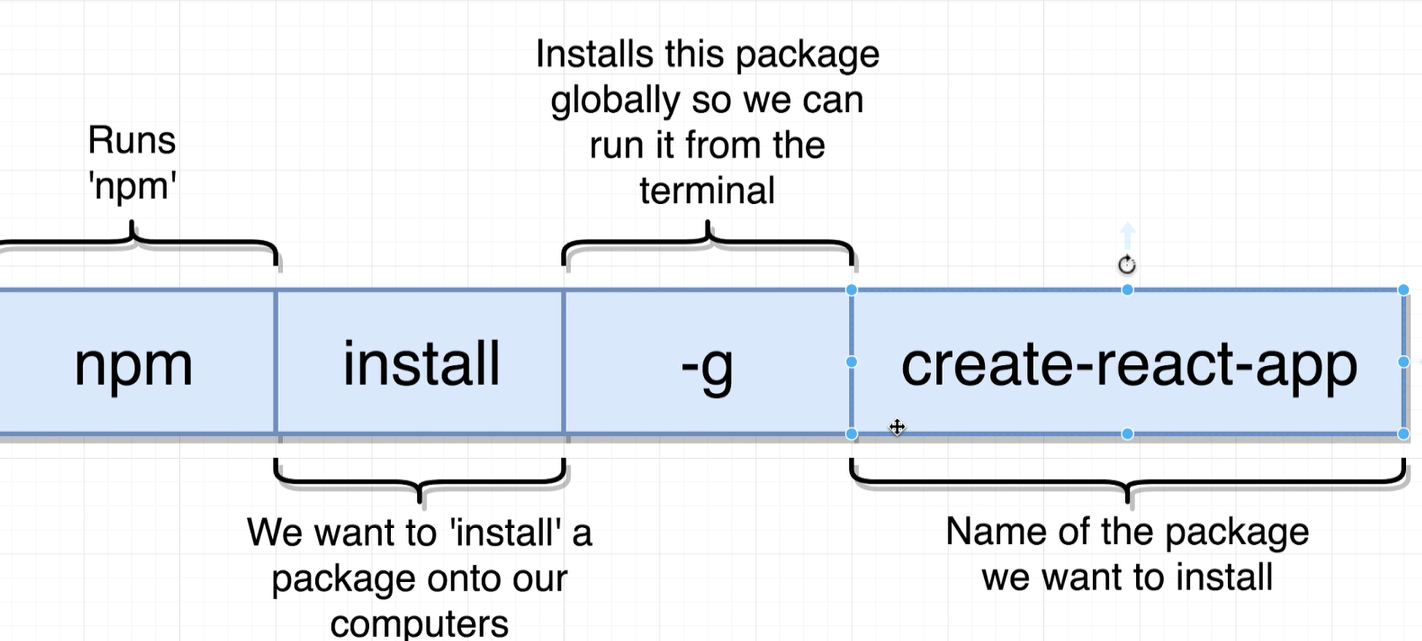
# **REACT JS**

## **What is ReactJS?**

React.js is an open-source **JavaScript library (Not a framework)** that is used for building user interfaces specifically for single-page applications. It’s used for handling the view layer for web and mobile apps. React also allows us to create reusable UI components.

**Create React App:**



**Alternatively**, we can use following commands,

npx create-react-app *app-name*

*To start a new Create React App project with TypeScript, we can run:*

npx create-react-app my-app --template typescript

***Start App:*** cd my-app

npm start

**Note:**

1. We have to first install nodejs on development server. Nodejs is not required to be installed on production (Any deployment) server.
2. App name should be in lower case.

### NPM - Manages packages but doesn't executing any. NPX - A tool for executing Node packages.

**Install Any Package:** npm install --save packageName

To install, any 3rd party package use npm install command

for ex: npm install --save faker

**Stop App:** Press “Control + C” at the terminal

When we run above command to create a react app, it adds (Installs) around 1800 packages. Why so many packages?

Answer: Under the hood, react uses [Babel](https://babeljs.io/) and [webpack](https://webpack.js.org/). Babel is consisting of tremendous number of packages, that’s the reason we are seeing so many packages.

**Build Production optimized build:** To create an optimized production build use “npm run build” command.

**File Paths:**

* / = Root Directory
* ./ = Current directory
* ../ = Parent of current directory (moves one directory backwards and starts there)
* ../../ = Two directories backwards

**Babel:** Babel is a free and open-source JavaScript transcompiler that is mainly used for following two purpose:

1. Compile JSX into regular JavaScript.  We only need Babel in development mode. When a React, app is shipped into production, it no longer needs to make transformations: the transformations will be hard-coded in place.

|  |  |
| --- | --- |
| const App = () => {  return <div>Pranamji!</div>;  }; | var App = function App() {  return React.createElement(  "div",  null,  "Pranamji!"  );  }; |

1. Babel is also used to convert ECMAScript 2015+ (ES6+) code into a backwards compatible version of JavaScript (EC5) in current and older browsers or environments. Babel is a popular tool for using the newest features of the JavaScript programming language.

|  |  |
| --- | --- |
|  |  |

For ex: Following arrow functions in our code will be transformed into ES5 compatible function expressions:

const fn = () => 1;

// Babel will convert it to following

var fn = function fn() {

return 1;

};

***Project Directory Structure:***

|  |  |
| --- | --- |
|  |  |

### **What is a component in ReactJS?**

Static websites tend to divide themselves into pages that may share UI with other pages or be completely different. ReactJS applications are split into components (e.g. dropdown, embedded video, carousel). This helps save time because developers only need to write a component once and can reuse it on any page.

***Basic Steps to show React Component on the screen.***

1. *Import the React and ReactDOM Libraries.*

|  |  |
| --- | --- |
| import React from 'react';  import ReactDOM from 'react-dom'; |  |

1. *Create a React Component*

const App = () => {

return <div>Pranamji, Shree Raj Shyamaji Sahay!</div>;

};

1. *Take a created react component and show it on the screen.*

ReactDOM.render(

<App />,

document.querySelector("#root")

);

***Note:*** Component name should be Pascal Case (First character should be upper case) only. If we use a lowercase for first character of component react is going to think that this is a normal vanilla HTML element (like div or span) that we're trying to render.

## **What is the difference between React, ReactDOM and React Native?**

React.js often referred to as React or ReactJS is a JavaScript library responsible for building a hierarchy of UI components or in other words, responsible for the rendering of UI components. It provides support for both frontend and server-side.

**ReactDOM** is the glue Connecting React to DOM. ReactDOM is a React library that allows us to use React to build web applications and websites. This is what React is most known for.

**React-Native:** It is a framework for building native iOS and Android app using JavaScript. React Native compiles to native app components, which makes it possible for us to build native mobile applications.

In React JS, React is the base abstraction of React DOM for the web platform, while with React Native, React is still the base abstraction but of React Native.

So, the syntax and workflow remain similar, but the components are different.

## **Why ReactJS**

* **JSX:** It allows developers to write their apps within JavaScript. It brings HTML directly into our JS.

**Inline Styling with JSX Rules:**

|  |  |
| --- | --- |
| Curly braces have got 2 usage here: -   * { .. } evaluates to an expression in JSX. * { key: value } implies a javascript object. Value will be always wrapped in string. |  |
| The style attribute accepts a JavaScript object with camelCased properties (Remove compound property delimeter e.g. “-“). | **HTML:**  <div style="border-color: red;"></div>  **JSX:**  <div style={{ borderColor: 'red' }}></div> |
| To specify a CSS class, use the className attribute.  **Why ClassName?** class is a keyword in javascript and JSX is an extension of javascript. So, to avoid collision, JSX uses className Attribute. | **HTML:**  <label class="caption" >Id:</label>  **JSX:**  <label className="caption" >Id:</label> |

**Referencing JS expression in JSX:** To render a Javascript expression in JSX, all we need to do is surround the expression in curly braces. Expression can be variable (Including Array variable), function or any other code.

|  |  |
| --- | --- |
| const buttonText = "Click Me";  <input type="button" value={buttonText}> </input> | const getButtonText = () => {  return "Click me";  };  <input type="button" value={getButtonText()}></input> |

**Note**: when we want to show one component inside of another component, we're going to treat it as though it were a JSX tag instead.

import CommentDetail from './CommentDetail';

//Create Component

const App = () => {

return (

<div className="ui container comments">

<CommentDetail />

</div>

);

};

**Multiple line in JSX:** When we split JSX over multiple lines for readability. While it is recommended (isn’t required) wrapping it in parentheses to avoid the pitfalls of [automatic semicolon insertion](https://stackoverflow.com/q/2846283).

return (

<div>

Pranam!

</div>

);

**Note:** We are not allowed to take a javascript object and reference it inside of JSX specifically where we would normally show text. This is because technically we can't show a JavaScript object as an attribute on an element.For ex:

const caption = {text: 'Submit’}

<input type="button" value={caption}></input>

To fix this issue, Use property of object. For ex:

<input type="button" value={caption.text}></input>

However, we can use javascript objects as long as we're not trying to print them up as text or something.

const style = { backgroundColor: "blue", color: "white" };

<input type="button" value="Submit" style={style}></input>

* **Components: Components** allow developers to break down complex UI. The idea of components is what makes ReactJS unique. Instead of worrying about the entire web app, it makes it possible to break the complex [UI/UX development](https://www.thinkwik.com/ui-ux-design-services.html) into simpler components. This is crucial in making every component more intuitive.
* **Props:** Pros in ReactJS make it possible to **pass the custom data from parent component to child components**.
* **State:** State is a JavaScript object that contains data that is relevant to a singular component. When using ReactJS, All the content that can change during the application will be placed it in a single location (State). For example, a user registration form might store whether or not the user has checked the "Terms and Conditions" checkbox in its state.
* **Virtual DOM:** The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called reconciliation.

In React, for every DOM object, there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy.

A virtual DOM object has the same properties as a real DOM object, but it lacks the power to directly change what’s on the screen (Real DOM).

Manipulating the real DOM is slow. Manipulating the virtual DOM is much faster, because nothing gets drawn onscreen. Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.

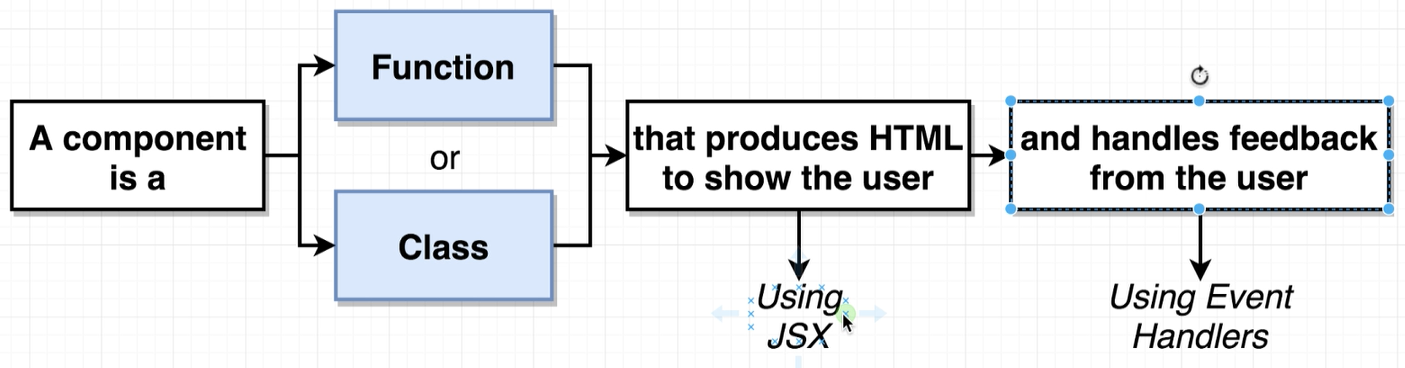
## **How Does Virtual DOM Work?**

Like the actual DOM, the Virtual DOM is a node tree that lists elements and their attributes and content as objects and properties. When we render JSX element using render() method, It creates a node tree (Virtual DOM) from React components and updates this tree in response to mutations in the data model, caused by actions. This sounds incredibly inefficient, but the cost is insignificant because the virtual DOM can update so quickly.

Each time the underlying data changes in a React app, a new Virtual DOM representation of the user interface (Component) is created. This is where things get interesting.

1. In React every UI piece is a component, and each component has a state. React follows the observable pattern and listens for state changes. When the state of a component changes, react updates the virtual DOM tree. Once the virtual DOM has been updated, react then compares the current version of the virtual DOM with the previous version of the virtual DOM. This process is called “diffing”.
2. Once React knows which virtual DOM objects have changed, then React updates those objects, and only those objects, on the real DOM (changes are drawn onscreen). This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost of updating the real DOM. React’s reputation for performance comes largely from this innovation. This is very much like applying a patch.

**React Component:** React lets us define components as classes or functions.

**

1. **Function Component:**  The simplest way to define a component is to write a JavaScript function.

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

This function is a valid React component because it accepts a single “prop” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

1. **Class Component:** To define a React component class, we need follow following rules:
   * Must be JS class
   * We need to extend React.Component
   * Must define a ‘render’ method that returns some amount of JSX. . All the other methods are optional.

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

### **Structuring Apps with Class-Based Components (Class vs functional Components):**

1. **Syntax:**

A functional component is just a plain JavaScript function which accepts props as an argument and returns a React element.

A class component requires us to extend from React.Component and create a render function which returns a React element. This requires more code but will also give us some benefits.

1. **State:**

Because a functional component is just a plain JavaScript function, we cannot use setState() in component. That’s the reason it’s called functional stateless components. So, every time we see a functional component, we can be sure that this particular component doesn’t have its own state.

If we need a state in our component, we will either need to create a class component or lift the state up to the parent component and pass it down the functional component via props.

**Note:** With react 16.8 Hooks update! We can now use the useState hook to use state in our functional components.

1. **Lifecycle Hooks:**

Another feature which we cannot use in functional components are lifecycle hooks. The reason is the same like for state, all lifecycle hooks are coming from the React.Component which we extend from in class components.

So, if we need lifecycle hooks we should probably use a class component.

**Note:** With react 16.8 Hooks update! We can now use the useEffect hook to use lifecycle event in our functional components.

**Note:** **React team strongly recommend against creating our own base component classes.** In React components, [code reuse is primarily achieved through composition rather than inheritance](https://reactjs.org/docs/composition-vs-inheritance.html).

* **Import and Export Component:** latest ES2015 provides a more advanced module importing/exporting pattern to import and export modules.

React uses the same features as mentioned above, and you we treat each React Component as a module itself. Thus, it is possible to import/export React Components and is one of the basic operations to be performed.

**Import:** In React we use the keyword import and from to import a particular module or a named parameter.

* **Importing default export:** Every module is said to have at most one default export. In order to import the default export from a file, we can use only the address and use the keyword import before it, or we can give a name to the import making the syntax as the following.

import GIVEN\_NAME from ADDRESS

* **Importing named values:** Every module can have several named parameters and in order to import one we should use the syntax as follows.

import { PARA\_NAME } from ADDRESS

And similarly, for multiple such imports we can use a comma to separate two parameter name within the curly braces.

* **Importing a combination of Default Exports and Named Values:** The title makes it clear what we need to see is that the syntax of the same. In order to import a combination, we should use the following syntax.

import GIVEN\_NAME, { PARA\_NAME, ... } from ADDRESS

**Export:** Importing is possible only if the module or named property to be imported has been exported in its declaration. In React we use the keyword export to export a particular module or a named parameter or a combination.

* **Exporting default export:** Every module have at most one default export. In order to export the default export from a file, we need to follow the syntax described below.

export default GIVEN\_NAME

* **Exporting named values:** Every module can have several named parameters and in order to export one we should use the syntax as follows.

export { PARA\_NAME }

And similarly, for multiple such exports we can use a comma to separate two parameter names within the curly braces.

* **Props:** Pros in ReactJS make it possible to **pass the custom data from parent component to child components**.

It allows us to customize or configure (How it looks or how user interacts with it) child component.

Data (props) is never passed up from the child to the parent, hence the phrase, unidirectional data flow. This is a powerful concept because it leads to a more predictable application and creates a single source of truth so that any changes to the parent’s state propagate to all its children consistently.

When we use the prop system its two-step process.

1. Pass data from the parent to the child
2. Child consumes or makes use of that data

|  |  |
| --- | --- |
| **Parent component:**  <CommentDetail  author={author.fullName}  age={author.age} | **Child Component:**  const CommentDetail = (props) => {  return (  <span>{props.author}</span>  <span>{props.age}</span>  );  }; |

#### **Default Props:** defaultProps can be defined as a property on the component (class or functional component) itself. It allow us to set the default props for the component. This is used for undefined props, but not for null props. For example:

CommentDetail.**defaultProps** = {

author: "Not spcified",

age: 'N/A'

};

If props.auther is not provided, it will be set by default to ‘Not Specified' and age to “N/A”, If props.author or props.age is set to null, it will remain null.

#### ***Communicating from child to parent:*** The standard way of doing this is to have the parent pass a function to the child through ***props***. The child then calls this function at some point and passes it a value that the parent is supposed to react to. We then write the functionality for the parent's reaction inside the parent component.

#### 

**Parent Component:**

**class** App **extends** React**.**Component **{**

onSearchBarSubmit **=** **(**term**)** **=>** **{**

console**.**log**(**term**);**

**};**

render**()** **{**

**return** **<**SearchBar onSearchBarSubmit**={this.**onSearchBarSubmit**}/>;**

**};**

**}**

**Child Component:**

**class** SearchBar **extends** React**.**Component **{**

**constructor(**props**)** **{**

**super(**props**);**

**};**

state **=** **{** term**:** "Pranam!" **}**

onFormSubmit **=** **(event)** **=>** **{**

**this.**props**.**onSearchBarSubmit**(this.**state**.**term**);**

**event.**preventDefault**();**

**};**

render**()** **{**

**return** **(**

**<form** className**=**"ui form" onSubmit**={this.**onFormSubmit**}>**

**<**input type**=**"text"

value**={this.**state**.**term**}**

onChange**={(**e**)** **=>** **{** **this.**setState**({**term**:** e**.**target**.**value**})** **}}** **/>**

**</form>**

**);**

**};**

**}**

### **Are React props immutable?**

React props are immutable. A component can only inherit them, but not modify them. If they do need to be modified, this can only happen by “asking” the parent component to do so. This can be done by using a callback or a hook.

#### **Component Reuse using composition over inheritance (Component Children):**

React has a powerful composition model, and It is recommended to use composition instead of inheritance to reuse code between components.

**props.children:** Some components don’t know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic “boxes”. To create this kind of component we can use props.children to render other component inside reusable parent component. This is a great way to create UI components, like cards, headers, and buttons.

**Comment Component**

const CommentDetail = (props) => {

return (

<div className="comment">

<a href="/" className="avatar">

<img alt="avatar" src={props.avatar}></img>

</a>

<div className="content">

<a href="/" className="author">{props.author}</a>

<div className="metadata">

<span className="date">{props.timeAgo}</span>

</div>

<div className="text">

{props.comment}

</div>

</div>

</div>

);

};

export default CommentDetail;

**Generic Reusable Component**

const ApprovalCard = (props) => {

return (

<div className="ui card">

<div className="content">

**{props.children}**

</div>

<div className="extra content">

<div className="ui two buttons">

<div className="ui basic green button">Approve</div>

<div className="ui basic red button">Decline</div>

</div>

</div>

</div>

);

};

export default ApprovalCard;

**Main Component**

|  |  |
| --- | --- |
| import faker from 'faker';  import CommentDetail from './CommentDetail';  import ApprovalCard from './ApprovalCard';  //Create Component  const App = () => {  return (  <div className="ui container comments">  <ApprovalCard>  <CommentDetail  author={faker.name.findName()}  timeAgo="Today at 3:00 PM"  avatar={faker.image.avatar()}  comment="Pranam" />  </ApprovalCard>  <ApprovalCard>  <CommentDetail  author={faker.name.findName()}  timeAgo="Yesterday at 4:00 AM"  avatar={faker.image.avatar()}  comment="Shree RajShyamaji" />  </ApprovalCard>  </div>  );  }; |  |

Occasionally, we may want to interact with the children, maybe mutating or separating them. We should treat this.props.children as an [opaque data structure](http://facebook.github.io/react/tips/children-props-type.html). Instead of working with this.props.children directly, use the [React.Children](http://facebook.github.io/react/docs/top-level-api.html#react.children) utilities to work with children.

