

React Basics – Part 1

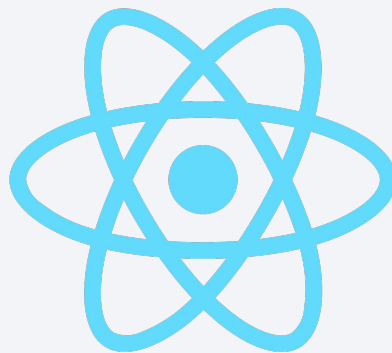
	What is React?	Props
Contents	Why use React?	State
	JSX	Lifecycle

What is React?

React is a **JavaScript library for building user interfaces**. It is an open-source, component-based, front-end library responsible only for the application's view layer.

React is **the most popular front-end JavaScript library** in the field of web development.

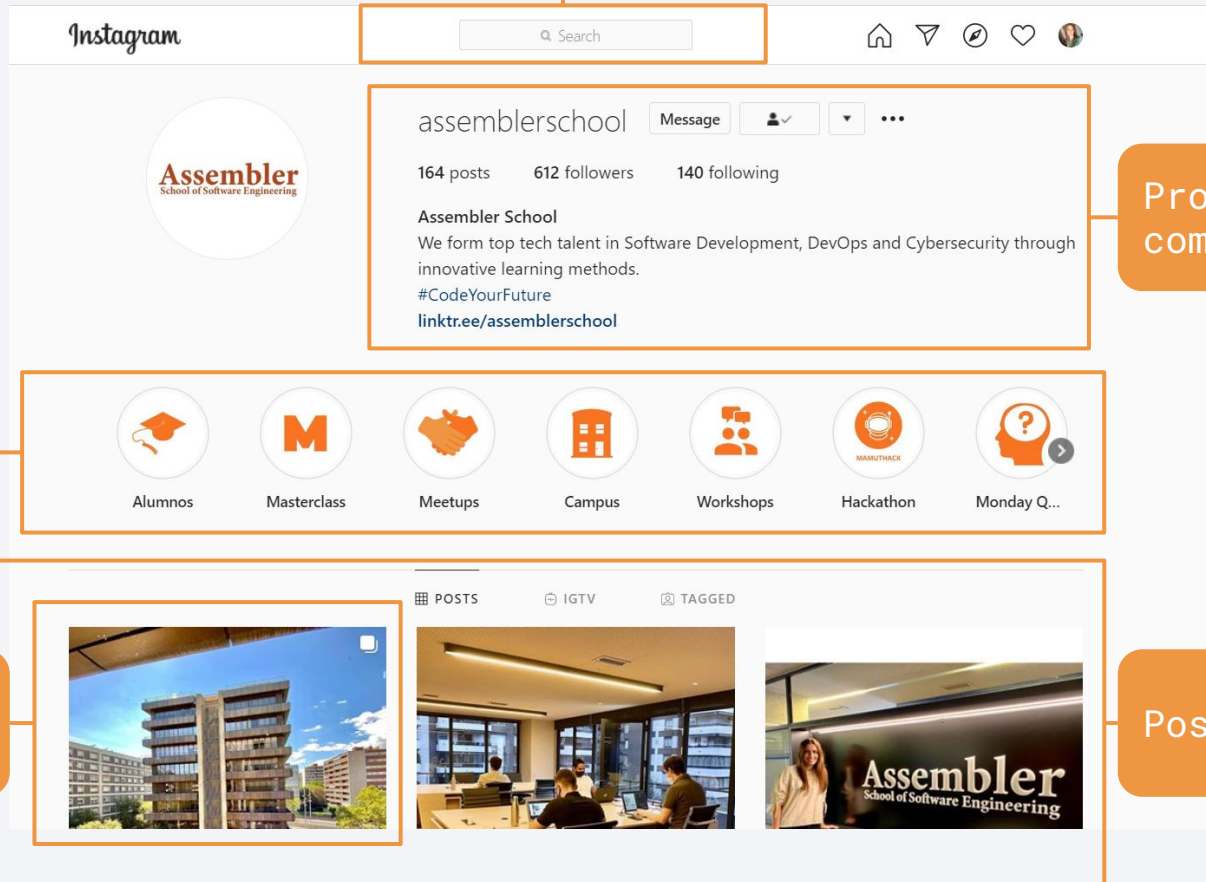
React was created by Jordan Walke, a software engineer at Facebook.



