**WHY REACT**

We mostly use OOP but React uses components. You can integrate other third party libraries like redux. The biggest competitor is Angular it is a full-fledged solution for an app or website and don’t need to integrate with third party. React provide flexibility with creating complex UI whereas angular make it easier to make a single page UI

BASICS

**COMPONENT-BASED ARCHITECTURE**

Build software based on reusable components of code. Each component consists of well-defined functions that can be inserted into an application

Components are reusable thus easily inserted anywhere needed thus can exist within the same space independently

Advantages

Many developers can work on same project without interfering on the other ones

Build and compose components

High performance rendering

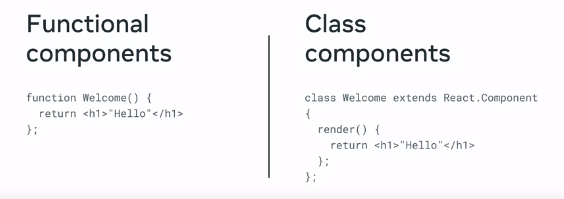
Component rendering

DOM – using html makes it complex thus making it a spaghetti code coz it’s difficult to understand

React prevents this by avoiding any manipulation of the DOM instead react provides a virtual DOM it is an in memory of a DOM or a clone thus minimizes updates

INTRO TO FXN COMPONENT

React provides 2 types of components



Let start with

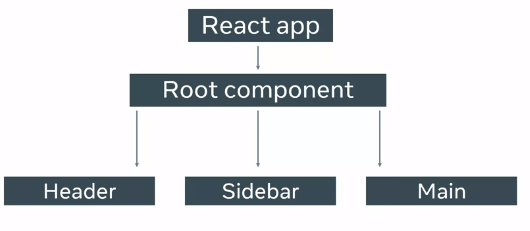
FXN COMPONENT

Only one component is rendered and it is the app component



Syntax to render a component is similar to a self -losing tag in html







 Similar to JS and has some HTML inside



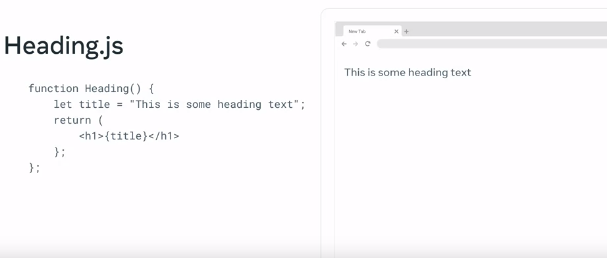
JSX is mixture of html and JavaScript



Notice how the first letter is capitalized that’s coz react components must be capitalized



Coz react treats lower case as html components thus able to distinguish btwn jsx and html



The rendering above happens behind the scene coz of transpiling – it’s the process of converting jsx to html

**Transpiling JSX**

**By the end of this reading, you will have learned how a component is built.**

**Introduction**

Components are a nice way to build websites in React because they allow you to build more modular apps. However, how do you build components using React, JSX, and JavaScript? You'll learn how this works in this lesson item.

**A browser cannot understand JSX syntax.**

This means that making a browser understand React code requires a lot of supporting technologies.

An example of such a technology is a **transpiler**.

A **transpiler** takes a piece of code and transforms it into some other code.

To understand why this is done, here is an example of an ES6 variable declaration:

1

Const PI = 3.14

This is perfectly valid ES6 syntax.

However, if you were using a very old computer, that computer will have an old browser. Perhaps that browser was built before ES6 came out in 2015.

This means that the JavaScript engine that is built into your old computer's browser is likely to be an ES5 JavaScript engine.

In ES5, the only way to declare a variable is the following:

1

Var pi = 3.14

What this means is that for this old browser to understand the ES6 code, the only way to do it is by **transpiling** it.

If you feel like it, you can try transpiling ES6 to ES5 code yourself, using [the es6console website](https://es6console.com/).

Now, let’s move the focus to another example of transpiling.

Let's say that you want to use a brand new, most modern ECMAScript syntax in an app. The only problem is that this new syntax is currently not supported by any browser; even an up-to-date browser.