**What About Inheritance?**

At Facebook, they have created thousands of components, and haven’t found any use cases where they would recommend creating component inheritance hierarchies.

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

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

## **Section 5: State in React Components**

**Overview**

1. State is a JavaScript object that contains data that is relevant to a singular component. When using ReactJS, All the content that can change during the application will be placed it in a single location (State).
2. Updating States or updating properties inside this javascript object on a component will cause our component to almost instantly re-render on the screen.
3. State must be initialized when component is created.

We can either do it in constructor or using abbreviation method.

Please note that when we use abbreviated syntax, babelJS is going to define the constructor function and it will initialize state inside constructor for us.

So basically, there is no difference between using constructor or shortened syntax initialize state.

**Constructor:**

class App extends React.Component {

constructor(props) {

super(props);

this.state = { lat: null, error: null };

}

}

**Alternate abbreviated way:**

class App extends React.Component {

state = { lat: null, error: null };

**}**

1. State can only be updated using the function ‘setState’ method except initialization in constructor, where we do direct assignment.

componentDidMount() {

window.navigator.geolocation.getCurrentPosition(

(pos) => {

this.setState({ lat: pos.coords.latitude });

},

(err) => {

this.setState({ error: err.message });

}

)

}

### **What is State in JavaScript**

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

**SetState Callback function:** setState is asynchronous. It means we can’t call setState on one line and assume state has changed on the next. To get upto date value use callback function of the setState function.

this.setState(state, callback);

The second parameter [this.setState()](https://linguinecode.com/post/master-react-state-and-props) accepts is the callback function, and that’s where we want to add our side effects.

For ex:

**this.**setState**({** boardAddModalShow**:** **true** **});**

// After setting a new state it still return a **false** value

console**.**log**(this.**state**.**boardAddModalShow**);**

The reason behind this is, state needs some time to mutate, and since console.log(this.state.boardAddModalShow)executes before the state mutates, we get the previous value as output. So, we need to write the console in the callback to the setState function.

**this.**setState**({** boardAddModalShow**:** **true** **},**

**()** **=>** **{** console**.**log**(this.**state**.**boardAddModalShow**);** **});**

According to React docs,

setState() does not immediately mutate this.state but creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to setState and calls may be batched for performance gains**.**

### **Props vs State in React**

In a React component, props are variables passed to it by its parent component. State on the other hand is still variables, but directly initialized and managed by the component.

Props should never be changed in a child component, so if there’s something going on that alters some variable, that variable should belong to the component state.

Props are also used to allow child components to access methods defined in the parent component. This is a good way to centralize managing the state in the parent component, and avoid children to have the need to have their own state.

Most of our components will just display some kind of information based on the props they received, and stay stateless.

### **Can you share a component’s state?**

Developers can share a component's state by passing it to child components (ones that are rendered by the parent) via their props. In a user registration form, we can disable the submit button until the user has accepted the "Terms and Conditions." The form can share the checkbox’ state via the button’s props.

**Can you share a component’s state with a parent or sibling component in React?**

No, you can’t. In React, data flows from the root level parents to the most deeply nested child. This is why it can be necessary to keep a central store of data and access it where you need it instead of endlessly passing it from parent to child.

**Why would they make setState async**

This is because setState alters the state and causes rerendering. This can be an expensive operation and making it synchronous might leave the browser unresponsive. Thus the setState calls are asynchronous as well as batched for better UI experience and performance.

### **Section -6 The Component Lifecycle:** Component lifecycle method is a function that we can optionally define inside of our class components. If implemented, it will be called automatically by react at certain points during a component's life cycle.

### There are three main phases of a component which including mounting, updating and unmounting.

|  |  |
| --- | --- |
|  |  |

#### **Mounting**

These methods are called in the following order when an instance of a react component is being created and inserted into the DOM:

1. **constructor ()**
2. **static getDerivedStateFromProps ()**
3. **render ()**
4. **componentDidMount()**
5. **constructor ():**

**Syntax:** constructor(props)

The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, we should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

* Initializing [local state](https://reactjs.org/docs/state-and-lifecycle.html) by assigning an object to this.state.
* Binding [event handler](https://reactjs.org/docs/handling-events.html) methods to an instance.

**Note:** We **should not call setState()** in the constructor(). Instead, if our component needs to use local state, **assign the initial state to this.state** directly in the constructor:

constructor(props) {

super(props);

// Don't call this.setState() here!

this.state = { counter: 0 };

this.handleClick = this.handleClick.bind(this);

}

Constructor is the only place where we should assign this.state directly. In all other methods, we need to use this.setState() instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use componentDidMount() instead.

**Note: Avoid copying props into state! This is a common mistake:**

constructor(props) {

super(props);

// Don't do this!

this.state = { color: props.color };

}

**The problem is that it’s both unnecessary (we can use this.props.color directly instead), and creates bugs (updates to the color prop won’t be reflected in the state).**

**Only use this pattern if we intentionally want to ignore prop updates.** In that case, it makes sense to rename the prop to be called initialColor or defaultColor. We can then force a component to “reset” its internal state by [changing its key](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) when necessary.

1. **\*static getDerivedStateFromProps()\*:**

Syntax: static getDerivedStateFromProps(props, state)

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

This is a *static* function that does not have access to “*this*“.

*getDerivedStateFromProps()* returns an object to update *state* in response to *prop* changes. It can return a *null* if there is no change to state.

This method also exists only for rare use cases where the state depends on changes in props in a component.

static getDerivedStateFromProps(props, state) {

if (props.currentRow !== state.lastRow) {

return {

isScrollingDown: props.currentRow > state.lastRow,

lastRow: props.currentRow,

};

}

// Return null to indicate no change to state.

return null;

}

1. **Render():** The render() method is the most used lifecycle method. This is because render() is the only required method within a class component in React.

As the name suggests it handles the rendering of component to the UI. It happens during the mounting and updating of our component.

Below is an example of a simple render() in React.

class Hello extends Component{

render(){

return <div>Hello {this.props.name}</div>

}

}

As we can see in the example above, the render() method returns JSX that is displayed in the UI. A render() can also return a null if there is nothing to render for that component.

**A render() method has to be pure with no side-effects,**

* + **We should return JSX nothing else.**
  + **We should not modify the component state or make network request.**
  + **We should avoid conditional renders (Conditional based content render), for ex.**

if (!this.state.error && this.state.lat)

return <SeasonDisplay lat={this.state.lat} />

return <div>Error: {this.state.error}</div>

Instead create helper function and call the helper method from render method.

**getRenderContent** = () => {

if (!this.state.error && this.state.lat) {

return <SeasonDisplay lat={this.state.lat} />

}

return <div>Error: {this.state.error}</div>

};

**render()**{

return <div> {this.getRenderContent()} </div>

};

* + it should return the same result each time it’s invoked, and it does not directly interact with the browser.
  + If we need to interact with the browser, perform work in componentDidMount() or the other lifecycle methods instead. Keeping render() pure makes components easier to think about.

1. **componentDidMount():** It is invoked immediately after a component is mounted (inserted into the tree).
   1. Initialization that requires DOM nodes should go here.
   2. If we need to **load data from a remote endpoint**, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don’t forget to unsubscribe in componentWillUnmount().

We may call setState() immediately in componentDidMount(). It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the render() will be called twice in this case, the user won’t see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, we should be able to assign the initial state in the constructor() instead. It can, however, be necessary for cases like modals and tooltips when we need to measure a DOM node before rendering something that depends on its size or position.

#### **Updating:** An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* 1. static getDerivedStateFromProps()
  2. shouldComponentUpdate()
  3. render()
  4. getSnapshotBeforeUpdate()
  5. componentDidUpdate()

B: **\*shouldComponentUpdate\*:** This lifecycle can be handy sometimes when we don’t want React to render our state or prop changes.

Anytime setState() is called, the component re-renders by default.

The shouldComponentUpdate() method is used to let React know if a component is not affected by the state and prop changes.

Keep in mind that this lifecycle method should be sparingly used, and it exists only for certain performance optimizations. We cannot update component state in shouldComponentUpdate() lifecycle.

**Caution:**Most importantly, do not always rely on it to prevent rendering of our component, since it can lead to several bugs.

shouldComponentUpdate(nextProps, nextState) {

return this.props.title !== nextProps.title ||

this.state.input !== nextState.input }

As shown in the example above, this lifecycle should always return a boolean value to the question, “***Should I re-render my component?***”

D: **\*getSnapshotBeforeUpdate()\*:**

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables our component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to componentDidUpdate().

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

A snapshot value (or null) should be returned.

class ScrollingList extends React.Component {

constructor(props) {

super(props);

this.listRef = React.createRef();

}

getSnapshotBeforeUpdate(prevProps, prevState) {

// Are we adding new items to the list?

// Capture the scroll position so we can adjust scroll later.

if (prevProps.list.length < this.props.list.length) {

const list = this.listRef.current;

return list.scrollHeight - list.scrollTop;

}

return null;

}

componentDidUpdate(prevProps, prevState, snapshot) {

// If we have a snapshot value, we've just added new items.

// Adjust scroll so these new items don't push the old ones out of view.

// (snapshot here is the value returned from getSnapshotBeforeUpdate)

if (snapshot !== null) {

const list = this.listRef.current;

list.scrollTop = list.scrollHeight - snapshot;

}

}

render() {

return (

<div ref={this.listRef}>{/\* ...contents... \*/}</div>

);

}

}

#### **E: componentDidUpdate:**

componentDidUpdate(prevProps, prevState, snapshot)

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

componentDidUpdate(prevProps) {

// Typical usage (don't forget to compare props):

if (this.props.userID !== prevProps.userID) {

this.fetchData(this.props.userID);

}

}

We may call setState() immediately in componentDidUpdate() but note that it must be wrapped in a condition like in the example above, or you’ll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance. If we’re trying to “mirror” some state to a prop coming from above, consider using the prop directly instead. Read more about why copying props into state causes bugs.

If our component implements the getSnapshotBeforeUpdate() lifecycle (which is rare), the value it returns will be passed as a third “snapshot” parameter to componentDidUpdate(). Otherwise this parameter will be undefined.

**Note:** componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.

#### **Unmounting:** This method is called when a component is being removed from the DOM:

#### A: **componentWillUnmount():** componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

#### You should not call setState() in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

#### **Error Handling:** These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

*A:* ***static getDerivedStateFromError(error):***This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

// Update state so the next render will show the fallback UI.

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, info) {

logComponentStackToMyService(info.componentStack);

}

render() {

if (this.state.hasError) {

// We can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

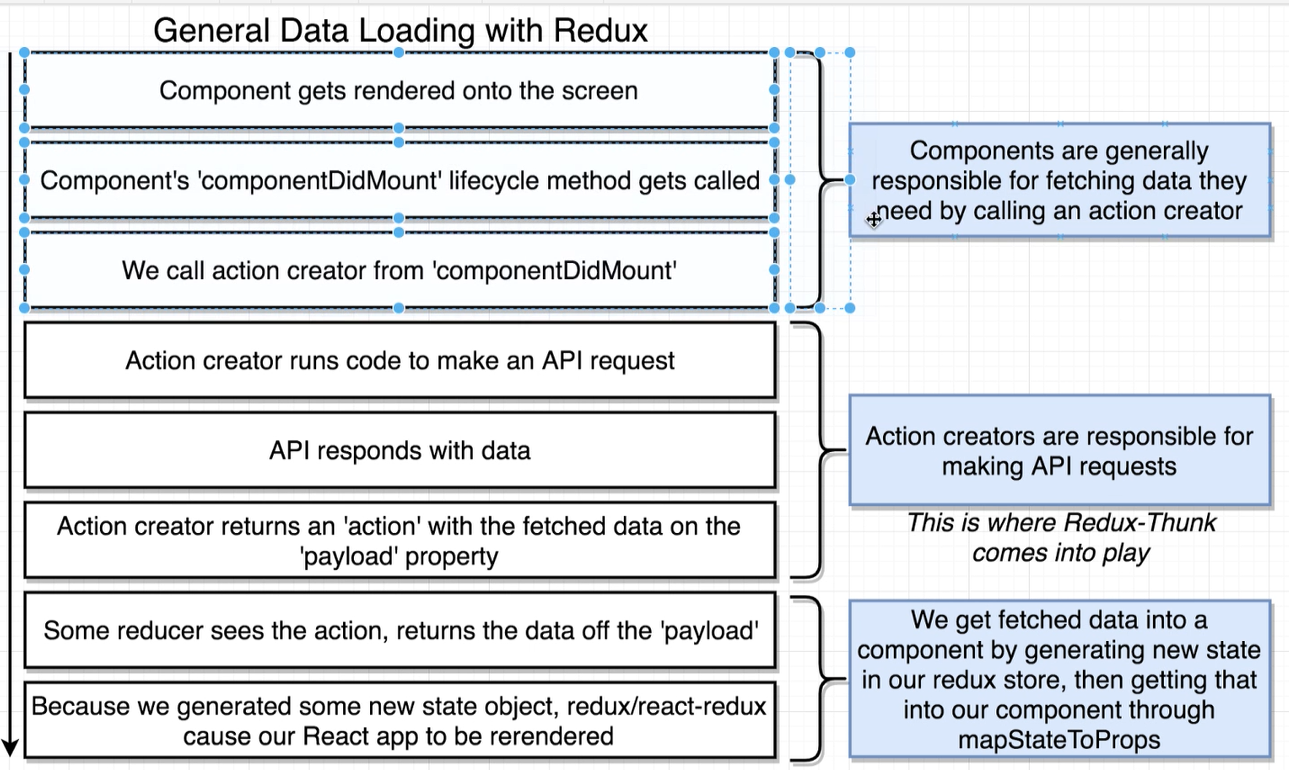
}

**B: componentDidCatch():** This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

error - The error that was thrown.

info - An object with a componentStack key containing information about which component threw the error.

componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors:

****

## **Understand one-way data bindings (Unidirectional Data Flow):**

## React apps are organized as a series of nested components.

## Unidirectional data flow is a technique that is also known as one-way data flow, in which means the data has one, and only one way to be transferred to other parts of the application. In essence, this means child components are not able to update the data that is coming from the parent component. In React, data coming from a parent is called **props**. Angular makes use of bi-directional binding in which the data flow takes place in both directions. React doesn’t support bi-directional binding to make sure you are following a clean data flow architecture. The major benefit of this approach is that data flows throughout our app in a single direction, giving you better control over it.

## **Section 7: Handling User Input with Forms and Events**

**Background:**

When we bind value to input element as following, react will create readonly input field. We won’t be able to change the value of input field.

<Form.Input placeholder="Title" value={activity.title} />

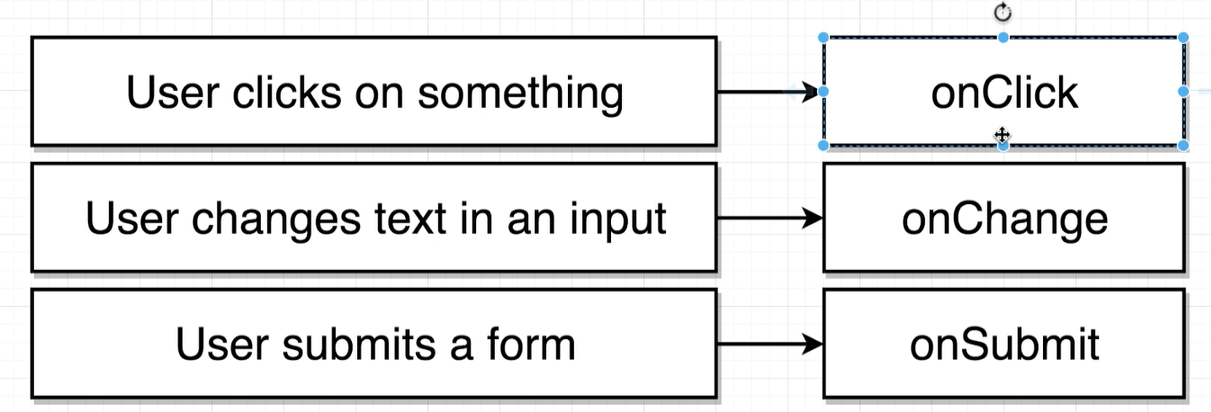
The reason our field is read only is because we've given our form inputs *value* property directly and because of this react wants to know about the values inside this particular input field.

if we remove value attribute from input field, we can enter and change the value of input field.

<Form.Input placeholder="Title" />

However, when we remove the value attribute, we are directly working with actual DOM element and React does not work directly with DOM elements instead react works with virtual DOM and pushes changes to actual DOM.

So, what we need to do is we need to give each of these input fields an **onChange** events and update the component states and then react will take care of updating the virtual DOM which will end up the actual DOM so that we can see the changes in our input.

******

### **Controlled vs Uncontrolled Components in React**

1. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself. In most cases, React Teams recommend using controlled components to implement forms.
2. **Uncontrolled Components:** Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components. It can also be slightly less code if we want to be quick and dirty. Otherwise, it is recommended to use controlled components.

To write an uncontrolled component, instead of writing an event handler for every state update, we can use a ref to get form values from the DOM.

For example, this code accepts a single name in an uncontrolled component:

**class** NameForm **extends** React**.**Component **{**

**constructor(**props**)** **{**

**super(**props**);**

**this.**handleSubmit **=** **this.**handleSubmit**.**bind**(this);**

**this.**input **=** React**.**createRef**();**

**}**

handleSubmit**(event)** **{**

**alert(**'A name was submitted: ' **+** **this.**input**.**current**.**value**);**

**event.**preventDefault**();**

**}**

render**()** **{**

**return** **(**

**<form** onSubmit**={this.**handleSubmit**}>**

Name**:** **<**input type**=**"text" ref**={this.**input**}** **/>**

**<**input type**=**"submit" value**=**"Submit" **/>**

**</form>**

**);**

**}**

**}**

1. **Controlled Components:** In a controlled component, the **form data is handled by the state within the react component**. The state within the component serves as “the single source of truth” for the input elements that are rendered by the component.
   * When changes are made to any of the input elements that have an event handler, the handler is fired.
   * The handler calls setState() as we can see below. This updates the state within the component.
   * When a state update occurs via setState(), it causes the component to re-render and the newly entered value is displayed in the element. This means our data (state) and UI (inputs) are always in sync. An input form element whose value is controlled by React in this way is called a “controlled component”.

This also means that the form component can respond to input changes immediately; for example, by: in-place feedback, like validations disabling the button unless all fields have valid data enforcing a specific input format, like credit card numbers etc.

* + We don’t need a form element on the page for the component to be a controlled component.

**class** SearchBar **extends** React**.**Component **{**

state **=** **{** term**:** "Pranam" **};**

render**()** **{**

**return** **(**

**<**input type**=**"text"

value**={this.**state**.**term**}**

onChange**={(**e**)** **=>** **this.**setState**({** term**:** e**.**target**.**value **})}**

**/>**

**);**

**};**

**}**

|  |  |  |
| --- | --- | --- |
| **Feature** | **uncontrolled** | **controlled** |
| one-time value retrieval (e.g. on submit) | Yes | Yes |
| validating on submit | Yes | Yes |
| instant field validation | **No** | Yes |
| conditionally disabling submit button | **No** | Yes |
| enforcing input format | **No** | Yes |
| several inputs for one piece of data | **No** | Yes |
| dynamic inputs | **No** | Yes |

### **Why do I need Keys in React Lists?**

 in React, when we use lists, each list item needs a unique key[.](https://medium.com/@adhithiravi?source=post_page-----dbb522188bbb----------------------)**Rendering a simple List component:**

**const** myList **=** **[{** id**:** 'a'**,** value**:** 'apple' **},**

**{** id**:** 'b'**,** value**:** 'orange' **},**

**{** id**:** 'c'**,** value**:** 'strawberry' **},**

**{** id**:** 'd'**,** value**:** 'blueberry' **},**

**{** id**:** 'e'**,** value**:** 'avocado' **}];**

**function** ListComponent**(**props**)** **{**

**const** listItems **=** myList**.**map**((**item**)** **=>**

**<**li**>{**item**}</**li**>**

**);**

**return** **(**

**<**ul**>{**listItems**}</**ul**>**

**);**

**}**

The code above shows a ListComponent that renders a list of items that are passed to it as props. In the render() method we have invoked the ListComponent and passed to it a list myList as the props.

when you run this code, you will notice that React also throws a warning.