Instagram

Let's take a look at an Instagram webpage example, entirely built using React, to get a better understanding of how React works.

Search component



Stories component

Single Post component

Profile Description component

Post List component

As the illustration shows, React divides the UI into multiple components, which makes the code easier to debug. This way, each component has its property and function.

Example

This is how the components would be distributed in code.

```
function App() {  
  return (  
    <div className="App">  
      <Search />  
      <ProfileDescription />  
      <Stories />  
      <SinglePost />  
      <PostList />  
    </div>  
  );  
}
```

Why use React?

Now that we know what React is, let's move on and see why React is the most popular front-end library for web application development.

Why React?

- **Improved performance:** React uses Virtual DOM and compares the components' previous states and updates only the items in the Real DOM that were changed, instead of updating all of the components again, as conventional web applications do.
- **Reusable components:** Components can be reused throughout the application, which in turn dramatically reduces the application's development time.

Why React?

- **Unidirectional data flow:** When designing a React app, developers often nest child components within parent components and it becomes easier to debug errors and know where a problem occurs in an application at the moment in question.
- **Small learning curve**
- **It can be used for the development of both web and mobile apps:** There is a framework called React Native, derived from React itself, that is used for creating beautiful mobile applications.
- **Dedicated tools for easy debugging:** React also has a Chrome extension that can be used to debug React applications.

Installing React

To start using React we only need to install the official **Create React App** package.

```
npm i -g create-react-app
```

```
// Command to create react app  
create-react-app myapp
```

Folder structure

The application structure in React is basically the **structure of folders, sub-folders and files included in a project**. Once we create a project in React , we get an overview of the application structure as shown in the image here.

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

public

public is where your **static files reside**.

If the file is not imported by your JavaScript application and must maintain its file name, put it here.

Files in the public directory will maintain the same file name in production, which typically means that they will be cached by your client and never downloaded again.

> node_modules

▼ public

★ favicon.ico

<> index.html

🖼 logo192.png

🖼 logo512.png

{ } manifest.json

☰ robots.txt

> src

💎 .gitignore

{ } package.json

{ } package-lock.json

📘 README.md

src

src is where your **dynamic files reside**.

If the file is imported by your JavaScript application or changes contents, put it here.

> node_modules

> public

▼ src

App.css

⚙ App.js

JS App.test.js

index.css

JS index.js

🖼 logo.svg

JS serviceWorker.js

JS setupTests.js

💎 .gitignore

{ } package.json

{ } package-lock.json

📘 README.md

JavaScript Syntax Extension

JSX

JSX

JSX is a **syntax extension to JavaScript**. It is used with React to describe what the user interface should look like. By using JSX, we can write HTML structures in the same file that contains JavaScript code. This **makes the code easier to understand and debug**, as it avoids the usage of complex JavaScript DOM structures.

JSX Example

In the example below, we declare a variable called **name** and then use it inside **JSX** by wrapping it in curly braces:

```
const fullName = "Josh Perez";

const element = <h1>Hello, {fullName}</h1>;

<PostList />;
```

Specifying Attributes with JSX

- You can use quotes to specify string literals as attributes.
- You can also use curly braces to embed a JavaScript expression in an attribute.

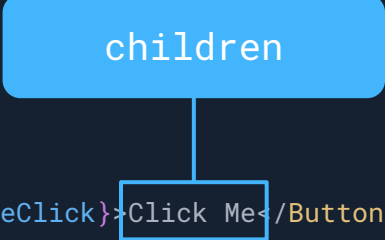
```
function App() {  
  return <Button id="1" />;  
}
```

```
function Button(props) {  
  return <button id={props.id}>Click id: {props.id}</button>;  
}
```

Specifying Children with JSX

JSX tags can also contain
child elements:

```
function App() {  
  function handleClick(event) {  
    console.log(event.target);  
  }  
  return <Button handleClick={handleClick}>Click Me</Button>;  
}  
  
function Button(props) {  
  return <button onClick={props.handleClick}>{props.children}</button>;  
}
```



Render

The **render()** method is used to convert JSX code to regular JavaScript at runtime. After translation, JSX code looks like this:

```
const element = <h1>Hello, world</h1>;
```

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

container where the
element will render

element that you want to render

Updating the Rendered Element

React elements are immutable. Once you create an element, you **can't change its children or attributes**. An element is like a single frame in a movie: **it represents the UI at a certain point in time**.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to `ReactDOM.render()`

React Only Updates What's Necessary

After looking at the workshop demo we can see that React DOM compares the element and its children to the previous one, and **only applies the DOM updates necessary** to bring the DOM to the desired state.

