  
  return {  
    type : 'DECREMENT' //name of the action  
  }  
}
```

- Step-3 : Create reducer

```
function reducer(state=initial-state, action){}
```

```
const counter = (state=0, action) => {  
  switch(action.type){  
    case "INCREMENT":  
      return state + 1;  
    case "DECREMENT":  
      return state - 1;  
  }  
}  
let store = createStore(counter)
```

Create a Redux application

- Step-4 : display store on console
- `store.subscribe(() => console.log(store.getState()))`
- Step-5 : dispatch the action
- `store.dispatch(increment()); //dispatches the increment action`

```
//DISPATCH
store.dispatch(increment()); //dispatches the increment action
store.dispatch(decrement()); //dispatches the decrement action
store.dispatch(decrement()); //dispatches the decrement action again
```

- Step-6 : Execute the app.
Start server
- `npm start`



Need for Redux

- Redux offers a solution to storing all your application state in one place called **“Store”**
- Components then **“dispatch”** state changes to store, not directly to other components
- Components that need to be aware of state changes can **“subscribe”** to the store
- The center of every Redux application is the store. A "store" is a container that holds your application's global state.
 - A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object:
 - You must never directly modify or change the state that is kept inside the Redux store
 - Instead, the only way to cause an update to the state is to create a plain action object that describes "something that happened in the application", and then dispatch the action to the store to tell it what happened.
 - When an action is dispatched, the store runs the root reducer function, and lets it calculate the new state based on the old state and the action
 - Finally, the store notifies subscribers that the state has been updated so the UI can be updated with the new data.

Actions, reducers and dispatchers

- An **action** is a plain JavaScript object that has a type field. You can think of an action as an event that describes something that happened in the application.
 - The type field should be a string that gives this action a descriptive name. Eg "todoAdded" or "depositFunds" or "incrementCounter"
- A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state:
(state, action) => newState.
 - You can think of a reducer as an event listener which handles events based on the received action (event) type.
- The Redux store has a method called **dispatch**. The only way to update the state is to call store.dispatch() and pass in an action object.
 - The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:

My original index.js

```
//original React imports
import {createStore} from 'redux';

//STORE -> GLOBALISED STATE

//ACTION -> INCREMENT
const increment = () => {
  return {
    type: 'INCREMENT' //name of the action
  }
}
const decrement = () => {
  return {
    type: 'DECREMENT' //name of the action
  }
}

//REDUCER
const counter = (state=0, action) => {
  switch(action.type){
    case "INCREMENT":
      return state + 1;
    case "DECREMENT":
      return state - 1;
  }
}
```

```
let store = createStore(counter);

//Display it in console
store.subscribe(() =>
  console.log(store.getState()));

//DISPATCH
store.dispatch(increment());
store.dispatch(decrement());

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

Create a Redux app

```
//reducers/counter.js
const counterReducer = (state=0, action) => {
  switch(action.type){
    case "INCREMENT":
      return state + 1;
    case "DECREMENT":
      return state - 1;
    default: return null;
  }
}
export default counterReducer;
```

```
//src/index.js
import {createStore} from 'redux';
import allReducers from "../reducers";

const store = createStore(allReducers);
```

```
//reducers/isLogged.js
const loggedReducer = (state=false, action) => {
  switch(action.type){
    case "SIGNIN":
      return !state;
    default:
      return state;
  }
}
export default loggedReducer;
```

```
//reducers/index.js
import counterReducer from "../counter";
import loggedReducer from "../isLogged";
import {combineReducers} from 'redux';

const allReducers = combineReducers({
  counter : counterReducer,
  isLogged:loggedReducer
})
export default allReducers;
```

chrome.google.com/webstore/detail/redux-devtools/lmhkpbekcpmknklieibfkpmmfib... ☆

