React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

React apps are built using modern JavaScript features, which are commonly known as ES6. Developers use React to develop Singe Page Applications. And you can also develop mobile applications with React Native.

Another important concept to know about when you're talking about HTML is the Document Object Model, or DOM.

Users need to be able to interact with elements on a web page. This means that an HTML document must be represented in a way that JavaScript code can query and update it. And that's the function of the DOM. It's a model of the objects in your HTML file.

And web developers interact with the DOM through JavaScript to update content, set up events and animate HTML elements.

**CSS**

CSS (Cascading Style Sheets) is the code that you use to style HTML. You need to be familiar with basic CSS concepts before you start learning React. This is because you will need to style your React components as well, and basic CSS knowledge will help your learning journey.

Before you learn React, make sure you are comfortable with these CSS styling options:

* Font styling (font size, font color, etc.)
* Flex Box Layout (Layout of items using CSS Flex Box Layout)
* CSS Selectors
* Position, Padding, Margins and Display
* Colors, Background and Icons

## JavaScript fundamentals and ES6

React is completely written in JavaScript and uses the more modern version of JavaScript which is ES6. While learning React, you should already know JavaScript fundamentals.

JavaScript is the programming language and React is a JavaScript UI library. This means the first step is to be proficient at JavaScript.

Here are some of the JavaScript topics that you need to be comfortable with before you begin your journey learning React.

* Data types
* Using var, let and const
* Conditionals and Loops
* Using objects, arrays and functions
* ES6 Arrow functions
* In-built functions such as map(), forEach() and promises.
* Destructuring Arrays and Objects
* Error Handling

## Package Manager (Node + npm)

React is a UI library, and you will encounter that many times you will need to add other packages to your React application. A package in JavaScript contains all the files needed for a module. To install these packages effectively and manage their dependencies you can use a package manager like NPM (Node Package Manager).

You can install npm by installing Node.js, which will then automatically install npm.

You need to be comfortable with using npm as your package manager, since you will be using npm to install packages within your React application.  Make sure you are aware of how to do the following with npm before you get started on this course.

* Installation command to install npm modules in your project
* Installing a package as a dev dependency
* Start command
* Updating npm version
* Navigating around the package.json file

Once you have become confident with these skills, you’ll be in a better position to learn and apply React concepts and prepare yourself for development of React apps.

# **JavaScript modules, imports - exports**

Before you start creating the next great app, let’s explore a little more about modules.

Modules can help you to save and access your code in a more structured way, and in this reading, you'll learn about some foundational concepts of working with JavaScript modules.

This knowledge is crucial in order to understand the syntax and the logic behind how the example React apps in this course are put together.

This reading will cover the three main concepts:

1. JavaScript modules
2. Module exports
3. Module imports

## ****JavaScript Modules****

In JavaScript, a module is simply a file.

The purpose of a module is to have more modular code, where you can work with smaller files, and import and export them so that the apps you build are more customizable and have more composable parts.

A module can be as simple as a single function in a separate file.

Consider the following function declaration:

function addTwo(a, b) {

    console.log(a + b);

}

Say that you have a file named **addTwo.js** that contains only the above code.

How would you make this file a JavaScript module?

All that you would need to do to make it a JavaScript module is use the export syntax.

## ****Module Exports****

There is more than one way to export a module in JavaScript.

While all the various syntactical differences are not listed, here are a few examples that will cover all the ways that the importing and exporting of JavaScript modules will be done in this course.

In general, there are two ways to export modules in JavaScript:

1. Using default exports
2. Using named exports

### **Default Exports**

You can have **one default export** per JavaScript module.

Using the above **addTwo.js** file as an example, here are two ways to perform a default export:

export default function addTwo(a, b) {

    console.log(a + b);

}

So, in the above example, you’re adding the **export default** keywords in front of the **addTwo** function declaration.

Here's an alternative syntax:

function addTwo(a, b) {

    console.log(a + b);

}

export default addTwo;

### **Named Exports**

Named exports are a way to export only certain parts of a given JavaScript file.

In contrast with default exports, you can export as many items from any JavaScript file as you want.

In other words, there can be only one default export, but as many named exports as you want.

function addTwo(a, b) {

    console.log(a + b);

}

function addThree(a + b + c) {

    console.log(a + b + c);

}

If you want to export both the **addTwo** and the **addThree** functions as named exports, one way to do it would be the following:

export function addTwo(a, b) {

    console.log(a + b);

}

export function addThree(a + b + c) {

    console.log(a + b + c);

}

Here's another way you could do it:

function addTwo(a, b) {

    console.log(a + b);

}

function addThree(a + b + c) {

    console.log(a + b + c);

}

export { addTwo, addThree };

## ****Importing Modules****

Just like when exporting modules in JavaScript, there are several ways to import them.

