Nathan Sebhastian

DIGESTING REACT

Learn how to build web apps with React

Digesting React

Learn how to build web applications with React

Nathan Sebhastian

Foreword

The year 2018 has been a great year for React. It wins the hearts of frontend developers and it also wins the job marketplace[1].

And React team itself doesn't seem to content with its position, with the new hooks feature being released as early as February 6th 2019. React is a revolutionary and very solid library for developing scalable and maintainable frontend, and it has proven track record from popular tech companies like AirBnB[2] and Netflix[3].

(I don't need to mention Facebook[4] now, do I?)

So yes, React is definitely at the top of the requirement list for many *nerd* jobs. And although learning React iself isn't that hard, the way to master React is complex and requires you to have a deep familiarity with its features.

For example, the hooks feature has been a great addition to React that opens up new possibilites of writing the code, but I think it also has made developers who want to learn React gets confused. Before hooks, using a JavaScript class is the only way to write a component that has business

[1] June 2018 Hacker News Hiring Trends
[2] Rearchitecting Airbnb's Frontend
[3] Netflix Likes React
[4] Does Facebook use React on Facebook.com?

logic applied to it. Now with hooks, you can use both JavaScript class and function to write your component. Which one should you use?

Outside of React itself, the React ecosystem is filled with many libraries that strive to achieve the same goals, and they are all equally good. For example, you can use Redux to manage complex React state. But should you use Redux? What about the Context API? Wait, there's also Overmind and Recoil library. Which one should you use?

If you don't know what I'm talking about, then this book will be perfect for you. I will help you to learn both React and how to start navigating its huge ecosystem of tools and libraries in a pace that you can absorb.

To experience the full benefit of this guide, you need to have the following requirements:

- Know basic website technology like HTML and CSS
- You're familiar with basic JavaScript knowledge (variables, functions, conditionals)
- Familiar with code editors like VSCode and SublimeText

Finally, welcome to *Digesting React* where you will learn how to build web applications with React **without getting exhausted**.

Introduction to React

React is a very popular JavaScript front-end library that has received lots of love from developers around the world for its **simplicity** and **fast performance**.

React was initially developed by Facebook as a solution to front end problems they are facing:

- DOM manipulation is an expensive operation and should be minimized
- No library specialized in handling front-end library at the time (there is Angular, but it's an ENTIRE framework.)
- Using a lot of jQuery is causing spaghetti code

Why developers love React? As a software developer myself, I can think of a few reasons why I love it:

It's minimalist

React takes care of only ONE thing: the user interface and how it changes according to the data you feed into it. You can think of React as the "V" in an MVC framework.

It has small learning curve

The core concepts of React are easy to learn, and you don't need months or 40 hours of video lectures to learn about them.

It's unopinionated

React can be integrated with lots of different technologies. On the front-end, you can use different libraries to handle Ajax calls (Axios, Superagent, or just plain old Fetch.) On the back-end, You can use PHP/Rails/Go/Python or whatever language you prefer.

Strong community support

To enhance React's capabilities, open source contributors have build an amazing ecosystem of libraries that enables us to make even more powerful application. But most open source libraries for React is optional. You don't need to learn them until after you master React fundamentals.

The bottom line is that with a small learning curve, React gives you incredible power in making your UI **flexible**, **reusable** and **spaghetti-free**.

Setting up your local computer environment

This section will help you install all the necessary tools to start writing React application in your computer. While you can use browser-based code editor like Code Sandbox and Codepen, I'd still recommend you to simply install things into your computer so that you won't be interrupted by bad internet connection or any downtime.

You will most likely use a local computer when working on a real project anyway, so let's do that now to make yourself comfortable. You can skip this section if you already have these tools installed:

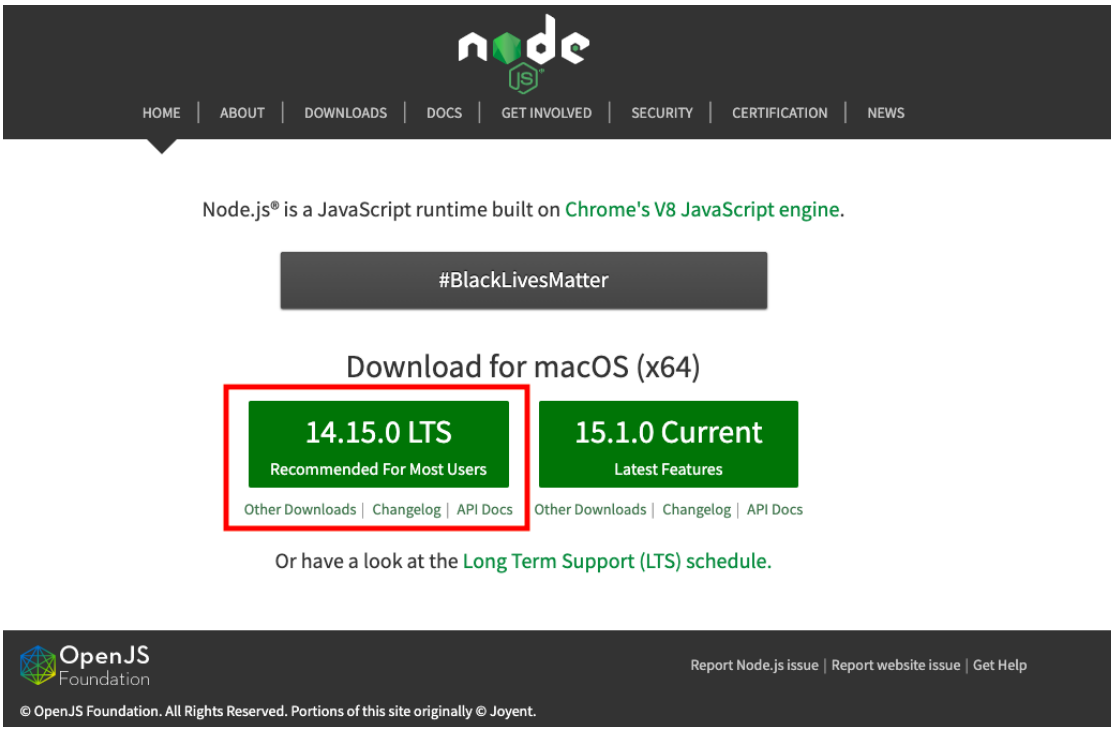
- Code editor (I recommend VSCode)
- Node.js
- Create React App
- React Developer Tools

Installing VSCode

I assume you already know the benefits of using a code editor over a regular one, so I'm going to recommend you to install VSCode if you haven't. VSCode is a free code editor that's really popular with developers today. Head over to VSCode website and download the version for your system.

Installing Node.js

To be able to create React projects on your computer, you need to install Node.js. Head over to its website at nodejs.org and download the most recent LTS version for your computer:


HOME
ABOUT
DOWNLOADS
DOCS
GET INVOLVED
SECURITY
CERTIFICATION
NEWS
#BlackLivesMatter
14.15.0 LTS
Recommended For Most Users
Other Downloads | Changelog | API Docs
15.1.0 Current
Latest Features
Other Downloads | Changelog | API Docs
OpenJS Foundation
Report Node.js issue | Report website issue | Get Help

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for macOS (x64)

Or have a look at the Long Term Support (LTS) schedule.

© OpenJS Foundation. All Rights Reserved. Portions of this site originally © Joyent.

Figure 1: Node.js website

Once installed, please open your command line and type in the following command:

```
node --version && npm --version
```

You'll see the versions of Node and NPM that has been installed into your computer. Now you're ready to create React applications.

Installing Create React App

In order to make React run properly without any errors, it needs a lot of configurations. Yes, you need to configure Babel and Webpack at minimum so that React can run when you hit the browser. Yet configurations should not stand in the way of getting started, and that's why Facebook created a utility tool called Create React App. It saves you from having to learn about configurations, and how to setup those configurations properly for React.

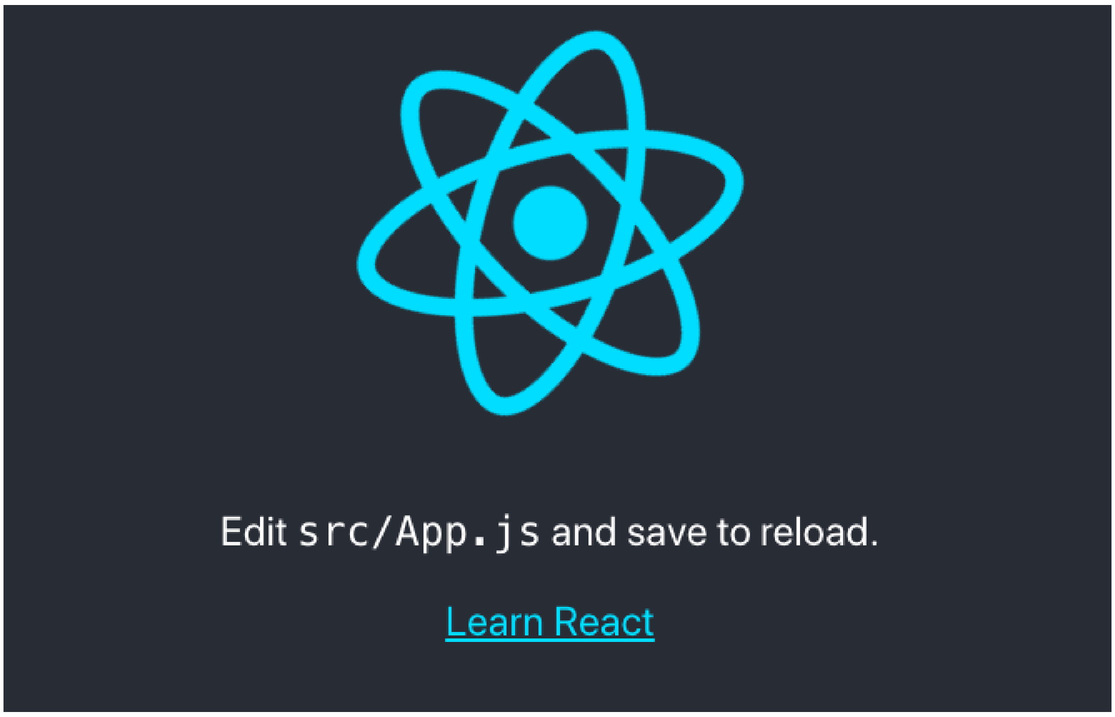
The only requirement to use this tool is that you have Node and NPM installed, so let's get one in your local computer by opening the Terminal and run the command:

```
npx create-react-app my-first-react-app
```

Once it's finished, navigate to the created directory and run the app using these commands:

```
cd my-first-react-app && npm start
```

A browser window will open and you'll be greeted with the index page of CRA apps:


Edit src/App.js and save to reload.

Learn React

Figure 2: The index page of CRA apps

There's a lot going on under the hood of this app, but the most important thing for you to know is that the entire React app code can be found in `src/` directory, and it's hooked into the `public/index.html` file.

If you want to learn about the configuration that's been setup by Create React App, you can read about my article Step by step React configuration from scratch. That's exactly what Create React App has saved you from.

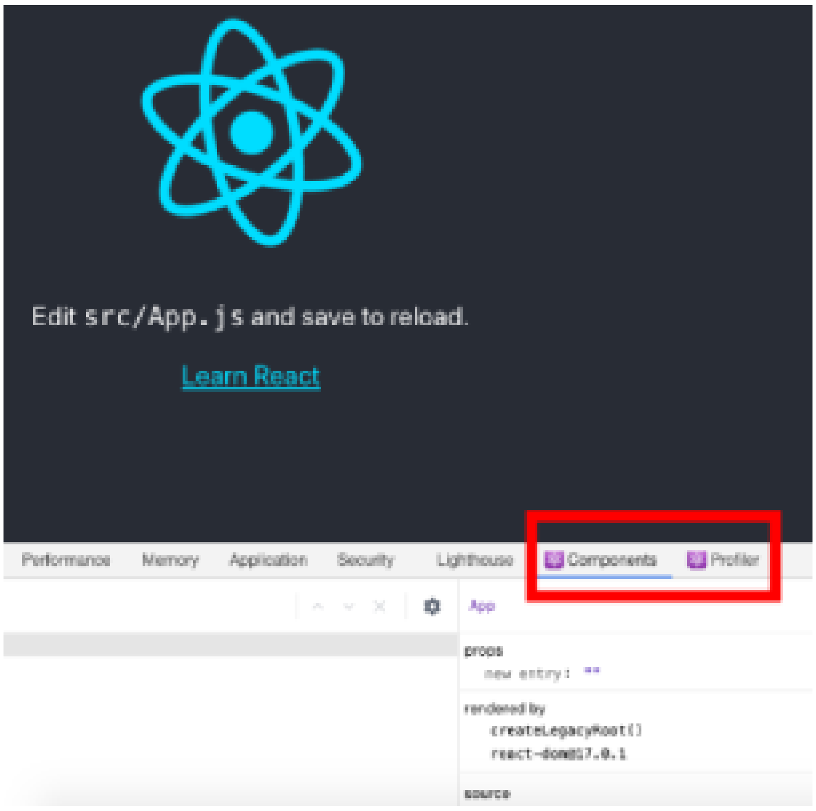
Installing React Developer Tools

React has a Developer Tool that allows you to inspect your React code at runtime from the browser. This is very useful to see changes in your React code as you'll see in the following chapters. To use it, you only need to download the right package for your environment:

- Chrome extension
- Firefox extension
- Standalone app (Safari, React Native, etc)

Opera users can enable Chrome extensions and then install the Chrome extension.

I recommend you use Chrome or Firefox extension, since they are easier to work with compared to the standalone version. To bring up the Developer Tool, simply open your browser developer tool and a new 'Components' and 'Profiler' tabs will appear on sites using React:



Edit src/App.js and save to reload.

Learn React

```
props
  new entry: ""
rendered by
  createLegacyRoot()
  react-dom@17.0.1
source
```

Figure 3: React dev tools in action

Similar to how you can inspect CSS in a regular HTML elements, you can inspect the state and props of React component using the developer tool. I will explain how you can use it later. Let's start learning React core concepts for now.

Part 1: React Core Concepts

This part will show you the building blocks of React applications. It will help you learn the idea of a **component** and how it can be used to split your application UI into small, manageable pieces.

You will also learn how your components know what to display on the screen based on the data you feed into it in the form of **props** and **state**.

To build a fully functional web application, you'll need more than one component, so you're going to learn how to **assemble multiple components** into a single application.

Next, you will learn about the **lifecycle** of a React class component, and how you can execute your code based on what lifecycle status your component is in.

Finally, you will learn how to **handle events** such as a button click with React. You will end part 1 with building a simple application that demonstrates how all these theories came together.

This part will focus on building components using JavaScript `class` syntax. You will learn about the `function` syntax in part two.

Introducing React components

A component in React is a single independent unit of a user interface. What you write inside a component will determine what should appear on the browser screen at a given time. Each component in React must return at least one UI element in order to run properly.

You can create a React component using both JavaScript `class` and `function` syntax.

Here's how you declare a class component in React:

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return <h1> Hello World </h1>
  }
}
```

A class component must always contain a `render()` function with a `return` statement or it will throw an error.

And here's how you declare a function component:

```
import React from 'react';

function MainComponent(){
  return <h1> Hello World </h1>
}
```

Just like class component, a function component must always have a `return` statement, but you don't need to create a `render()` function with it.

When you want a component to render nothing, you can return a `null` or `false` instead of an element.

```
import React from 'react';

function MainComponent(){
  return null
}

//or

class MainComponent extends React.Component {
  render(){
    return false
  }
}
```

Since React is a JavaScript library, all React component code goes under the `.js` file extension. You might find some public code use `.jsx` extension, but most compilers treat them as the same, so it's better to use `.js` extension because it's the universal JavaScript code format.

Returning multiple elements

A component must always return a single element. When you need to return multiple elements, you need to wrap all of it in a single element like a `<div>` :

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return (
      <div>
        <h1>Hello World!</h1>
        <h2>Learning to code with React</h2>
      </div>
    )
  }
}
```

But this will make your application renders extra `<div>` nodes into the browser. To avoid cluttering your application, you can render an empty tag:

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return (
      <>
        <h1>Hello World!</h1>
        <h2>Learning to code with React</h2>
      </>
    )
  }
}
```

With this, you won't render any extra nodes. You can do the same with React function components.

Rendering to the screen

A React component needs to be rendered into the screen by using the `ReactDOM.render()` function which takes two arguments: the component you want to render and the containing HTML element to render the component into. This means that you need to have an HTML file in your React project, or React won't know where to render the component.Usually, a very basic HTML document with a div is enough:

```
<div id="root"></div>
```

Next, you render the component into the `div` . Notice how ReactDOM is imported from `react-dom` package in the example below:

```
import React from "react";
import ReactDOM from "react-dom";

class MainComponent extends React.Component {
    render(){
    return <h1>Hello World</h1>
  }
}

ReactDOM.render(
  <MainComponent />,
  document.querySelector("#root")
)
```

The first argument is the `<MainComponent>` and the second argument is the `<div>` element with id "root".

Here's a render example in Code Sandbox that you can tweak around with.

Writing comments in React components

Writing comments in React components can be done just like how you comment in regular JavaScript classes and functions. You can use the double forward-slash syntax `//` to comment any code:

```
class MainComponent extends React.Component {
    render(){
    // const name = "John"
    // const age = 28
    return <h1>Hello World</h1>
  }
}
```

Or you can use the forward-slash and asterisk format `/* */` like this:

```
class MainComponent extends React.Component {
    render(){
    /*
    const name = "John"
    const age = 28
    */
    return <h1>Hello World</h1>
  }
}
```

Generally, the forward-slash and asterisk format for comments is used for writing real comments like license and other documentation for developers to read instead of commenting out code:

```
/** @license React v16.13.1
 * react.development.js
 *
 * Copyright (c) Facebook, Inc. and its affiliates.
 *
 * This source code is licensed under the MIT license found in the
 * LICENSE file in the root directory of this source tree.
 */
```

But there is no strict rule that says you can't use it for commenting out code too.

Your first exercise

It's time to create your first React application. Open a terminal in your computer and create a new React application with Create React App. Let's name this app `my-first-react-app` :

```
npx create-react-app my-first-react-app
```

Move into the folder and run the application with `npm start` :

```
cd my-first-react-app && npm start
```

Once the application is loaded into the browser, open your application with a code editor. Inside, you will see both `src/` and `public/` folders. The `src/` folder is where the code responsible for rendering your application. Open the file `App.js` and you'll see the component that gets rendered by `index.js` .

Your assignment is very simple:

1. Change the text `Edit <code>src/App.js</code> and save to reload` into `Hello World from React!`
2. Remove the anchor element "Learn React" from the screen

You might see some unfamiliar syntax inside the `return` statement. Don't worry about it, I will explain in the next chapter.

Once you make the changes, save the file and return to the browser. The script from Create React App will automatically refresh the browser and display the changes for you. Great job!

Next, you're going to learn about JSX, the template language of React.

JSX: The template language of React

In the previous chapter, you've learned that a component must always have a `return` statement that contains elements to render on the screen:

```
class MainComponent extends React.Component {
  render(){
    return <h1> Hello World </h1>
  }
}
```

The tag `<h1>` looks like a regular HTML tag, but it's actually a special template language included in React called JSX.

JSX is an extension of JavaScript that produces JavaScript powered React elements. It can be assigned to JavaScript variable and can be returned from function calls. For example:

```
class MainComponent extends React.Component {
  render(){
    const element = <h1> Hello World! </h1>
    return element
  }
}
```

The example above is a valid JSX code.

Adding class attribute

Just like ordinary HTML, JSX can use `class` attribute with the `className` keyword (because the keyword `class` is reserved by JavaScript.)

```
const App = <h1 className='text-lowercase'>Hello World!</h1>;
```

Writing comments inside React JSX

Commenting inside React JSX syntax is a bit confusing because while JSX gets rendered just like normal HTML by the browser, JSX is actually an enhanced JavaScript that allows you to write HTML in React.

You can't write comments as you might do in HTML and XML with `<!-- -->` syntax. The following example will throw `Unexpected token` error:

```
export default function App() {
  return (
    <div>
      <h1>Hello World~ </h1>
      <!-- <p>My name is Bob</p> -->
      <p>Nice to meet you!</p>
    </div>
  );
}
```

To write comments in JSX, you need to use JavaScript's forward-slash and asterisk syntax, enclosed inside a curly brace `{/* comment */}` .

Here’s an example:

```
export default function App() {
  return (
    <div>
      <h1>Commenting in React and JSX~ </h1>
      {/* <p>My name is Bob</p> */}
      <p>Nice to meet you!</p>
    </div>
  );
}
```

And here’s for multiple lines of JSX:

```
export default function App() {
  return (
    <div>
      {/* <h1>Commenting in React and JSX~ </h1>
      <p>My name is Bob</p>
      <p>Nice to meet you!</p> */}
    </div>
  );
}
```

It may seem very annoying that you need to remember two different ways of commenting when writing React code. But don’t worry!

Most modern IDEs like VSCode and CodeSandbox already know about this issue. They will write the right comment syntax for you automatically when you press on the comment shortcut `CTRL+/` or `command+/` for macOS.

Using JavaScript inside JSX

You can embed JavaScript expression inside the element by using curly brackets `{}` :

```
const lowercaseClass = 'text-lowercase';
const text = 'Hello World!';
const App = <h1 className={lowercaseClass}>{text}</h1>;
```

This is what makes React element distinct from HTML element. You can't embed JavaScript directly by using curly braces in HTML.

Instead of creating a whole new templating language, you just need to use JavaScript functions to control what is being displayed on the screen.

For example, let's say you have an array of users that you'd like to show:

```
const users = [
  { id: 1, name: 'Nathan', role: 'Web Developer' },
  { id: 2, name: 'John', role: 'Web Designer' },
  { id: 3, name: 'Jane', role: 'Team Leader' },
]
```

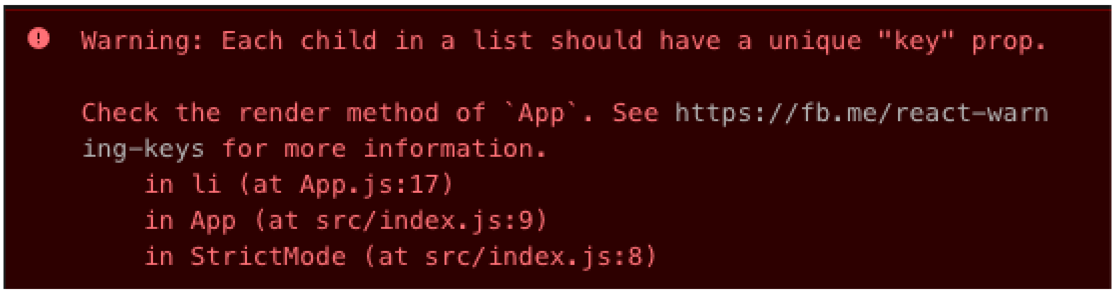
You can use the `map()` function to loop over the array:

```
import React from "react"

function App() {
  const users = [
    { id: 1, name: 'Nathan', role: 'Web Developer' },
    { id: 2, name: 'John', role: 'Web Designer' },
    { id: 3, name: 'Jane', role: 'Team Leader' },
  ]
  return (
    <>
      <p>The currently active users list:</p>
      <ul>
      {
        users.map(function(user){
          // returns Nathan, then John, then Jane
          return (
            <li> {user.name} as the {user.role} </li>
          )
        })
      }
      </ul>
    </>
  )
}
```

Inside React, you don't need to store the return value of the `map` function in a variable. The example above will return a list element for each array value into the component.

While the above code is already complete, React will trigger a warning that you need to put "key" prop in each child of a list (the element that you return from `map` function):



```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`. See https://fb.me/react-warn
ing-keys for more information.
    in li (at App.js:17)
    in App (at src/index.js:9)
    in StrictMode (at src/index.js:8)
```

Figure 4: React needs a key inside each child

The "key" prop is a special prop that React will use to determine which child element have been changed, added, or removed from the list. you won't use it actively in any part of your array rendering code, but since React needs the props, then let's give it.

It is recommended that you put the unique identifier of your data as the key value. In the example above, you can put the `id` of each user as the key of each `<li>` element:

```
return (
  <li key={user.id}>
    {user.name} as the {user.role}
  </li>
)
```

When you don't have any unique identifiers for your list, you may use the array index as the last resort:

```
{
  users.map(function(user, index){
    return (
      <li key={index}>
        {user.name} as the {user.role}
      </li>
    )
  })
}
```

Now that you know how flexible and powerful JSX is, let's learn about composing components next.

Assembling multiple components as one

Up until this point, you've only rendered a single component into the browser. But applications build using React can comprise of tens and hundreds of components. **Composing or assembling components** is the process of forming the user interface by using loosely coupled components in a top-down approach.

It's kind of like making a robot out of lego blocks, as I will show you in the following demo:

```
class ParentComponent extends React.Component {
  render(){
    return (
      <>
        <UserComponent />
        <ProfileComponent />
        <FeedComponent />
      </>
    )
  }
}

class UserComponent extends React.Component {
  render(){
    return <h1> User Component </h1>
  }
}

class ProfileComponent extends React.Component {
  render(){
    return <h1> Profile Component </h1>
  }
}

class FeedComponent extends React.Component {
  render(){
    return <h1> Feed Component</h1>
  }
}
```

From the example above, you can see how the `<ParentComponent>` renders three children components:

- `<UserComponent>`
- `<ProfileComponent>`
- `<FeedComponent>`

Both class components and function components can compose components this way. You can even render a class component inside a function component and vice versa.

The composition of many components will form a single tree of React components, and a single component will be rendered into the DOM using `ReactDOM.render()` :

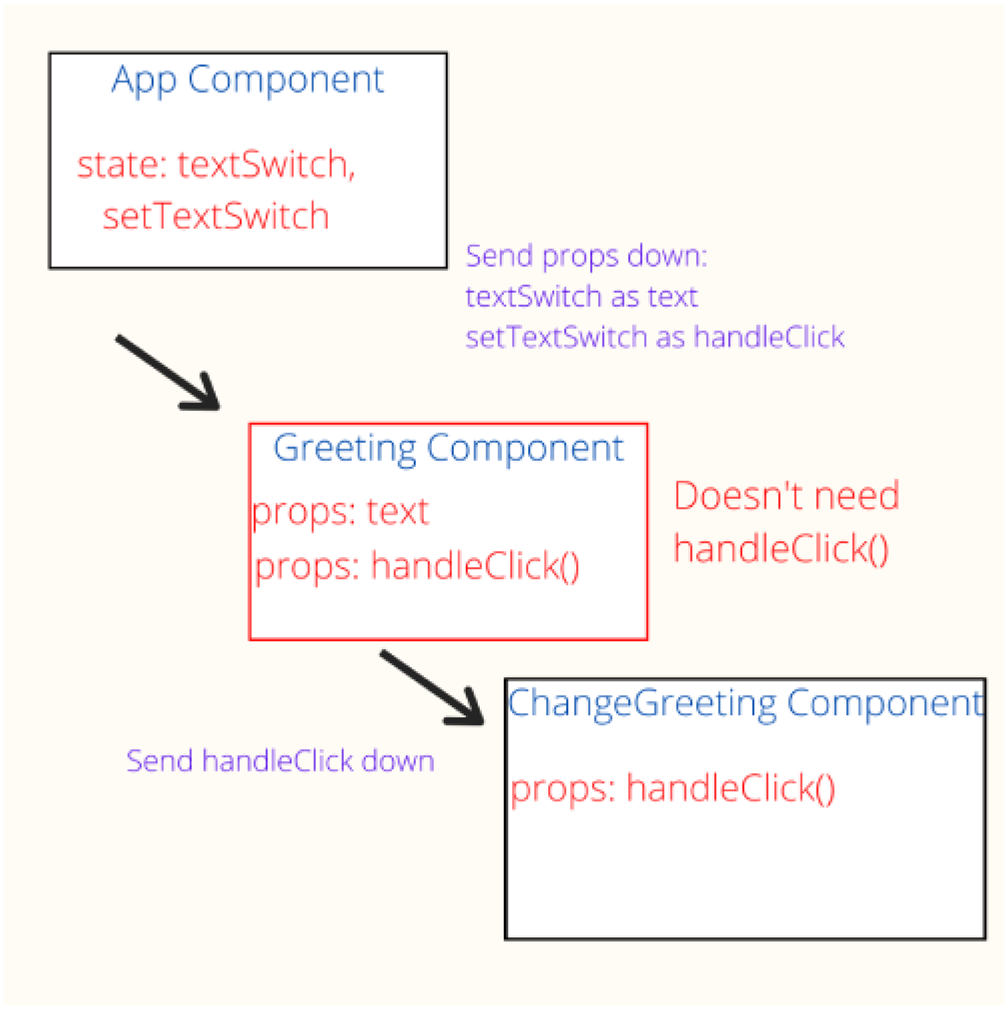


Figure 5: React tree of components

By composing multiple components, you can split the user interface into independent, reusable pieces, and develop each piece in isolation.

Custom functions and this keyword

You can create custom functions to run inside a component by declaring the function name without the `function` keyword. Here's an example:

```
import React from 'react';

class MainComponent extends React.Component {

  formatName(firstName, lastName) {
    return firstName + ' ' + lastName;
  }

  render(){
    const firstName = "Bruce"
    const lastName = "The Batman"
    return (
      <>
        <h1>Hello World!</h1>
        <h2>I'm {this.formatName(firstName, lastName)}</h2>
      </>
    )
  }
}
```

In the example above, the function `formatName()` is a custom function that gets called by the `render()` function using the `this.formatName()` syntax. The `this` keyword refers to the JavaScript class `MainComponent` which owns the function `formatName()` .

In the following chapters, you will find that React class components will use `this` keyword to access properties owned by the class.

State and props

In React library, props and states are both means to make a component more dynamic. Props (or properties) are inputs passed down from a parent component to its child component. On the other hand, states are variables defined and managed by the component.

For example, let's say we have a that calls a :

```
class ParentComponent extends React.Component {
  return <ChildComponent />
}
```

You can pass a prop from ParentComponent into ChildComponent by adding new attributes after the component name. For example, the `name` prop with value `John` is passed to ChildComponent below:

```
class ParentComponent extends React.Component {
  return <ChildComponent name="John" />
}
```

After that, the child component will accept assign the props into its own `this.props` object. What to do with the prop being passed down to the child is of no concern to the parent component. You can simply output the `name` prop like this:

```
class ChildComponent extends React.Component {
  render(){
    return <p> Hello World! my name is {this.props.name}</p>
  }
}
```

Passing down multiple props

You can pass as many props as you want into a single child component, just write the props next to the previous. Here's an example:

```
class ParentComponent extends React.Component {
  render(){
    return (
      <ChildComponent
        name="John"
        Age={29}
        isMale={true}
        hobbies={["read books", "drink coffee"]}
        occupation="Software Engineer"
      />
    );
  }
}
```

It will all be passed accordingly into the ChildComponent's `this.props` object.

Also please remember: you need to **pass either a string or a JavaScript expression** inside curly brackets as the value of props. This is because your component call is in JSX syntax, and you need to use curly brackets to write an expression.

Props are immutable

Meaning that a prop's value can't be changed no matter what happens. The following example will still output the value passed into it instead of "Mark":

```
class ChildComponent extends React.Component {
  render(){
    this.props.name = "Mark";
    return <p> Hello World! my name is {this.props.name}</p>
  }
}
```

But what if you need variables that might change later? This is where state comes in. States are arbitrary data that you can define, but they are initialized and managed by a component. Here's how you define a class component state:

```
class ParentComponent extends React.Component {
  state = {
    name: "John"
  }
}
```

The state initialized in the class can be accessed from `this.state` object.

Please note that you mustn't change the state value by reassigning the variable. The following code is considered wrong even though it works:

```
class ParentComponent extends React.Component {
  state = {
    name: "John"
  }

  render() {
    this.state.name = "Mark"
    return <div>{this.state.name}</div>;
  }
}
```

You need to use the `setState` function which is inherited from React's `Component` class. You need to pass in an object just like when you declare the initial state:

```
class ParentComponent extends React.Component {
  state = {
    name: ""
  };

  render() {
    if (this.state.name === "") {
      this.setState({ name: "Mark" });
    }
    return <div>{this.state.name}</div>;
  }
}
```

Also, **never** call on the `setState` without a condition since it will cause an infinite loop. The following code will throw an error:

```
class ParentComponent extends React.Component {
  state = {
    name: ""
  };
  render() {
    this.setState({ name: "Mark" });
    return <div>{this.state.name}</div>;
  }
}
```

Binding a custom function

You can pass state into any children component, but if you want to update the state from a child component, you need to create a custom function in the parent component and pass it down to the child component:

```
class ParentComponent extends React.Component {
  state = {
    name: "John"
  };

  setName(name) {
    this.setState({ name: name });
  }

  render() {
    return (
      <ChildComponent
        name={this.state.name}
        setName={this.setName.bind(this)}
      />
    );
  }
}
```

The `bind` keyword used inside the `setName` props is needed because when you pass the function to other components, the function will be called in a different context (in `ChildComponent` instead of `ParentComponent`)

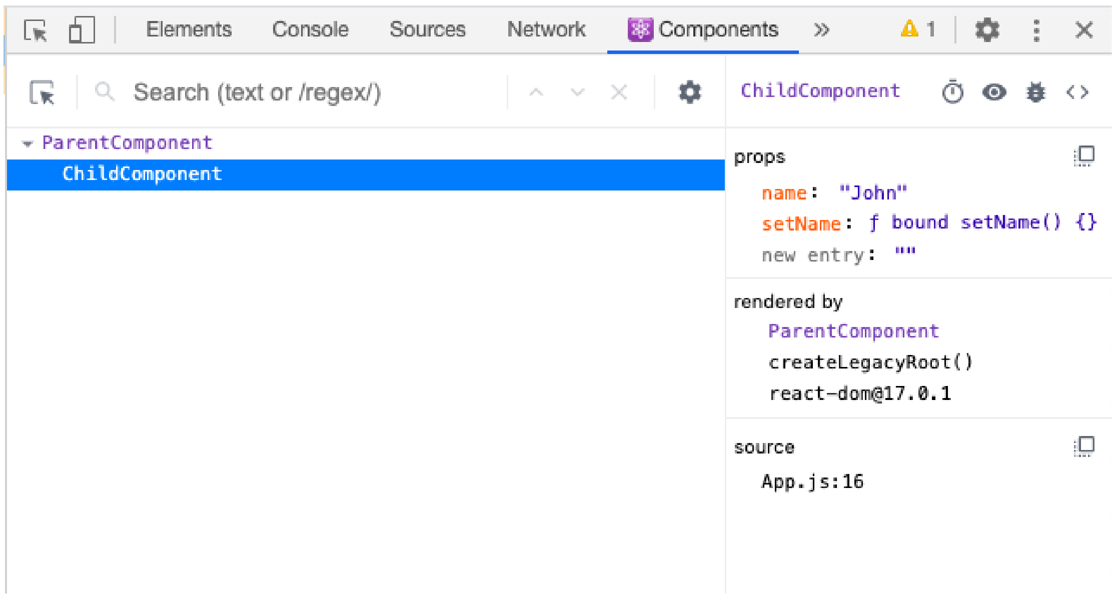
When a child component needs the state to change, you can do so by calling on the `setName` function. In the following example, I put a button to change the value of `name` state when it gets clicked:

```
class ChildComponent extends React.Component {
  render() {
    return (
      <>
        <p> Hello World! my name is {this.props.name}</p>
        <button
          onClick={() => this.props.setName("Mark")}>
            Change name
        </button>
      </>
    );
  }
}
```

Notice how the `<button>` above has an `onClick` event attribute. You will explore that in the next chapter.

Using React DevTools to inspect state and props

To help ease your development, you can use React DevTools that you've installed previously to inspect the current state and props value of your components. Head over into the *components* tab and click on one of the components. Here's an example of `ChildComponent` detail in the inspector:



```
ParentComponent
  ChildComponent
```

```
ChildComponent

props
  name: "John"
  setName: ƒ bound setName() {}
  new entry: ""

rendered by
  ParentComponent
  createLegacyRoot()
  react-dom@17.0.1

source
  App.js:16
```

Figure 6: DevTools component inspector

When you click on the button, the `name` prop value will change accordingly. You can inspect the `ParentComponent` to view the value of state in it. This will be useful as you code your application in the exercise projects later.

Conclusion

You've just learned the difference between props and state in React. Both features are simply arbitrary variables that you can use to make your React components more dynamic. In most cases, states are initialized at top-level components and passed down as props into children components.

When you need to change the prop value, the child component can send a signal — usually function call when a button is clicked or something similar — to the parent component to change the state. This will make the props value being passed down from the parent component to child component changes.

The components will then render the user interface accordingly.

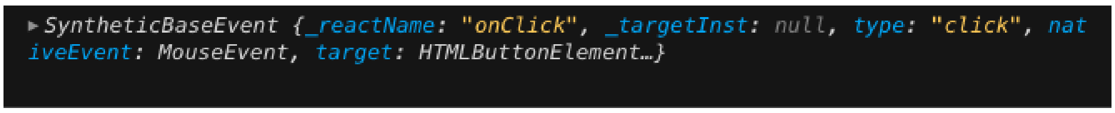
React event handlers

React has an internal event system that gets triggered every time a certain action is taken by the user. For example, an event can be triggered when you click on a button with the `onClick` prop:

```
class LogButton extends React.Component {
  handleClick = (event) => {
    console.log("Hello World!");
    console.log(event);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

When you click on the button above, the `event` variable will be logged as a `SyntheticBaseEvent` object in your console:



```
▸SyntheticBaseEvent {_reactName: "onClick", _targetInst: null, type: "click", nat
iveEvent: MouseEvent, target: HTMLButtonElement…}
```

Figure 7: React's SyntheticBaseEvent log

The `event` argument passed into `handleClick` is React's own Synthetic event. It will always get send into your event handler function.

React's Synthetic events are basically wrappers around the native DOM events. They are helper functions created to make sure the events have consistent properties across different browsers.

These specific events are baked into React library to help you in creating the proper responses to a user's actions. General use cases where you need to make use of these event handlers include listening to user inputs and storing form input data in React's state.

Storing form input with onChange event handler

The `onChange` event handler is a prop that you can pass into JSX's input elements. In React, `onChange` is used to handle user input in real-time.

If you want to build a form in React, you need to use this event to track the value of input elements.

Here's how you add the `onChange` handler to an input:

```
import React from "react"

class App extends React.Component {
  render(){
    return (
      <input
```

```
        type="text"
        name="firstName"
        onChange={ (event) => console.log("onchange is triggered") } />
    )
  }
}
```

Now whenever you type something into the input box, React will trigger the function that we passed into the `onChange` prop.

In regular HTML, form elements such as `<input>` and `<textarea>` usually maintain their own value:

```
<input id="name" type="text">
```

Which you can retrieve by using the `document` selector:

```
var name = document.getElementById("name").value;
```

In React however, it is encouraged for developers to store input values in the component's state object. This way, React component that renders the form elements will also control what happens on subsequent user inputs. First, you create a state for the input:

```
import React from "react"

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

Then, you create an input element and call the `setState` function to update the `name` state. Every time the `onChange` event is triggered, React will pass the `event` argument into the function that you define inside the prop:

```
import React from "react"

class App extends React.Component {
  state = {
    name: ''
  }

  render(){
    return (
      <input
        type="text"
        name="firstName"
        onChange={ (event) => this.setState({ name: event.target.value}) }  />
    )
  }
}
```

Finally, you use the value of `name` state and put it inside the input's `value` prop:

```
return (
  <input
    type="text"
    name="firstName"
    onChange={
      (event) => this.setState({ name: event.target.value})
    }
    value={this.state.name} />
)
```

You can retrieve input value in `event.target.value` and input name in `event.target.name` . You can also separate the `onChange` handler into its own function.

The `event` object is commonly shortened as `e`

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

class App extends React.Component {
  state = {
    name: ''
  }

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

  render(){
    return (
      <input
        type="text"
        name="firstName"
        onChange={ this.handleChange.bind(this) }
        value={this.state.name} />
    )
  }
}
```

The `bind` keyword is used when you call on the `handleChange` function so that you can call on `this.setState` inside it (`this` will still refers to `App` class instead of `handleChange` function)

This pattern of using React's onChange event and the component state will encourage developers to use state as the "single source of truth". Instead of using Regular JavaScript to retrieve input values, you retrieve them from the state.

Preventing default event behavior

Since Synthetic events are just wrappers, the internal default behavior of the DOM object will still be triggered. One problem with the native DOM events is that it sometimes triggers a behavior that you don't need.

For example, a form's submit button in React will always trigger a browser refresh to submit the data into a backend system. This is bad because the behavior you defined in the `onSubmit` event function will be ignored by the browser. Try the following example:

```
import React from "react";

class App extends React.Component {
  state = {
    name: ''
  }

  handleSubmit(event) {
    console.log(this.state.name);
  };

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

Because of the default DOM event behavior, the `handleSubmit` function will be ignored and your log will not get written on the console.

This may be good in the past where the entire form validation and processing happens in the backend, but modern web applications tend to run the form validation process on the client-side in order to save time and bandwidth. To do so, you need to run your own defined behavior.

To cancel the native behavior of the submit button, you need to use React's `event.preventDefault()` function:

```
handleSubmit(event) {
  event.preventDefault();
  console.log(name);
};
```

And that's all you need. Now the default event behavior will be canceled, and any code you write inside handleSubmit will be run by the browser.

You can also write the `preventDefault()` function on the last line of your function:

```
handleSubmit(event) {
  console.log(name);
  console.log("Thanks for submitting the form!");
  event.preventDefault();
};
```

But for collaboration and debugging purposes, it’s always better to write the prevent function **just below your function declaration**. That way you won’t cause a bug by forgetting to put the prevent function too.

React component's lifecycle methods

All React components must have a `render` method, which returns some element that will be inserted into the DOM. Indeed, `ReactDOM.render` is called on a pure HTML element, which in most of our example so far, use the `<div>` tag with id `root` as its entry point.

That's why when we do this:

```
class sampleComponent extends React.Component {
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

ReactDOM.render(
  <sampleComponent />,
  document.getElementById('root')
);
```

The `<h1>` element will be added into the DOM element with id `root` :

```
<div id='root'>
  <h1>Hello World</h1>
</div>
```

Even though you can't see it in the browser, there's a fraction of time before React component `render` or insert this `<h1>` element into the browser and after it, and in that small fraction of time, you can run special functions designed to exploit that time.

This is what lifecycle functions in a React component do: it executes at a certain time *before* or *after* a component is rendered to the browser.

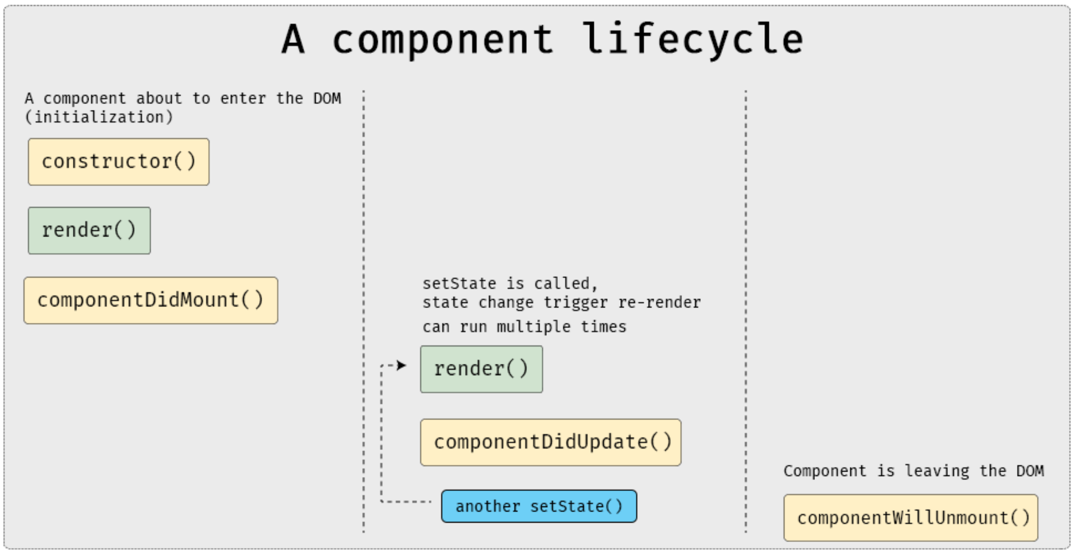


Figure 8: The lifecycle graph

When a component is first inserted into the DOM (or the `root` element), it will run the `constructor` method. At this point, nothing is happening in the browser.

Then React will run the component `render` method, inserting the JSX you write into the DOM. After `render` is finished, it will immediately run the `componentDidMount` function.

When you call on `setState` , the `render` function will be called again after state is changed, with componentDidUpdate function immediately run after it .

`componentWillUnmount` function will run before the component rendered element is removed from the DOM.

The theory might seem complex, but as you will see in the following chapters, lifecycle functions are situational code, and they are used only for specific use cases.

The constructor function

The `constructor` function is run on the initialization of a React component. It is widely used as the place where state is initialized:

```
class sampleComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number : 0
    }
  }
}
```

The function `super` will call on the parent `constructor` (specifically, the `React.Component` `constructor`) so that you can call on `this` :

```
class sampleComponent extends React.Component {
  constructor(props) {
    // this will cause error
    this.state = {
      number : 0
    }
    super(props);
  }
}
```

The `props` are being passed into `super` so that you can call on `this.props` on the constructor. If you're not using `props` in the constructor at all, you can omit it.

You might notice that on the previous chapters, you can also initiate state outside of the constructor:

```
class sampleComponent extends React.Component {
  state = {
    number: 0
  }
}
```

Both are valid state declaration, but the constructor style is widely adopted as the conventional style to class components, so you will find most React code use it.

The bottom line for `constructor` function — initialize your state there.

render function

You have seen this function in previous chapters, so it must be familiar for you. The `render` function is used to write the actual JSX elements, which is returned to React and hooked into the DOM tree.

Before returning JSX, you can write regular JavaScript syntax for operation such as getting state value, and embed it into the JSX:

```
render() {
  const { name, role } = this.state;
  return (
    <div>My name is {name} and I'm a {role}</div>
  )
}
```

The componentDidMount function

The most common use of `componentDidMount` function is to load data from backend services or API. Because `componentDidMount` is called after render is finished, it ensures that whatever component manipulation you do next, like `setState` from fetched data, will actually update state from its initial value.

A data request to backend services might resolve faster than the component is inserted into the DOM, and if it did, you will do a `setState` faster than the `render` method finished. That will cause React to give you a warning. The most common use of `componentDidMount` looks like this:

```
class sampleComponent extends React.Component {

  componentDidMount() {
    this.fetchData().then(response => {
      this.setState({
        data: response.data
      });
    });
  }

  fetchData = () => {
    // do a fetch here and return something
  }
}
```

But `componentDidMount` is limited to running only once in a component lifecycle. To address this limit, let's learn about the next lifecycle function.

The componentDidUpdate function

Since `componentDidMount` is run only once in a component lifetime, it can't be used to fetch data in response to state change. Enter `componentDidUpdate` function. This function is always run in response to changes in the component, remember the diagram again:

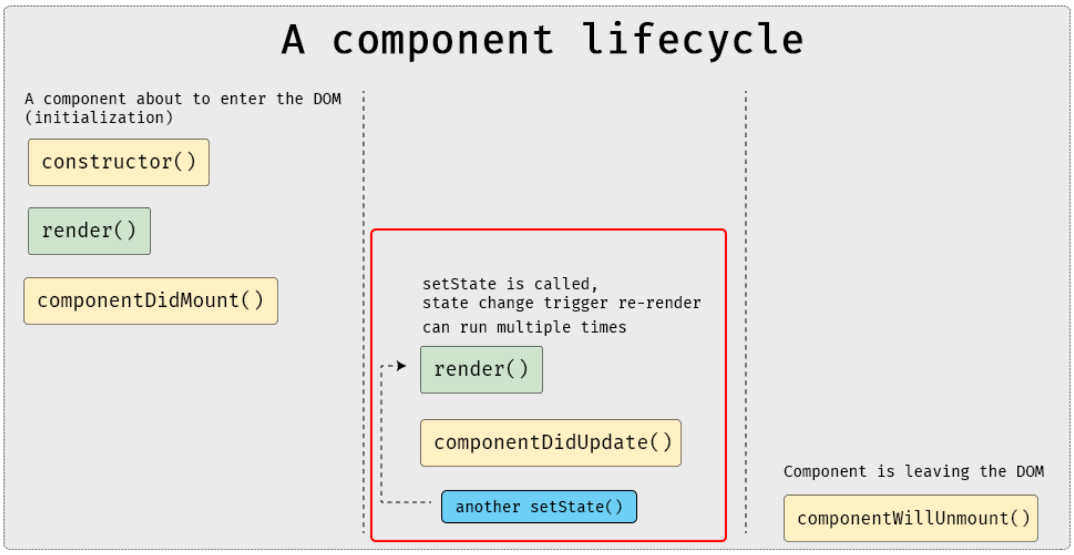

A component lifecycle
A component about to enter the DOM (initialization)
constructor()
render()
componentDidMount()
setState is called, state change trigger re-render can run multiple times
render()
componentDidUpdate()
another setState()
Component is leaving the DOM
componentWillUnmount()

Figure 9: componentDidUpdate graph

An easy example would be to log the new state after a re-render.

```
class SampleDidUpdate extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }

  incrementState = () => {
    const { number } = this.state;
    this.setState({
      number: number + 1
    });
  };

  decrementState = () => {
    const { number } = this.state;
    this.setState({
      number: number - 1
    });
  };

  componentDidMount() {
    const { number } = this.state;
    console.log(`The current number is ${number}`);
  }

  componentDidUpdate() {
    const { number } = this.state;
    console.log(`The current number is ${number}`);
  }

  render() {
    const { number } = this.state;
    return (
      <>
        <div> The current number is {number}</div>
        <button onClick={this.incrementState}>Add number</button>
        <button onClick={this.decrementState}>Subtract number</button>
      </>
    );
  }
}
```

A demo is available here. Notice how `didMount` and `didUpdate` is identical in everything but name. Since user can change the keyword after the component did mount into the DOM, subsequent request won't be run by `componentDidMount` function. Instead, `componentDidUpdate` will "react" in response to the changes after `render` function is finished.

The componentWillUnmount function

The final function `componentWillUnmount` will run when the component is about to be removed from the DOM. This is used to cleanup things that would be left behind by the component.

To try out this function, let's create two child component and one parent component.

```
class ChildComponentOne extends React.Component {
  componentWillUnmount() {
    console.log("Component One will be removed");
  }

  render() {
    return <div>Component One</div>;
  }
}

class ChildComponentTwo extends React.Component {
  componentWillUnmount() {
    console.log("Component Two will be removed");
  }

  render() {
    return <div>Component Two</div>;
  }
}
```

This child components will do a simple `div` render with componentWillUnmount function that logs a text into the console. Then the parent component will render one of them based on the current state it's in.

```
class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }

  switchState = () => {
    const { number } = this.state;
    this.setState({
      number: number === 0 ? 1 : 0
    });
  };

  render() {
    const { number } = this.state;
    let component = number ? <ChildComponentOne /> : <ChildComponentTwo />;
    return (
      <>
        {component}
        <button onClick={this.switchState}>Switch</button>
      </>
    );
  }
}
```

When you click on the Switch button, the component that will be removed from the DOM will log a message, then leave and be replaced with the new component. You can try it here.

When to use it? It's actually very situational, and the best use of `componentWillUnmount` is to shut down some external service listener your component is subscribed into.

Conclusion

React's lifecycle methods are used for running codes that needs to be automatically run when the component is created, added, and removed from the DOM.

The lifecycle methods bring more control over what happens at each specific time during your component lifetime, from its creation to its destruction, allowing you to create dynamic applications in the process.

Writing CSS for React Components

There are four different ways to write CSS for React components. Let's learn what they are and which one you should start with.

Inline styling

React components are composed of JSX elements. But just because you're not writing regular HTML elements doesn't mean you can't use the old inline style method.

The only difference with JSX is that inline styles must be written as an object instead of a string.

Here is a simple example:

```
function App() {
  return (
      <h1 style={{ color: "red" }}>Hello World</h1>
  );
}
```

In the style attribute above, the first set of curly brackets will tell your JSX parser that the content between the brackets is JavaScript (and not a string). The second set of curly bracket will initialize a JavaScript object.

Style property names that have more than one word are written in camelCase instead of using the traditional hyphenated style. For example, the usual `text-align` property must be written as `textAlign` in JSX:

```
function App() {
  return (
      <h1 style={{ textAlign: "center" }}>Hello World</h1>
  );
}
```

Because the style attribute is an object, you can also separate the style by writing it as a constant. This way, you can reuse it on other elements as needed:

```
const pStyle = {
  fontSize: '16px',
  color: 'blue'
}

export default function App() {
  return (
      <p style={pStyle}>The weather is sunny today.</p>
  );
}
```

If you need to extend your paragraph style further down the line, you can use the object spread operator. This will let you add inline styles to your already-declared style object:

```
const pStyle = {
  fontSize: "16px",
  color: "blue"
};

export default function App() {
  return (
    <>
      <p style={pStyle}>
        The weather has a small chance of rain today.
      </p>
      <p style={{ ...pStyle, color: "green", textAlign: "right" }}>
        When you go to work, bring your umbrella with you!
      </p>
    </>
  );
}
```

Inline styles are the most basic example of a CSS in JS styling technique.

One of the benefits in using the inline style approach is that you will have a simple component-focused styling technique. By using an object for styling, you can extend your style by spreading the object. Then you can add more style properties to it if you want.

But in a big and complex project where you have a hundreds of React components to manage, this might not be the best choice for you.

You can't specify pseudo-classes using inline styles. That means `:hover` , `:focus` , `:active` , or `:visited` go out the window rather than the component.

Also, you can't specify media queries for responsive styling. Let's consider another way to style your React app.

CSS stylesheets

When you build a React application using Create React App, you will automatically use webpack to handle asset importing and processing.

The great thing about webpack is that, since it handles your assets, you can also use the JavaScript `import` syntax to import a `.css` file to your JavaScript file. Just create a regular CSS file in your project folder:

```css
/* style.css */
.paragraph-text {
  font-size: 16px;
  color: 'blue';
}
```

And you can import the CSS file and use the class name in JSX elements that you want to style, like this:

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

function App() {
  return (
    <>
      <p className="paragraph-text">
        The weather is sunny today.
      </p>
    </>
  );
}
```

This way, the CSS will be separated from your JavaScript files, and you can just write CSS syntax just as usual.

You can even include a CSS framework such as Bootstrap in your React app using this approach. All you need to is import the CSS file into your root component.

This method will enable you to use all of the CSS features, including pseudo-classes and media queries. But the drawback of using a stylesheet is that your style won't be localized to your component.

All CSS selectors have the same global scope. This means one selector can have unwanted side effects, and break other visual elements of your app.

Just like inline styles, using stylesheets still leaves you with the problem of maintaining and updating CSS in a big project.

CSS Modules

A CSS module is a regular CSS file with all of its class and animation names scoped locally by default.

Each React component will have its own CSS file, and you need to import the required CSS files into your component.

In order to let React know you're using CSS modules, name your CSS file using the `[name].module.css` convention.

Here's an example:

```css
/* App.module.css */
.BlueParagraph {
  color: blue;
  text-align: left;
}
.GreenParagraph {
  color: green;
  text-align: right;
}
```

Then import it to your component file:

```jsx
import React from "react";
import styles from "./App.module.css";

function App() {
  return (
    <>
      <p className={styles.BlueParagraph}>
        The weather is sunny today.
      </p>
      <p className={styles.GreenParagraph}>
        Still, don't forget to bring your umbrella!
      </p>
    </>
  )
}
```

When you build your app, webpack will automatically look for CSS files that have the `.module.css` name. Webpack will take those class names and map them to a new, generated localized name.

Here is the sandbox for the above example. If you inspect the blue paragraph, you'll see that the element class is transformed into `_src_App_module__BlueParagraph` .

CSS Modules ensures that your CSS syntax is scoped locally.

Another advantage of using CSS Modules is that you can compose a new class by inheriting from other classes that you've written. This way, you'll be able to reuse CSS code that you've written previously, like this:

```css
.MediumParagraph {
  font-size: 20px;
}
.BlueParagraph {
  composes: MediumParagraph;
  color: blue;
  text-align: left;
}
.GreenParagraph {
  composes: MediumParagraph;
  color: green;
  text-align: right;
}
```

Finally, in order to write normal style with a global scope, you can use the :global selector in front of your class name:

```css
:global .HeaderParagraph {
  font-size: 30px;
  text-transform: uppercase;
}
```

You can then reference the global scoped style like a normal class in your JavaScript file:

```
<p className="HeaderParagraph">Weather Forecast</p>
```

Styled components

Styled Components is a library designed for React and React Native. It combines both the CSS in JS and the CSS Modules methods for styling your components.

Let me show you an example:

```
import React from "react";
import styled from "styled-components";

// Create a Title component
// that renders an <h1> tag with some styles
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

function App() {
  return <Title>Hello World!</Title>;
}
```

When you write your style, you're actually creating a React component with your style attached to it. The funny looking syntax of styled.h1 followed by backtick is made possible by utilizing JavaScript's tagged template literals.

Styled Components were created to tackle the following problems:

- **Automatic critical CSS:** Styled-components keep track of which components are rendered on a page, and injects their styles and nothing else automatically. Combined with code splitting, this means your users load the least amount of code necessary.
- **No class name bugs:** Styled-components generate unique class names for your styles. You never have to worry about duplication, overlap, or misspellings.
- **Easier deletion of CSS:** It can be hard to know whether a class name is already used somewhere in your codebase. Styled-components makes it obvious, as every bit of styling is tied to a specific component. If the component is unused (which tooling can detect) and gets deleted, all of its styles get deleted with it.
- **Simple dynamic styling:** Adapting the styling of a component based on its props or a global theme is simple and intuitive, without you having to manually manage dozens of classes.
- **Painless maintenance:** You never have to hunt across different files to find the styling affecting your component, so maintenance is a piece of cake no matter how big your codebase is.
- **Automatic vendor prefixing:** Write your CSS to the current standard and let styled-components handle the rest.

You get all of these benefits while still writing the same CSS you know and love – just bound to individual components.

If you'd like to learn more about styled components, you can visit the documentation and see more examples.

Which one should you start with?

As you will practice building some React projects in the next chapter, I recommend you to go with **CSS stylesheets** method because it's probably most familiar to you.

CSS Modules and Styled Components are better methods to handle complex front-end styling, but since these are practice projects, you don't need to think about scaling your CSS for large React projects.

Thank You For Reading Digetsting React Preview

If you decided to learn more about React and support my effort in making coding skills available to everyone, please consider purchasing a copy of this book at https://sebhastian.com/digesting-react/

It includes the rest of the chapters along with the practical projects.

For any question, you can email me at nathan@sebhastian.com

Always remember to improve your skills everyday :)

Regards,

Nathan Sebhastian the React Geek