ReactJS

# React : The Big Picture

# Why React?

## History

2011 – created by Facebook

2012 – Used on Instagram

2013 – Open sourced

Some people dismissed it because of mixup of markup and logic in single file. But over time, many people embraced it.

2014 – embraced by many large companies

2015 – React Native released for native mobile applications

2016 April – React 15 released

Today – Over 30K components are in production at Facebook

Full time dev staff for regular releases, bug fixes and documentation.

Used by many in Fortune 500 companies

Let’s explore six key reasons for choosing React.

1. Flexibility
2. Developer experience
3. Corporate investment
4. Community support
5. Performance
6. Testability
7. Flexibility : Learn once, write everywhere.

Once you learn ReactJs, you can build UI for huge variety of platforms. It’s more flexible than opinionated frameworks unlike Angular and Ember.

Where can we use React? – Started for Web applications. Now we can develop static websites using Gatsby and Phenomic, truly native mobile apps using React Native, installable desktop using Electron to run on Mac and Windows, also supports Server rendering using NextJS and finally can create Virtual Reality using ReactVR.

React Renderers

React is highly versatile because the renderer is separate from React iself. For web apps use react-dom, for mobile use react-native and for VR environments use react-vr renderer. There are over dozens of renderers including WebGL, CLI, PDF and word documents, canvas.

React-dom provides a simple function called ReactDomServer.renderToString() that renders your component to string of HTML for server side rendering. There are multiple libraries for SSR : NextJS, Gtasby and Phenomic, even for static websites. Since React is a lightweight library you can use it with existing apps too.

Facebook used React to slowly replace its php application. Since React is used by Facebook, its supported by multiple browsers including latest version of IE.

1. Developer experience

You’ll rarely use the docs. There are few APIs to learn.

For example:

import React from ‘react’;

function HelloWorld(props) {  
 return <div>Hello {props.name} </div>  
}

A simple hello world component can be rendered like this. The function receives an object and render its name property encapsulated within a div tag.

You can also declare a React component using a standard JavaScript class.

import React from ‘react’;

class HelloWorld extends React.Component {  
 return() {  
 return <div>Hello {props.name} </div>  
 }  
}

The HTML sitting inside JavaScript is called JSX.

JSX compiles to JS.

For example:

<h1 color=”red”>Heading here</h1>

Is compiled to:

React.createElement(“h1”, {color: “red”}, “Heading here”)

First parameter is the element tag, second parameter is an object with all attributes for the element and then third parameter is the markup that should sit inside the element.

It’s recommended to use JSX because for someone familiar with HTML, it’s easier to read.

Traditional frameworks like Amber, Vue and Angular seek to enhance the power of HTML by inventing their own syntax for example : looping.

“JS” in HTML

Angular : <div \*ngFor=”let user of users”>

Vue : <div v-for=”user in users”>

Ember: {{ #each user in users }}

React went the opposite route. Instead of trying to make HTML powerful, React handles HTML in JavaScript. You don’t need to learn new framework specific keywords, conditionals, looping, you just use JavaScript. JavaScript already has a function for looping called map().

“HTML” in JS

React : { users.map(createUser) }

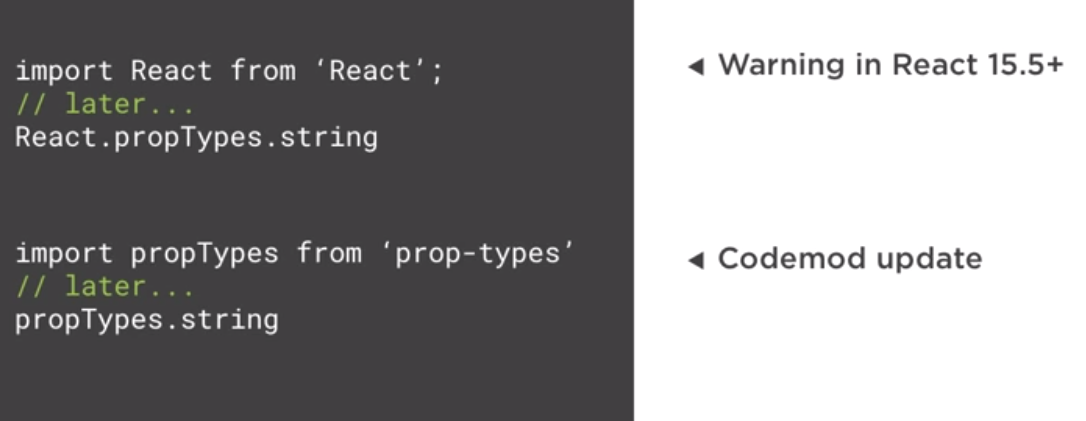
Traditional libraries put fake JavaScript in HTML and React puts fake HTML in JavaScript. React encourages you to get better at JavaScript and in doing so, get better at React.

Use create-react-app to create a new React application and npm start to serve your application. In React, each component is isolated and while developing, each change is applied instantly in the browser. If you make a mistake, you get a detailed error message in the browser and IDE. If you need to debug the code, just put ‘debugger;’.

You can use Visual Studio Code or online code editors such as Code Sandbox, GitPod etc for developing your app.

1. Corporate Investment

Facebook is deeply committed to React. Although React is open sourced, four of the top contributors of React are full time Facebook employees. And Facebook dev team maintains document for each release of React and plans for future. Because of Facebook’s deep existing commitment to React, when breaking changes occur in React, Facebook provides a codemod that automates the change. Codemod is command line tool that you can point at your code to automate the changes. So you can automatically update older React components to the latest specifications.



For example, in React 15.5, propTypes was moved to a different module and all the projects that were built with older versions of React started getting errors after upgrading. Using codemod, developers were able to update their code as per the changes of new version of React. We can rely on React because Facebook must rely on codemods to update the code that they create.

1. Community

Since 2013, React’s popularity has steadily grown to over 75K stars on Github. This makes React one of the most popular library on Github. Today it has over 1K contributors. React components are being downloaded 1.5 million times every single week. On Stackshare, where companies list down the technology stack that they use, over 5K companies have reported using React.

Companies using React : Apple, Adobe, Microsoft, Amazon, Twitter, BBC, Tesla, Netflix, Airbnb, Dropbox, Slack, Reddit, PayPal.