The exact syntax depends on how the module was exported.

Say that you have two modules in a folder.

The first module is **addTwo.js** and the second module is **mathOperations.js**.

You want to import the **addTwo.js** module into the **mathOperations.js** module.

### **Importing a Module that was Exported as Default**

Consider the previous example of exporting the **addTwo** function as a default module:

// addTwo.js module:

function addTwo(a, b) {

    console.log(a + b);

}

export default addTwo;

To import it into the **mathOperations.js** module, you could use the following syntax:

import addTwo from "./addTwo";

// the rest of the mathOperations.js code goes here

So, you could start this import with the **import** keyword, then the name under which you’ll use this imported code inside the **mathOperations.js** file. You would then type the keyword **from**, and finally the location of the file, without the .js extension.

Contrast the above import of the default **addTwo export** with the different import syntax if the **addTwo** function was instead a named export:

import { addTwo } from "./addTwo";

// the rest of the mathOperations.js code goes here

### **To view your code and instructions side-by-side**, select the following in your VS Code toolbar:

1. View -> Editor Layout -> Two Columns
2. To view a file in Preview mode, right click on the file and **Open Preview** (in the EXPLORER sidebar)
3. Select your code file in the code tree, which will open it up in a new VS Code tab.
4. You can drag any file over to the second column to view the contents in that column.
5. Great work! You can now see instructions and code at the same time.

# **Additional reading**

Below you will find links to helpful additional readings.

* [nodejs.org](https://nodejs.org/en/)
* [npmjs.com](https://www.npmjs.com/)
* [reactjs.org](https://reactjs.org/)
* <https://create-react-app.dev/>
* [VS Code](https://code.visualstudio.com/)

True or false? React enables developers to build SPAs (single page applications).

* + True
* **Correct**
* Well done. A technology that loads a single web page and performs updates to the DOM on this single web page based on user interaction with this web page in known as a SPA. Indeed, React allows developers to build SPAs.

Recall the concept of the single-page application or SPA for short?

This is a one-page website where some of the pages content changes based on user

interaction.

SPA differ from traditional website,

where each web page needs to be loaded as the user navigates around the site.

This frequent reload can be considered inefficient as some of the same content

such as logos, navigation, and footers need to be loaded again.

And one of the most popular ways to build an SPA is with the React library

from Meta.

React used to run many of the world's most popular websites.

Often on these sites, you're type a search term into the sites input box.

The site then returns the relevant content or results.

But you might notice that even though the content has updated,

the sites URL doesn't change.

As SPA only load the content as required, they can be ideal for businesses and

enterprises who need a web app that offers rich user interfaces,

speed scalability, and flexibility.

As an aspiring developer, you may feel like there are a lot of new concepts and

processes to understand when working with React but you can be sure that you will

have an opportunity to get familiar with and use them.

In this video, you will learn about the basics of React by exploring the concepts

of the component-based architecture components and the virtual DOM.

Let's begin with components.

One of the core building blocks of React.

When meta released the React library,

it included the concept of something called component-based architecture.

This is essentially a design philosophy for

building software based on reusable components of code.

Each component consists of well-defined functionality that can be inserted

into an application without requiring modification of other components.

Because components are reusable, they can be used multiple times and

easily inserted anywhere we're needed.

This results in components that can exist within the same space yet

interact independently from each other.

One of the advantages of developing using components is that many developers can

work on the same project without interfering with

the code of other developers' components.

As you may recall, modern front-end web development revolves around

the concept of creating standalone parts of the user interface, or UI for short.

Well, in react, these standalone parts are created using

components which form the foundation of all UI design.

It's important to know that all UI is composed of simple components that can be

combined into more complex components.

In fact,

you can think of an entire website is just a collection of components.

For example, consider the product check out page of an e-commerce web application.

The page consists of three sections, a header, a payment section, and a sidebar.

The header section contains the company logo with a navigation menu and

a button to view the shopping cart.

The payment section area contains a form where the user inputs their payment

information.

Finally, there's a sidebar with the order summary information.

As the components are self-contained, they have their own HTML, CSS,

and JavaScript logic for functionality.

For example, the payment section component has a JavaScript function that

submits a payment when a button is clicked.

It's important to know that the use of components in website UI design is not

limited to just React.

Many website's front end or UI are built on the foundations of components and

compose ability.

But React is a powerful tool for

streamlining the process of building components and composing them.

It performs these actions efficiently as components are rendered to the DOM without

significantly impacting the browser's resources.