“Warning: Each child in an array or iterator should have a unique ‘key’ prop..

Notice here that the warning is about using a unique key. Keys are necessary to improve performance of our React app.

# **How do you use Keys in Lists?**

Keys help React identify which items have changed (added/removed/re-ordered). To give a unique identity to every element inside the array, a key is required.

**function** ListComponent**(**props**)** **{**

**const** listItems **=** myList**.**map**((**item**)** **=>**

**<**li **key={item.id}>**

**{**item**.**value**}**

**</**li**>**

**);**

**return** **(**

**<**ul**>{**listItems**}</**ul**>**

**);**

**}**

In the code snippet above we can notice that we have included a key to each list item.

# **Can I just use indexes as keys? — Only under some exceptions**

React recommends that we do not use indexes as keys, since it could impact performance negatively and could lead to some unstable component behavior.

When we use indexes as keys, reordering a list, or adding and removing items from a list can cause issues with the component state, when indexes are used as keys. If the key is an index, reordering an item changes it. Hence, the component state can get mixed up and may use the old key for a different component instance.

Therefore, avoid this practice, and make sure unique ids are generated to be assigned as key.

What are some exceptions where it is safe to use index as key?

* If our list is static and will not change.
* The list will never be re-ordered.
* The list will not be filtered (adding/removing items from the list).
* There are no ids for the items in the list.

## **Refs and the DOM:**

Refs provide a way to access DOM nodes or React elements created in the render method.

refs are used to get reference to a DOM (Document Object Model) node or an instance of a component in a React Application i.e. refs would return the node we are referencing.

**Creating Refs:** 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**}** **/>;**

**}**

**}**

**Accessing Refs:** When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

**const** node **=** **this.**myRef**.**current**;**

The value of the ref differs depending on the type of the node:

* When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.
* When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current.
* You may not use the ref attribute on function components because they don’t have instances.

Following react component set the div height dynamically based on image height.

// JavaScript source code

**import** React **from** 'react'**;**

**class** ImageCard **extends** React**.**Component **{**

**constructor(**props**)** **{**

**super(**props**);**

**this.**state **=** **{** gridRowEndSpan**:** 0 **};**

**this.**imageRef **=** React**.**createRef**();**

**}**

componentDidMount**()** **{**

**this.**imageRef**.**current**.**addEventListener**(**'load'**,** **this.**onImageLoad**);**

**};**

onImageLoad **=** **()** **=>** **{**

**const** imgHeight **=** **this.**imageRef**.**current**.**clientHeight**;**

**const** gridRowEndSpan **=** Math**.**ceil**(**imgHeight **/** 10**);**

**this.**setState**({** gridRowEndSpan**:** gridRowEndSpan **});**

**};**

render**()** **{**

**const** **{** description**,** urls **}** **=** **this.**props**.image;**

**return** **(**

**<**div style**={{** gridRowEnd**:** `span ${this.state.gridRowEndSpan}` **}}>**

**<**img ref**={this.**imageRef**}** alt**={**description**}**

src**={**urls**.**regular**}** **/>**

**</**div **>**

**);**

**}**

**};**

**export** **default** ImageCard**;**

# **What is the key prop?**

React key prop is a special prop that can be added to any component in order to explicitly tell React its identity. The common scenario where a component requires a key prop is within a list: React needs it in order to understand which items have been added, removed or, more generically, modified. This allows for advance DOM manipulations, preventing nodes from being destroyed and recreated for no reason. The key prop also prevents list items from losing state between renders.

## **Key prop works even on non-list components**

What it might sounds as a surprise, however, is that the key prop can be used on any component regardless and if used correctly can solve unwanted state persistency in a really clean way.

For instance, let’s assume that you are building a chat app with a list of rooms on the left and the active room on the right together with a textarea where the user can write a message:

For instance, let’s assume that we are building a chat app with a list of rooms on the left and the active room on the right together with a textarea where the user can write a message:

**class** App **extends** React**.**Component **{**

state **=** **{** activeChat**:** **null};**

render**()** **{**

**return** **(**

**<**div**>**

**<**ChatList onClick**={**chat **=>** **this.**setState**({** activeChat**:** chat **})}** **/>**

**<**Chat activeChat**={this.**state**.**activeChat**}** **/>**

**</**div**>**

**);**

**}**

**}**

**class** ChatList **extends** React**.**Component **{**

// ... implementation

**}**

**class** Chat **extends** React**.**Component **{**

state **=** **{** message**:** '' **};**

render**()** **{**

**if(!this.**props**.**activeChat**)**

**return** **null;**

**return** **(**

**<**div**>**

**{this.**props**.**activeChat**.**messages**.**map**(**m **=>** **(**

/\* render messages \*/

**)}**

**<textarea** value**={this.**state**.**message**}** **/>**

**</**div**>**

**);**

**}**

**}**

Whenever the user clicks on a ChatList item, the App activeChat state attribute changes. Based on the current implementation however, when that happens the message is persisted instead of being reset.

One solution is to define componentWillReceiveProps, check if activeChat prop has changed and if so, reset the message field manually. This solution works fine, however requires you to be the owner of the component. If Chat is a component written by someone else with its own state, we might have hard times trying to reset it.

**Discarding current component instance**

Enters the key prop: as we said earlier, it helps React identifying a component but it also can be used to tell React that the component identity has changed, forcing a full re-instantiation of that component. Tweaking the example above:

**<**Chat

key**={this.**state**.**activeChat **&&** **this.**state**.**activeChat**.**id**}**

activeChat**={this.**state**.**activeChat**}**

**/>**

Now the state of the Chat component is bound to activeChat.id. On activeChat change, React will discard the current Chat instance and create a new one with a fresh state.

**Be aware of the performance implications**

While this is a nice trick that might reduce the complexity of your app, it is important to remember that this approach makes React discard the entire component instance and DOM tree. In the above example, most of the Chat component will be redrawn anyway on activeChat change, so it might be a good enough solution. On real word applications, you should limit non-list components with keys and avoid setting it on top level ones, as it might be the cause of (hard to spot) performance issues.

## **The 'this' Keyword & Binding in JavaScript**

In this lesson, we study how the notorious 'this' keyword works in JavaScript in the context of explicit, implicit, new, and global binding.

### **What is this?**

the value of this depends on what context it is used in. So, if it is used in a function, it’s value will depend on how that function is invoked, i.e., the call site, In javascript, Anytime we want to figure out what's the value of this is going to be inside function, we don't look at the function instaed look at where the function is called i.e. look at what is to the left of the dot when the function gets called.. Let’s go through the ways that this can be assigned in JavaScript.

### Implicit Binding

When the dot notation is used to call a function that is in an object or an object of a class, we say that this was bound implicitly. For example, consider the code sample below:

class Developer {

  constructor(firstname, lastname) {

    this.firstname = firstname;

    this.lastname = lastname;

  }

  getName() {

    return `${this.firstname} ${this.lastname}`;

  }

}

var me = new Developer('Robin', 'Wieruch');

console.log(me.getName()); // 'this' is me

var hobbit = new Developer('Frodo', 'Baggins');

console.log(hobbit.getName()); // 'this' is 'hobbit'

### Left of the Dot Rule

**Whatever is to the left of the dot is what this is.** For example, we call getName() first through the me object and hence firstname and lastname of the me object, i.e., ‘Robin’ Wieruch’ get printed. Then we call getName() through the hobbit object and so this refers to that hobbit object and prints its attributes.

### **Global Context**

When this is used outside of any context such as a class, function, or object, it refers to the global object. The global object in the browser is usually the window object. Download the file below, which simply prints the global this object, and open it with a browser of our choice. Then examine our browser’s console (inspect element > console). It will say something like “window{document:…”. That is the global window object that this refers to. In the case of a terminal and in our case the global object is undefined.

<html>

    <head>

        <script>console.log(this)</script>

    </head>

</html>  
  
// this is undefined in terminals/command prompts

console.log(this);

### **Understanding Call, Bind and Apply Methods in JavaScript**

const person = {  
 firstName: 'John',  
 lastName: 'Doe',  
 printName: function() {  
 console.log(this.firstName + ' ' + this.lastName);  
 }  
};

person.printName(); // John Doe

const printFullName = person.printName;

printFullName(); // undefined undefined

# Why does this happen?

# Here, we are storing a reference of person.printName to printFullName variable. After that, we are calling it without an object reference, so “this” will now refer to the window (global) object or undefined (in strict mode).

# If the script is in strict mode, this refers to undefined, so console.log() will return an error.

const counter = {  
 count: 0,  
 incrementCounter: function() {  
 console.log(this);  
 this.count++;  
 }  
}

document.querySelector('.btn').addEventListener('click', counter.incrementCounter);

What would be the value of “this” inside incrementCounter() method?

In the above snippet, “this” keyword refers to the DOM element where the event happened, not the counter object.

So, we can see that “this” keyword inside a function refers to different objects depending on how the function is called and sometimes we accidentally lose reference to “this” variable. So how can we prevent that from happening?

# Call(), Bind(), and Apply() to Rescue: We use call, bind and apply methods to set the this keyword independent of how the function is called. This is especially useful for the callbacks.

We know that functions are a special kind of objects in JavaScript. So, they have access to some methods and properties. To prove functions are objects, we can do something like this, for example:

function greeting() {  
 console.log('Hello World');  
}

greeting.lang = 'English'; // Prints 'English'  
console.log(greeting.lang);

JavaScript also provides some special methods and properties to every function object. So, every function in JavaScript inherits those methods. Call, bind, and apply are some of the methods that every function inherits.

# **Bind():** The bind method creates a new function and sets the “*this”* keyword to the specified object.

**Syntax:** **function.bind(thisArg, optionalArguments)**

For ex:Let’s suppose we have two person objects.

const john = {  
 name: 'John',  
 age: 24,  
};

const jane = {  
 name: 'Jane',  
 age: 22,  
};

Let’s add a greeting function:

function greeting() {  
 console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);  
}

We can use the bind method on the greeting function to bind the *“this”* keyword to john and jane objects.

For example:

const greetingJohn = greeting.bind(john);

// Hi, I am John and I am 24 years old

greetingJohn();

const greetingJane = greeting.bind(jane);

// Hi, I am Jane and I am 22 years old

greetingJane();

Here greeting.bind(john) **creates a new functio**n with this set to john object, which we then assign to greetingJohn variable. Similarly for greetingJane.

We can also use bind in case of callbacks and event handlers. For example:

const counter = {  
 count: 0,  
 incrementCounter: function() {  
 console.log(this);  
 this.count++;  
 }  
}

document.querySelector('.btn').addEventListener('click', **counter.incrementCounter.bind(counter));**

In the above example, the *“this”* keyword inside the incrementCounter method will now correctly refer to the counter object instead of the event object.

## Bind() can also accept arguments

We can also pass extra arguments to the bind method. The general syntax for this is function.bind(this, arg1, arg2, ...).

For example:

function greeting(lang) {  
 console.log(`${lang}: I am ${this.name}`);  
}

const john = {  
 name: 'John'  
};

const jane = {  
 name: 'Jane'  
};

const greetingJohn = greeting.bind(john, 'en');  
greetingJohn();

const greetingJane = greeting.bind(jane, 'es');  
greetingJane();

In the above example, **the bind method creates a new function with certain parameters predefined** (lang in this case) and this keyword set to the john and jane objects.

# Call(): The call method sets the *“this”* inside the function and immediately executes that function.

**Note:** The difference between call() and bind() is that the call() sets the this keyword and executes the function immediately and it does not create a new copy of the function, while the bind() creates a copy of that function and sets the this keyword.

## Syntax: **function.call(thisArg, arg1, agr2, ...)**

For example:

function greeting() {  
 console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);  
}

const john = {  
 name: 'John',  
 age: 24,  
};

const jane = {  
 name: 'Jane',  
 age: 22,  
}; // Hi, I am John and I am 24 years old

greeting.call(john);// Hi, I am Jane and I am 22 years old  
greeting.call(jane);

Above example is similar to the bind() example except that call() does not create a new function. We are directly setting the this keyword using call().

## Call () can also accept arguments

Call() also accepts a comma-separated list of arguments. The general syntax for this is function.call(this, arg1, arg2, ...)

For example:

function greet(greeting) {  
 console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);  
}

const john = {  
 name: 'John',  
 age: 24,  
};

const jane = {  
 name: 'Jane',  
 age: 22,  
};// Hi, I am John and I am 24 years old  
greet.call(john, 'Hi');// Hi, I am Jane and I am 22 years old  
greet.call(jane, 'Hello');

# **Apply():** The apply() method is similar to call(). The difference is that the apply() method accepts an array of arguments instead of comma separated values.

## Syntax: **function.apply(thisArg, [argumentsArr])**

For example:

function greet(greeting, lang) {  
 console.log(lang);  
 console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);  
}

const john = {  
 name: 'John',  
 age: 24,  
};

const jane = {  
 name: 'Jane',  
 age: 22,  
};// Hi, I am John and I am 24 years old

greet.apply(john, ['Hi', 'en']);// Hi, I am Jane and I am 22 years old  
greet.apply(jane, ['Hola', 'es']);

# Conclusion: We have learned that how this keyword behaves differently in JavaScript than in other object-oriented languages. The call, bind and apply methods can be used to set the “this” keyword independent of how a function is called. The bind method creates a copy of the function and sets the “this” keyword, while the call and apply methods sets the “this” keyword and calls the function immediately.

function greet(greeting, lang) {

console.log(lang);

console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

name: 'John',

age: 24,

};

//The bind method creates a copy of the function and sets the “this” keyword so we have to add additional set of brackets to invoke function created by bind.

greet.bind(john)();

//Call will invoke the function directly.

greet.call(john,"en");

### Caveat: arrow functions and ‘this’

Let’s​ see how the “this” keyword works with them. Consider the code block below. What do you think should get printed?

class Developer {

  constructor(firstname, lastname) {

    this.firstname = firstname;

    this.lastname = lastname;

  }

}

var me = new Developer('Robin', 'Wieruch');

// declare getName() outside of the Developer class using arrow functions

var getName = () => console.log(this.firstname);

const printMyName = getName.bind(me);

printMyName();

Yep, this gives us an error. **This is because methods like apply(), and bind(), etc. don’t have any effect on this in an arrow function in Javascript. The value of this remains the same as it was when the function was called.** If we want to bind to a different value, we need to use a function expression.

# **Redux:**

**What is Redux:** Redux is a **state container** for JavaScript applications. It is also known as a pattern for managing application state.

Redux is used mostly for application state management. To summarize it, Redux maintains the state of an entire application in a single immutable state tree (object), which can’t be changed directly. When something changes, a new object is created (using actions and reducers).

#### **ACTIONS**

An action is like a message that we send (i.e. dispatch) from the application (user interactions, internal events such as API calls, and form submissions) to our central Redux store. The store gets information only from actions. Internal actions are simple JavaScript objects that have a type property (usually constant), describing the type of action and payload of information being sent to the store.

Redux enforces that a type field exists in the action.

**{**

type: LOGIN\_FORM\_SUBMIT,

payload: {username: ‘alex’, password: ‘123456’}

**}**

#### **Action creators**

Actions are created with action creators. They are just function that creates and return actions.

Action creators must return plain objects with a type property.

function authUser(form) {

return {

type: LOGIN\_FORM\_SUBMIT,

payload: form

}

}

#### **dispatch(action)**

Dispatches an **action**. This is the only way to trigger a state change. Once action is dispatched it will go to middleware if any and to reducer finally.

dispatch(authUser(form));

#### **Reducers:** *In Redux, reducers are a pure function that takes current state of the application and an action and then return a new state.*

It's called a reducer because it’s based on the array reduce method ([Array.prototype.reduce(reducer, ?initialValue)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce), where it accepts a callback (reducer) and lets us get a single value out of multiple values, sums of integers, or an accumulation of streams of values.

Overall goal of reducer is to take some existing data, some action and then modify and return that existing data based upon the contents of an action.

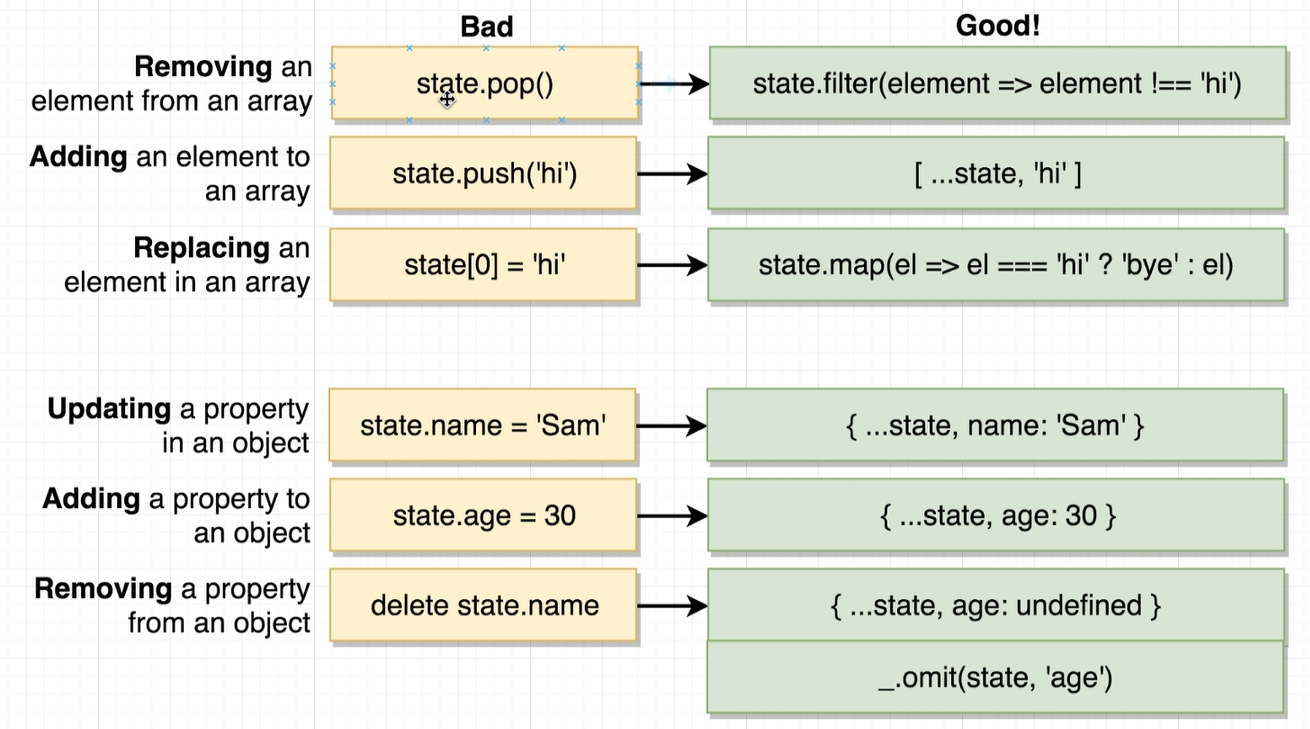
When we first start up the redux application each reducer is going to be automatically called exactly one time.

It's very important that the reducer stays pure.

**Reducer Rule**:

* Must return any value besides ‘undefined’.
* Reducers produce state only using current state (First argument of action reducer) and the action (second argument of action reducer)
* Never mutate its arguments, instead generate new value using its arguments; If we mutate the arguments react is not going to throw an exception but our application will not work as expected.

**Reason:** Redux performs reference equality check on prev. state and on new state after invoking reducer to check weather its needs to re-render the component or not. If we update value in the same object or add new item in array, react won’t detect any changes as new state data/object will point to same (previous state) object in memory so it will not re-render the component and we will never see any updated content on the screen.



* It should be pure functions; it should only use its arguments (current state i.e. First argument and the action i.e. second argument) nothing else. It should not use e.g. Date.now() or Math.random(), make API call, access DOM element, routing transitions, etc…

For more complex apps, using the combineReducers() utility provided by Redux is possible (indeed, recommended). It combines all of the reducers in the app into a single index reducer. Every reducer is responsible for its own part of the app’s state, and the state parameter is different for every reducer. The combineReducers() utility makes the file structure much easier to maintain.

If an object (state) changes only some values, Redux creates a new object, the values that didn’t change will refer to the old object and only new values will be created. That’s great for performance.