Microsoft open sourced their React library for Office UI.

Google has their own Material UI for React.

Deep community investments has led to wide variety of mature related projects. For routing, use React Router. Do you want to handle complex data flows, consider Redux and Mobx. Automated testing with Jest also from Facebook. Want a library for RESTFul API calls, check out GraphQL. Want server side rendering, try NextJS.

1. Performance

React team recognized that JavaScript is fast but it’s the DOM that makes it slow. They realized that updating the DOM is expensive so they found that updating the DOM in efficient way would help in performance.

Before React most libraries would unintelligently update the DOM to reflect the new state. This often led to redrawing a significant portion of the page even when only a minor change had occurred. In contrast, React monitors the values of each component’s state. When a component’s state changes, React compares the existing DOM state to what the new DOM looks like and then determines the least expensive way to update the DOM. Benefits of this approach: It avoids layout thrashing, which is when a browser has to calculate the position of each element when DOM element changes. And being efficient is important in a world of mobile devices so conserving battery and CPU usage is a concern. This also enables React’s simple programming model, when data changes, React effectively updates the DOM and there is nothing extra that you have to do. The comparison happens in memory so it’s fast.

Today many libraries use similar approach but React’s performance remains competitive. React provides many performance optimizations but they are rarely necessary.

ReactJS with React DOM only weight 35K and there are other React alternatives available such as Preact and Inferno that provide light weight library but they miss out on some features of React to help keep the size down.

1. Testability

Final reason is testability. Testing the front end is hard. React’s design is very friendly to automated testing.

Traditional UI tests vs React

1. It’s a hassle to setup. You have to carefully wire together multiple open source projects to get it to work. With React, testing is already configured for you out of the box.
2. Traditional UI tests require a browser. React uses Node to quickly test the app in memory.
3. Browser based UI tests are slow. In memory tests are faster.
4. Browser based UI tests are more integration oriented whereas React allows us to write Reliable Deterministic unit tests focused on single component in isolation.
5. Browser based tests are time consuming and hard to maintain. React tests can be written quickly in Jest and Enzyme.

With React, the vast majority of your tests contain pure functions. A pure function always returns the same output for the same input. It has no side effects. React’s component style makes it trivial to test your app.

For example:

function HelloWorld(props) {  
 return <div>Hello {props.name} </div>  
}

If we always pass the word “World” to this function then the output is always going to be Hello World. This is reliable, deterministic and has no side effects. It relies upon no global state. This design can scale upto highly complex UIs. Any one of following testing frameworks can work with React : Mocha, Jasmine, Tape, QUnit, Ava, Jest. For React, Jest is most popular and it’s created by Facebook. Every time you hit save during test runs, the automated tests are re-ran.

# Tradeoffs

Let’s consider six key tradeoffs when designing React.

1. Framework vs Library.
2. Concise vs Explicit
3. Template-centric vs JavaScript-centric.
4. Separate template vs Single-file component
5. Standard vs non-standard.
6. Community backed vs Corporate backed
7. Framework vs Library

Competitors like Angular and Ember are framework whereas React is a library. Here’s few advantages to choosing a framework. A framework contains more opinions. This reduces design fatigues and has less setup overhead. Frameworks can help enforce maintain consistency.

However React’s library approach, react is significantly lighter and you can use React in existing applications. You can use your React component to gradually update your application. This is why Facebook is slowly migrating its server side rendered PHP code to React. React allows you to only pull in the features that you need for your project. React can be used as an optional framework.

Angular comes with many features bundled such as Testing, HTTP library, Routing, Internationalization, Animation, Form validation and CLI. In contrast, since React is a feature composed library, you select the features you need and you add them in. Since React is popular, there are so many options for features you need.

1. Concise vs Explicit

React trades conciseness for predictability and explicitness. You spend a little more time wiring things together so that helps things not fall apart.

Example:

Frameworks like Angular and Knockout has two-way binding as a way to avoid typing by automatically keeping form inputs in sync with underline data. This was popular until React came. It was popular because it required less code. React supports one-way binding. It requires a little more code. With React, you declare an explicit change handler and you reference it on your input. This extra work has some benefits, you can transform and validate inputs before updating states and do some performance optimization as desired. This makes it easy to debug and handle when an error occurs. Although React helped repopularize one way binding, Angular and other frameworks have embraced it as well. You can use libraries in React for two way binding but it’s not recommended. Also you don’t need to declare separate change handler for each input. There are simpler patterns for centralizing your change handlers. So in real world, the amount of code you write in React is not substantially large because you have typically single change handler per component.

1. Template-centric vs JavaScript centric

Angular, Vue and Ember seek to make HTML more powerful by inventing their own unique syntax for writing code in HTML. React takes the opposite approach and utilizes the power of JavaScript to handle HTML. This makes React elegant. Let’s consider a simple Conditional which shows a tag when isAdmin is true.

Angular : <h1 \*ngIf=”isAdmin”>Hi Admin</h1>

Vue : <h1 v-if=”isAdmin”>Hi Admin</h1>

Ember: <h1>{{ if isAdmin ‘Hi Admin’}}</h1>

Here, Angular conditional is written inside a string and the ngIf directive must be prefixed with asterisk. Similar for Vue and Ember.

React : { isAdmin && <h1>Hi Admin</h1>}

With React, we can use JavaScript’s logical && operator. The right hand side only runs when left hand side is true. Since the code is plain JavaScript, we get autocomplete support as we type the code and if we type something invalid, we get error messages.

Now let’s consider a loop in each technology.

Angular: <div \*ngFor=”let user of users”>{{ user.name}}</div>

Vue: <div v-for=”user in users”>{{ user.name }} </div>

Ember: {{ #each users as |user| }}  
<div>{{ user.name }}</div>  
{{ /each }}

With Angular, you say ngFor then use a syntax that looks like JavaScript but that’s declared inside a string. Similar for Vue. With Ember, you use Ember’s pound each helper which is lengthy.

React : users.map(user => <div>{ user.name } </div>)

With React, you use JavaScript’s built in map() that takes an arrow function which displays the user’s name. So it’s preferable because the syntax is plain JavaScript.

Finally let’s see how each handles clicking a button.