However, by transpiling the new most-modern JavaScript syntax into something that modern browsers can understand, it is able to convert some code that the browser cannot comprehend, into code that it can comprehend, run, and produce a result from.

Likely the most popular site that shows off how this works is [Babel](https://babeljs.io/). As the heading of the website reads, "Babel is a JavaScript Compiler".

This finally brings you to the point of this discussion about transpiling JavaScript code.

What Babel does is this: it allows you to transpile JSX code (which cannot be understood by a browser) into plain JavaScript code (which can be understood by a browser).

This is where React and JSX come in.

For React code to be understood by a browser, you need to have a **transpiling step** in which the JSX code gets converted to plain JavaScript code that a modern browser can work with.

To demonstrate how this works, let’s use the **Heading** component from the previous lesson.

Add the JSX code into [the online Babel repl](https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=GYVwdgxgLglg9mABACQKYEMAmMwHMAUADgE5yEDOAlIgN4BQijixqUIxSAPABYCMAfDRJlyAOlhQANqgC-nAPR9-dGUA&debug=false&forceAllTransforms=false). Repl stands for "read-eval-print loop" and it accepts code you write, evaluates it, and produces some result. In the specific case of [the online Babel repl](https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=GYVwdgxgLglg9mABACQKYEMAmMwHMAUADgE5yEDOAlIgN4BQijixqUIxSAPABYCMAfDRJlyAOlhQANqgC-nAPR9-dGUA&debug=false&forceAllTransforms=false), that result is some transpiled code. Here's a more detailed explanation.

If you've visited the above-linked URL, you'll find a web page that has two panels. On the left, there's source JSX code:

function Heading (props) {

    return <h1>{props.title}</h1>

}

..and on the right, there's the transpiled, plain JavaScript code. However, ensure that you select the **classic** runtime for React in the left sidebar.

function Heading(props) {

  return /\*#\_\_PURE\_\_\*/React.createElement("h1", null, props.title);

}

If you now analyze the difference between the source JSX code and the transpiled, plain JavaScript code, dis-regarding the comment, here's the body of the Heading function:

React.createElement("h1", null, props.title);

So, here you have a React object, and this object has a **createElement()** method on it. The method is invoked with three arguments:

1. **"h1"**
2. **null**
3. **props.title**

The first argument is the DOM element to render - in this case, an **h1** element. The second property is any HTML attribute that should be added, and there's a null here - meaning, there should be an object with some data, but there isn't any data so instead of the object there's the null value. The third property is the contents of the inner HTML of the DOM element specified as the first argument - in this case, the contents of the inner HTML of the **h1** element.

Now let’s use Babel again, and this time transpile the **render** syntax for the **Heading** component:

1

<Heading title="This is the heading text!"></Heading>

Again using [the Babel repl](https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=DwCQpghgJglgdgcwAQBcYoDZgLwCIAqAFjAM5KmqFhJXTzIpgAeKAhLgHzAD04diHIA&debug=false&forceAllTransforms=false&modules=false&shippedPro), and as can be confirmed in [the link](https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=DwCQpghgJglgdgcwAQBcYoDZgLwCIAqAFjAM5KmqFhJXTzIpgAeKAhLgHzAD04diHIA&debug=false&forceAllTransforms=false&shippedProposals=false&c), the output of the tranpilation is the following code. Ensure that you select the **classic** runtime for React in the left sidebar.

/\*#\_\_PURE\_\_\*/

React.createElement(Heading, {

  title: "This is the heading text!"

});

Again, you have the **React.createElement()** method call, and this time, the first item to render is **Heading**, and then you have an object as the second argument (instead of a null that you had in the previous transpilation example).

This brings me to an interesting question: What is the minimum code that a component must have to be able to show something on the screen when rendered?

You can see the answer below:

function Example()

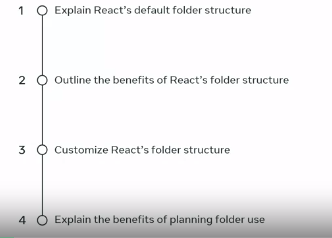
    return <div>An element</div>

}

export default Example

**THE REACT PROJECT STRUCTURE**

By the end you will learn



Default folder structure

* public

for now it is important to know index.html

* src

you will mostly use this

**Customizing the project**

So far, you’ve learned about React components, but now you will focus on learning how to customize the project. You will learn about the software development approach, detailing the creation of separate associated files, the requirements gathering and the subsequent folder structure to be created.