**const** LoginComponent **=** **(**state **=** initialState**,** action**)** **=>** **{**

**switch** **(**action**.**type**)** **{**

// This reducer handles any action with type "LOGIN"

**case** "LOGIN"**:**

**return** state**.**map**(**user **=>** **{**

**if** **(**user**.**username **!==** action**.**username**)** **{**

**return** user**;**

**}**

**if** **(**user**.password** **==** action**.password)** **{**

**return** **{**

**...**user**,**

login\_status**:** "LOGGED IN"

**}**

**}**

**});**

**}**

**}**

## **Store**

Store is the object that holds the application state and provides a few helper methods to access the state, dispatch actions and register listeners. The entire state is represented by a single store. Any action returns a new state via reducers. That makes Redux very simple and predictable.

**const store =** createStore**(LoginComponent);**

store is the created by calling createStore, a function from the Redux library. createStore takes a reducer as the first argument.

With Redux, there’s one general state in the store, and each component has access to the state. This eliminates the need to continuously pass state from one component to another.

We may also pass an initial state to createStore, useful for server side rendering and state preloading.

The store has the following responsibilities:

* Holds application state;
* Allows access to state via [getState()](https://redux.js.org/api/store#getState);
* Allows state to be updated via [dispatch(action)](https://redux.js.org/api/store#dispatchaction);
* Registers listeners via [subscribe(listener)](https://redux.js.org/api/store#subscribelistener);
* Handles unregistering of listeners via the function returned by [subscribe(listener)](https://redux.js.org/api/store#subscribelistener).

**Redux Example Code**

**const** CREATE\_POLICY **=** "CREATE\_POLICY"**;**

**const** DELETE\_POLICY **=** "DELETE\_POLICY"**;**

**const** CREATE\_CLAIM **=** "CREATE\_CLAIM"**;**

//Action Creator

**const** createPolicy **=** **(**name**,**amount**)** **=>** **{**

**return{**

type**:** CREATE\_POLICY**,**

payload**:** **{**

name**:** name**,**

amount**:** amount

**}**

**};**

**};**

**const** deletePolicy **=** **(**name**)** **=>{**

**return** **{**

type**:** DELETE\_POLICY**,**

payload**:{**

name**:** name

**}**

**};**

**};**

**const** createClaim **=** **(**name**,** claimAmount**)** **=>{**

**return** **{**

type**:** CREATE\_CLAIM**,**

payload**:{**

name**:** name**,**

claimAmount**:** claimAmount

**}**

**};**

**};**

//Reducers

**const** policyManager **=** **(**currentPolicies **=** **[],** action **=** **null)** **=>{**

**if(**action**.**type **===** CREATE\_POLICY**){**

**return** **[...**currentPolicies**,** action**.**payload**.**name**];**

**}**

**else** **if(**action**.**type **===** DELETE\_POLICY**){**

console**.**log**(**action**);**

**return** currentPolicies**.**filter**(**phName **=>** phName **!==** action**.**payload**.**name**);**

**}**

**return** currentPolicies**;**

**};**

**const** accounting **=** **(**currentAmountOfMoney **=** 100**,** action**=** **null)** **=>{**

**if(**action**.**type **===** CREATE\_CLAIM**){**

**return** currentAmountOfMoney **-** action**.**payload**.**claimAmount**;**

**}**

**else** **if(**action**.**type **===** CREATE\_POLICY**){**

**return** currentAmountOfMoney **+** action**.**payload**.**amount**;**

**}**

**return** currentAmountOfMoney**;**

**};**

**const** claimProcessor **=** **(**currentClaims **=** **[],** action **=** **null)** **=>** **{**

**if(**action**.**type **===** CREATE\_CLAIM**){**

//Create new Array using spread operator,

//Following statement will add new item to array and will return it.

**return** **[...**currentClaims**,** action**.**payload**];**

**}**

**return** currentClaims**;**

**};**

//Create Store

**const** **{**createStore**,** combineReducers**}** **=** Redux**;**

**const** insuranceCompany **=** combineReducers**({**

policyHolders**:**policyManager**,**

accounting**:**accounting**,**

claimHistories**:**claimProcessor

**});**

**const** store **=** createStore**(**insuranceCompany**);**

//Dispatch Action

store**.**dispatch**(**createPolicy**(**"Raj"**,**10**));**

store**.**dispatch**(**createPolicy**(**"Shyama"**,**20**));**

store**.**dispatch**(**createPolicy**(**"Aanchal"**,**30**));**

store**.**dispatch**(**deletePolicy**(**"Aanchal"**));**

store**.**dispatch**(**createClaim**(**"Shyama"**,**20**));**

console**.**log**(**store**.**getState**());**

//Output

**{**

"policyHolders"**:** **[**"Raj"**,** "Shyama"**],**

"accounting"**:** 140**,**

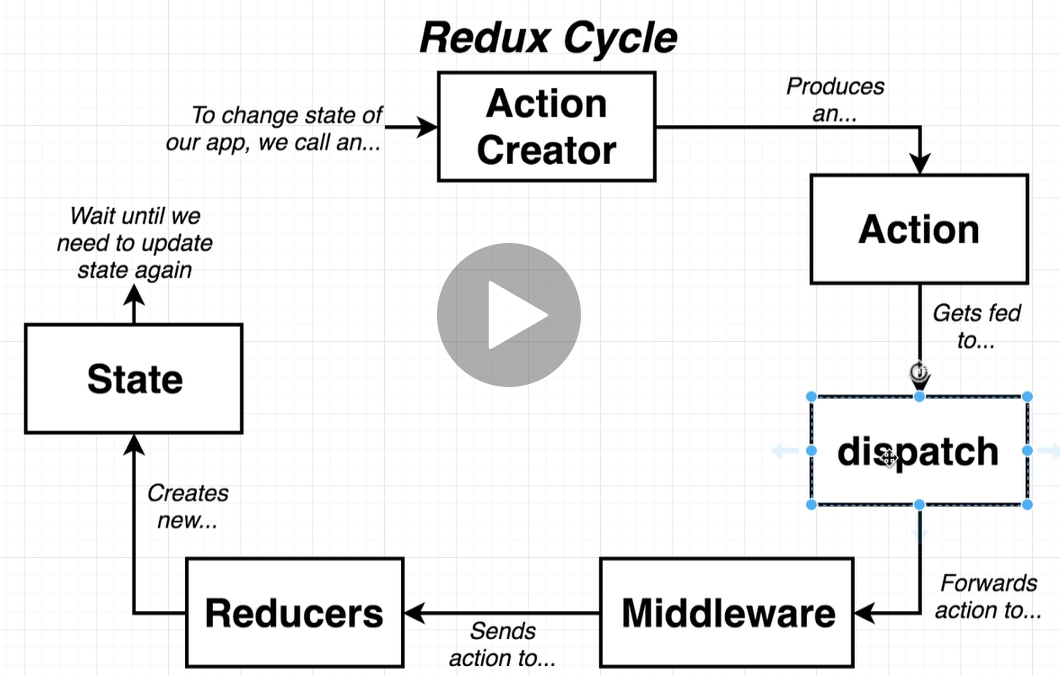
"claimHistories"**:** **[{**

"name"**:** "Shyama"**,**

"claimAmount"**:** 20

**}]**

**}**



**Three Principles:** Redux can be described in three fundamental principles:

### **Single source of truth:** The [state](https://redux.js.org/glossary#state) of our whole application is stored in an object tree within a single [store](https://redux.js.org/glossary#store).

This makes it easy to create universal apps, as the state from our server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables us to persist our app's state in development, for a faster development cycle. Some functionality which has been traditionally difficult to implement - Undo/Redo, for example - can suddenly become trivial to implement, if all of our state is stored in a single tree.

### **State is read-only:** The only way to change the state is to emit an [action](https://redux.js.org/glossary#action), an object describing what happened.

This ensures that neither the views nor the network callbacks will ever write directly to the state. Instead, they express an intent to transform the state. Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for. As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

### **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure [reducers](https://redux.js.org/glossary#reducer).

Reducers are just pure functions that take the previous state and an action, and return the next state. Remember to return new state objects, instead of mutating the previous state. We can start with a single reducer, and as our app grows, split it off into smaller reducers that manage specific parts of the state tree. Because reducers are just functions, we can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.

### **What is Pure Function?**

* Given the same input, will always return the same output.

Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

* Produces no side effects - stuff like accessing a global variable, making an async call or waiting for a promise to resolve have no place in here.
* **Pure functions** return a new value based on arguments passed to them.
* They don’t modify existing objects; instead, they return a new one.
* They will never call non-pure functions, e.g. Date.now() or Math.random().

# **React-Redux**

React Redux package lets our React components to read data from a Redux store, and dispatch actions to the store to update data.

It consists of four main building blocks:

1. A **single, centralized state** (i.e. a global JS object you could say) which is **not** directly accessible or mutable
2. **Reducer functions** that contain the logic to change and update the global state (by returning a new copy of the old state with all the required changes)
3. **Actions And Action Creators,** that can be dispatched to trigger a reducer function to run.
4. **Subscriptions** to get data out of the global state (e.g. to use it in React components), The connect method sets up a subscription behind the scenes.

**Provider:**React Redux provides <Provider />, which **makes the Redux store available to the rest of app**. To make the store available to our app, we have to wrap our app (Root component) with the <Provider /> API provided by React Redux.

Since any React component in a React Redux app can be connected, most applications will render a <Provider> at the top level, with the entire app’s component tree inside of it.

Normally, we can’t use a connected component unless it is nested inside of a <Provider>.

import reducers from './reducers';

<Provider store={createStore(reducers)}>

<App />

</Provider>

**connect():** it **connects a React component with the Redux store**. It also turn the state and action into component props.

It provides its connected component with the pieces of the data it needs from the store, and the functions it can use to dispatch actions to the store.

It does not modify the component class passed to it; instead, it returns a new, connected component class that wraps the component we passed in.

The connect function takes four different parameters, all optional. By convention, they are called:

* mapStateToProps?: Function
* mapDispatchToProps?: Function | Object
* mergeProps?: Function
* options?: Object:
* **mapStateToProps:** It is a function that we would use to provide the store data to our component.

If a mapStateToProps function is specified, the new wrapper component will subscribe to Redux store updates. This means that any time the store is updated, mapStateToProps will be called. The results of mapStateToProps must be a plain object, which will be merged into the wrapped component’s props. If we don't want to subscribe to store updates, pass null or undefined in place of mapStateToProps.

This function should be passed as the first argument to connect, and will be called every time when the Redux store state changes.

The object that we return from the mapStateToProps() function is going to show up as props inside of our component, in below example songs will be available as *“this.props.counter”.*

* **mapDispatchToProps:** It connects Redux actions to React props. This way a connected React component will be able to send messages to the store. this parameter can either be a function, or an object.
  + If it’s a function, it will be called once on component creation. It will receive dispatch as an argument, and should return an object full of functions that use dispatch to dispatch actions.

//util/constants.js

**export** **const** INCREMENT\_ACTION **=** "INCREMENT\_ACTION"**;**

//actions/index.js

**import** **\*** as constants **from** '../util/constants'**;**

**export** **const** incrementAction **=** **(**value**)** **=>** **{**

**return** **{**

type**:** constants**.**INCREMENT\_ACTION**,**

payload**:** value

**};**

**};**

//reducers/index.js

**import** **\*** as constants **from** '../util/constants'**;**

**export** **const** incrementReducer **=** **(**state **=** 0**,** action**)** **=>** **{**

**if** **(**action**.**type **===** constants**.**INCREMENT\_ACTION**)** **{**

**return** state **+** action**.**payload**;**

**}**

**return** state**;**

**};**

//components/App.js

**import** React **from** 'react'**;**

**import** **{** connect **}** **from** 'react-redux'**;**

**import** **{** incrementAction **}** **from** '../actions'**;**

**class** App **extends** React**.**Component **{**

render**()** **{**

**const** **{** counter**,** onIncrementClick **}** **=** **this.**props**;**

**return** **(**

**<**div**>**

**<button** onClick**={()** **=>** **{** onIncrementClick**(**1**);** **}}** **>**Increment 1**</button>**

**<button** onClick**={()** **=>** **{** onIncrementClick**(**10**);** **}}** **>**Increment 10**</button>**

**<**b**>**Total**:** **</**b**>** **{**counter**}**

**</**div**>**

**)**

**};**

**}**

**const** mapStateToProps **=** **(**state**)** **=>** **{**

**return** **{** counter**:** state **}**

**};**

**const** mapDispatchToProps **=** **{**

onIncrementClick**:** incrementAction

**};**

**export** **default** connect**(**mapStateToProps**,** mapDispatchToProps**)(**App**);**

//index.js

**import** React **from** 'react'**;**

**import** ReactDOM **from** 'react-dom'**;**

**import** **{** Provider **}** **from** 'react-redux'**;**

**import** **{** createStore **}** **from** 'redux'**;**

**import** **{** incrementReducer **}** **from** './reducers'**;**

**import** App **from** './components/App'**;**

**const** store **=** createStore**(**incrementReducer**);**

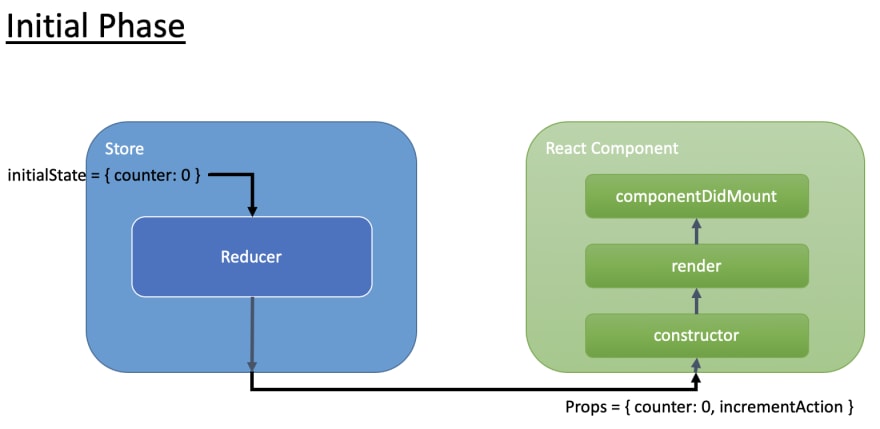
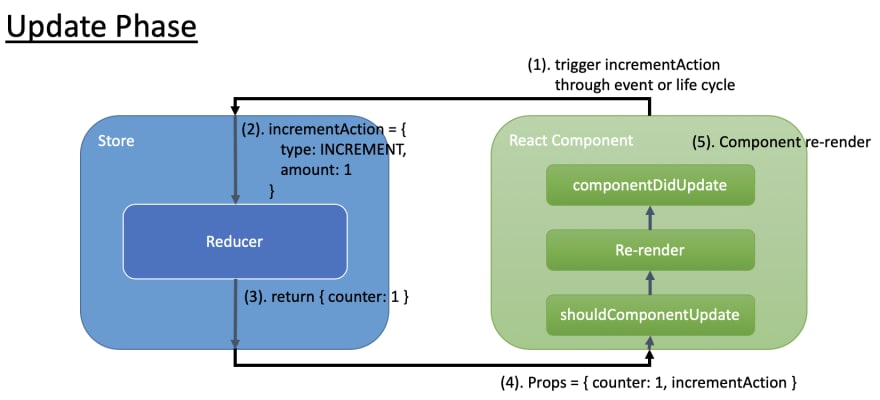
ReactDOM**.**render**(**

**<**Provider store**={**store**}>**

**<**App **/>**

**</**Provider**>,**

**document.**querySelector**(**"#root"**));**

After integrating Redux into React, the visualization should look like this.  
[](https://res.cloudinary.com/practicaldev/image/fetch/s--D2eR0iPA--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/i.imgur.com/LNpmQzn.png)  
[](https://res.cloudinary.com/practicaldev/image/fetch/s--r9x5HtZD--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/i.imgur.com/bAJNw3d.png)

Whenever user clicks on button, connect method will call action reducer and will pass current state and action to it. In turn action reducers will update the state based on action and it will cause component to re-render.

## **connect() Returns**

The return of connect() is a wrapper function that takes component and returns a wrapper component with the additional props it injects.

# **MobX**

MobX is a state management solution that helps in managing the local state within our app. MobX can have multiple stores to store the state of the application

## **Core concepts:**

1. **Observable state:** MobX adds observable capabilities to existing data structures like objects, arrays and class instances. This can simply be done by annotating our class properties with the @observable decorator (ES.Next).

**import** **{** observable **}** **from** "mobx";

**import** **{** createContext **}** **from** "react";

**class** TodoStore **{**

id **=** Math**.**random**();**

@observable title **=** "";

@observable finished **=** **false;**

**}**

**export default** createContext(new TodoStore());

Using observable is like turning a property of an object into a spreadsheet cell. But unlike spreadsheets, these values can be not only primitive values, but also references, objects and arrays.

If our environment doesn't support decorator syntax, don't worry. We can read [here](http://mobxjs.github.io/mobx/best/decorators.html) about how to set them up. Or we can skip them altogether, as MobX can be used fine without decorator *syntax*, by leveraging the *decorate* utility. Many MobX users do prefer the decorator syntax though, as it is slightly more concise.

**import** **{** decorate**,** observable **}** **from** "mobx"

**class** Todo **{**

id **=** Math**.**random**()**

title **=** ""

finished **=** false

**}**

decorate**(**Todo**,** **{**

title**:** observable**,**

finished**:** observable

**})**

1. **Actions:** Actions are all the things that alter the state. MobX will make sure that all changes to the application state caused by our actions are automatically processed by all derivations and reactions.

**class** TodoStore **{**

@observable title **=** "";

@action setTitle = (title:string) => {

this.title = title;

}

**}**

anything that modifies an observable or has side effects is decorated with the action.

It takes a function and returns a function with the same signature, but wrapped with [transaction](https://mobx.js.org/refguide/api.html#transaction), [untracked](https://mobx.js.org/refguide/api.html#untracked), and [allowStateChanges](https://mobx.js.org/refguide/api.html#untracked). Especially the fact that [transaction](https://mobx.js.org/refguide/api.html#transaction) is applied automatically yields great performance benefits; actions will batch mutations and only notify computed values and reactions after the (outer most) action has finished. This makes sure intermediate or incomplete values produced during an action are not visible to the rest of the application until the action has finished.

It is advised to use (@) action on any function that modifies observables or has side effects.

## **Computed values (derivations):** With MobX we can define values that will be derived automatically when relevant data is modified. By using the [@computed](http://mobxjs.github.io/mobx/refguide/computed-decorator.html) decorator or by using getter / setter functions when using (extend)Observable (Of course, we can use decorate here again as alternative to the @ syntax).

**class** TodoList **{**

@observable todos **=** **[]**

@computed get unfinishedTodoCount**()** **{**

**return** **this.**todos**.**filter**(**todo **=>** **!**todo**.**finished**).**length

**}**

**}**

### MobX will ensure that unfinishedTodoCount is updated automatically when a todo is added or when one of the finished properties is modified. Computations like these resemble formulas in spreadsheet programs like MS Excel. They update automatically and only when required.

1. ***Reactions:*** Reactions are similar to a computed value, but instead of producing a new value, a reaction produces a side effect for things like printing to the console, making network requests, incrementally updating the React component tree to patch the DOM, etc. In short, reactions bridge reactive and imperative programming.

**Usage:** reaction(() => data, data => { sideEffect }, options?)

It takes two functions, the first one (the *data* function) indicates which field/property is tracked and returns data that is used as input. The second one, the *effect* function. the side effect won't be run directly when created, but only after the data expression returns a new value for the first time.

**class** TodoStore **{**

@observable title **=** ""**;**

**constructor(){**

reaction**(**

**()** **=>** **this.**title**,**

**(**title**)** **=>** console**.**log**(**title**);**

**);**

**}**

**@action** setTitle **=** **(**title**:** string**)** **=>** **{**

**this.**title **=** title**;**

**}**

**}**

In this case, we are tracking title and in case of any changes to title we are just logging the value to console.

### **Custom reactions**

Custom reactions can simply be created using the [autorun](http://mobxjs.github.io/mobx/refguide/autorun.html), [reaction](http://mobxjs.github.io/mobx/refguide/reaction.html) or [when](http://mobxjs.github.io/mobx/refguide/when.html) functions to fit our specific situations.