Angular: <button (click)=”delete()”>Delete</button>

Vue: <button v-on:click=”delete”>Delete</button>

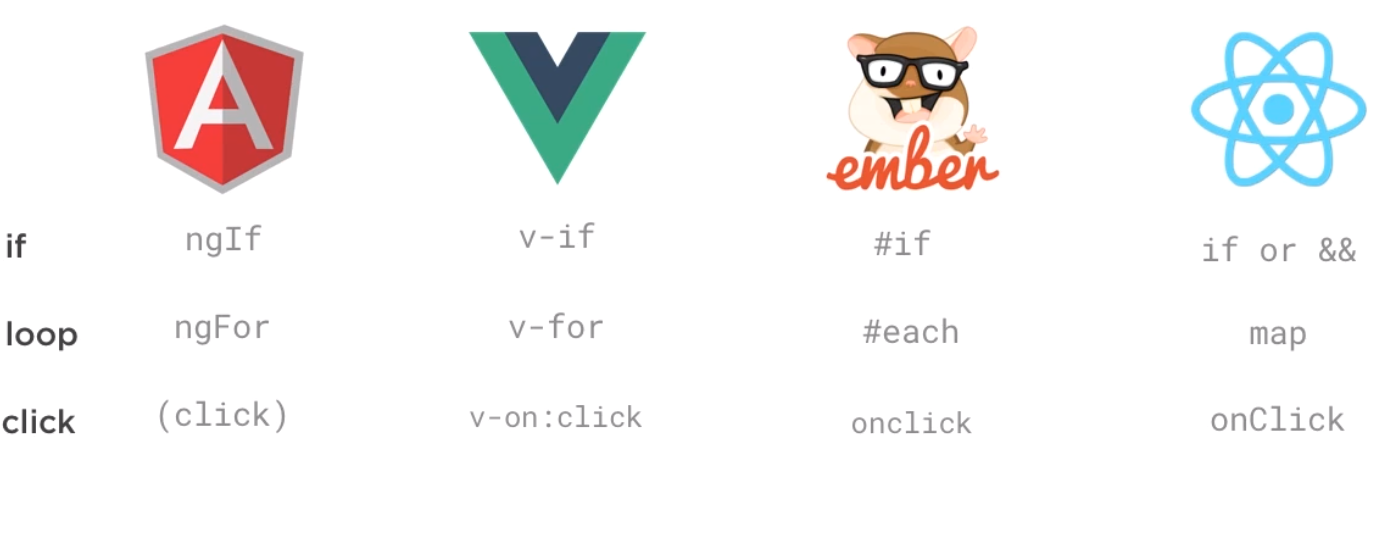
Ember: <button onclick={{ action ‘delete’ }}>Delete</button>

With Angular, you have parenthesis around the events, and unlike traditional event handlers, you also have to put parenthesis after the event handler method. This wouldn’t work if it were real JavaScript. With Vue, you put v-on before the events. With Ember, you specify plain onclick but inside you Ember specific convention to wrap the method in string with action keyword.

React: <button (click)=”delete()”>Delete</button>

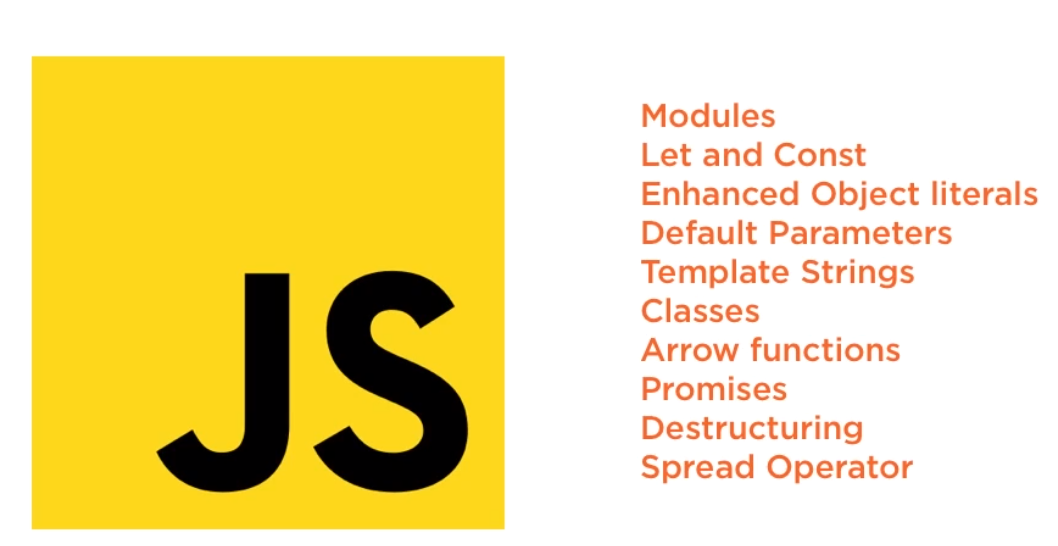
Finally in React, you use native click handler name but its camel case since JSX uses JavaScript casing rules, otherwise the only unique syntax is you specify the function name inside single braces.

If you know JavaScript, then you can easily write conditionals, loops and event handlers in React. This is why React’s API is so small.



Let’s see the benefit of each approach.

Template-centric : It requires little JS knowledge. Template languages provide a streamlined API for performing core functions. You focus on enhancing the template with framework specific syntax. These syntaxes are useful for avoiding confusion with JS binding and this keyword. Template languages is preferable because of principle called Rule of least power. It’s counter-intuitive but less powerful languages can be preferable because they can protect from misuse by only allowing to perform a small set of prescribed operations. For example, Angular’s template syntax only supports JavaScript syntax of comparison. Compared to that, React has little framework specific syntax because you spend more time learning syntax of the framework which JavaScript already supports. React has fewer concepts to learn. Most of the code is JavaScript in React. React’s code is easy to read and it improves JavaScript learnings.



In order to learn React, you need to get better at JavaScript because this means your skills transfer to all JavaScript code even if you use framework or library other than React.

1. Separate template vs single file

Unlike MVC template where model, view and controller are separately maintained in files, in React, all the markup and logic exist in the same file and they together make a component. Because React recognizes that although they are logically separate, they must be intertwined to do anything useful so in React, each component is a separate concern for e.g. Button component, datepicker component, accordion etc. With React, you can still keep CSS in a separate file.