**Building a Layout**

Imagine that you've been given the task of building a somewhat more complex website layout using React.

At this point, you still don't know too much about how React works, but even with your limited knowledge, you can still build some relatively interesting designs.

Currently, you need to build a simple typography-focused layout for a coding blog.

This means that you will not have to use images, which simplifies your task significantly.

The layout you're supposed to build will consist of the following sections:

* Main navigation
* Promo (main advertisement)
* A list of newest posts' previews (intros)
* The footer

**Organizing Your Code**

Keeping in mind the above structure, how would you organize your code?

This is where React docs can help. They suggest two approaches:

1. Grouping by features
2. Grouping by file type

They also advise not to nest folders too deep, and to keep things simple and not overthink it.

They even say that if you're just starting out, you shouldn't spend more than five minutes setting up a project.

Taking this advice into account, you might say that for a small project like this, you could keep it as simple as just adding a **components** folder and moving all your components into it. This is exactly what you’ll do next.

**Building The App**

Since this is app's focus is on customization, let’s name the app **customizing-example***.*

What follows is the command to run **in a suitable folder on your own computer**. By "a suitable folder", I mean: "a folder where you feel comfortable installing a boilerplate React application". This also includes that the folder you chose will need to be accessible for your user on your OS (Operating System).

npm init react-app customizing-example

This will produce a brand-new starter app with a familiar structure.

Inspecting the **src** folder of the starter app, it looks like this:

src/

    App.js

    App.test.js

    index.css

    index.js

    logo.svg

    reportWebVitals.js

    setupTests.js

Then simply add a components folder to it, like this:

src/

    components/

    App.js

    App.test.js

    index.css

    index.js

    logo.svg

    reportWebVitals.js

    setupTests.js

Since the components folder is currently empty, you can add a component for each of the sections of the typography-focused blog. Here's the structural update:

src/

    components/

        Nav.js

        Promo.js

        Intro1.js

        Intro2.js

        Intro3.js

        Footer.js

    App.js

    App.test.js

    index.css

    index.js

    logo.svg

    reportWebVitals.js

    setupTests.js

At this point, there's no need to complicate things. You have the **Nav** component, the **Promo** component, the **Intro1**, **Intro2**, and the **Intro3** component. Finally, there's also a **Footer.js** component.

This means you've fully planned the app, based on some best practices as suggested by the official React docs website, and based on the level of complexity of the project itself. Since this project is relatively simple, this structure feels right.

In this reading, you’ll just build all the components inside the components folder, and then, in the upcoming lesson items, import them into the **App.js** file.

**Building Components**

For now, let’s just build those components. After you've added the **components** folder, you’ve also added all the functional component files. Since they are all currently empty, you can start adding them, one by one.

Heres' the contents of the **Nav.js** file:

function Nav() {

    return (

        <nav className="main-nav">

            <ul>

                <li>Home</li>

                <li>Articles</li>

                <li>About</li>

                <li>Contact</li>

            </ul>

        </nav>

    );

};

export default Nav;

Next, you can focus on the **Promo.js** file:

function Promo() {

    return (

        <div className="promo-section">

            <div>

                <h1>Don't miss this deal!</h1>

                </div>

                <div>

                <h2>Subscribe to my newsletter and get all the shop items at 50% off!</h2>

            </div>

        </div>

    );

};

export default Promo;

Once you’ve finished the promo section, you can focus on the Intro components.