Components and Props

Components let you split the UI into **independent**, **reusable pieces**, and think about each piece in **isolation**.

Function component

The simplest way to define a component is to write a JavaScript function:

```
// Using a functional component  
function Welcome(props) {  
  // Notice that a functional component should have a return statement  
  return <h3>Hello {props.name}</h3>;  
}
```


Class component

You can also use an ES6 class to define a component:

```
class SecondWelcome extends React.Component {  
  // Notice that class components should have a render method  
  render() {  
    return <h1>Hello {this.props.name} from a class</h1>;  
  }  
}
```

What are props?

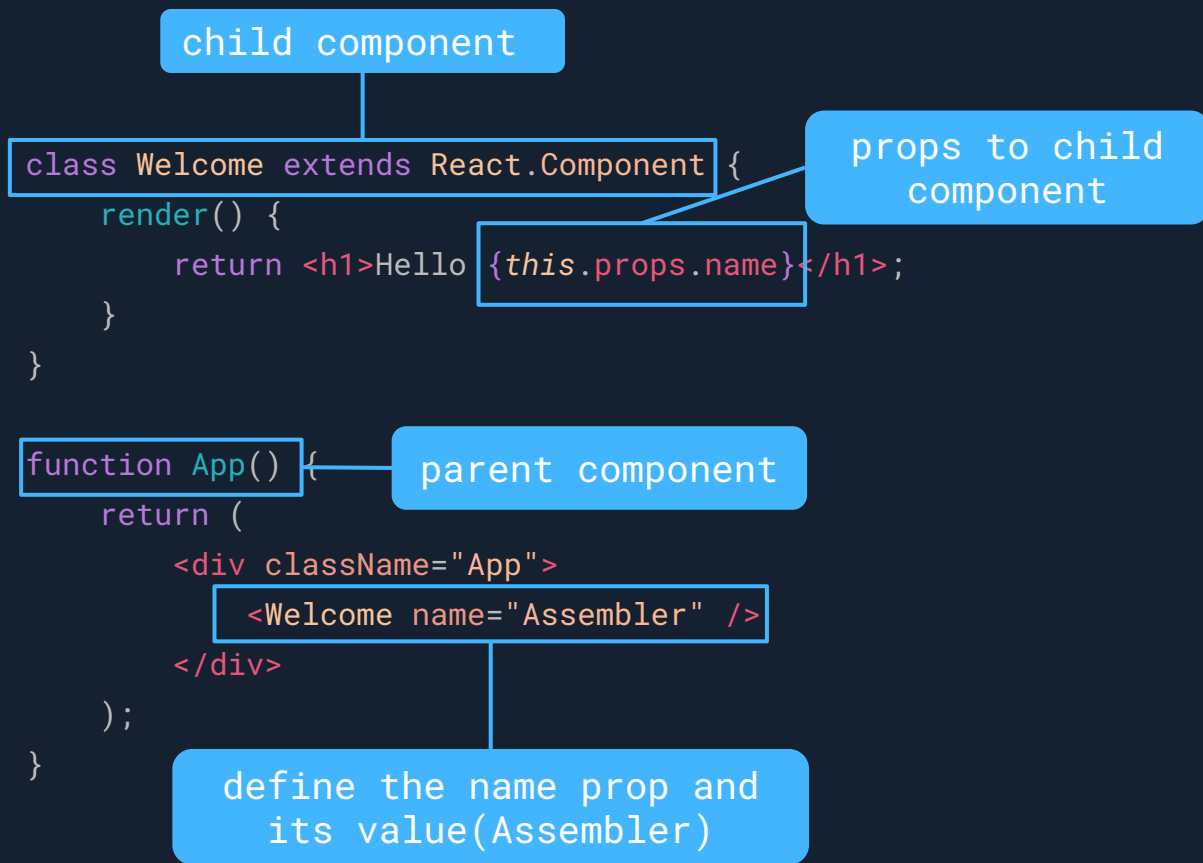
React is a **component-based library** which divides the UI into little reusable pieces. In some cases, those components need to communicate (send data to each other) and the way to pass data between components is by using **props**.

“**props**” is a special keyword in React, which stands for **properties** and is being used for **passing data from one component to another**.