1. Standard vs non-standard

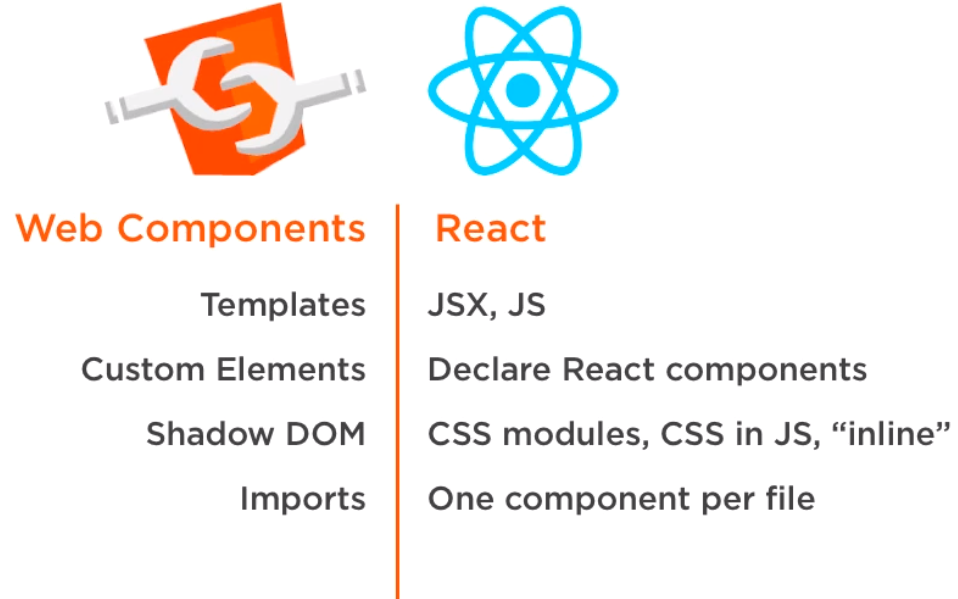
React is one of many non-standard component libraries just like Angular, Vue etc. But the WebComponent standard has been around for so years without much usage yet. Why aren’t many people building web pages with standardized Web components yet?

First let’s understand what a web component standard is. The web component standard consists of four core technologies:

1. Templates : contains markup
2. Custom Elements : to expand HTML with custom elements
3. Shadow DOM: encapsulating styles and prevent it leaking outside
4. Imports : Bundling HTML, CSS and JS into a single line for importing.

Then why not use the Web component standards? : Because the browse support remains spotty. The template tag isn’t supported in any version of IE. Html imports is supported only in Chrome, Opera and Android. Custom elements are only supported in Chrome, Opera and newest versions of Android browsers. Similar story with Shadow DOM. So over the years, it’s pretty clear that browser vendors have shown little interest in supporting full suite of HTML5 Web component features. So you need to use pollyfills to make it all work cross browser. So when you’re pulling in extra JavaScript, you need to ask why you’re choosing a poorly supported standard like Web Components instead of a popular technology like React. Secondly, Web Components don’t enable anything new. Everything that you can do in Web Components can be accomplished today in a cross-browser-friendly way using a variety of modern JavaScript libraries including React.

Let’s consider core features of Web components against React.



Besides that, React and other frameworks like Angular keep improving their libraries. Web Components only run in the browser whereas React can run on Virtual Reality too.

The web component standard may take off at some point but currently the majority of developers continue to reach for the tools like React because they are innovating more quickly, they offer strong user experience and they run cross browser.

1. Community vs corporate backing

Many JavaScript libraries are community driven. React is open source but it’s backed by Facebook. This means React is driven by Facebook’s needs. So if your apps are very different from what Facebook is building then React may not be ideal for you. But React’s corporate backing because Facebook provides full time staff for React that carefully plans releases, prepares documentation, blog posts and on going support for bugs. Facebook is 5th most valuable company today.

# Why not React?

JSX differs from HTML

Build Step

Potential version conflicts

Old features in searches

Decision fatigue

License (React now uses a standard MIT open source license)

1. JSX differs from HTML

Although JSX appears 99% similar to HTML, there are few differences such as htmlFor instead of for, className instead of class, inline styles are in JSON format and finally, comments are handled in JavaScript style instead of HTML style.

If your concern is about how to convert HTML from your existing app into its equivalent in react, there are some tools to do that.

1. You can either find and replace the syntaxes.
2. You can use online HTML to JSX compiler
3. OR you can use htmltojsx package on npm. It’s a command line tool that does the same thing as the online HTML to JSX compiler.
4. Build step required

When you use JSX, you need to compile your code into JavaScript which seems like an additional build step. But when you are building modern web applications, you’re going to continue to use build steps for purposes such as Bundling and Minifying, Transpiling, Testing and Linting. Good for React developers is that TypeScript and Babel are popular transpilers that can compile JSX into JS. And there are many variety of boilerplate today that make it easy to get started and have build steps built in automatically to transpile JSX into JS. “Create-react-app” does this transpiling automatically.

1. Version conflicts

You can’t run two versions of the React at the same time on the same page. So all the components must use the same version of React to be useful on a page. Since React is a lean component library, you’d often use React compatible libraries but it also means if you want to use latest version of React Router then you need to use React 15+.

A workaround for this is:

1. Standardize on a version that your team is going to use.
2. Upgrade React when upgrading libraries
3. Upgrade as a team
4. Old stuff online

Because React is an older library, many of questions and answers will be for older versions of react so some of the content of features and examples are outdated because there are features that are extracted away.

For e.g. instead of  
“import { render } from ‘react’;” use “import { render } from ‘react-dom’;”

React.createClass was extracted to a separate library called ‘create-react-class’. So you need to reference a separate library to create a new React class.

Similarly, only some teams like to use PropTypes in react, it was extracted to its own library called ‘prop-types.

1. Decision fatigue

# React : Getting started

JavaScript experience is required.

Topics:

- variables and types  
- objects and arrays  
- functions and classes  
- loops and conditionals

Learn JS:

- <https://jscomplete.com/beginning-js>

- <https://jscomplete.com/js-labs>

Common problems beginners usually face with React: <https://jscomplete.com/react-cfp>

React is a JavaScript library for building user interfaces. You’ll need to use other libraries to form a solution. React does not assume anything about any library or solution.

React is declarative, it means we tell React what we want and not how to accomplish what we want. React will take care of how translate our declarative descriptions which we write in React and translate that into user interfaces.

## React’s basic concepts

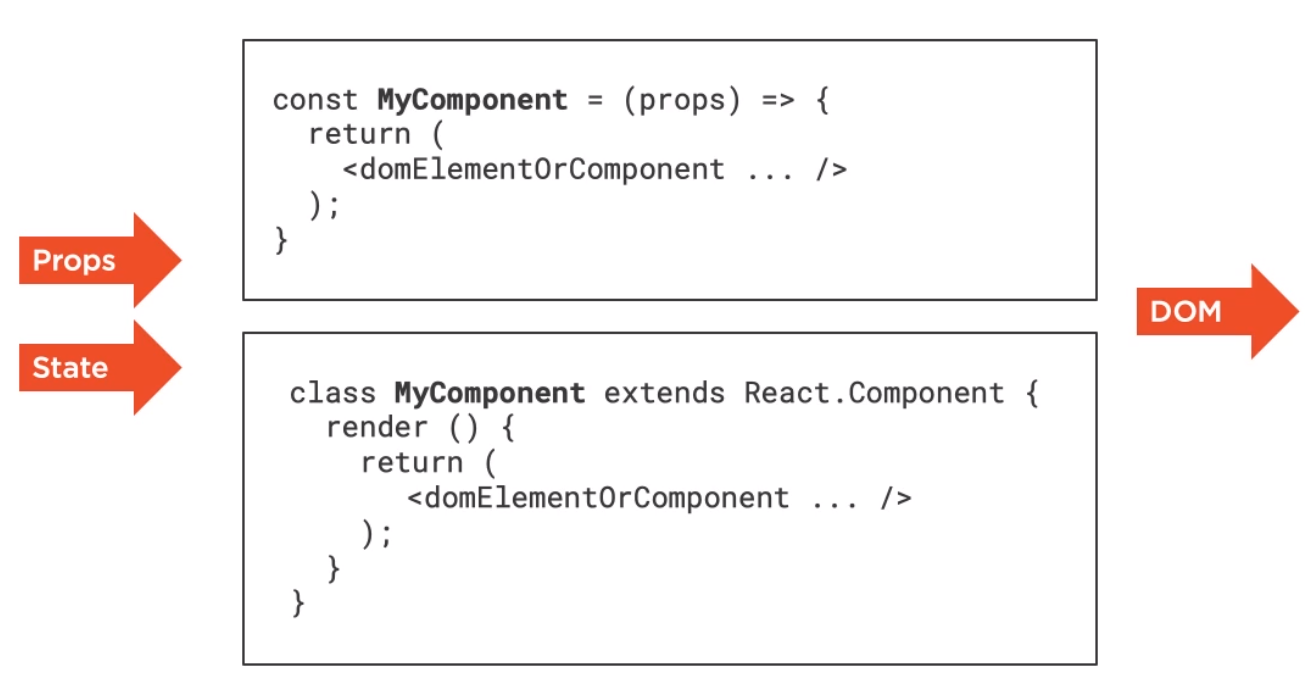
1. Components
2. like functions
3. Input: props, state | Output : UI
4. Reusable and composable
5. Easy to use : <Component />
6. Can manage a private state
7. Reactive updates

When input changes, the library reacts and changes the output i.e. UI, this is self-explanatory why the library was named React. It automatically updates parts of the DOM.

1. Virtual views in memory

Generates HTML using JavaScript. React uses “Tree Reconciliation” algorithm to update views in DOM and then update only parts that need to be changed.

React Component can be a function component or a class component. They can be stateful or they can be purely presentational. Prefer to use function components because they are simpler.



Components take props and states as input and they output what is called JSX which looks like HTML but is just JavaScript. Props are just attributes that HTML elements can have. State input is an internal property that helps React auto reflect changes in browser. In a component, state input can be changed but props are fixed values, they are immutable. A component can change its internal state, not its properties.

## Build your first React component

You can practice React on <https://jscomplete.com/playground>. Download and add React Devtools extension, it allows us to inspect and interact with any React application. You can see the state of components by opening Chrome Dev Tools and in the React tab. You can even change the values of the components and the change would be reflected by the browser.

First practice session can be done at : <https://jsdrops.com/rgs1.1>

This online editor hosts an HTML element called ‘mountNode’, you can use this element as the entry point for your React application.

function Hello() {  
 return <div>Hello React!</div>;  
}  
ReactDOM.render(  
 <Hello />,  
 document.getElementById(‘mountNode’);   
);

A component name has to start with an uppercase letter. Above component Hello() takes no input and returns a div, it doesn’t have any state. To display a component, we need to instruct ReactDOM library to render it. ReactDOM’s render takes the component as first parameter and the HTML element where the component should be rendered.

In real application, you must keep an HTML for entry point for your react and pass its name to the ReactDOM.render() function. We are telling React to take over this element and render all content within this element.

Now if you copy above React code and try to run it in browser console, it won’t run because it’s not valid JavaScript.

But in react application, this JSX syntax is compiled into JavaScript. You can try online BabelJS compiler to convert your JSX code into JS code to understand how React works.

*Input: <div>Hello React!</div>*

*Output: React.createElement(‘div’, null, ‘Hello React!’);*

If we replace our code with above compiled code, it would produce the same output.

This is true for ReactDOM.render() too.

That line gets converted to :

*ReactDOM.render(  
 React.createElement(Hello, null),  
 document.getElementById(‘mountNode’),  
);*

Output would be the same if we replaced both lines of code as follows:

*function Hello() {  
 return React.createElement(‘div’, null, ‘Hello React!’);  
}  
ReactDOM.render(  
 React.createElement(Hello, null),  
 document.getElementById(‘mountNode’);   
);*

Now let’s change our component’s name from Hello to Button and make it return HTML button.

*function Button() {  
 return <button>TEST</button>;  
}  
ReactDOM.render(  
 <Button />,  
 document.getElementById(‘mountNode’);   
);*

Run it and you should see a button with “TEST” as label. Now we intentionally rendered an HTML button with a component Button because capitalization is important in React. Otherwise React would try to render the HTML element button instead of our component Button. Try renaming the Button to button and pass the smaller case button in the render() function and you will see a button with no label.

## Your First React Hook

Continuing with previous example for Button, the code is available at : <https://jsdrops.com/rgs1.2>

Our component renders a stateless button. We need to make a button to increment a counter. We need a state object. To use a state object, React has a special function called useState(). We are going to call this function and it will return two items. The first item is a state object (getter) and second item is a function to update the state object (setter). The state object can be anything a string, an array, a number or anything else. For this use case, we need a number. We will name this state object ‘counter’ and the updater function will be called ‘setCounter’. JavaScript functions can return only a single values so this useState() function returns an array with exactly two items in the sequence : [stateObject, updaterFunction].

To make this work, we need variables to hold the result. For that we use a special syntax:  
*const [currentStateValue, functionToUpdateStateValue] = useState(initialStateValue);*

For our use case:  
*const [counter, setCounter] = useState();*

This special syntax uses JavaScript’s Array Destructuring feature to capture the two items of the array into these two variables which we have named ‘counter’ and ‘setCounter’.

We can also pass an initial value to useState as an argument. We’ll set that value to 0.

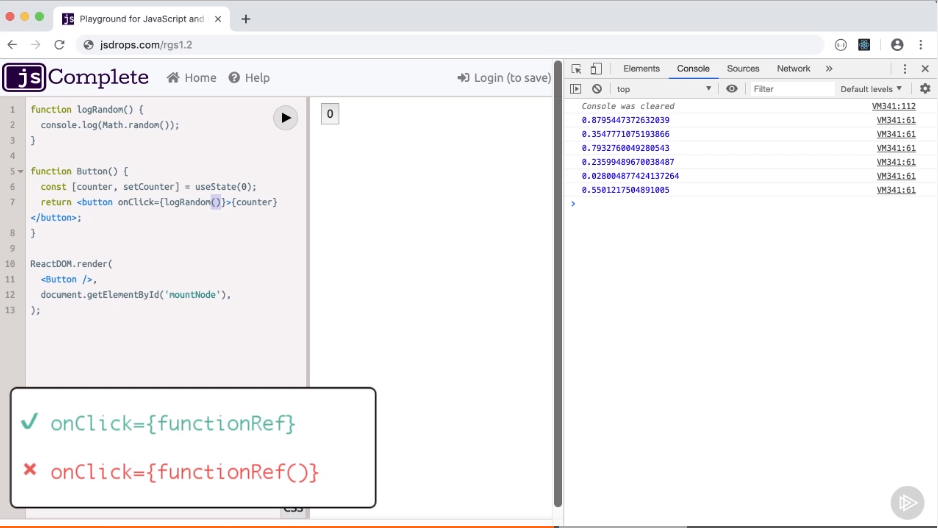
To display our state in the HTML, we will dynamic expressions supported by JSX, these expressions are placed within curly braces. So all we have to do to display the counter variable on our button label is place it within { } braces.

*function Button() {  
 const [counter, setCounter] = useState(0);  
 return <button>{counter}</button>;  
}*

Now to use the setCounter function, we need to introduce an event on the button, e.g. click event, which will call this setCounter function every time the button is clicked. In JSX, the click event name should be in camelCase. We define an onClick attribute on the button element and unlike the DOM version of the event which receives a string, this attribute will get a reference to our setCounter in curly braces. It means instead of calling our function on click, we are passing the function itself as a parameter. Let’s create a dummy function which will log random numbers.

*function logRandom() {  
 console.log(Math.random());  
}  
function Button() {  
 const [counter, setCounter] = useState(0);  
 return <button onClick={logRandom}>{counter}</button>;  
}*

When you run the code, a button will show with 0 as label and every time you click, it will log random numbers.



You can also pass a function inline, which is known as Arrow functions.

*function logRandom() {  
 console.log(Math.random());  
}  
function Button() {  
 const [counter, setCounter] = useState(0);  
 return <button onClick={ () => console.log(Math.random()); }>{counter}</button>;  
}*

Now what we need to do is call setCounter function and the argument to set counter will be the new value for our ‘counter’ state variable. But because this onClick accepts functions as parameters, so we have to use Arrow functions and call our setCounter inside this arrow function.

*function logRandom() {  
 console.log(Math.random());  
}  
function Button() {  
 const [counter, setCounter] = useState(0);  
 return <button onClick={ () => setCounter(counter + 1); }>{counter}</button>;  
}  
ReactDOM.render(  
 <Button />,  
 document.getElementById(‘mountNode’);   
);*

Now we run this code and on every click, the counter value will increase by 1.

This useState function is called a hook in React. It hooks a component into a state. We did not update the UI, we only supplied the value using an updater function and React updated the UI.

## Learn one-way data flow

Now we will try to create more components and pass data from one component to another. React supports one-way data flow.

Our previous code is all in one line. Let’s make it more readable. Instead of defining an arrow function on onClick attribute, we will create a function separately for the incrementing counter and then pass that function reference to onClick.

*function Button() {  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = () => setCounter(counter+1);  
 return (  
 <div>  
 <button onClick={incrementCounter} increment={1}>  
 {counter}  
 </button>  
 </div>  
 );  
}  
ReactDOM.render(  
 <Button />,  
 document.getElementById('mountNode'),  
);*

We will split our one Button component into two: one component for displaying a button and another one for incrementing counter. This new displaying component will not have its own state and that’s okay, not every component is expected to have their state.

So we define a Display function as a component.

*function Display() {  
 return (  
 <div>....</div>  
 );  
}*

But this will not be rendered by React because it’s not included in ReactDOM.render() function. But we can’t just pass an HTML-like sequence of elements and expect the buttons to be displayed:

*ReactDOM.render(  
 <Button /><Display />,  
 document.getElementById('mountNode'),  
);*

This won’t work because each one of these component’s markup gets translated into a function call as follows:

*React.createElement(Button, null)*  
OR  
*React.createElement(Display, null)*

And JavaScript can’t have two function calls as parameter without being separated by a comma. We have few options to fix this problem. We can render an array of elements here.

*ReactDOM.render(  
 [ <Button /><Display /> ],  
 document.getElementById('mountNode'),  
);*

The other option is to make these React elements a child of another element.

*ReactDOM.render(  
 <div>  
 <Button /><Display />  
 </div>,  
 document.getElementById('mountNode'),  
);*

In fact, React has its own element <React.Fragment> for wrapping our component elements and we don’t have to introduce a new DOM parent such as DIV parent in above example.

*ReactDOM.render(  
 <React.Fragment>  
 <Button /><Display />  
 </React.Fragment>,  
 document.getElementById('mountNode'),  
);*

React also supports empty tags in place of React.Fragment and this will be compiled just like React.Fragment.

*ReactDOM.render(  
 <>  
 <Button /><Display />  
 <>,  
 document.getElementById('mountNode'),  
);*

We can keep using <DIV> as parent for continuing this session.

Next, we can extract the markup parameter into its own component. It can have any name. Let’s call it App.

*function App() {  
 return (  
 <div>  
 <Button />  
 <Display />  
 </div>  
 );  
}*

And then use this App component in render().

*ReactDOM.render(  
 <App />,  
 document.getElementById('mountNode'),  
);*

Since we are going to display the counter’s value in the new Display component, then the code should be in Display component but if we do that then the setCounter function will not be available in Button component. So instead we will move the useState() function code one level up in the App component.

*function Button() {  
 return (  
 <button onClick={ incrementCounter }>  
 </button>  
 );  
}  
  
function Display() {  
 return (  
 <div>…</div>  
 );  
}  
  
function App() {  
 return (  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = () => setCounter(counter+1);  
 <div>  
 <Button />  
 <Display />  
 </div>  
 );  
}*

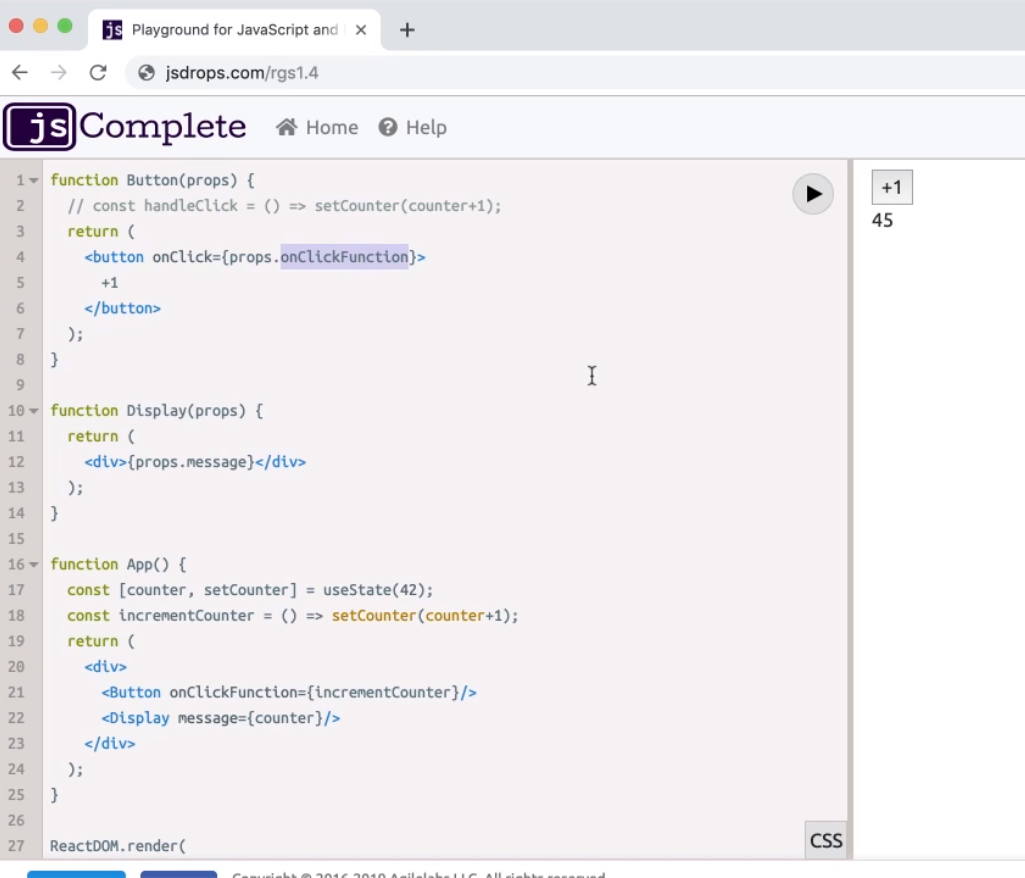
Now we want the value of counter to flow into the Display component. So we are going to use props object in React. To pass a data from one component to another, we will set an attribute on the that component’s element markup and the value will be accessible inside the function component through props object. All function components receive this props object even when we are not passing anything.

Because the button component can receive many attributes, this props object is accessed like a name-value pair. So we can display the message

*function Display(props) {  
 return (  
 <div> { props.message } </div>  
 );  
}  
  
function App() {  
 return (  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = () => setCounter(counter+1);  
 <div>  
 <Button />  
 <Display message={ counter } />  
 </div>  
 );  
}*

So now we are able to pass the counter’s value from App to Display. Parents can pass data down to child components. Similarly parents can also pass behavior to child components, which is what we need to do next. We will pass a reference to incrementCounter method from App to Button component as a prop. In JavaScript, functions are objects and you can pass any object as a parameter.

*function Button(props) {  
 return (  
 <button onClick={ props.onClickFunction }>  
 +1  
 </button>  
 );  
}  
  
function Display() {  
 return (  
 <div>…</div>  
 );  
}  
  
function App() {  
 return (  
 const [counter, setCounter] = useState(42);  
 const incrementCounter = () => setCounter(counter+1);  
 <div>  
 <Button onClickFunction={ incrementCounter } />  
 <Display message={ counter } />  
 </div>  
 );  
}*



Components reusability

One of the selling points of components is re-usability, making a component generic enough so we can reuse it. Let’s make the Button component more generic. Assume we can pass a value and the button will increment the counter with that value. So we will try to make a +1 button, +5 button etc. The weight value for increment will be passed in an attribute on Button. We can pass this value in double quotes but the Button component will receive it as a string so we should pass the value in curly braces.