Here's **Intro1.js:**

function Intro1() {

    return (

        <div className="blog-post-intro">

            <h2>I've become a React developer!</h2>

            <div>

                <p>I've completed the React Basics course and I'm happy to announce that I'm now a Junior React Developer!</p>

                <p className="link">Read more...</p>

            </div>

        </div>

    );

};

export default Intro1;

Here's the code for the **Intro2.js** component:

function Intro2() {

    return (

        <div className="blog-post-intro">

            <h2>Why I love front-end web development</h2>

            <div>

                <p>In this blog post, I'll list 10 reasons why I love to work as a front-end developer.</p>

                <p className="link">Read more...</p>

            </div>

        </div>

    );

};

export default Intro2;

You can finish the previews for my blog posts with the code for **Intro3.js** component:

function Intro3() {

    return (

        <div className="blog-post-intro">

            <h2>What's the best way to style your React apps?</h2>

            <div>

                <p>There are so many options to choose from. Here's a high-level overview of the popular ones.</p>

                <p className="link">Read more...</p>

            </div>

        </div>

    );

};

export default Intro3;

There's just one more thing left to code, the **Footer** component, so here it is:

function Footer() {

    return (

        <div className="copyright">

            <p>Made with love by Myself</p>

        </div>

    );

};

export default Footer;

that you have completed all the components for the app, here are a few more interesting things about the syntax.

These are:

* The use of the **className** attribute in JSX
* The use of separate components for repetitive code
* Where are all the props?
* Why was I not using the **<a>** element for empty links?

**Discussing the Syntax**

Now let’s briefly discuss the four bullet points above.

Why use the **className** attribute in the JSX syntax?

Well, with JSX, it looks like HTML so much that it's easy to forget that it's actually JavaScript code - not HTML.

While regular HTML does indeed have a **class** attribute, which is used to list one or more CSS classes to be used on a given HTML element, this cannot really work in JSX. The reason is that JSX is a special kind of JavaScript syntax, and the word **class** is a reserved keyword in JSX. That's why the React team had to make a compromise and so **className** is used in JSX to list one or more CSS classes to be used on a given element or component.

But why use **Intro1.js, Intro2.js,** and **Intro3.js**? Isn't one of the tenets of coding the DRY approach - that is, the "Don't repeat yourself" approach?

Indeed, it is. However, there are still a few concepts to discuss before you learn how to re-use a single component with variations in its content. This has to do with data in components, but don’t worry, we’ll be getting to that later.

The third question is about the **props** object. It has been mentioned before, but so far it hasn't been used. It hasn’t been used in this example either.

The answer to this question has to do with the next lesson, titled ***Component Use and Styling***. In this lesson, you’ll see in practice how you can make components work better, with the help of **props**.

The final question is about not using the **<a>** element for empty links in my app.

The answer here depends on whether those links are "internal" - inside an app, or "external", meaning, leading to some external link, such as [*https://www.coursera.org*](https://www.coursera.org/). If the links are internal to the app - as they are envisioned here - using the **<a>** tag is simply not the React way of doing things. You'll learn why that is the case when discussing the use of React Router.

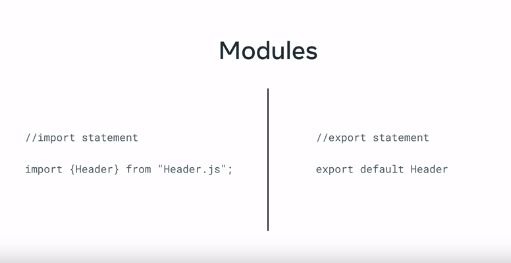
**Conclusion**

Having finished this reading, you have now learned about the software development approach, detailing the creation of separate associated files, the requirements gathering, and the subsequent folder structure to be created.

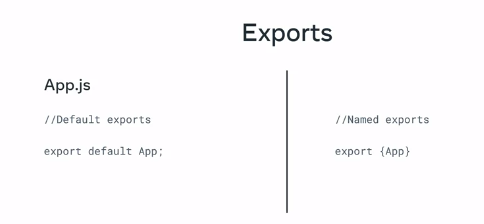
**IMPORTING COMPONENTS**







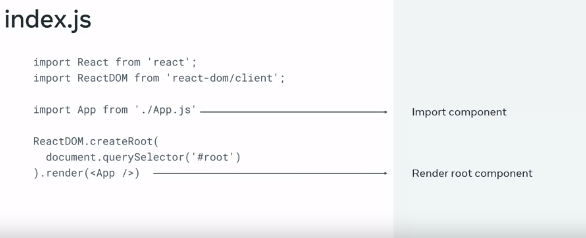




Default is when the function name is same as the file name while name export used when you want the fxn name to be different from the file name