Using props

Props step by step:

- Firstly, we pass the props to child component(s)
- Then we define an attribute and its value(data) to parent component
- Finally, we render the props data



Extracting Components

Don't be afraid to split components into smaller components.

For example, consider this App component:

```
import React from "react";
import img from "../img/react-components-intro.png";

import "../App.scss";

function App() {
  return (
    <div className="App">
      <h1>Hello Assembler</h1>
      <div>
        <img src={img} />
        <div>
          <p>Code your future!</p>
        </div>
      </div>
    </div>
  );
}
```

Extracting components

Let's extract a few components from App component.

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps.

```
function Welcome(props) {  
  return <h3>Hello {props.name}</h3>  
}  
  
function AssemblerLogo(props) {  
  return <img src={props.img} />  
}  
  
function AssemblerText(props) {  
  return <p>Code your future!</p>  
}
```

Rendering components

Now we can render the components in our App component.

```
function App() {  
  return (  
    <div className="App">  
      <Welcome name="developer" />  
      <div>  
        <AssemblerLogo img={img} />  
        <div>  
          <AssemblerText />  
        </div>  
      </div>  
    </div>  
  );  
}
```

State

Like props, state holds information about the component. However, the kind of information and how it is handled is different.

Stateless component

By default, a component has no state. The Welcome component from the example is stateless:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello {this.props.name}</h1>;  
  }  
}
```


So when would you use state?

When a component needs to keep track of information between renderings the component itself can create, update, and use state.

We'll be working with a fairly simple component to see **state** working in action.

Modify status

Changing the state should only be done from inside a component through the method **this.setState()**.

React exposes setState on the component instance to update it so that react finds out data change and can re-render the view.

State example

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      title: "hi",
      counter: 2,
    };
    this.increment = this.increment.bind(this);
  }

  increment() {
    this.setState({ counter: this.state.counter + 1 });
  }

  render() {
    return (
      <div>
        <h2>{this.state.counter}</h2>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Lifecycle Methods

React components have several lifecycle methods which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting

Mounting means **putting elements into the DOM**. React has several built-in methods that gets called, in this order, when mounting a component, but these are the common ones:

- constructor()
- render()
- componentDidMount()

The **render()** method is required and will always be called, the others are optional and will be called if you define them.

Constructor

The **constructor()** method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

```
class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { favoriteColor: "red" };  
  }  
  render() {  
    return (  
      <h1>My Favorite Color is {this.state.favoriteColor}</h1>  
    );  
  }  
}
```

render

The **render()** method is required, and is the method that actually outputs the HTML to the DOM.

```
class Example extends React.Component {  
  render() {  
    return <h1>hello-mundo</h1>;  
  }  
}
```

componentDidMount

The **componentDidMount()** method is called after the component is rendered. This is where you run statements that requires that the component is already placed in the DOM.

Example

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoriteColor: "red" };
  }

  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoriteColor: "yellow" })
    }, 1000)
  }

  render() {
    return (
      <h1>My Favorite Color is {this.state.favoriteColor}</h1>
    );
  }
}
```

Updating

The next phase in the lifecycle is when a **component is updated**.

A component is updated whenever there is a **change in the component's state or props**.

React calls the following methods, in this order, when a component is updated:

- `render()`
- `componentDidUpdate()`

The **`render()`** method is required and will always be called, the others are optional and will be called if you define them.

Example

We can see it in the following example:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoriteColor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoriteColor: "yellow" });
    }, 1000);
  }
  componentDidUpdate() {
    console.log("The updated favorite is " + this.state.favoriteColor);
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoriteColor}</h1>
        <div id="mydiv"></div>
      </div>
    );
  }
}
```

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- **`componentWillUnmount()`**

Example

We have the
following component:

```
class Parent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { show: true };  
    this.handleRemove = this.handleRemove.bind(this);  
  }  
  handleRemove() {  
    this.setState((prevState) => ({  
      ...prevState,  
      show: !prevState.show,  
    }));  
  }  
  render() {  
    return (  
      <div>  
        <button onClick={this.handleRemove}>Delete Example</button>  
        {this.state.show ? <Child /> : null}  
      </div>  
    );  
  }  
}
```

componentWillUnmount()

This method is called exactly moments before the component is removed from the render.

```
class Child extends React.Component {  
  componentWillUnmount() {  
    console.log("Example component is about to be unmounted.");  
  }  
  render() {  
    return <h1>Hello Assembler!</h1>;  
  }  
}
```

Questions?