*function App() {  
 return (  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = () => setCounter(counter+1);  
 <div>  
 <Button onClickFunction={ incrementCounter } increment={1} />  
 <Display message={ counter } />  
 </div>  
 );  
}*

We have added increment attribute and replicated the markup for different values. And then we are accessing this for label in Button using props.

*function Button(props) {  
 return (  
 <button onClick={ props.onClickFunction }>  
 + { props.increment }  
 </button>  
 );  
}  
function App() {  
 return (  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = () => setCounter(counter+1);  
 <div>  
 <Button onClickFunction={ incrementCounter } increment={1} />  
 <Button onClickFunction={ incrementCounter } increment={5} />  
 <Button onClickFunction={ incrementCounter } increment={10} />  
 <Button onClickFunction={ incrementCounter } increment={100} />  
 <Display message={ counter } />  
 </div>  
 );  
}*

This code is still incrementing by 1 because the setCounter has hard-coded value 1. So to fix this, we will pass a parameter to the arrow function ‘incrementCounter’ and use it instead of 1.

And inside the Button function component, assign a new function that calls props.onClickFunction with the increment value, because the onClick attribute expects a function reference, it doesn’t call so we can’t pass value to it.

*function Button(props) {  
 const handleClick = () => props.onClickFunction(props.increment);  
 return (  
 <button onClick={ handleClick }>  
 + { props.increment }  
 </button>  
 );  
}  
function App() {  
 return (  
 const [counter, setCounter] = useState(0);  
 const incrementCounter = (incrementValue) => setCounter(counter + incrementValue);  
 <div>  
 <Button onClickFunction={ incrementCounter } increment={1} />  
 <Button onClickFunction={ incrementCounter } increment={5} />  
 <Button onClickFunction={ incrementCounter } increment={10} />  
 <Button onClickFunction={ incrementCounter } increment={100} />  
 <Display message={ counter } />  
 </div>  
 );  
}*

## Tree Reconciliation in action

Now we will understand why React is so popular and how React is proven useful for building UI. In this example, we have two DIVs on our page. Both these DIVs have Hello text inside them. But one of them uses plain HTML and JavaScript. The JavaScript code updates the innerHTML with Hello HTML text. The second DIV is added by React and calls render(). Note that the multiline HTML in this example is enclosed between **backticks** (`) instead of **single quotes**; this multiline string is known as template string.

Source:

***document.getElementById('mountNode').innerHTML = `***

***<div>***

***Hello HTML***

***</div>***

***`;***

***ReactDOM.render(***

***React.createElement(***

***'div',***

***null,***

***'Hello React',***

***),***

***document.getElementById('mountNode2'),***

***);***

***// currentTime: (new Date).toLocaleTimeString()***

***// setInterval(fn, 1000);***

Output:



This is pretty simple example. Let’s add some more elements. Let’s add an input box to both DIVs. In plain JavaScript, it’s very easy.

***document.getElementById('mountNode').innerHTML = `***

***<div>***

***Hello HTML***

***<input/>***

***</div>***

***`;***

We can do the same in React by adding more arguments after the 3rd argument to React.createElement().

***ReactDOM.render(***

***React.createElement(***

***'div',***

***null,***

***'Hello React',***

***React.createElement("input", null)***

***),***

***document.getElementById('mountNode2'),***

***);***

Similarly we can add some more elements, for e.g. let’s display a time inside a pre tag.

***document.getElementById('mountNode').innerHTML = `***

***<div>***

***Hello HTML***

***<input />***

***<pre>${(new Date).toLocaleTimeString()}</pre>***

***</div>***

***`;***

***ReactDOM.render(***

***React.createElement(***

***'div',***

***null,***

***'Hello React',***

***React.createElement('input', null),***

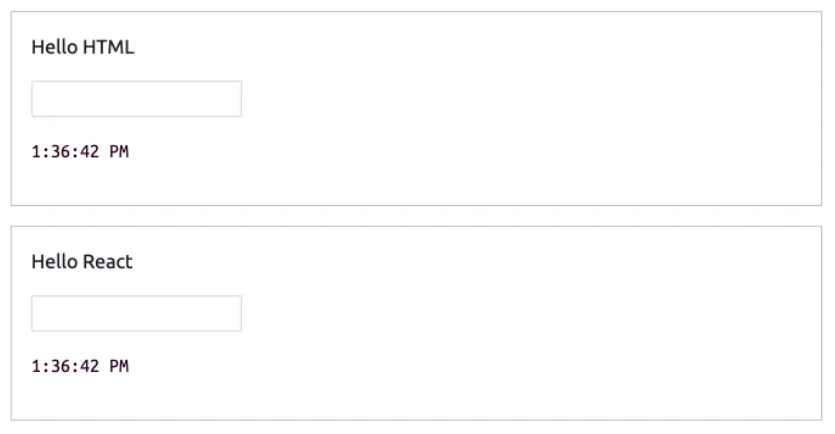
***React.createElement('pre', null, (new Date).toLocaleTimeString())***

***),***

***document.getElementById('mountNode2')***

***);***

And we get following output:



Going by the effort to add more HTML, it’s obvious that React’s way of adding elements is complicated than plain JavaScript and HTML. So why go through all this trouble and what it is worth in React ? The answer is not in how React renders a view first time, it’s how React updates the UI again and again.

So now we will render above HTML on an interval of 1 second. Following code will render those two DIVs every second and update the time in the pre tag too.

Source code :

const render = () => {  
 document.getElementById('mountNode').innerHTML = `  
 <div>  
 Hello HTML  
 <input />  
 <pre>${(new Date).toLocaleTimeString()}</pre>  
 </div>  
 `;

ReactDOM.render(  
 React.createElement(  
 'div',  
 null,  
 'Hello React',  
 React.createElement('input', null),  
 React.createElement('pre', null, (new Date).toLocaleTimeString())  
 ),  
 document.getElementById('mountNode2')  
 );  
 };  
  
setInterval(render, 1000);

When you run above code, it will render two div sections, one using HTML and JavaScript and another using React. This code is being called on an interval of 1 second and the DOM is being re-rendered.