For example the following autorun prints a log message each time the amount of unfinishedTodoCount changes:

autorun**(()** **=>** **{**

console**.**log**(**"Tasks left: " **+** todos**.**unfinishedTodoCount**)**

**})**

### **Observer:** The observer HoC / decorator subscribes React components automatically to any observables that are used during render. As a result, components will automatically re-render when relevant observables change. But it also makes sure that components don't re-render when there are no relevant changes.

useContext allows us access our stores from anywhere in our react to application without passing it down as props.

The real context is primarily used when some data needs to be accessible by many components at different nesting levels. In our case we want all of our React components to be able to use our stores that’s the reason we are using context here.

* observer is provided through the separate [mobx-react package](https://github.com/mobxjs/mobx-react).
* If your code base doesn't have any class based components, we can also the [mobx-react-lite package](https://github.com/mobxjs/mobx-react-lite), which is smaller

**import** React**,** **{** useContext **}** **from** "react"**;**

**import** TodoStore **from** "TodoStore"**;**

**import** **{**observer**}** **from** "mobx-react-lite"**;**

**const** Todo **=** **()** **=>{**

**const** store **=** useContext**(**TodoStore**);**

**const** **{**id**,** title**}** **=** store**;**

**return** **(**

**<**div**>**

**{**title**}:** **{** finished **}**

**</**div**>**

**)**

**}**

**export** **default** observer**(**Todo**);**

Note that observer *only* subscribes to observables used during the *own* render of the component. So, if observables are passed to child components, those have to be marked as observer as well. This also holds for any callback-based components.

**Guideline:**

# **Writing asynchronous actions**

The action wrapper / decorator only affects the currently running function, not functions that are scheduled (but not invoked) by the current function! This means that if we have a setTimeout, promise.then or async construction (Async is just syntactic sugar over promises.then), all those callback function won’t be executed inside “Action” context.

For ex:

@action deleteActivity **=** **async** **(**id**:** string**)** **=>** **{**

**this.**setIsDeleting**(true);**

**await** activityService**.delete(**id**);**

//Following line won’t be executed inside action context.

**this.**activityRegistry**.delete(**id**);**

**this.**setSelectActivity**(**""**);**

**this.**setIsDeleting**(false);**

**this.**setShowFormFlag**(false);**

**}**

As per guideline if we have callbacks, and in that callback if we are changing the state, those callbacks should be wrapped in action as well!

A simple fix is to extract the callbacks to actions. Although this is clean and explicit, it might get a bit verbose with complex async flows. Alternative, we can wrap the promise callbacks with the action keyword or runInAction.

However, problem with nested action is TypeScript does not apply type inference on them. Instead of this we can use “runInAction” and run only the state modifying part of the callback in an action. The advantage of this pattern is that it encourages us to not litter the place with action, but rather put all the state modifications as much as possible at the end of the whole process:

***runInAction(name?, thunk)***

runInAction is a simple utility that takes a code block and executes in an (anonymous) action. This is useful to create and execute actions on the fly, for example inside an asynchronous process. runInAction(f) is sugar for action(f)()

**Middleware:** Redux middleware function provides an extension point (a medium) to interact with dispatched action before they reach the reducer. Middleware is function that called with every action that we dispatch. Middleware has the ability to modify, stop the actions.

Multiple middlewares can be combined together to add new functionality, and each middleware requires no knowledge of what came before and after.

It is used for logging, crash reporting, making asynchronous API call, routing, and more.

[**Why do we need middleware for async flow in Redux?**](https://stackoverflow.com/questions/34570758/why-do-we-need-middleware-for-async-flow-in-redux)

Without middleware, Redux store **only supports synchronous data flow** (action that returns only plain objects). If try to invoke asynchronous function as shown below, we will get the error.

**Reason:**

1: It returns the promise not the plain object.

2: If we remove the async and await keywords, by the time action reaches to reducers, we won’t have fetched data from server.

To solve the problem, we need custom middleware that solves the problem by calling reducer only when we have retrieved the data from server.

**export** **const** fetchPosts **=** **async** **()** **=>** **{**

**const** response **=** **await** jsonAPI**.**get**(**"/posts"**);**

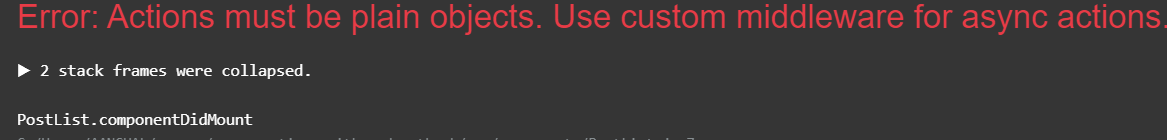
**return** **{**

type**:** constants**.**FETCH\_POSTS**,**

payload**:** response

**};**

**};**

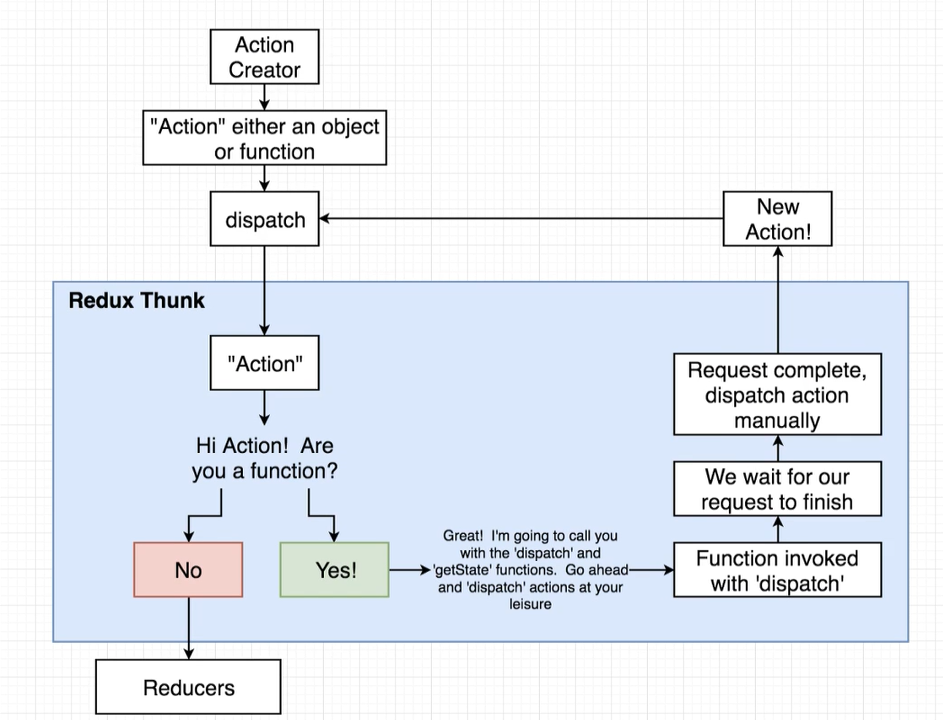


### **What is redux thunk and its purpose**

Redux Thunk [middleware](https://github.com/reactjs/redux/blob/master/docs/advanced/Middleware.md) allows us to write action creators that return a function instead of an action. redux thunk mainly used to support asynchronous actions.

The thunk can be used **to delay the dispatch of an action**, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState as parameters.

When an action creator returns a function, that function will get executed by the Redux Thunk middleware. This function doesn't need to be pure; it is thus allowed to have side effects, including executing asynchronous API calls. The function can also dispatch actions—like those synchronous actions we defined earlier.



//Action Creator

**export** **const** fetchPosts **=** **()** **=>** **{**

**return** **async** **function** **(**dispatch**,** getState**)** **{**

**const** response **=** **await** jsonAPI**.**get**(**"/posts"**);**

**return** dispatch**({**

type**:** constants**.**FETCH\_POSTS**,**

payload**:** response

**});**

**}**

**};**

//Posts.js

**import** **\*** as constants **from** '../util/constants'**;**

**export** **const** posts **=** **(**state **=** **[],** action**)** **=>** **{**

**switch** **(**action**.**type**)** **{**

**case** constants**.**FETCH\_POSTS**:**

**return** action**.**payload**;**

**default:**

**return** state**;**

**}**

**};**

//Main React App

**import** React **from** 'react'**;**

**import** ReactDOM **from** 'react-dom'**;**

**import** **{** createStore**,** applyMiddleware **}** **from** 'redux'**;**

**import** **{** Provider **}** **from** 'react-redux'**;**

**import** thunk **from** 'redux-thunk'**;**

**import** App **from** './components/App'**;**

**import** reducers **from** './reducers'**;**

**const** store **=** createStore**(**reducers**,** applyMiddleware**(**thunk**));**

ReactDOM**.**render**(**

q **<**Provider store**={**store**}>**

**<**App **/>**

**</**Provider**>,**

**document.**querySelector**(**"#root"**)**

**);**

## **React Router:**

### **What is a router?**

Single-page applications (SPAs) **rewrite sections of a page** rather than loading entire new pages from a server. they just look like multiple pages because they contain components which render like separate pages.

These applications are easy to deploy and greatly improve the user experience, among other advantages.

However, they also bring challenges.

One of them is browser history. As the application is contained in a single page, it cannot rely on the browser’s forward/back buttons per se. It needs something else.

Something that, according to the application’s state, changes the URL to push or replace URL history events within the browser. At the same time, it also needs to rebuild the application state from information contained within the URL.

Router to rescue: A router allows our application to navigate between different components, changing the browser URL, modifying the browser history, and keeping the UI state in sync.

At its core, what React Router does is conditionally render certain components to display depending on the *route* being used in the URL (/ for the home page, /about for the about page, etc.).

React Router includes three main packages:

* **react-router:** This is the core package for the router
* **react-router-dom:** It contains the [DOM](https://www.w3.org/TR/WD-DOM/introduction.html) bindings for React Router. In other words, the router components for websites. This is the only package we have to install.
* **react-router-native:** It contains the [React Native](https://facebook.github.io/react-native/) bindings for React Router. In other words, the router components for an app development environment using React Native.

For a web app, we have to just install react-router-dom NPM package.

React Router keeps our UI in sync with the URL. It has a simple API with powerful features like lazy code loading, dynamic route matching, and location transition handling built right in

Here we are importing three components,

* Link component will create HTML link to the pages.
* Route component will define the routes.
* Router component will handle the logic of routing. When user click the link, it checks whether this link exist in route definition. If it exists, then the router will change the URL in browser and route will render the correct component.

#### **Link components:**

#### HTML <a /> tag will create a server-side link. So, each time, a user clicks on the route, it won’t check the router or the routes. Instead it simply redirects the page in the browser to that route.

React-Link component to the rescue, Link components check the router and the router check the route and load the component without reloading the page in the browser. That’s why it is called as client-side routing. It doesn’t load the page from the server while clicking on the Link component.

#### **Anchor tags vs Link components in React:**

**When to use Anchor tag**: If linking to an external page that is not a part of our React application, use an anchor tag link. However, when we have to navigate to URL within App, we should not use Anchor tag, as It will refresh the page (Retrieves the page and all the resources from the server) and starts the apps again. Instead to when we want user to navigate with in our app use Link tag instead.

**When to use link tag**: Link accept to props which defines the URL it wants to link.

If linking to a different URL within our app, use the <Link> component. This will (depending on the routing library you’re using) still ultimately render a semantically valid anchor tag (<a />), but doesn’t refresh the page and gives us additional prop options that can be super handy.

<Link to=**"/"**>**Go to Home Page**</Link>

Above link component will be rendered as anchor tag on the page as following.

<a href=**"/"**>**Go to Home Page**</a>

How Link tag works: Whenever a user clicks on the anchor tag (anchor tag will be rendered on the page internally when we use link tag, as <a /> tag is valid HTML element), react router is going to automatically prevent browser from fetching the resource from the server. However, URL is still going to change, history object is going to be updated and also going to send it off to the BrowserRouter component. Then BrowserRouter component going to communicate URL to the different route components. Route components then going to re-render new set of components depending upon the path that the user is now visiting.

We can also assign “Link” and “To” component to other HTML as follow.

<Button as={Link} to=**"/ManageActivity/**:id” basic color=**'blue'** content=**'Edit'** />

This will be rendered as anchor tag.

<a class=**"ui blue basic button"** role=**"button"** href=**"/ManageActivity/75f15285-0bae-4e57-b62b-01202e87269c"**>**Edit**</a>

**Route Component:** Routes specify the path to different components in a ReactJS application.Each route will be identified in a <Route> component. The <Route> component will take two properties: path and component. When a path matches the path given to the <Route> component, it will return the component specified.

***Route parameters:* Route params** are parameters whose values are set dynamically in a page’s URL. This allows a route to render the same component while passing that component the dynamic portion of the URL, so that it can change its data based on the parameter.

For ex: This is a dynamic route with placeholders for `:id` that will be updated on-the-fly based on user input.



This is an example of a dynamically updating route. The first part of the path, the "/movie/” part, doesn’t change, but the :id is a route parameter that is set dynamically — usually from a user’s actions, like clicking a link or typing an input and clicking a search button.

**Note:**

* **Multiple Path:** Pathaccepts string or array of string as paths that path-to-regexp@^1.7.0 understands.

<Route path={["/users/:id", "/profile/:id"]}>

<User />

</Route>

* **Nested Routes:** To define nested routes, we have to nest Route objects within each other.

For ex: in following example, when user clicks on "/activities" link, it will render both NavBar and ActivityDashboard component, and when user goes to root path “/” it will it will only display home page component.

<React.Fragment>

<Route path="/" exact component={HomePage} />

<Route path=**{'/(.+)'}** render={() => (

<React.Fragment>

<NavBar />

<Route path="/activities/:id" component={ActivityDetails} />

<Route path="/activities" exact component={ActivityDashboard} />

</React.Fragment>

)} />

</React.Fragment>

* **Key:** If we need our component to be fully unmounted and remounted, we can use key (which a React special prop for identifying components from render to render).

**For ex:** Ifuser is currently on"/createActivity" page and if he navigates to “/manageActivity/5” page or vice versa, without key attributes, react won’t remount the component. To we need our component to be fully unmounted and remounted, we have to use key attribute.

<Route key={location.key}

path={["/createActivity", "/manageActivity/:id"]}

component={ActivityForm} />

using “key” will force reset a component state, when value of key changes in this case when url will change.

## **Accessing dynamic route params in a component**

### When we use Route inside our App, Router passes History, Location and Match object as props to the components. We can use these props to get parameter from match object of props;

Typically, the component being rendered on redirect will access the dynamic route info it needs from the URL.

Accessing dynamic path parameter in a URL:



Inside a React component that needs URL path data, this is the code you’ll have to use to access the route’s actual ID, which was shown as a placeholder in the route path as :id.

**Router:** The common low-level interface for all router components. Typically, apps will use one of the following high-level routers instead:

* [<BrowserRouter>](https://reacttraining.com/react-router/web/api/BrowserRouter)
* [<HashRouter>](https://reacttraining.com/react-router/web/api/HashRouter)
* [<MemoryRouter>](https://reacttraining.com/react-router/core/api/MemoryRouter)
* [<NativeRouter>](https://reacttraining.com/react-router/native/api/NativeRouter)
* [<StaticRouter>](https://reacttraining.com/react-router/core/api/StaticRouter)

The most common use-case for using the low-level <Router> is to synchronize a custom history with a state management lib like Redux or Mobx. Note that this is not required to use state management libs alongside React Router, it’s only for deep integration.

**BrowserRouter:** A Router that uses the HTML5 history API (pushtate, replaceState and the popstate event) to keep our UI in sync with the URL.

* The widely popular router and a router for modern browsers which user HTML5 pushState API. (i.e., pushState, replaceState and popState API).
* It routes as normal URL in browser, we can’t differentiate whether it is server rendered page or client rendered page through the URL.
* It assumes, our server handles all the request URL (eg., /, /about) and points to root index.html. From there, BrowserRouter take care of routing the relevant page.
* It accepts forceRefresh props to support legacy browsers which doesn’t support HTML5 pushState API
* **Note:** When wedeploy React App to a web server, and visit a route of app directly, e.g. via https://myapp.com/route/123 Apache tries to map that URL to a file in the public folder. In this case it looks for /route/123.html which obviously doesn’t exist – therefore the 404 error.
  + Easy solution of this issue is use HashRouter but to use clean URLs without the hash #.
  + Other solution is to redirect all requests to *index.html* where our app lives, we have to modify/create the *.htaccess* file on Apache or web.config file on IIS.
  + Deploying the React app to a sub directory: BrowserRouter has a prop called basename where we can specify our sub-directory path:

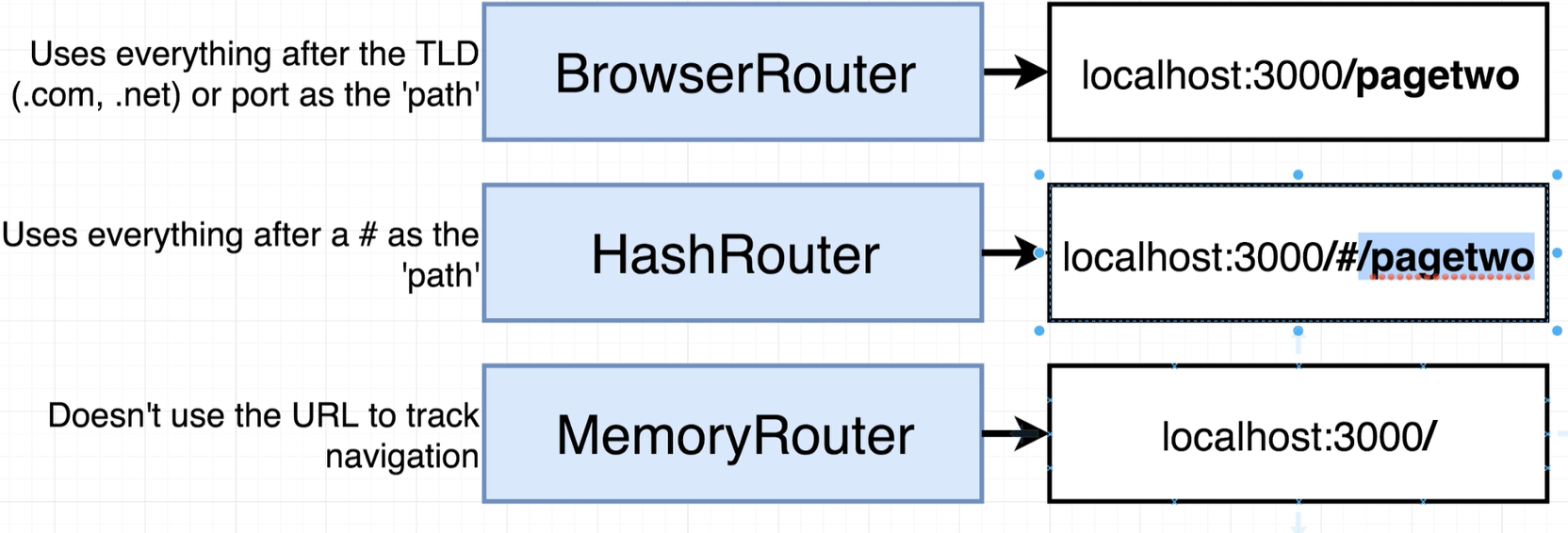
<BrowserRouter basename="/the-app">

**HashRouter:** A Router that uses the hash portion of the URL (i.e. window.location.hash) to keep our UI in sync with the URL.

* Whenever, there is a new route get rendered, it updated the browser URL with hash routes. (eg., /#/about)
* Hash portion of the URL won’t be handled by server, server will always send the index.html for every request and ignore hash value. Hash value will be handled by react router.
* It is used to support legacy browsers which usually doesn’t support HTML pushState API
* It doesn’t need any configuration in server to handle routes
* This route isn’t recommended by the team who created react router package. Use it only if you need to support legacy browsers or don’t have server logic to handle the client side routes

**MemoryRouter:** A router which doesn’t change the URL in our browser instead it keeps the URL changes in memory.

* It is very useful for testing and non browser environments
* But in browser, it doesn’t have history. So, we can’t go back or forward using browser history



#### **How React Router works:**

When react application loads inside the browser, browser router component creates an object of its own called The History object its history object is going to look at the URL and it’s going to extract the portion that react needs that is everything after the domain name or the port.

<BrowserRouter>

<div>

<Route path="/" exact component={PageOne} />

<Route path="/Page2" component={PageTwo} />

</div>

</BrowserRouter>

|  |  |
| --- | --- |
|  | Then browser router component is then going to communicate that path down to the route components. The router components are going to decide either to show themselves or hide themselves depending upon the path inside of the URL that the user is visiting and the path property that was passed when it was created. |

#### **How Paths Get Matched:**

Please note that, Inside react application, we can have multiple route components that match a given URL and all matching react components will be shown on the page.