This is called component rendering and you'll learn more about this and

its associated render method later.

You may recall that the DOM is a logical tree-like structure representing the HTML

document and it uses nodes to describe the various parts of the document.

Before React, you could still build components style layouts.

However, it involved much more complicated DOM manipulation and

code making the layouts more complex and harder to work with.

This resulted in something known as spaghetti code,

a term developers use in web development to describe code that is complex,

convoluted, and difficult to understand like spaghetti.

Reacts prevents this spaghetti code by avoiding any manipulation of the DOM.

Instead, React to provide something known as the virtual DOM.

You may recall that this is an in-memory representation or

clone of the real DOM which minimizes updates to the DOM itself.

React uses the virtual DOM to update the browser Dom only when needed.

This ensures that the update is as minimal as possible,

increasing the application speed and performance.

In this video, you learned about the basics of React by exploring the concepts

of the component-based architecture components and the virtual DOM.

Recall how you learned about functions in JavaScript.

They are reusable blocks of code that can take an input,

perform some procedure or calculation,

and then return an output.

Well, a React component acts

much like a traditional JavaScript function.

In this video, you'll continue

your exploration of the structure of

the React architecture by

learning about functional components.

Component types, JSX and transpiling.

React provides two types of components,

functional components and class components.

They behave very similar in React to

traditional functions, and classes in JavaScript.

Don't worry about class components for now.

You'll learn more about them later.

Instead, let's just focus on functional components,

which acts like a JavaScript function.

In the default React application,

only one component is

rendered and it's the app components located

inside the index.js file

that's located inside the source folder.

It's important to know that every React app

must contain at least one component,

and it's called the root components.

This component is loaded using the import statement.

You'll learn more about the import

statement in React later.

For now, just know that it's used to

import code needed for React to work,

such as the import React and import React DOM statements.

The syntax to render a component is very

similar to a self-closing tag in HTML,

you just place the component name inside the left and

right angle brackets and don't forget the forward slash.

The roots components can

contain other components that developers

create to represent

the various UI part of the application.

Like in the e-commerce example

that you learned about earlier.

Recall that this component is ultimately

converted to a DOM fragment and placed

into the existing DOM as a child of

the HTML div element with an ID of roots.

This div element is then rendered to the browser.

If you analyze the app component,

you'll notice that it looks very similar to

a JavaScript function with some HTML code inside.

You may also notice an export default statement.

You'll learn more about this soon.

For now, just know that you need it

to make your components available.

Now that you're familiar with the concept

of functional components,

let's explore how web developers create them in React.

React is scripted using

a special syntax called JavaScript XML or JSX.

For many Reacts developers,

this is known as a syntax extension to JavaScript.

What is JSX syntax like?

Let's find out by going back to

our React default app component.

Recall that in the return statements of the app function,

it seemed that some HTML content is returned.

Well, this content is not exactly HTML.

It's JSX.

JSX syntax looks very similar to HTML.

What are its advantages is that it allows you to write

JavaScript code inside what looks like HTML elements.

In fact, you can think of JSX as

a combination of custom HTML and JavaScript.

This allows you to make your website dynamic.

You'll learn more about the differences between

HTML and JSX later.

For now, just know that you can place this syntax

inside the return statement of a functional component.

It's also important to note that

a React component won't render until it's used as

a JSX element just like

a JavaScript function declaration and

won't run until it's called or invoked.

Now you know what JSX is.

Let's explore the steps involved to

create a React components which will contain

some JSX code inside

a heading 1 HTML element

to display some text on a webpage.

First, you create the component,

which is basically just a JavaScript file,

since its purpose is to return some heading text,

you name the file Heading.js.

Notice that the first letter of

the component name is capitalized.

This is because there's a difference in how React treats

capitalized, and non capitalized component names.

It's important to remember that

all component names in React must

be capitalized. Why is this?

Well, because React treats

lowercase components as regular HTML elements.

Capitalizing a component name helps React to

distinguish JSX elements from HTML elements.

Now let's continue with our component.

Next inside the app.js file,

create a function named Heading.

The function name must also be declared using

a capital letter for the first letter of the function.

Then inside the function body,

you create a variable named title and assign it

the string value of this is some heading text.

Now you're ready to create

the return statement of the function.

Inside the parentheses of the return,

insert a heading 1 tag,

and inside it place the variable named title.

To make React evaluate the title variable,

you need to place it inside curly brackets.

If you didn't use curly brackets,

you'd get the word title instead

of this is some heading text.

At this point, it's worth remembering that while

you are creating HTML like syntax,

you are actually coding inside a JavaScript file.

Because of this, you can output

a variable inside your JSX code,

something you cannot do when writing static HTML.

The overall syntax instructs React to render

the heading HTML element with

whatever text value that is

stored within the variable named title.

This rendering happens behind the scenes

because of something called transpiling.

You can think of transpiling as a process of converting

JSX to HTML and you'll learn more about this later.

In this video you learned about

functional components and how to create them in React.

You also learned about JSX,

which acts like a combination of HTML, CSS,

and JavaScript that you can use to generate

dynamic content inside your functional components.

Finally, you explore the concepts

of rendering and transpiling.

If you'd like to learn about

these concepts in more detail.

There's a link to an additional reading

at the end of this lesson.

True or false? For a component to render something on the page, it needs to return one or more JSX elements.



False



True.

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:

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:

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: "use strict";

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);

<h1>{props.title}</h1> ⬄ React.createElement("h1", null, 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:

<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&shippedProposals=false&c), 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:

"use strict";

/\*#\_\_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

Question 1

True or False: You can declare a JavaScript function to be used as a component in React.

**1 / 1 point**



True



False

**Correct**

Yes, it is correct that a function declaration is a way to create a component in React.

### 2.

Question 2

True or False: You can spread the return statement over multiple lines by surrounding the code that follows the return keyword with an opening and a closing parenthesis.

**1 / 1 point**



True



False

**Correct**

Yes, it is correct that you can surround multiple lines of code in a pair of parentheses so that all that code can be returned from a component.

### 3.

Question 3

Is this valid component code? **function Example() { return (<h1>Example</h1>) }**

**1 / 1 point**



Yes



It would be valid, if it was spread over multiple lines.



No

**Correct**

Correct. This code shows a component named Example. It is defined as a function, it does not receive any parameters, and it has a return statement with <h1>Example</h1> text in a pair of parentheses.

# **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>

}

… if you used the Babel transpiler to transpile this JSX syntactic sugar code down to plain 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**:

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.

Marquer comme terminé

J'aime

Je n'aime pas

Signaler un problème

* The arrow function has a single parameter, so you do not need to add parentheses around the item parameter (to the left of the arrow)
* Since the arrow function fits on one line of code, you don’t need to use curly braces around the function body, or the return keyword; it's implicit

Arrow functions are used extensively in JSX in React, and getting used to their syntax and being able to "mentally parse" it as you read it is an important skill to have and helps you get better at writing React apps.

Now that you have completed this reading, you’ve learned about some alternative approaches, specifically by using function expressions and arrow functions.

## ****Using ternary expressions in JSX****

Let’s examine an example of a component which uses a ternary expression to randomly change the text that is displayed.

function Example() {

    return (

        <div className="heading">

            <h1>{Math.random() >= 0.5 ? "Over 0.5" : "Under 0.5"}</h1>

        </div>

    );

};

Inside the **<h1>** element, the curly braces signal to React that you want it to parse the code inside as regular JavaScript.

Then, inside the curly braces, you can add a ternary statement. Every ternary statement conceptually, expressed in pseudo-code, works like this:

Inside the **<h1>** element, the curly braces signal to React that you want it to parse the code inside as regular JavaScript.

Then, inside the curly braces, you can add a ternary statement. Every ternary statement conceptually, expressed in pseudo-code, works like this:

comparison ? true : false

Inside the **<h1>** element, the curly braces signal to React that you want it to parse the code inside as regular JavaScript.

Then, inside the curly braces, you can add a ternary statement. Every ternary statement conceptually, expressed in pseudo-code, works like this:

const bool = false;

function Example(props) {

    return (

        <h2>The value of the toggleBoolean prop is: {props.toggleBoolean.toString()}</h2>

    );

};

export default function App() {

    return (

        <div className="App">

            <Example toggleBoolean={!bool} />

        </div>

    );

};

Below you will find links to helpful additional resources.

* [Components and props](https://reactjs.org/docs/components-and-props.html)
* [Intoducing JSX](https://reactjs.org/docs/introducing-jsx.html)
* [Styling and CSS in React](https://reactjs.org/docs/faq-styling.html)
* [Introducing expressions in JSX](https://reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx)

Marquer comme terminé

J'aime

Je n'aime pas

Signaler un problème

Hooks also come with a set of rules, that you need to follow while using them. This applies to all React hooks, including the **useState** hook that you just learned.

* You can only call hooks at the top level of your component or your own hooks.
* You cannot call hooks inside loops or conditions.
* You can only call hooks from React functions, and not regular JavaScript functions.

### 3.

Question 3

True or false? State data is the data inside a component that a component can mutate.

**1 point**



True



False

* + True

# **Additional resources**

Below you will find links to helpful additional resources.

* [React docs website URL which discusses the issue in depth](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html)
* [Data flows down](https://reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down)
* [The Power Of Not Mutating Data](https://reactjs.org/docs/optimizing-performance.html#the-power-of-not-mutating-data)
* [Add Inverse Data Flow](https://reactjs.org/docs/thinking-in-react.html#step-5-add-inverse-data-flow)
* [Component state](https://reactjs.org/docs/faq-state.html)
* [State: A Component's Memory](https://beta.reactjs.org/learn/state-a-components-memory)
* [Sharing State Between Components](https://beta.reactjs.org/learn/sharing-state-between-components)
* [State as a Snapshot](https://beta.reactjs.org/learn/state-as-a-snapshot)
* [Basic useState examples](https://beta.reactjs.org/apis/usestate#examples-basic)
* [Synchronizing with effects - putting it all together](https://beta.reactjs.org/learn/synchronizing-with-effects#putting-it-all-together)
* [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)
* [The event loop in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop)

## Before Single-Page Apps

Before the advent of modern JavaScript frameworks, most websites were implemented as multi-page applications. That is, when a user clicks on a link, the browser navigates to a new webpage, sends a request to the web server; this then responds with the full webpage and the new page is displayed in the browser.

This can make your application resource intensive to the Web Server. CPU time is spent rendering dynamic pages and network bandwidth is used sending entire webpages back for every request. If your website is complex, it may appear slow to your users, even slower if they have a slow or limited internet connection.

To solve this problem, many web developers develop their web applications as Single Page Applications.

## Single-Page Apps

You’re using many Single Page Applications every day. Think of your favorite social network, or online email provider, or the map application you use to find local businesses. Their excellent user experiences are driven by Single Page Applications.

A Single Page Application allows the user to interact with the website without downloading entire new webpages. Instead, it rewrites the current webpage as the user interacts with it. The outcome is that the application will feel faster and more responsive to the user.

## How Does a Single-Page App Work?

When the user navigates to the web application in the browser, the Web Server will return the necessary resources to run the application. There are two approaches to serving code and resources in Single Page Applications.

1. When the browser requests the application, return and load all necessary HTML, CSS and JavaScript immediately. This is known as bundling.
2. When the browser requests the application, return only the minimum HTML, CSS and JavaScript needed to load the application. Additional resources are downloaded as required by the application, for example, when a user navigates to a specific section of the application. This is known as lazy loading or code splitting.

Both approaches are valid and are used depending on the size, complexity and bandwidth requirements of the application. If your application is complex and has a lot of resources, your bundles will grow quite large and take a long time to download – possibly ending up slower than a traditional web application!

Once the application is loaded, all logic and changes are applied to the current webpage.

Let’s look at an example

Imagine there is a webpage that has a Label and a Button. It will display a random movie name when the button is clicked.

In a traditional website, when the button is clicked, the browser will send a POST request to the web server. The web server will return a new web page containing the button and movie name, and the web browser renders the new page.

In a Single Page Application, when the button is clicked, the browser will send a POST request to a web server. The web server will return a JSON object. The application reads the object and updates the Label with the movie name.

See, more efficient!

But what if we need to have multiple pages with different layouts in our application?

## Practical Differences Between Single-Page Apps and Multi-Page Apps

You have a web application that has a navigation bar on top and two pages. One page shows the latest news, and the other shows the current user’s profile page. The navigation bar contains a link for each page.

In a traditional website, when the user clicks the Profile link, the web browser sends the request to the web server. The web server generates the HTML page and sends it back to the web browser. The web browser then renders the new web page.

In a Single Page Application, different pages are broken into templates (or views). Each view will have HTML code containing variables that can be updated by the application.

The web browser sends the request to the web server, and the web server sends back a JSON object. The web browser then updates the web page by inserting the template with the variables replaced by the values in the JSON object.

## Anchor Tag Elements in Single-Page Elements

A single-page application can’t have regular anchor tag elements as a traditional web app can.

The reason for this is that the default behavior of an anchor tag is to load another HTML file from a server and refresh the page. This page refresh is not possible in a SPA that's powered by a library such as React because a total page refresh is not the way that a SPA works, as explained earlier in this lesson item.

Instead, a SPA comes with its own special implementation of anchor tags and links, which only give an illusion of loading different pages to the end user when in fact, they simply load different components into a single element of the real DOM into which the virtual DOM tree gets mounted and updated.

That's why navigation in a single-page app is fundamentally different from its counterpart in a multi-page app. Understanding the concepts outlined in this lesson item will make you a more well-rounded React developer.