Different btwn component and module

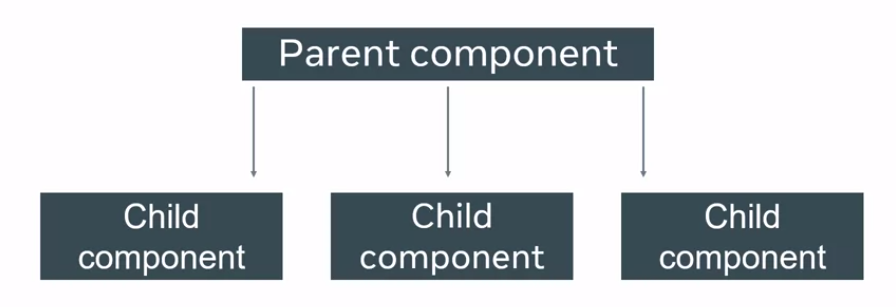




From react you use import followed by the file name. react has no strict rule on how you put files into folders

However there are few approaches likegrouping similar files together

**PRINCIPLE OF PROPS COMPONENTS**



Remember it is one directional but dev use other way

LIMITATION

1. Not possible to send data from child to parent component

2. Pure function- return same value for same function. It never modifies it’s own props



# Dissecting props

Recall that much like parameters in a JavaScript function which allow you to pass in values as arguments, React uses properties, or **props**, to pass data between components. But how exactly do they work?

In this reading, you’ll use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Remember first that JSX code in React is just syntactic sugar - meaning, a nicer way to write some hard-to-read code.

For the browser to understand this syntactic sugar, you need to transpile JSX down to plain JavaScript code. You have a resource online, at the URL of [babeljs.io](https://babeljs.io/), which allows you to inspect the results of this transpiling. Once you visit the website, make sure to navigate to the Try it out link in the main navigation.

For example, let’s say you have a component that returns a piece of JSX:

function App() {

  return <h1>Hello there</h1>

}

JavaScript code, you’d get back some unusual code:

"use strict";

function App() {

    return /\*#\_\_PURE\_\_\*/React.createElement("h1", null, "Hello there");

}

You just want to focus on the **React.createElement("h1", null, "Hello there");** part. You can ignore the rest.

This means that the **createElement** function receives three arguments:

1. The wrapping element to render.
2. A null value (which is there to show an absence of an expected JavaScript object value).
3. The inner content that will go inside the wrapping element.

Interestingly, the inner content that will go inside the wrapping element can also be a call to the **createElement** function.

For example, let’s say you have a slightly more complex JSX element structure:

function App() {

  return (

    <div>

    <h1>Hello there</h1>

    </div>

  )

}

… the transpiled return statement in plain JavaScript again returns two **createElement** functions:

"use strict";

function App() {

  return /\*#\_\_PURE\_\_\*/React.createElement("div", null, /\*#\_\_PURE\_\_\*/React.createElement("h1", null, "Hello there"));

}

If you format this output, remove the **"use strict"** line, and remove the **\_\_PURE\_\_** comments, you get a more readable output:

function App() {

  return React.createElement(

    "div",

    null,

    React.createElement("h1", null, "Hello there")

  );

}

So now the third argument of the outer-most **React.createElement** call is another **React.createElement** call.

This is how you can nest as many elements as you want.

This means that a nested JSX structure is just a bunch of nested **React.createElement** calls, passed in to other **React.createElement** calls as their third argument.

## The second – null – argument

The second argument of **null** can – in this case – be replaced with an empty object.

In that case, your code would contain a pair of curly braces instead of the word **null**:

"use strict";

function App() {

  return React.createElement(

    "div",

    {},

    React.createElement("h1", {}, "Hello there")

  );

}

This object is referred to as the props object. It is the main mechanism of sending data from a parent component to a child component in React.

The way this works is described in React docs using the following code:

React.createElement(

  type,

  [props],

  [...children]

)

## The third argument (...children)

This is the inner content that will go inside the wrapping element. It's what makes it possible to nest elements inside other elements, mimicking the way that HTML works.

In this reading you’ve learned how to use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.