**Path Matching Formula:** extractPath.contains(path)

If extracted path contains the path specified in route component it will math the path and shows the component on the page,

|  |  |  |  |
| --- | --- | --- | --- |
| **URL:** | **Extracted Path** | **Route Component Path** | **Will Match?** |
| <http://localhost:88/page> | /page | **Path=”/”** | **Yes** |
| **Path=”/page”** | **Yes** |
| Path=”/page/5” | No |

To override partial matching of route, we can use “exact” param.

**Exact Parameter:** The exact param disables the partial matching for a route and makes sure that it only returns the route if the path is an EXACT match to the current url.

**Path Matching Formula:** extractPath === path

**Switch Component:** There are two route matching components: Switch and Route.

When a <Switch> is rendered, it searches through its children <Route> elements to find one whose path matches the current URL. When it finds one, it renders that <Route> and ignores all others. This means that we should put <Route> with more specific (typically longer) paths before less-specific ones.

This is really handy when we have nested routes such as the below:

<Switch>

<Route path="/accounts/new" component={AddForm} />

<Route path={`/accounts/:accountId`} component={Profile} />

<Route component={NotFound} />

</Switch>

Say we put the above code in a <Router/> component — we would see that both {AddForm} and {Profile} would render, since “/accounts/new” could look like either Route to a Router component. Router components render inclusively of all route matches. The Switch component will render only first match. This makes it ideal for these nested scenarios.

Note, if none of the path match with the URL, in that {NotFound} component will be rendered.

## **Navigation:** There are two of navigation with in the any system.

* **Intentional Navigation:** When user clicks on “Link” component.
* **Programmatic Navigation:** System/App will navigate the user.
  + **<Redirect> Component:** We can redirect using <Redirect> component by simply passing the route we want to redirect to and rendering the component. It already comes loaded in the react-router-dom library.

**import** **{** Redirect **}** **from** "react-router-dom"**;**

The easiest way to use this method is by maintaining a redirect property inside the state of the component.

state **=** **{** redirect**:** **null** **};**

render**()** **{**

**if** **(this.**state**.**redirect**)** **{**

**return** **<**Redirect to**={this.**state**.**redirect**}** **/>**

**}**

**return(**

// Our Code goes here

**)**

**}**

Whenever we want to redirect to another path, we can simply change the state to re-render the component, thus rendering the <Redirect> component.

**this.**setState**({** redirect**:** "/someRoute" **});**

**Note:** This is the recommended way to navigate other than the <Link> method.

Discussed in detail towards the end of the post.

The downside of this method is that in cases like when we want to redirect directly from a redux action, we cannot do so.

* + **useHistory Hook:** As of release 5.1.2, react-router ships with some new hooks that can help us access the state of the router.

**import** **{** useHistory **}** **from** "react-router-dom"**;**

**function** App**()** **{**

**let** **history** **=** useHistory**();**

**}**

After this, we can use the .push() method to redirect to any route we want.

**history**.push('/someRoute')

* + **History prop:** Every component that is an immediate child of the <Route> component receives history object (along with match and location) as a prop. This is the same history (library) which keeps history of the session of React Router. We can thus use its properties to navigate to the required paths.

**this.**props**.history.**push**(**"/first"**);**

A common problem that we can encounter here is that in components which are not immediate child to the <Route> component, there is no history prop present. This can be easily solved using the withRouter function.

### **withRouter:** withRouter is a function provided in the react-router-dom library that helps us access the history prop in components which are not immediate children to the <Route> components. **import** **{** withRouter **}** **from** "react-router-dom"**;**

Now to get the history prop inside our component, we need wrap our component with withRouter while exporting it.

**export** **default** withRouter**(**our Component**);**

We can now access the history prop same as above to do our required navigations.

* **Custom History Object:**

The methods we learned above can cover most of the cases that we'll ever encounter while building a react app, However, every time we need to redirect from, say example a redux/mobx action, we always have to pass history to the action, unnecessarily increasing the number of arguments. This method can thus be used to get a neater code.

In this method, we make our custom history instance which we can import in other files to redirect.

// Inside /utils/createBrowserHistory.js

**import** **{** createBrowserHistory **}** **from** 'history'**;**

**export** **default** createBrowserHistory**();**

As <BrowserRouter> uses its own history and does not accept any outer(custom) history property, we have to use <Router> instead of <BrowserRouter> component.

**import** **{** Router **}** **from** "react-router-dom"**;**

**import** **customBrowserHistory** **from** "./utils/createBrowserHistory"**;**

**function** App**(){**

**return(**

**<**Router **history={customBrowserHistory}>**

// Routes go here

**<**Router**>**

**)**

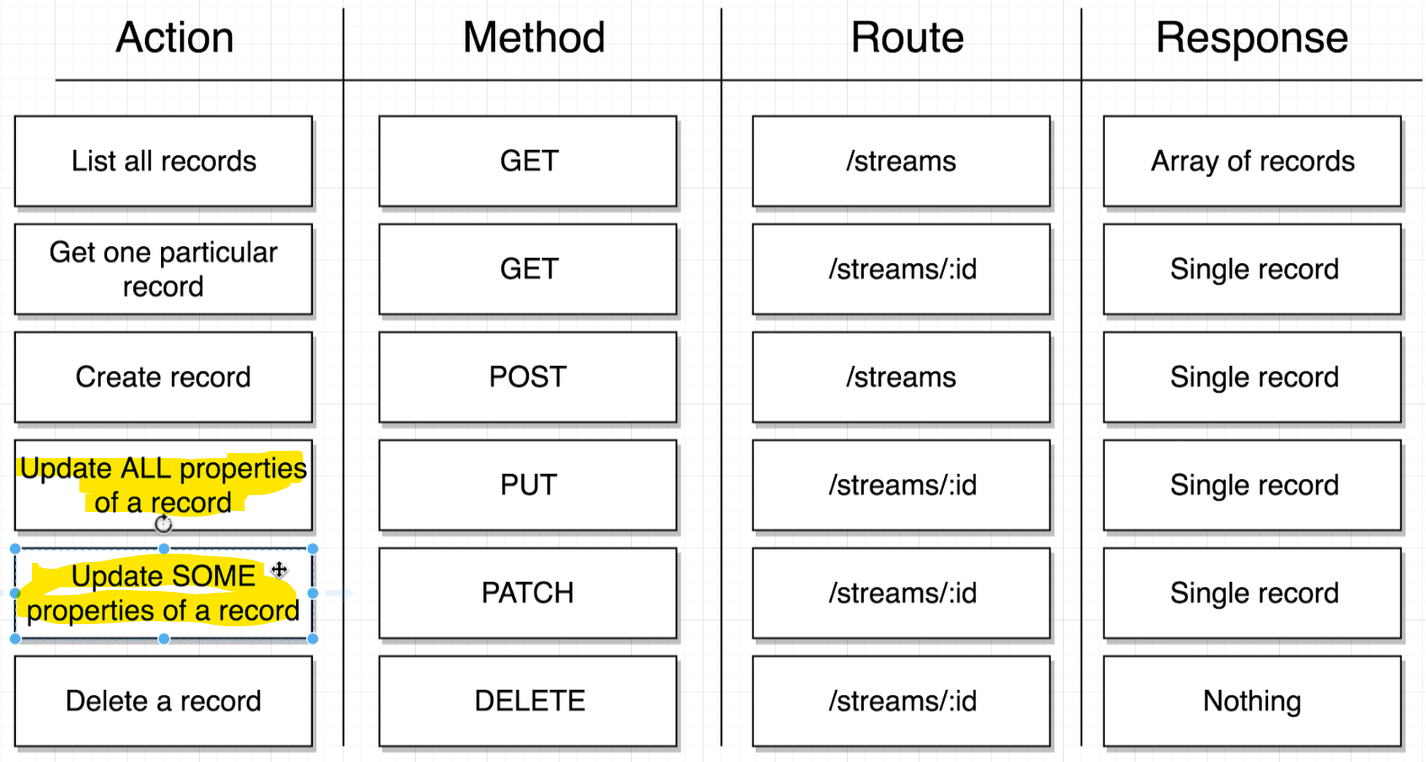
**}**

After this, we can import this history instance in whichever file we want to redirect from.

**import** **customBrowserHistory** **from** "./utils/createBrowserHistory"**;**

**customBrowserHistory.**push**(**"/somePath"**);**

**Note:** With React-Router, each component needs to be designed to work in isolation, in other words it should fetch its own data instead of depending upon other component. if we rely upon the user first going to page A component and then over to page B component eventually, It may break the Page B Component if a user is going to go straight to page B and the data that we might expect to have is not going to actually exist as user went directly to Page B Component instead of going to Page A first and then Page B.



## **Higher-Order Component (HOC):** is essentially a design pattern, also known as a Decorator Pattern.

In *ReactJS*, a **HOC** is a component that wraps another component by adding extra functionality or extra properties. In other words, a higher-order component (HOC) is basically a function that takes a React component as its argument and returns another React component, usually an enhancement of the original.

This allows abstraction from some commonly used logic and keeps our code **DRY**. It is how we distribute complex component structure between other components in *ReactJS* and a way to decouple our application logic and UI. For instance, we may use container component as a **HOC** for our presentational Button component.

# ***Container components:*** Container components, on the other hand, have logic to set state or have functions to emit events up to a parent component.

# The general rule of the thumb is to keep our component as simple as possible with a Single Responsibility Principle design principle in mind, which essentially means our component must do one thing, but do it well. Most often, these types of components are the HOCs that accommodate few presentational components. Presentational components Writing simple components may reduce our overall application complexity. Here is where presentational components come into play. These components should have minimal to no logic. Presentational components accept data and emit events to a callback that they receive as part of its props. Essentially, this type of component renders UI and executes a function that was passed into it when some action in its UI happens. This type of component is a building block and is sometimes referred to as a Lower-Order Component (or LOC).

## **Redux Form:** Redux forms provides a way to manage our form state in Redux. It is a Higher-Order-Component (HOC) that uses react-redux to make sure HTML forms in React use Redux to store all of its state.

Redux-form primarily consists of three things:

1. A Redux reducer that listens to dispatched redux-form actions to maintain form state in Redux. In react app, we have to import it
2. A React component decorator that wraps entire form in a Higher Order Component (HOC) and provides functionality via props.

**export** **default** reduxForm**({**

**form:** 'StreamCreateForm'

**})(**StreamCreate**);**

1. A Field component to connect individual field inputs to the Redux store.

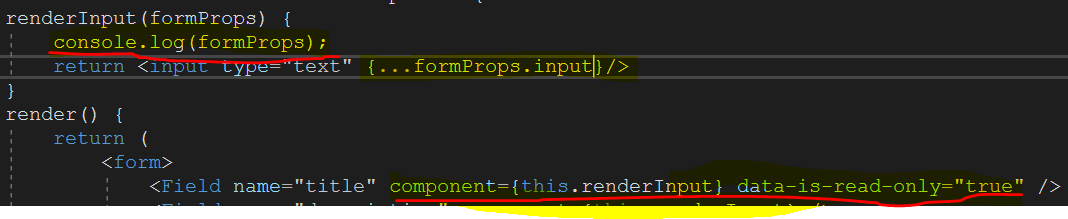
<Field name="inputName" component="input" type="text" />

**Field:** The <Field/> component connects each input to the redux store. The basic usage goes as follows:

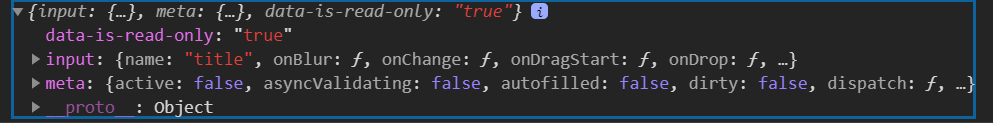
<Field name="inputName" component="input" type="text" />

It creates an HTML <input/> element of type text. It also passes additional props such as value, onChange, onBlur, etc. Those are used to track and maintain the input state under the hood.

## **Props:** These are props that Field will pass to our wrapped component. **The props provided by redux-form are divided into input and meta objects.**



**Console.Log:**



**Any custom props passed to Field will be merged into the props object on the same level as the input and meta objects.**

### **Input Props:** The props under the input key are what connects input component to Redux and are meant to be destructured into our <input/> component.

* **input.checked :** boolean [optional]

An alias for value only when value is a boolean. Provided for convenience of destructuring the whole field object into the props of a form element.

* **input.name:** String

The name prop passed in.

* **input.onBlur(eventOrValue)** : Function

A function to call when the form field loses focus. It expects to either receive the React SyntheticEvent or the current value of the field.

* **input.onChange(eventOrValue)** : Function

A function to call when the form field is changed. It expects to either receive the React SyntheticEvent or the new value of the field.

* **input.onDragStart(event) :** Function

A function to call when the form field receives a dragStart event. Saves the field value in the event for giving the field it is dropped into.

* **input.onDrop(event)**: Function

A function to call when the form field receives a drop event.

* **input.onFocus(event):** Function

A function to call when the form field receives focus.

* **input.value:** any

The value of this form field. It will be a boolean for checkboxes, and a string for all other input types. If there is no value in the Redux state for this field, it will default to ''. This is to ensure that the input is controlled. If you require that the value be of another type (e.g. Date or Number), you must provide initialValues to our form with the desired type of this field.

#### **Meta Props:** The props under the meta key are metadata about the state of this field that redux-form is tracking for you.

* **meta.active:** boolean

true if this field currently has focus. It will only work if we are passing onFocus to our input element.

* **meta.autofilled:** boolean

true if this field has been set with the AUTOFILL action and has not since been changed with a CHANGE action. This is useful to render the field in a way that the user can tell that the value was autofilled for them.

* **meta.asyncValidating:** boolean

true if the form is currently running asynchronous validation because this field was blurred.

* **meta.dirty:** boolean

true if the field value has changed from its initialized value. Opposite of pristine.

* **meta.dispatch:** Function

The Redux dispatch function.

* **meta.error:** String [optional]

The error for this field if its value is not passing validation. Both synchronous, asynchronous, and submit validation errors will be reported here.

* **meta.warning:** String [optional]

The warning for this field if its value is not passing warning validation.

* **meta.invalid:** boolean

true if the field value fails validation (has a validation error). Opposite of valid.

* **meta.pristine:** boolean

true if the field value is the same as its initialized value. Opposite of dirty.

* **meta.submitting:** boolean

true if the field is currently being submitted

* **meta.touched:** boolean

true if the field has been touched. By default this will be set when the field is blurred.

* **meta.valid:** boolean

true if the field value passes validation (has no validation errors). Opposite of invalid.

* **meta.visited:** boolean

true if this field has ever had focus. It will only work if you are passing onFocus to our input element.

#### **Redux Form:** Modern web apps are all about managing state. State boils down to data. Data comes from forms.

**Redux Form Props:** The props listed on this page are the props that redux-form generates to give to our decorated form component.

|  |  |  |
| --- | --- | --- |
|  |  |  |

* **handleSubmit(eventOrSubmit): Function**

A function meant to be passed to <form onSubmit={handleSubmit}> or to <button onClick={handleSubmit}>. It will run validation, both sync and async, and, if the form is valid, it will call this.props.onSubmit(data) with the contents of the form data.

<form className=**"ui form"** onSubmit={this.props.handleSubmit(this.onMyHandler)}>

#### **validate: (values:Object, props:Object) => errors:Object [optional]**

a synchronous validation function that takes the form values and props passed into our component. If validation passes, it should return {}. If validation fails, it should return the validation errors in the form { field1: <String>, field2: <String> }. Defaults to (values, props) => ({}).

|  |  |
| --- | --- |
| **Sample Form:** When user clicks on submit button =>  **import** React **from** 'react'**;**  **import** **{** Field**,** reduxForm **}** **from** 'redux-form'**;**  **import** **{** connect **}** **from** 'react-redux'**;**  **import** **{** createStream **}** **from** '../../actions'**;** |  |

**Form Initial Values:**If we want to set the initial default values to our form fields, redux-form allows us to that as well!

### **Pass initialValues prop in form instance component**

We pass a prop called initialValues into the form instance component and the value is an **object** with the **keys being the name of the field** and the **values being the initial value** which we want to add.

reduxForm**({**

**form:** 'StreamCreateForm'**,**

validate**:** validate,

initialValues: { title: 'This is default title' }

**});**

### **Add initialValues prop in mapStateToProps:**

We can use the initalValues prop in mapStateToProps.

* Convert form to a React Component
* Import connect from redux-react
* Add initialValuesprop in mapStateToProps and retrieve initialValues from the redux store

**const** mapStateToProps **=** **(**state**,** props**)** **=>** **({**

initialValues**:** state**.**initialName**,** // retrieve name from redux store

**});**

* To pass the initialValues props into the redux form, we use the connect function
* Add enableReinitialize : true When set to true, the form will reinitialize every time the initialValues prop changes

**const** wrappedForm **=** reduxForm**({**

**form:** 'StreamCreateForm'**,** // a unique identifier for this form

enableReinitialize**:** **true**

**})(**StreamCreate**);**

**export** **default** connect**(**mapStateToProps**)(**wrappedForm**);**

**class** StreamCreate **extends** React**.**Component **{**

renderError **=** **({** error**,** touched **})** **=>** **{**

**if** **(**error **&&** touched**)** **{**

**return** **(**

**<**div className**=**"ui error message"**>**

**<**div className**=**"header"**>**

**{**error**}**

**</**div**>**

**</**div**>**

**);**

**}**

**};**

renderInput **=** **({** input**,** caption**,** meta **})** **=>** **{**

**const** className **=** `field ${(meta.error && meta.touched ? 'error' : '')}`**;**

**return** **(**

**<**div className**={**className**}>**

**<**label**>{**caption**}</**label**>**

**<**input type**=**"text" **{...**input**}** autoComplete**=**"off" **/>**

**{this.**renderError**(**meta**)}**

**</**div**>**

**);**

**};**

onMyHandler **=** **(**formValues**)** **=>** **{**

**this.**props**.**createStream**(**formValues**);**

**};**

render**()** **{**

**return** **(**

**<form** className**=**"ui form error" onSubmit**={this.**props**.**handleSubmit**(this.**onMyHandler**)}>**

**<**Field name**=**"title" component**={this.**renderInput**}** caption**=**"Title" **/>**

**<button** className**=**"ui button primary"**>**Submit**</button>**

**</form>**

**);**

**}**

**};**

**const** validate **=** **(**formValues**)** **=>** **{**

**let** errors **=** **{};**

**if** **(!**formValues**.**title**)** **{**

errors**.**title **=** "Title is required field"**;**

**}**

**return** errors**;**

**};**

**const** wrappedForm **=** reduxForm**({**

**form:** 'StreamCreateForm'**,**

validate**:** validate,

initialValues: { title: 'This is default title' }

**})(**StreamCreate**);**

**export** **default** connect**(null,** **{** createStream **})(**wrappedForm**);**

**React-final-form:** React Final Form is a thin React wrapper for [Final Form](https://final-form.org/), which is a subscriptions-based form state management library that uses the [Observer pattern](https://en.wikipedia.org/wiki/Observer_pattern), so only the components that need updating are re-rendered as the form's state changes.

By default, React Final Form subscribes to all changes, but if we want to fine tune our form to optimized blazing-fast perfection, we may specify only the form state that we care about for rendering UI. We can think of it a little like GraphQL's feature of only fetching the data our component needs to render, and nothing else.

**<Form/>:** Form component, surrounds entire form and manages the form state. It can inject form state and functionality, e.g. we can pass a handleSubmit function to our <form> element, via render props.

# import { Form } from 'react-final-form';

The only two required props are [onSubmit](https://final-form.org/docs/react-final-form/types/FormProps#onsubmit) and one of [component](https://final-form.org/docs/react-final-form/types/FormProps#component), [render](https://final-form.org/docs/react-final-form/types/FormProps#render), or [children](https://final-form.org/docs/react-final-form/types/FormProps#children).

* **onSubmit:** onSubmit is a function that will be called with the values of your form when the user submits the form and all validation passes. Your onSubmit function will not be called if there are validation errors.