chrome web store

tshrilita@gmail.com

Extensions > Redux DevTools

Redux DevTools

Offered by: remotedevio

★★★★★ 528 | Developer Tools | 1,000,000+ users

Add to Chrome

github.com/zalmoxisus/redux-devtools-extension

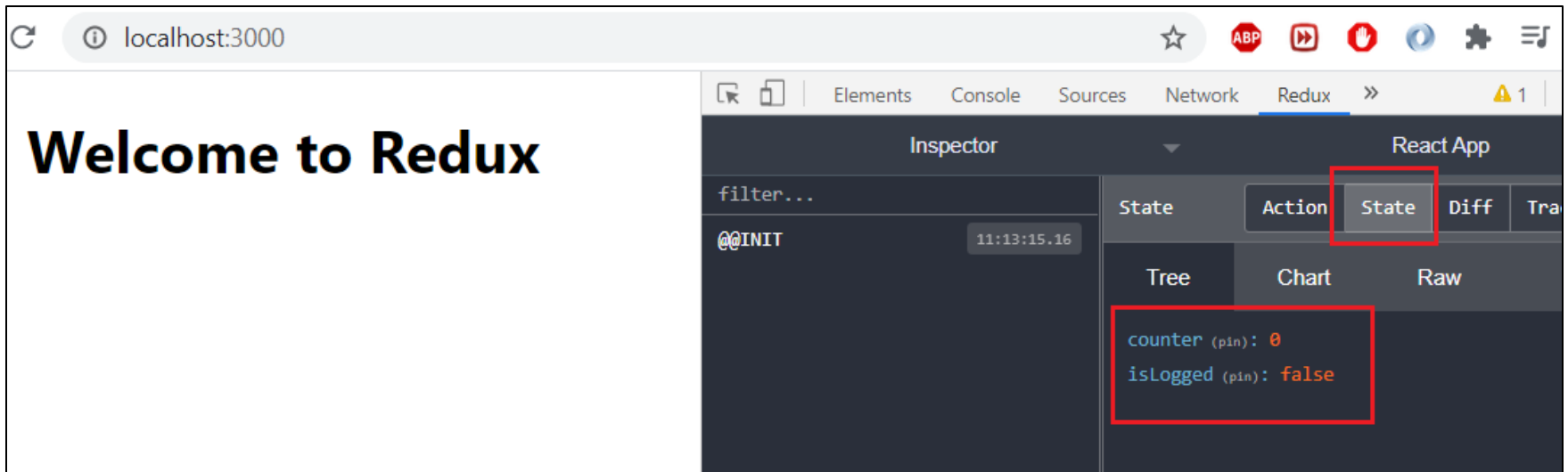
1. With Redux

1.1 Basic store

For a basic Redux store simply add:

```
const store = createStore(
  reducer, /* preloadedState, */
  + window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

```
//src/index.js
//Original code:
const store = createStore(allReducers);
//New code:
const store = createStore(allReducers,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__());
```



```
//src/index.js
import {createStore} from 'redux';
import allReducers from "././reducers";
import {Provider} from 'react-redux';
```

```
const myStore = createStore(allReducers,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__());
```

```
ReactDOM.render(
  <React.StrictMode>
    <Provider store={myStore}>
      <App />
    </Provider>
  </React.StrictMode>,
```

```
const counterReducer =
  (state=0, action) => {...}
```


App.js : displaying state

```
import './App.css';
import {useSelector} from 'react-redux';

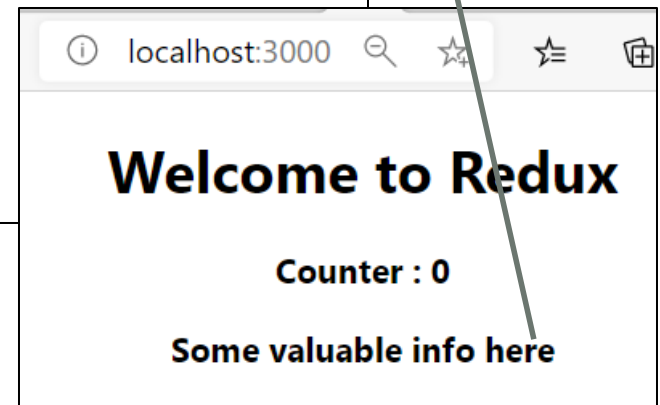
function App() {
  const counter = useSelector(state => state.counter);
  const isLogged = useSelector(state => state.isLogged);

  return (
    <div className="App">
      <h1>Welcome to Redux</h1>
      <h3>Counter : {counter}</h3>
      {isLogged ? <h3> Some valuable info here</h3> : ''}
    </div>
  );
}

export default App;
```

Allows you to extract data from the Redux store state, using a selector function.

I set the isLogged state to true



Modifying state

```
import './App.css';
import {useSelector, useDispatch} from 'react-redux';
import {increment} from './actions';
import {decrement} from './actions';

function App() {
  const counter = useSelector(state => state.counter);
  const isLogged = useSelector(state => state.isLogged);
  const dispatch = useDispatch();

  return (
    <div className="App">
      <h1>Welcome to Redux</h1>
      <h3>Counter : {counter}</h3>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
      {isLogged ? <h3> Some valuable info here</h3> : ''}
    </div>
  );
}
export default App;
```

```
//actions/index.js
export const increment = () => {
  return {
    type : 'INCREMENT'
  }
}

export const decrement = () => {
  return {
    type : 'DECREMENT'
  }
}
```

Welcome to Redux

Counter : 3