### **Render Form:** There are three ways to render a <Form/> component:

|  |  |
| --- | --- |
| **Prop** | **Type** |
| <Form component/> | React.ComponentType |
| <Form render/> | Function |
| <Form children/> | Function |

The only important distinction is that if we pass a component prop, it will be rendered with [React.createElement()](https://reactjs.org/docs/react-api.html#createelement), resulting in component actually being in the React node tree, i.e. inspectable in [DevTools](https://github.com/facebook/react-devtools#react-developer-tools-). However, it is recommended to use render prop.

The most important thing that <Form/> will pass to our render function is the handleSubmit function in props. handleSubmit will call event.preventDefault() to stop the default browser submission process.

<Form onSubmit={onMyFormSubmit}>

{props => (

<form onSubmit={props.handleSubmit}>

... fields go here...

<button type="submit">Submit</button>

</form>

)}

</Form>

**<Field/>:** A component that registers a field with the containing form, subscribes to field state, and injects both field state and callback functions, onBlur, onChange, and onFocus via a render prop.

The <Field/> will rerender any time the field state it is subscribed to changes. By default, it subscribes to *all* field state. we can control which field state it subscribes to with the subscription prop.

There are four ways to render a **<**Field**/>** component**:**

|  |  |
| --- | --- |
| **Prop** | **Type** |
| **<**Field component**/>** | **<**Field component**/>** 'input' or 'select' or 'textarea |
| **<**Field component**/>** | **<**Field component**/>** React**.**ComponentType |
| **<**Field render**/>** | **<**Field render**/>** Function |
| **<**Field children**/>** | **<**Field children**/>** Function |

The only important distinction is that if we pass a component to the component prop, it will be rendered with React.createElement(), resulting in your component actually being in the React node tree, i.e. inspectable in DevTools.

**FieldState:** FieldState is an object containing the following values. Depending on your subscription when calling form.registerField(), some of the values may not be present.

* Active
* Blur
* Change
* Data
* Dirty
* DirtySinceLastSubmit
* Error
* Focus
* initial
* length
* modified
* modifiedSinceLastSubmit
* name
* pristine
* submitError
* submitFailed
* submitSucceeded
* submitting
* touched
* valid
* validating
* value
* visited

# **FieldRenderProps:** Field provides prop object that separates out the values and event handlers intended to be given to the input component from the meta data about the field. The input can be destructured directly into an <input/> like so: <input {...props.input}/>.

<https://final-form.org/docs/react-final-form/api/Field>

FieldRenderProps, FormFieldProps

**Portal:** Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. Portal makes it possible for a component to always have the same DOM-parent independent of its actual parent in the react tree.

ReactDOM.createPortal(child, container)

# The first argument (child) is any renderable React child, such as an element, string, or fragment. The second argument (container) is a DOM element.

**Usage:**

Normally, when we return an element from a component’s render method, it’s mounted into the DOM as a child of the nearest parent node:

However, sometimes it’s useful to insert a child into a different location in the DOM to avoid restrictions imposed by the CSS styles of the parent element. For example, if the parent has the wrong z-index or prevents overflows. For example, dialogs, hovercards, and tooltips.

**Event Bubbling Through Portals**

Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way. Features like context work exactly the same regardless of whether the child is a portal, as the portal still exists in the React tree regardless of position in the DOM tree.

This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing React tree, even if those elements are not ancestors in the DOM tree. In other words, React events are still bubbling properly. Assuming the following HTML structure:

<html>

<body>

<div id=**"app-root"**></div>

<div id=**"modal-root"**></div>

</body>

</html>

A Parent component in #app-root would be able to catch an uncaught, bubbling event from the sibling node #modal-root.

Catching an event bubbling up from a portal in a parent component allows the development of more flexible abstractions that are not inherently reliant on portals. For example, if we render a <Modal /> component, the parent can capture its events regardless of whether it’s implemented using portals.

***React.Fragments:*** A common pattern in React is for a component to return multiple elements. Fragments let us group a list of children without adding extra nodes to the DOM.

This is common pattern for a component to return a list of children. In most cases the wrapper div is “irrelevant” and is only added because React components require us to return only one element. This kind of behavior results in useless markup and sometimes even invalid HTML to be rendered, which is bad.

For example, we could have a component Table that renders an HTML table and inside that table the columns are rendered with another component called Columns.

class Table extends React.Component {

render() {

return (

<table>

<tr>

<Columns />

</tr>

</table>

);

}

}

We can make use of fragments with <React.Fragments> syntax. So, we could write the Columns component as follows.

<React.Fragment>

<td>Hello</td>

<td>World</td>

</React.Fragment>

# **Context**

**What is Context?** Context provides a way to pass data deep into the component tree without having to pass props down manually at every level. 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.

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

**Note:** Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

If we only want to avoid passing some props through many levels, component composition (using props.children) is often a simpler solution than context.

**API**

### **React.createContext:**

const MyContext = React.createContext(defaultValue);

### Creates a Context object. We can later use created context to create providers and consumers.

### When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.

### The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them. Note: passing undefined as a Provider value does not cause consuming components to use defaultValue.

We can define the Context object in a separate file or right next to a component in a component file. We can also have multiple Context objects in one and the same app.

### **Context.Provider**

With the Context created, we can now provide it to all components that should be able to interact with it (i.e. read data from it or trigger a method stored in Context).

Context should be provided in a component that wraps all child components that eventually need access to the Context.

For data that should be available in our entire app, you have to provide Context in our root component (e.g. <App />) therefore. If you only need Context in a part of our app, we can provide it on a component a little further down the component tree.

<MyContext.Provider **value**={/\* some value \*/}>

<**UserCreate />**

</MyContext.Provider>

Please note the value prop on <ShopContext.Provider>: The value you set here is forwarded to the wrapped child components. And if the value changes, it will also change in the child components.

The fact that updates to the data passed to value are received by consumers of our Context and allow us to use the Context API as a global state management tool.

**All consumers that are descendants of a Provider will re-render whenever the Provider’s value prop changes.** The propagation from Provider to its descendant consumers (including .contextType and useContext) is not subject to the shouldComponentUpdate method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as Object.is.

To consume context, we have two options:

## Using static contextType

## Using Context.Consumer

* **Class.contextType:** The **contextType** property on a component class can be assigned a Context object created by React.createContext(). This lets us consume the nearest current value of that Context type using **this.context**. We can reference this in any of the lifecycle methods including the render function.

Class.Context only allows us to subscribe to a single context. If we need to get information from multiple components use consumer component.

class MyClass extends React.Component {

static contextType = MyContext;

render() {

let value = this.context;

/\* render something based on the value \*/

}

}

**Context.Consumer:** A consumer is where the stored information ends up. It can request data via the provider and manipulate the central store if the provider allows it.

A React component that subscribes to context changes. This lets us subscribe to a context within a [function component](https://reactjs.org/docs/components-and-props.html#function-and-class-components).

Requires a [function as a child](https://reactjs.org/docs/render-props.html#using-props-other-than-render). The function receives the current context value and returns a React node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the defaultValue that was passed to createContext().

renderValue = value => value === 'English' ? "Submit" : "प्रस्तुत";

<MyContext.Consumer>

{value => {value => this.renderValue(value)}/\* render based on the context value \*/}

</MyContext.Consumer>

we can use the normal React state management solution to update the data in that component => state and setState()

#### **What is a provider?**

The provider acts as a delivery service. When a consumer asks for something, it finds it in the context and delivers it to where it's needed.

#### **What is a consumer?**

A consumer is where the stored information ends up. It can request data via the provider and manipulate the central store if the provider allows it.

#### **Does the React Context API make Redux obsolete?**

The Context API makes one feature of Redux obsolete - the central store. If you don't use any of Redux's other features we can replace the whole library with this new native (to React) solution.

#### **Context caveats**

* Like all good things in code, there are some caveats to using Context:
* Don't use Context to avoid drilling props down just one or two layers. Context is great for managing state which is needed by large portions of an application. However, prop drilling is faster if you are just passing info down a couple of layers.
* Avoid using Context to save state that should be kept locally. So if you need to save a user's form inputs, for example, use local state and not Context.
* Always wrap the Provider around the lowest possible common parent in the tree - not the app's highest-level component. No need for overkill.
* Lastly, if you pass an object as our value prop, monitor performance and refactor as necessary. This probably won't be needed unless a drop in performance is noticeable.

**Will React’s Context API replace Redux?**

Indeed, there are reasons that hint towards React Context being better than Redux.

It’s built into React and we therefore need no extra third-party dependencies - a smaller bundle and improved project maintainability are the result. The API is also relatively straight-forward to use once we got the hang of it (especially when using hooks). We also don’t need a package like redux-thunk to handle asynchronous actions.

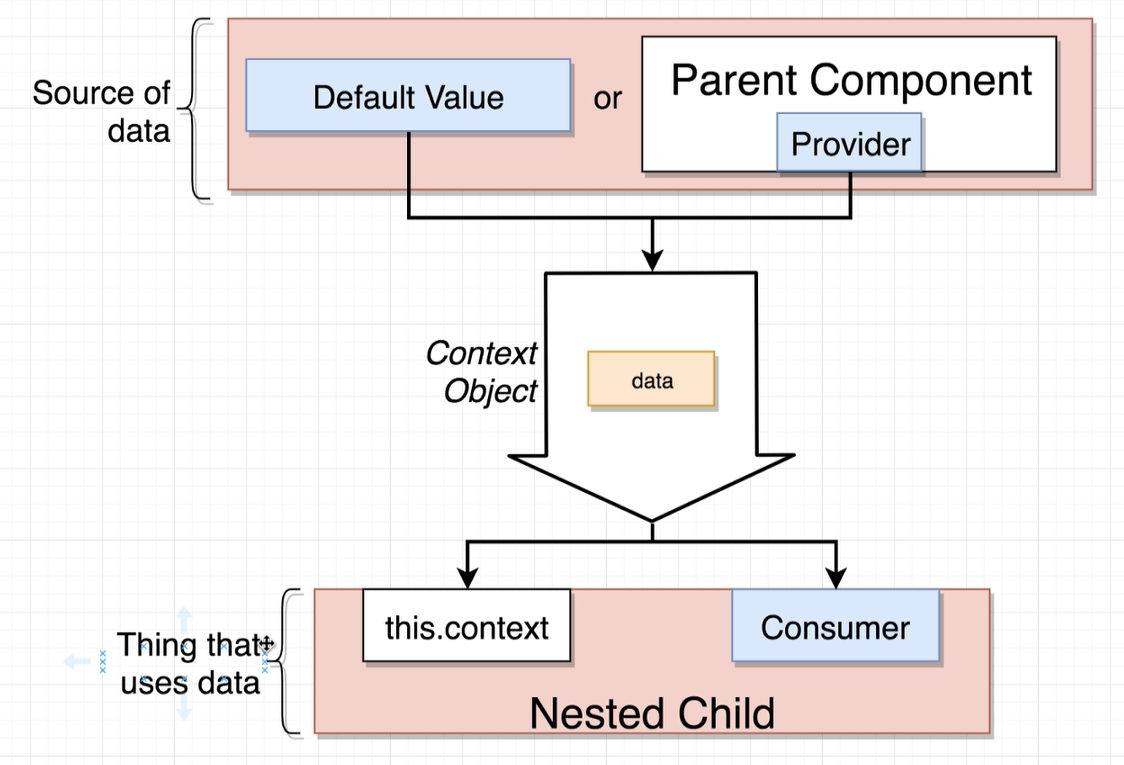
Redux doesn’t offer an immediate, obvious advantage other than the ability to add in middleware maybe.

But there is an important gotcha!

The Context API (currently) is not built for high-frequency updates, it’s not optimized for that. The react-redux people ran into this problem when they tried to switch to React Context internally in their package.

"new context is ready to be used for low frequency unlikely updates (like locale/theme). It’s also good to use it in the same way as old context was used. I.e. for static values and then propagate updates through subscriptions. It’s not ready to be used as a replacement for all Flux-like state propagation."

So, for the moment, it seems like you might want to look into using React Context for low-frequency updates (e.g. theme changes, user authentication) but not use it for the general state management of our application.

****

//App.JS

**import** UserCreate **from** './UserCreate'**;**

**import** LanguageSelector **from** './LanguageSelector'**;**

**import** **{** LanguageStore **}** **from** '../contexts/LanguageContext'**;**

**import** ColorContext **from** '../contexts/ColorContext'**;**

**class** App **extends** React**.**Component **{**

render**()** **{**

**return** **(**

**<**div className**=**"ui container"**>**

**<**LanguageStore**>**

**<**LanguageSelector **/>**

**<**ColorContext**.**Provider value**=**'red'**>**

**<**UserCreate **/>**

**</**ColorContext**.**Provider**>**

**</**LanguageStore**>**

**</**div**>**

**);**

**}**

**}**

**export** **default** App**;**

//LanguageSelector.js

**import** LanguageContext **from** '../contexts/LanguageContext'**;**

**class** LanguageSelector **extends** React**.**Component **{**

**static** contextType **=** LanguageContext**;**

getText **=** **()** **=>** **{**

**return** **this.**context**.**language **===** "English" **?** "Select a language: " **:** "भाषा चुनें: "**;**

**};**

onLanguageChange **=** **(**lang**)** **=>** **{**

**//** change lang. in language store.

**this.**context**.**onLanguageChange**(**lang**);**

**};**

render**()** **{**

**return** **(**

**<**React**.**Fragment**>**

**<**label**>** **{this.**getText**()}</**label**>**

**<**i className**=**"flag us" onClick**={()** **=>** **this.**onLanguageChange**(**'English'**)}** **/>**

**<**i className**=**"flag in" onClick**={()** **=>** **this.**onLanguageChange**(**'Hindi'**)}** **/>**

**</**React**.**Fragment **>**

**);**

**}**

**};**

**export** **default** LanguageSelector**;**

//UserCreate.js

**import** Field **from** './Field'**;**

**import** Button **from** './Button'**;**

**const** UserCreate **=** **()** **=>** **{**

**return** **(**

**<**div className**=**"ui form"**>**

**<**Field **/>**

**<**Button **/>**

**</**div**>**

**);**

**};**

**export** **default** UserCreate**;**

//Button.Js

**import** React **from** 'react'**;**

**import** LanguageContext **from** '../contexts/LanguageContext'**;**

**import** ColorContext **from** '../contexts/ColorContext'**;**

**class** Button **extends** React**.**Component **{**

renderValue **=** lang **=>** lang **===** 'English' **?** "Submit" **:** "प्रस्तुत"**;**

renderButton **=** **(**color**)** **=>** **{**

**return** **(**

**<button** className**={**`ui button ${color}`**}>**

**<**LanguageContext**.**Consumer**>**

**{**param **=>** **this.**renderValue**(**param**.**language**)}**

**</**LanguageContext**.**Consumer**>**

**</button>**

**);**

**}**

render**()** **{**

**return** **(**

**<**ColorContext**.**Consumer**>**

**{(**color**)** **=>** **this.**renderButton**(**color**)}**

**</**ColorContext**.**Consumer**>**

**);**

**};**

**}**

**export** **default** Button

//Field.js

**import** LanguageContext **from** '../contexts/LanguageContext'**;**

**class** Field **extends** React**.**Component **{**

**static** contextType **=** LanguageContext**;**

render**()** **{**

**const** **text** **=** **this.**context**.**language **===** 'English' **?** "Name" **:** 'नाम'**;**

**return** **(**

**<**div className**=**"ui field"**>**

**<**label**>{text}</**label**>**

**<**input **/>**

**</**div**>**

**);**

**}**

**}**

**export** **default** Field**;**

//ColorContext.js

**export** **default** React**.**createContext**(**'red'**);**

//LanguageContext.js

**const** Context **=** React**.**createContext**(**'English'**);**

**export** **class** LanguageStore **extends** React**.**Component **{**

state **=** **{** language**:** 'English' **};**

onLanguageChange **=** **(**lang**)** **=>** **{**

**this.**setState**({** language**:** lang **});**

**};**

render**()** **{**

**return** **(**

**<**Context**.**Provider

value**={{**

**...this.**state**,**

//Function that will allow consumer to change lang.

onLanguageChange**:** **this.**onLanguageChange

**}}>**

**{this.**props**.**children**}**

**</**Context**.**Provider**>**

**);**

**}**

**};**

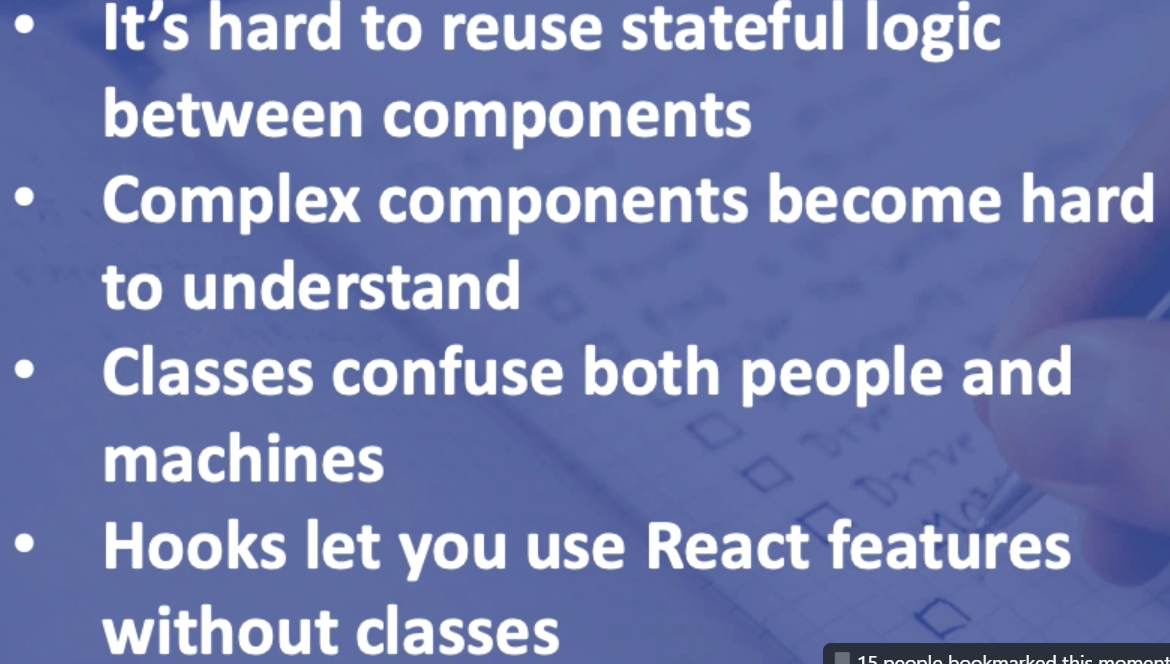
**export** **default** Context**;**

|  |  |
| --- | --- |
|  |  |

# **Hooks**

**Hooks** are special functions that let us use **React** state and lifecycle features without writing class.

**Hooks** don't work inside classes — they let us use **React** without classes. (React team don't recommend rewriting our existing components overnight but we can start using **Hooks** in the new ones if we'd like.)



## **1: State Hook:** useState is a Hook that **allows us to have state variables in functional components**.

## There are two types of components in React: class and functional components.

## Class components are ES6 classes that extend from React.Component and can have state and lifecycle methods:

Functional components are functions that just accept arguments as the properties of the component and return valid JSX. As we can see, there are no state or lifecycle methods.

However, since React 16.8 we can use Hooks, which are functions with names starting with use, to add state variables to functional components and instrument the lifecycle methods of classes.

useState allow us to “preserve” some values between the function calls. Normally, variables “disappear” when the function exits but state variables are preserved by React.

|  |  |
| --- | --- |
| 1**:** **import** React**,** **{** useState **}** **from** 'react'**;**  2**:**  3**:** **function** Example**()** **{**  4**:** **const** **[**count**,** setCount**]** **=** useState**(**0**);**  5**:**  6**:** **return** **(**  7**:** **<**div**>**  8**:** **<**p**>**You clicked **{**count**}** times**</**p**>**  9**:** **<button** onClick**={()** **=>** setCount**(**count **+** 1**)}>**  10**:** Click me  11**:** **</button>**  12**:** **</**div**>**  13**:** **);**  14**:** **}** |  |

* Line 1: We import the useState Hook from React. It lets us keep local state in a function component.
* Line 4: Inside the Example component, we declare a new state variable by calling the useState Hook. It returns a pair of values, to which we give names. We’re calling our variable count because it holds the number of button clicks. We initialize it to zero by passing 0 as the only useState argument. The second returned item is itself a function. It lets us update the count so we’ll name it setCount.
* Line 9: When the user clicks, we call setCount with a new value. React will then re-render the Example component, passing the new count value to it.

**Argument in useState:** The only argument to the useState() Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string. In our example below, we just want a number for how many times the user clicked, so pass 0 as initial state for our variable.

**What does useState return?** It returns a pair of values: the current state and a function that updates it. This is why we write const [count, setCount] = useState(). This is similar to this.state.count and this.setState in a class, except you get them in a pair.

We use [“array derestructuring”](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Array_destructuring) syntax to get return value assigned to variable. It means that we’re making two new variables count and setCount, where count is set to the first value returned by useState, and setCount is the second. It is equivalent to this code.

**let** countStateVariable **=** useState**(**0**);** // Returns a pair

**let** count **=** fruitStateVariable**[**0**];** // First item in a pair

**let** setCount **=** fruitStateVariable**[**1**];** // Second item in a pair

**why is useState not named createState instead?**

“Create” wouldn’t be quite accurate because the state is only created the first time our component renders. During the next renders, useState gives us the current state. Otherwise it wouldn’t be “state” at all! There’s also a reason why Hook names always start with use.

### **Tip: Using Multiple State Variables:** Declare different state variables if we want to use more than one:

// Declare multiple state variables!

**const** **[**age**,** setAge**]** **=** useState**(**42**);**

**const** **[**fruit**,** setFruit**]** **=** useState**(**'banana'**);**

We don’t have to use many state variables. State variables can hold objects and arrays just fine, so we can still group related data together. However, unlike this.setState in a class, updating a state variable always replaces it instead of merging it.

**Callback function:** React.useState doesn’t have accept callback function that gets called after React state has actually been modified, unlike setState callback function in class component.

To perform side effects after state has change, we must use the React.useEffect hook.

React.useEffect accepts a callback function to perform our side effects or actions, and it accepts a list of dependencies in the form of an array. That hook function will only activate if the values in the list change.

## **UseEffect:**

React.useEffect(callback, deps);

Lifecycle methods in class-based components are very important. Sometimes we want to fetch some data from an API when rendering a component(**componentDidMount)**, sometimes we want to do specific action when our component updates (**componentDidUpdate)**.

We can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

The Effect Hook lets us perform **side effects in function components**. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.

There are two common kinds of side effects in React components: those that don’t require cleanup and those that do.

## **Effects Without Cleanup**

#### **What does**useEffect**do?** By using this Hook, we tell React that our component needs to do something after render. React will remember the function we passed (it as our “effect”), and call it later after performing the DOM updates.

By default, it runs after the first render and after every update. (Instead of thinking in terms of “mounting” and “updating”, we might find it easier to think that effects happen “after render”. React guarantees the DOM has been updated by the time it runs the effects.

#### **How does**useEffect**works?**

**function** Example**()** **{**

**const** **[**count**,** setCount**]** **=** useState**(**0**);**

useEffect**(()** **=>** **{**

**document.**title **=** `You clicked ${count} times`**;**

**});**

**}**

We pass a function to the useEffect Hook. This function we pass is our effect. Inside our effect, we set the document title using the document.title browser API. We can read the latest count inside the effect because it’s in the scope of our function. When React renders our component, it will remember the effect we used, and then run our effect after updating the DOM. This happens for every render, including the first one.

Please note that the function passed to useEffect is going to be different on every render. This is intentional. In fact, this is what lets us read the count value from inside the effect without worrying about it getting stale. Every time we re-render, we schedule a different effect, replacing the previous one. In a way, this makes the effects behave more like a part of the render result — each effect “belongs” to a particular render.

**Tip:** Unlike componentDidMount or componentDidUpdate, effects scheduled with useEffect don’t block the browser from updating the screen. This makes our app feel more responsive. The majority of effects don’t need to happen synchronously. In the uncommon cases where they do (such as measuring the layout), there is a separate useLayoutEffect Hook with an API identical to useEffect.

**Effects with Cleanup:** Earlier, we looked at how to express side effects that don’t require any cleanup. However, some effects do. For example, we might want to set up a subscription to some external data source. In that case, it is important to clean up so that we don’t introduce a memory leak! Let’s compare how we can do it with classes and with Hooks.

**Example:** In a React class, we would typically set up a subscription in componentDidMount, and clean it up in componentWillUnmount.

For example, let’s say we have a ChatAPI module that lets us subscribe to a friend’s online status. Here’s how we might subscribe and display that status using a functional component:

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

**function** FriendStatus**(**props**)** **{**

**const** **[**isOnline**,** setIsOnline**]** **=** useState**(null);**

useEffect**(()** **=>** **{**

**function** handleStatusChange**(status)** **{**

setIsOnline**(status.**isOnline**);**

**}**

ChatAPI**.**subscribeToFriendStatus**(**props**.**friend**.**id**,** handleStatusChange**);**

**return** **()** **=>** **{**

ChatAPI**.**unsubscribeFromFriendStatus**(**props**.**friend**.**id**,** handleStatusChange**);**

**};**

**},** **[**props**.**friend**.**id**]);** // Only re-subscribe if props.friend.id changes

**if** **(**isOnline **===** **null)** **{**

**return** 'Loading...'**;**

**}**

**return** isOnline **?** 'Online' **:** 'Offline'**;**

**}**

There is no special code for handling updates because useEffect handles them *by default*. It cleans up the previous effects before applying the next effects. To illustrate this, here is a sequence of subscribe and unsubscribe calls that this component could produce over time:

// Mount with { friend: { id: 100 } } props

ChatAPI**.**subscribeToFriendStatus**(**100**,** handleStatusChange**);** // Run first effect

// Update with { friend: { id: 200 } } props

ChatAPI**.**unsubscribeFromFriendStatus**(**100**,** handleStatusChange**);** // Clean up previous effect

ChatAPI**.**subscribeToFriendStatus**(**200**,** handleStatusChange**);** // Run next effect

// Update with { friend: { id: 300 } } props

ChatAPI**.**unsubscribeFromFriendStatus**(**200**,** handleStatusChange**);** // Clean up previous effect

ChatAPI**.**subscribeToFriendStatus**(**300**,** handleStatusChange**);** // Run next effect

// Unmount

ChatAPI**.**unsubscribeFromFriendStatus**(**300**,** handleStatusChange**);** // Clean up last effect

This behavior ensures consistency by default and prevents bugs that are common in class components due to missing update logic.

## **Tips for Using Effects:**

### **Use Multiple Effects to Separate Concerns:** One of the problems we outlined in the [Motivation](https://reactjs.org/docs/hooks-intro.html#complex-components-become-hard-to-understand) for Hooks is that class lifecycle methods often contain unrelated logic, but related logic gets broken up into several methods.

componentDidMount**()** **{**

**document.**title **=** `You clicked ${this.state.count} times`**;**

ChatAPI**.**subscribeToFriendStatus**(**

**this.**props**.**friend**.**id**,**

**this.**handleStatusChange

**);**

**}**

componentDidUpdate**()** **{**

**document.**title **=** `You clicked ${this.state.count} times`**;**

**}**

componentWillUnmount**()** **{**

ChatAPI**.**unsubscribeFromFriendStatus**(**

**this.**props**.**friend**.**id**,**

**this.**handleStatusChange

**);**

**}**

Note how the logic that sets document.title is split between componentDidMount and componentDidUpdate. The subscription logic is also spread between componentDidMount and componentWillUnmount. And componentDidMount contains code for both tasks.

Just like [we can use the *State* Hook more than once](https://reactjs.org/docs/hooks-state.html#tip-using-multiple-state-variables), we can also use several effects. This lets us separate unrelated logic into different effects

**const** **[**count**,** setCount**]** **=** useState**(**0**);**

useEffect**(()** **=>** **{**

//This effect will run in case of count value changes.

**document.**title **=** `You clicked ${count} times`**;**

**},** **[**count**]);**

**const** **[**isOnline**,** setIsOnline**]** **=** useState**(null);**

useEffect**(()** **=>** **{**

**function** handleStatusChange**(status)** **{**

setIsOnline**(status.**isOnline**);**

**}**

ChatAPI**.**subscribeToFriendStatus**(**props**.**friend**.**id**,** handleStatusChange**);**

**return** **()** **=>** **{**

ChatAPI**.**unsubscribeFromFriendStatus**(**props**.**friend**.**id**,** handleStatusChange**);**

**};**

**},** **[**props**.**friend**.**id**]);** // Only re-subscribe if props.friend.id changes

**Hooks let us split the code based on what it is doing** rather than a lifecycle method name. React will apply *every* effect used by the component, in the order they were specified.

**Why did we return a function from our effect?** This is the optional cleanup mechanism for effects. Every effect may return a function that cleans up after it. This lets us keep the logic for adding and removing subscriptions close to each other. They’re part of the same effect!

**When exactly does React clean up an effect?** React performs the cleanup when the component unmounts. However, as we learned earlier, effects run for every render and not just once. This is why React *also* cleans up effects from the previous render before running the effects next time.

**Note:** We don’t have to return a named function from the effect. We called it cleanup here to clarify its purpose, but you could return an arrow function or call it something different.

***Recap:*** We’ve learned that useEffect lets us express different kinds of side effects after a component renders. Some effects might require cleanup so they return a function. Other effects might not have a cleanup phase, and don’t return anything.

#### **Tip: Optimizing Performance by Skipping Effects**

## In some cases, cleaning up or applying the effect after every render might create a performance problem. In class components, we can solve this by writing an extra comparison with prevProps or prevState inside componentDidUpdate:

componentDidUpdate**(**prevProps**,** prevState**)** **{**

**if** **(**prevState**.**count **!==** **this.**state**.**count**)** **{**

**document.**title **=** `You clicked ${this.state.count} times`**;**

**}**

**}**

This requirement is common enough that it is built into the *useEffect* Hook API. We can tell React to *skip* applying an effect if certain values haven’t changed between re-renders. To do so, pass an array as an optional second argument to *useEffect*:

useEffect**(()** **=>** **{**

**document.**title **=** `You clicked ${count} times`**;**

**},** **[**count**]);** // Only re-run the effect if count changes

In the example above, we pass [count] as the second argument. What does this mean? If the count is 5, and then our component re-renders with count still equal to 5, React will compare [5] from the previous render and [5] from the next render. Because all items in the array are the same (5 === 5), React would skip the effect. That’s our optimization.

When we render with count updated to 6, React will compare the items in the [5] array from the previous render to items in the [6] array from the next render. This time, React will re-apply the effect because 5 !== 6. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

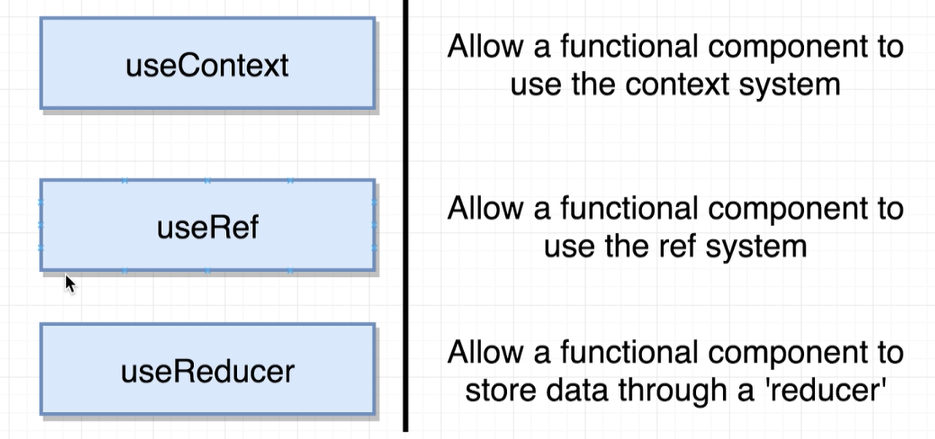
This also works for effects that have a cleanup phase:

useEffect**(()** **=>** **{**

**...**

**},** **[**props**.**friend**.**id**]);** // Only re-subscribe if props.friend.id changes

If we want to run an effect and clean it up only once (on mount and unmount), we can pass an empty array ([]) as a second argument. This tells React that our effect doesn’t depend on *any* values from props or state, so it never needs to re-run. This isn’t handled as a special case — it follows directly from how the dependencies array always works.

[https://reactjs.org/docs/hooks-reference.html#usecontext](https://reactjs.org/docs/hooks-reference.html#usecontext)

### ***useRef:*** Sometimes we want to store a value in our component for future reference, but we don’t want to trigger a re-render (as this value isn’t expected to change often or has little or no UI impact).

const refContainer = useRef(initialValue);

useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

A common use case is to access a child imperatively:

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

// `current` points to the mounted text input element

inputEl.current.focus();

};

return (

<>

<input ref={inputEl} type="text" />

<button onClick={onButtonClick}>Focus the input</button>

</>

);

}

Essentially, useRef is like a “box” that can hold a mutable value in its .current property.

You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with <div ref={myRef} />, React will set its .current property to the corresponding DOM node whenever that node changes.

However, useRef() is useful for more than the ref attribute. It’s handy for keeping any mutable value around similar to how you’d use instance fields in classes.

This works because useRef() creates a plain JavaScript object. The only difference between useRef() and creating a {current: ...} object our self is that useRef will give you the same ref object on every render.

Keep in mind that useRef doesn’t notify you when its content changes. Mutating the .current property doesn’t cause a re-render. If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a callback ref instead.

## **Rules of Hooks**

Hooks are JavaScript functions, but they impose two additional rules:

* Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions.
* Only call Hooks from React function components. Don’t call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — our own custom Hooks. We’ll learn about them in a moment.)

**Different Type of Hooks**

[**https://reactjs.org/docs/hooks-rules.html**](https://reactjs.org/docs/hooks-rules.html)

# **Adding Custom Environment Variables:** React project can consume variables declared in environment as if they were declared locally in JS files. By default, we will have NODE\_ENV defined for us, and any other environment variables starting with REACT\_APP\_

**Note:** We must create custom environment variables beginning with REACT\_APP\_. Any other variables except NODE\_ENV will be ignored to avoid accidentally exposing a private key on the machine that could have the same name. Changing any environment variables will require us to restart the development server if it is running.

These environment variables will be defined for us on “*process.env”*. For example, having an environment variable named REACT\_APP\_NOT\_SECRET\_CODE will be exposed in our JS as process.env.REACT\_APP\_NOT\_SECRET\_CODE.

During the build, process.env.REACT\_APP\_NOT\_SECRET\_CODE will be replaced with the current value of the REACT\_APP\_NOT\_SECRET\_CODE environment variable. Remember that the NODE\_ENV variable will be set for you automatically.

We can define environment variable either in your shell or in a .env file.

## Adding Temporary Environment Variables in Your Shell

Defining environment variables can vary between OSes. It’s also important to know that this manner is temporary for the life of the shell session.

set "REACT\_APP\_NOT\_SECRET\_CODE=abcdef" && npm start

## **Adding Development Environment Variables In .env**

To define permanent environment variables, create a file called .env in the root of project such as (.env.development, .env.production):

REACT\_APP\_NOT\_SECRET\_CODE=abcdef

**Post Build:** It allow us dosomething after the build and will be called by npm/Yarn automatically. For ex. Following command will move build to wwwroot directory.   
"postbuild": "move build ../SocialNetwork.API/wwwroot",

# **axios.create([config]):** Create a new instance of axios with a custom config

const instance = axios.create({

baseURL: 'https://some-domain.com/api/',

timeout: 1000,

withCredentials: true,

headers: {'X-Custom-Header': 'foobar'}

});

export default instance;

# **Function.prototype.bind():** The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

x**:** 42**,**

getX**:** **function()** **{**

**return** **this.**x**;**

**}**

**}**

**const** unboundGetX **=** module**.**getX**;**

console**.**log**(**unboundGetX**());** // The function gets invoked at the global scope

// expected output: undefined

**const** boundGetX **=** unboundGetX**.**bind**(**module**);**

console**.**log**(**boundGetX**());**

// expected output: 42

1. **Array.Map:** The map() method creates a new array with the results of calling a function for every array element. The map() method calls the provided function once for each element in an array, in order.

**Note:** this method does not change the original array.

**const** nums **=** **[**10**,** 20**,** 30**,** 40**];**

**const** newNums **=** nums**.**map**((**item**,**idx**)** **=>{**

**return** item **\***idx**;**

**});**

console**.**log**(**nums**);** //[10, 20, 30, 40]

console**.**log**(**newNums**);** //[0,20,60,120]

1. In Node when we can create a file like src/actions/index.js, and then when another file imports it they can leave off the index part and simply import using

**import** defaultType **from** '../actions';

This makes it ever so slightly easier to import the model vs. if we named the file "foo.js" and thus had to

**import** defaultType **from** '../actions/foo';

If we don't specify a file name in the import statement webpack is going to automatically to import index.js file.

1. By default, when we import component in React, react variable name follows pascal case.

**import** {Provider} **from** 'react-redux';

1. **npm init:** create a package.json file

**Useful NPM Packages:**

* **Lodash**
  + **mapKeys:**
  + **omit:**
* **JSON Server**
* **Axios**
* **Faker**
* **Redux Dev Tools for Chrome and Firefox**
* **MobX Dev Tools for Chrome and Firefox**
* react-toastify
* **react-final-form**
* **final-form**
* **react-widgets**
* **react-infinite-scroller**
* moment – Format Date
* react-widgets-moment
* revalidate – To validate Form Field
* **~~react-widgets-date-fns (npm install react-widgets-date-fns date-fns@2.0.0-alpha.7 --save)~~**
* [**https://api.unsplash.com**](https://api.unsplash.com)
* <https://stephengrider.github.io/playgrounds/>
* <https://developers.themoviedb.org/>

<https://securityheaders.com/>

Web Pack

Index.tsx file

React Native vs DOM

ClassName

src={}

start inside scripts method

react hooks why?

Props

### **Presentational and Container Components**

React bindings for Redux separate presentational components from container components. Here's a summary of the differences between presentational and container components.

|  | **Presentational Components** | **Container Components** |
| --- | --- | --- |
| **Purpose** | How things look (markup, styles) | How things work (data fetching, state updates) |
| **Aware of Redux** | No | Yes |
| **To read data** | Read data from props | Subscribe to Redux state |
| **To change data** | Invoke callbacks from props | Dispatch Redux actions |
| **Are written** | By hand | Usually generated by React Redux |

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux store.

Technically we could write the container components by hand using store.subscribe(). However it’s not advised, because React Redux makes many performance optimizations that are hard to do by hand. For this reason, rather than write container components, generate them using the connect() function provided by React Redux

<https://www.sitepoint.com/es6-enhanced-object-literals/>

Key Interpolation Syntax

<https://medium.com/javascript-in-plain-english/the-only-react-router-set-up-you-will-ever-need-9f36ddee03a5>

!! Operator:

In Javascript, every value has an associated boolean, true or false, value. Values that are associated with boolean true are said to be truthy. Values that are associated with boolean false values are said to be falsy.

The following values are considered by JavaScript to be **falsey**s:

* Empty string: ""
* 0
* null
* undefined
* NaN
* False

So every other value will be a truthy value if it doesn’t fall into the above list.

 First things first, there is no **!!**operator in JS... there is only the single **!**.

## **So, why double exclamation marks?**

In some cases you may want to cast a variable to be explicitly boolean. Why? Well, the number one reason is that most of time developers do not use type safe comparison operators.

The type safe comparison operators are:

* Strictly equal: ===
* Strictly unequal: !==

When using the type safe comparison operators you are both checking that the values are equal (or unequal) and that their type is the same. Without the type safe comparison operators you are allowing the JavaScript engine the freedom to coerce your variables to true or false based on the truthy/falsey logic.

To cast your JavaScript variables to boolean, simply use two exclamation signs:

function() {

var name = 'Brian';

//alert 'string'

window.alert(typeof name);

//cast to boolean

var bool = !!name;

//alert 'boolean'

window.alert(typeof bool);

}

In the example code above we are casting the string "Brian" to a boolean value. Therefore the second alert will indicate that the variable is now a boolean value.

One of the benefits of using the **!!**is to convey that your design decision was to only return a **true** or **false** value. When you or someone else looks at that code and a **!!**is encountered, the only possible return values are **true** or **false**. It is a matter of contention if readability is improved with the use of this logical tool.

For ex: following function will return either true or false

**function** isUserLoggedIn**(){**

Return **!!this.**user**;**

**}**

Refresh Token:

