# **1. Golden Rules**

1. **Always test your code!**
2. Use Google! **ESPECIALLY** if you don’t know or can’t remember something
   1. **DO NOT OVER-GOOGLE (do not pass page 2 of results)**
   2. **DO NOT USE CODE YOU DO NOT UNDERSTAND**
   3. **DO NOT COPY PASTE FROM GOOGLE! (read the code, understand it, and recreate it your own way)**
   4. **DO NOT USE EXAMPLES THAT USE CLASS COMPONENTS**
3. Make mistakes! **Seriously, don’t be scared of making mistakes**. The worst thing that can happen is you go back and fix it. Mistakes are a great way to learn (it’s called **trial and error** for a reason).
4. Good code is lazy and efficient! **Efficiency and laziness go hand in hand.**
5. Ask your classmates! There is no shame in not knowing something or asking for help. **Leverage the community!**
6. Hop in the queue! If you get stuck on something for 15 minutes or more, ask us.
7. Never overestimate how much time you have!
8. Don’t Panic!
9. **Don’t use terminal commands you don’t know!**
10. Static code is illegal code!

# 

# **2. React Concepts**

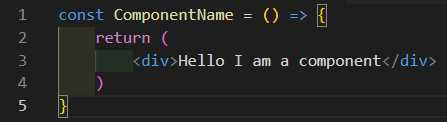
## **2.1 Components**

### **2.1.1 What is a Component?**

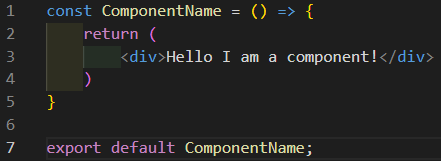
* A component is a piece of code that returns JSX onto our website .
* A component should always be in its own file. The file should have the same name as the component. Both the component’s name and the file’s name start with capital letters, and camelCasing applies.
* As a rule of thumb, all components are reusable. This is why we ensure our code is dynamic. You can code components to be single use, but this generally goes against the convention.
* A component is a function. **DO NOT USE CLASS COMPONENTS!**

### **2.1.2 What Does it Look Like?**

This is the most basic skeleton of a component:



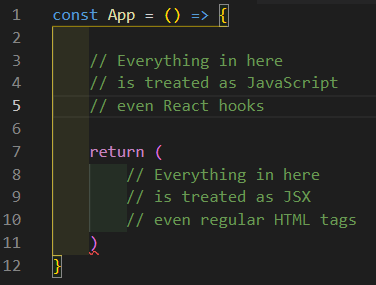
This is a basic component file:



### **2.1.3 JS vs JSX**

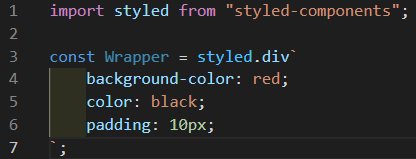
A component can, and usually does, contain both JS and JSX. The division is pretty simple:

* JS: anything **ABOVE** the component’s **return** statement is treated as pure JS
* JSX: anything **INSIDE** the component’s **return** statement is JSX

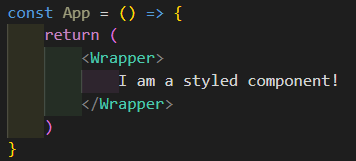


### **2.1.4 Styled Components**

A styled component is a customized piece of code with styles applied to it. To put it simply, it’s HTML and CSS mixed into one convenient package.



In the screenshot above, we can see a Wrapper component that is in fact just a regular HTML div with CSS styled applied to it. When we put Wrapper in the code, the styles will be applied onto it in the browser.



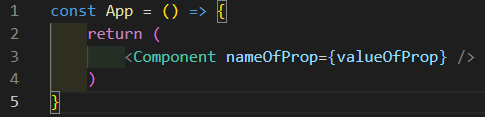
## **2.2 Props**

### **2.2.1 What is a Prop?**

* A prop is a unique variable that is passed from a parent component to a child component. The word "prop" is short for "property".
* A component can contain an infinite number of props.
* A prop is a custom HTML attribute. We can name it whatever we want.
* A prop is declared in the tags of the child component being called.
* The prop’s name is set on the left side of the = and the value is set on the right side. The value of a prop is not always a JavaScript variable, but when it is it must be encased in curly braces {}.
* The name that the prop is given in the parent is the name it will have in the child component.

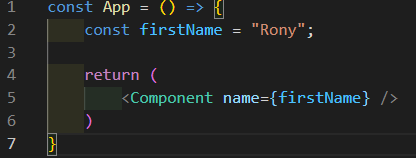
### **2.2.2 Passing Down a Prop**

A very basic example of a prop being passed down from <App /> (the parent) to <Component /> (the child):



A more concrete example:

* Here we can see that a variable called firstName containing the string "Rony" is being passed down from <App /> to <Component />
* In <Component />, the prop is being called "name"



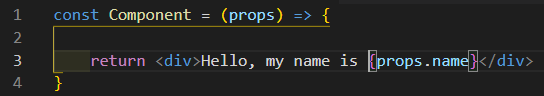
### **2.2.3 Receiving a Prop**

There are 2 ways of receiving properties: (building on the screenshots above)

1. Using the umbrella term props.
2. Destructuring the keys from props.

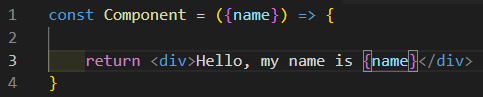
#### **2.2.3.1 Using the Term props**

The term props is an object that contains all of the individual properties passed down from parent to child in the form of key:value pairs. To access a property, you call it as a key to the props object (using dot notation or bracket notation).



#### **2.2.3.2 Destructuring the Prop**

An alternative method (and generally considered better/cleaner) to receiving the properties is to destructure them directly in the function parameters. This allows us to avoid the use of the term props entirely.



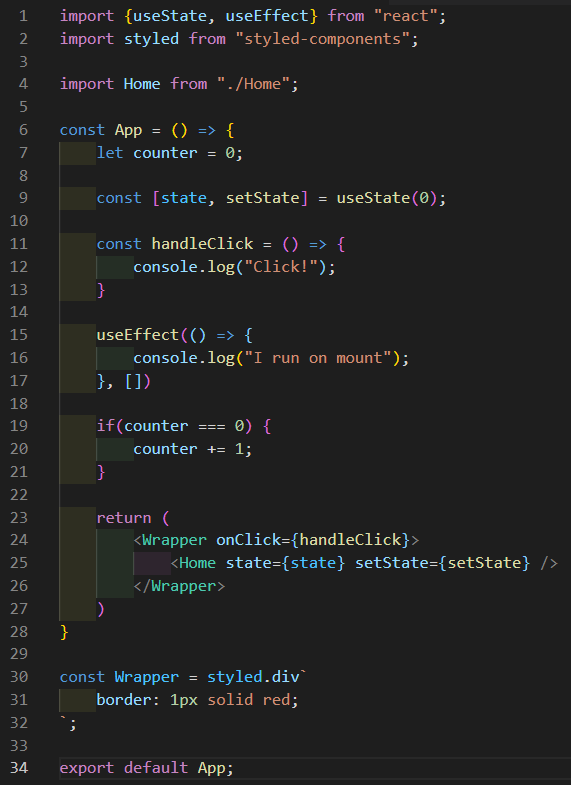
## **2.3 Order of Declaration**

### **2.3.1 What is it for?**

The order of declaration in React, like in every language, is important and is helpful for keeping your code clear, clean, organized and readable. Just like in the JavaScript cheat sheet, specific blocks should be grouped together.

### **2.3.2 The order**

1. Import libraries
2. Import components
3. Create component function
   1. Regular variables
   2. Variables that use hooks (useState, useParams, etc.)
   3. Functions
   4. useEffect
   5. Regular code
   6. Return statement
4. Styled components
5. [Export statement](#_j3mq4l81a8fd)



## **2.8 URL Navigation**

React based websites are a single page application (S.P.A.), meaning they only actually have a single page. Almost all website pages are html files. In React we just have the one: index.html. However React allows us to simulate URL navigation by using URL-based component rendering (rendering a component based on the URL).

### **2.8.1 Using <Route>**

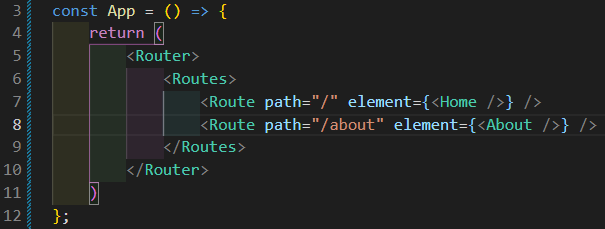
The <Route> component lets us render specific components on our page based on the URL of the site. <Route> should only be rendered inside of a <Routes>, which in turn should only be rendered inside a <BrowserRouter>, often renamed <Router>. The Route component has multiple props, but the ones we care about are:

1. path
2. element

The <Route> component can also [receive URL Parameters](https://docs.google.com/document/d/1cNnWOKo7j7JppHCdMXsC0T5R_hW9a1J9qCPnXXYNEw4/edit#heading=h.cvf4mc9gu0tp).

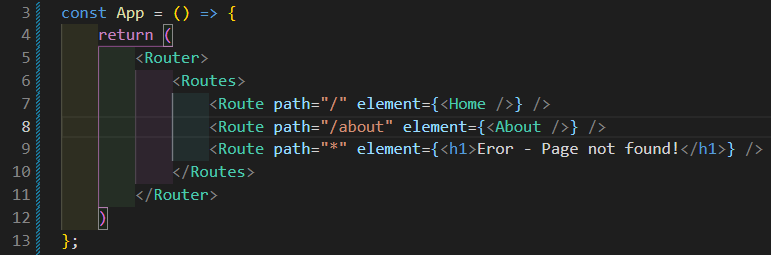
#### **2.8.1.1 The path Property**

The path prop works almost like an event listener for the URL of the website. If the URL matches the path of the <Route>, the <Route> will be triggered and render its contents on the page.



#### **2.8.1.2 The element Property**

The element prop is what the <Route> will render once it has been triggered. It can contain simple HTML, or [entire components](#_ffpnucahyu28).



The "\*" means everything; the <Route> with a path of "\*" is set up to trigger on any URL that doesn’t match the preceding routes. In this example instead of showing a component it shows a simple <h1> tag.

The other routes will trigger on specific paths and will render the component inside the element prop.

#### **2.8.1.3 Receiving URL Parameters**

The <Route> component can receive something called a URL parameter. A URL parameter is literally just a variable that is stored in the URL. It is an excellent way for 2 non-related components (components that are not parent-child) to communicate information with each other. The way to define a URL parameter is inside the path of the <Route> by placing a : in front of a word. That word will become the name of the URL parameter.

As you can see here, the parameter will be called itemId. This variable will be available in the <ItemDetails /> component only through the URL. In order to grab it, we must make use of the [useParams() hook](#_l2sd4txsjjgr).

##### 

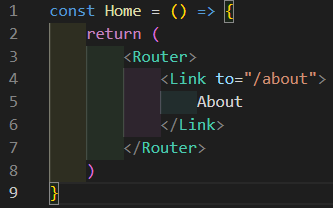
### **2.8.2 Using <Link>**

The <Link> component is, for all intents and purposes, an HTML <a> tag that is modified to be compatible with React. Rather than loading a new page, the <Link> component changes the URL of the website when clicked. The change in URL is what triggers the <Route> components to render their children. Just like the <Route> component, <Link> can only be created inside a <BrowserRouter>, often renamed <Router>. <Link> can take many properties, but we only care about the to property. The <Link> component can also [send URL Parameters](https://docs.google.com/document/d/1cNnWOKo7j7JppHCdMXsC0T5R_hW9a1J9qCPnXXYNEw4/edit#heading=h.1f9guvbogwb4).

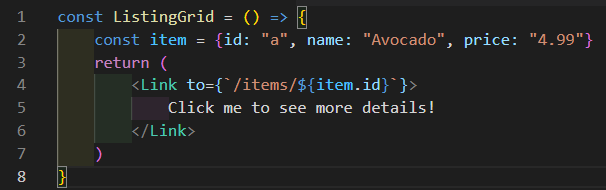
#### **2.8.2.1 The to Property**

The to property is very much the only one that matters (for our purposes) when using the <Link> component. When the link is activated by being clicked, the to property is what triggers the change of the URL.

In this example, we change the URL from "http://localhost:3000/" to "http://localhost:3000/about".



#### **2.8.2.2 Sending URL Parameters**

The <Link> component can send something called a URL parameter. A URL parameter is literally just a variable that is stored in the URL. It is an excellent way for 2 non-related components (components that are not parent-child) to communicate information with each other. The way to define a URL parameter is inside the to of the <Link> by placing a variable inside the string. For us to be able to put a variable inside a string, we need to use backticks interpolation `${variable}`. Because backticks are a JavaScript element, we need to also wrap them in curly braces {}.

## **2.5 Rendering**

In React, rendering just means to put something on our web page. **WE DO NOT USE THE render() FUNCTION!**

### **2.5.1 Rules of rendering**

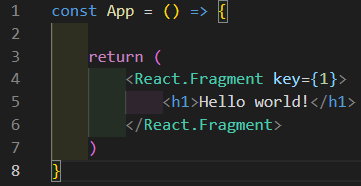
In order for things to render correctly in React, the components and HTML tags inside the **return** statement **MUST HAVE A SINGLE PARENT!** The simplest way to do this is to either use a <div> tag or a React fragment. React fragments come in 2 forms:

* <React.Fragment> </React.Fragment>
* <> </>

#### **2.5.1.1 <React.Fragment>**

The full version of the **<React.Fragment>** allows you to give a single parent to your elements and also allows the use of the [key property](#_nzprhb4c49ld) if needed.

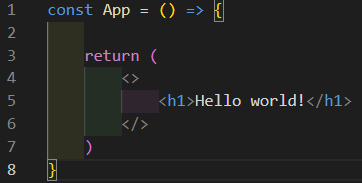
* Pros: allows the use of the key prop
* Cons: longer to write



#### **2.5.1.2 <>**

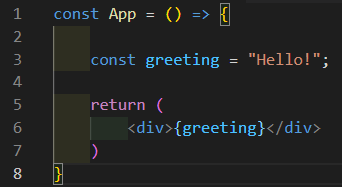
The short version of the **<React.Fragment>** allows you to give a single parent to all your elements but does not allow the use of the [key property](#_nzprhb4c49ld).

* Pros: shorter/lazier to write
* Cons: does not allow the use of the key prop



### **2.5.2 Rendering with return**

All components render things on our website through the **return** statement. Anything that is inside this **return** is treated as JSX. React cannot recognize pure JavaScript. This is why we use the curly braces {} to let React know that we are inserting JavaScript into its **return** statement.



### **2.5.3 Rendering with map**

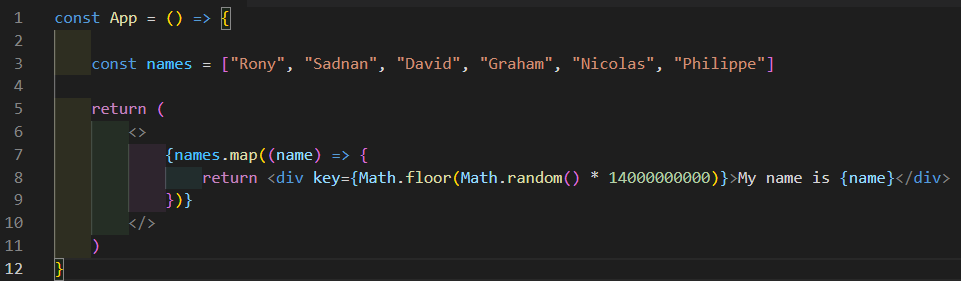
In React, we want our code to be as dynamic as possible. Most of the time we will be dealing with arrays that contain objects [{key:value}, {key:value}, {key:value}] and we want to render each object through a single dynamic component. To do this, React gives special powers to the map method. In JS, the map method returns a brand new array; in React, the map method returns JSX onto our website.

#### **2.5.3.1 The key Prop**

What the map does is clone whatever JSX it returns onto the page array.length number of times. When it does this, React needs a way to identify each JSX element in a unique way. This is what the key prop does. **Every element** generated by the map, whether it’s a React component or an HTML tag **MUST** have a key property. The key is the unique identifier for each element generated by the map. As a rule of thumb, we want to avoid using the index parameter generated by the map function because if we have more than one map on the page, you will have identical keys and React does not like this.

The best 2 valid options are:

1. Using the id provided in the object.
2. If no id is provided, use a very large random number.
   1. I set it to 14 billion because the odds are the same as winning the lottery.
   2. If you ever get a key warning when setting it to 14 billion, go buy a ticket
   3. When you win don’t forget who gave you the tip 😉

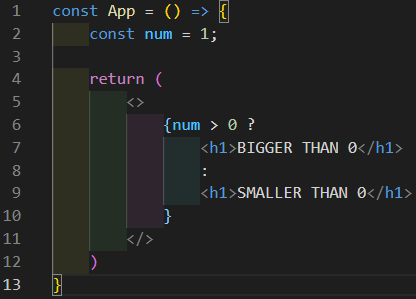


### **2.5.4 Conditional Rendering**

In React, we can render elements on our page based on a condition. This is known as conditional rendering. **It is essential when rendering state variables on the page. Also, conditional rendering only takes one condition!**

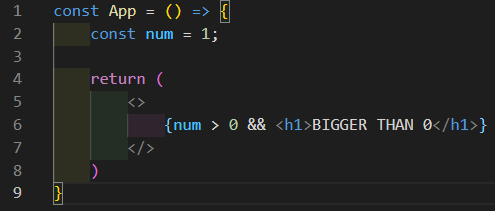
#### **2.5.4.1 The Ternary Operator**

The [ternary](https://docs.google.com/document/d/1KURc9AZOmHga6UD4mCsNAW55r3x8es_x9L3kwdi0kQ0/edit#heading=h.pxpeyajhc3v8) operator ?, similar to map, works differently in React. It renders things based on a condition and can render based on if the condition is true or false.



#### **2.5.4.2 The && Operator**

Just like the other operators above, the [and](https://docs.google.com/document/d/1KURc9AZOmHga6UD4mCsNAW55r3x8es_x9L3kwdi0kQ0/edit#heading=h.o1evf840xgvd) operator && has its own rules in React. It renders things based on a condition and can only render if the condition is true.



## **2.6 Hooks**

React hooks are incredibly powerful tools that allow us to "hook" the inner functionality of React to access, set, and manipulate them to fit our needs.

### **2.6.1 Rules of Hooks**

All hooks must follow these rules:

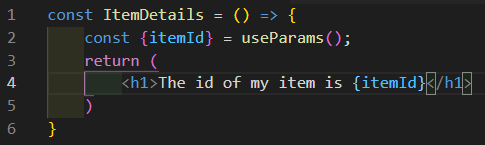
1. A hook can only be declared inside of a function component
   1. An exception can be made for [custom hooks](#_m10umdeb6feu)
2. A hook can never be created inside of a condition (ex: **if** statement)
3. A hook starts with use

### **2.6.2 useParams()**

The useParams() hook lets us grab variables from the URL and use it inside our component. To do this, we must use [destructuring](https://docs.google.com/document/d/1KURc9AZOmHga6UD4mCsNAW55r3x8es_x9L3kwdi0kQ0/edit#heading=h.62j1k1b9ohsd) on the hook itself. The same rules apply because the useParams() hook is an object that contains the name of the URL parameter as the key, and the contents of the parameter as the value.

In this example, useParams() is equal to {itemId: "a"}.

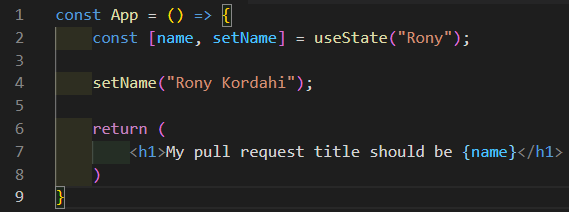
* Based on screenshots [#1](https://docs.google.com/document/d/1cNnWOKo7j7JppHCdMXsC0T5R_hW9a1J9qCPnXXYNEw4/edit?usp=sharing#heading=h.v4ika8blq2p1) and [#2](https://docs.google.com/document/d/1cNnWOKo7j7JppHCdMXsC0T5R_hW9a1J9qCPnXXYNEw4/edit?usp=sharing#heading=h.gq1snz9pmlq).



### **2.6.3 useState()**

The useState() hook lets us create semi-persistent variables. What this means is that the value of the variable will not get reset if the component is re-rendered. A component is re-rendered when the value of a state variable is changed. As long as our component is on the page, the value of the state variable will never be lost. **A refresh is not a re-render! Refreshing will clear all the values on the page.**

* State variables are created in pairs, along with a setState function.
  + The name of the setState function will depend on the name of the state variable
* A state variable can be created with an initial value
* A state variable cannot be modified through the use of the assignment operator =.
* **THE ONLY THING THAT CAN MODIFY THE VALUE OF A STATE VARIABLE IS ITS SETSTATE FUNCTION!**

****

### **2.6.4 useEffect()**

The useEffect() hook lets us execute a piece of code when

* On component mount (on load)
* A specific variable’s value has changed
* When any state variable’s value changes
* On component unmount (unload)

The useEffect() hook takes 2 parameters:

1. An anonymous callback function (function with no name)
2. A dependency array

The useEffect() hook also has an unsubscribing function which is used for cleanup

#### **2.6.4.1 Parameter: Callback Function**

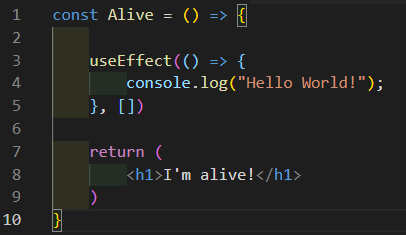
The anonymous callback function is simply an arrow function () => {} that is executed when the useEffect() hook is triggered.

#### **2.6.4.2 Parameter: Dependency Array**

The dependency array is an array of variables that the useEffect() hook observes. The array can hold an infinite number of variables to observe. When the value of one of the variables inside this array changes, the useEffect() hook is triggered. When the hook is triggered, the callback function is executed.

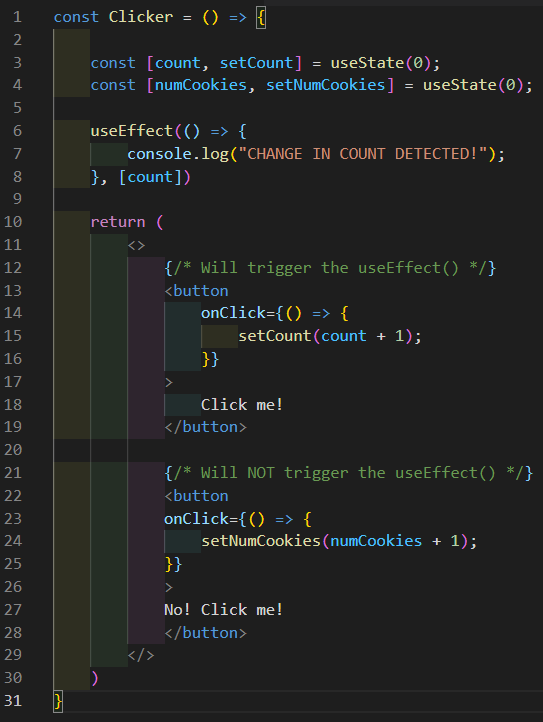
#### **2.6.4.3 On Mount**

To make sure useEffect() only happens once on mount and never again, we keep the dependency array empty.



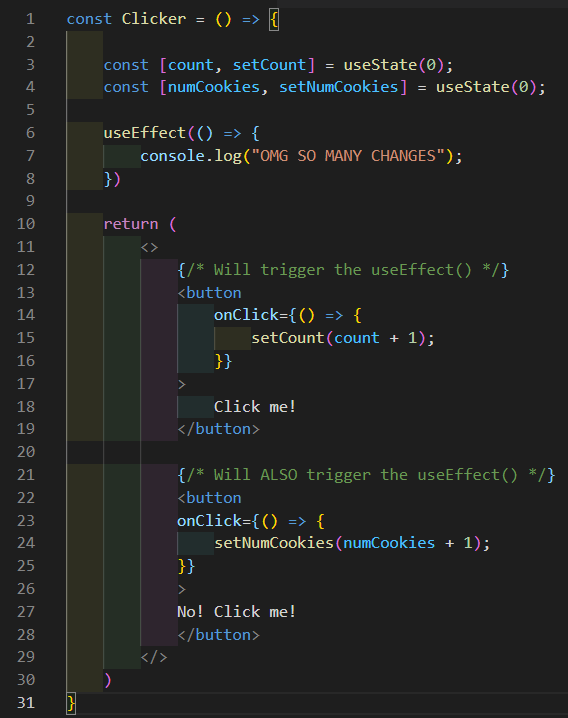
#### **2.6.4.4 On Variable Change**

To make useEffect() trigger every time the **value of a variable changes**, we need to put that variable inside the dependency array of the useEffect().



#### **2.6.4.5 On Any Change**

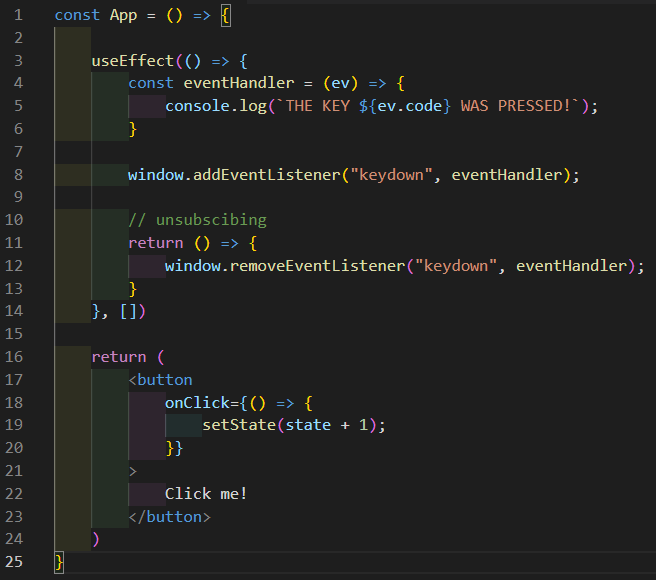
To make useEffect() trigger every time the **value of any state variable changes**, we do not put the dependency array of the useEffect().



#### **2.6.4.6 Unsubscribing (on unmount)**

The useEffect() hook’s unsubscribing function is used for cleaning up most of the time. We can also use it to run a final piece of code, to change the value of a useContext() or useReducer() if it is needed, but this is a very rare case. Also we haven’t seen those hooks yet.

* To unsubscribe, we must use a **return** statement followed by an anonymous arrow function () => {}
* The code inside the arrow function is executed only when the component is unmounted (removed from the page)



#### **2.6.4.7 Warning**

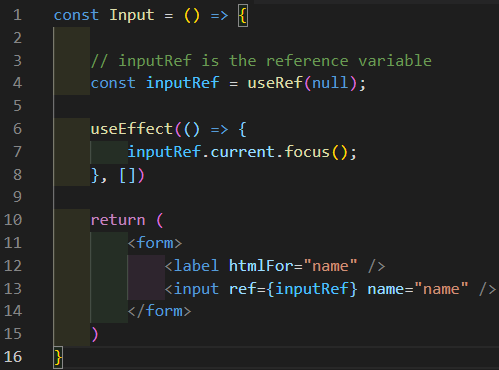
The useEffect() hook can cause infinite loops if you have a state variable in its dependency array and you are modifying that state variable in the useEffect(). Infinite loops are bad and must be avoided!

**ESPECIALLY** if you are using a fetch(). If you run an infinite loop while using a fetch(), you are committing a cyber-attack and risk a hefty fine and/or legal action. Please be careful and only set your fetch() inside a useEffect() with an empty dependency array!

### **2.6.5 useRef()**

The useRef() hook lets us focus on a specific element on the page. Focusing on an element means to have it selected. Most of the time, we want to focus on our element when the component is loaded (on mount). To do this, we run the code to focus on the element inside a useEffect() set up to only run on mount.

* When initiating a reference variable, we set the initial value to **null**.
* You can set any initial value that you want. It doesn’t really seem to do anything so **null** is used by convention.
* To set the reference variable, the element takes a special property called ref that is reserved for the useRef() hook.
* Inside the useEffect(), we execute our reference command ref.current.focus().



### **2.6.6 useContext()**

One of the most powerful features of React is that we can import anything we want inside a specific component when it is needed. The useContext() hook is a great tool that allows us to make any variable, function, or state available globally across our application from one convenient location. The useContext() is a bit unique because it makes use of 2 very special properties:

1. {children}
2. value

###### and has several steps that are required in order to be able to set it up and use it:

1. Create the useContext().
2. Create the Context.Provider component.
3. Initiate the variables that your Context.Provider will export.
4. Export the variables in your Context.Provider.
5. Export the Context.Provider and the context variable.
6. In the index.js file, import your Context.Provider and wrap it around the <App /> component.
7. Import and consume the context (import the variables) in the components that require it.

#### **2.6.6.1 The {children} prop**

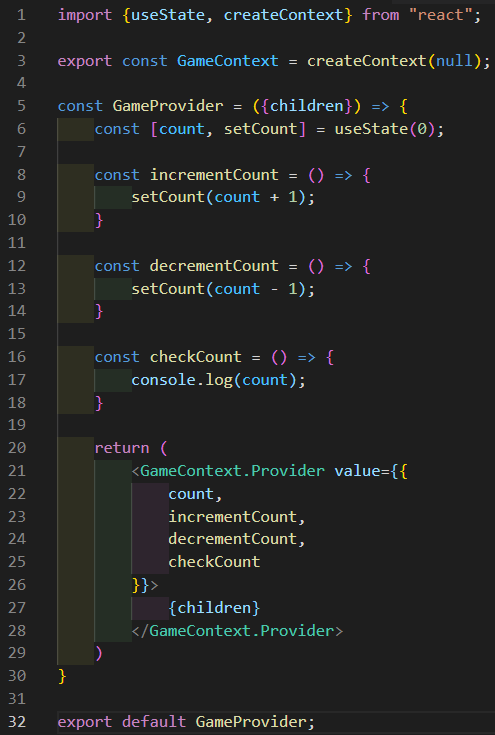
The {children} prop is a special prop in React. Similar to how the props property is an umbrella term for all properties, {children} is an umbrella term for all children components contained inside tags. The useContext() hook makes use of this special prop to allow all the components inside our application to have access to the exported variables.

#### **2.6.6.2 The value prop**

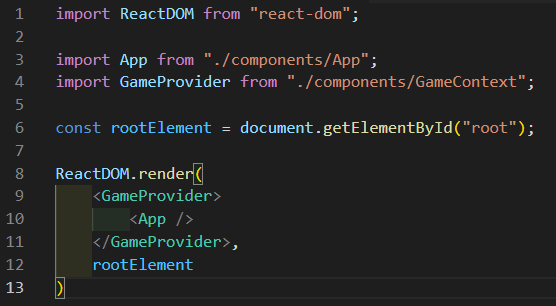
The value prop is a special prop that allows the Context.Provider to export all of its variables to the rest of the application. The value prop is what the components are accessing when they consume the context.

Inside the GameContext.js file, we follow steps 1 through 5 (listed [above](#_l2mi9zw7yj5k)):

* By convention, we generally do a named export on the context variable, and a default export on the provider component. It doesn’t matter really, as long as there’s only one default export in the file.



Inside the index.js file, we do step 6 (listed [above](#_l2mi9zw7yj5k)):



Inside the component that requires the variables and functions provided by the Context.Provider, we do step 7 (listed [above](#_l2mi9zw7yj5k)):

* Notice that we did not import the count state variable because we don’t need it in this component.



### **2.6.7 useReducer()**

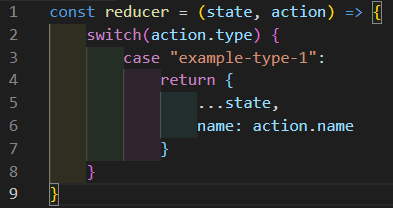
The useReducer() hook is basically a more complex but powerful version of the useState() hook because it gives us full control over the setState function. The useReducer() hook also goes hand in hand with the useContext() hook, making our powerful state available globally in our entire application. When creating a state through useReducer(), an initial value of state and of setState must be provided.

#### **2.6.7.1 The reducer function**

The reducer function is our custom setState function for the userReducer() hook. We call it reducer by convention. It receives 2 parameters:

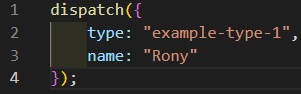
1. state
2. action

The state parameter is received automatically, and it is our state variable that we have created with the useReducer() hook. The action parameter is an object that contains all the necessary information required to modify our state variable. Typically it must receive a key called type and it can receive extra data. Inside the reducer function, we make use of a **switch** to find out what type of action (action.type) we’re executing. When we find our **case**, the state will be modified accordingly through the **return** statement.



#### **2.6.7.2 The dispatch function**

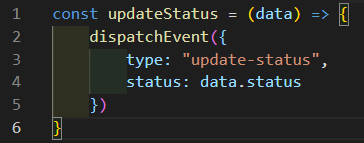
The dispatch function is very unique. Its sole purpose is to call the reducer function and to give it the action object. To do this, we call our dispatch function and put a brand new object inside it as an argument, making sure we never forget the type key. We can also provide additional data inside this object.



#### **2.6.7.3 Dispatching functions**

Dispatching functions are simple functions whose sole purpose is to call the dispatch function. They are very useful for a few reasons:

1. Since useReducer() is almost always used with useContext(), it’s better to export the dispatching functions rather than the dispatch function.
2. They can receive data and pass it along to the dispatch function.
3. They are very reusable.



#### **2.6.7.4 Particularities**

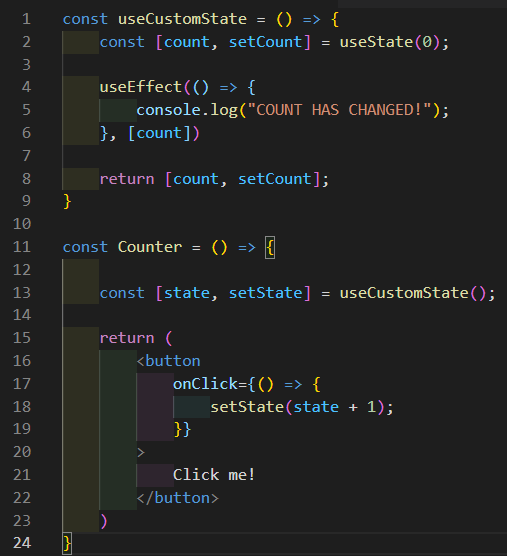
It is important to know that when using the useReducer() hook, the initial values get assigned backwards. The reducer function goes inside dispatch, and the initialState (a variable that contains the initial values of our state) variable goes inside state.



### **2.6.8 Custom hooks**

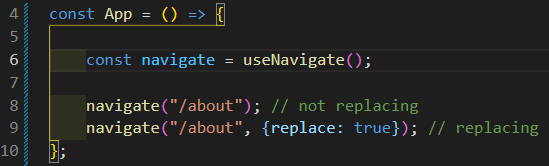
Custom hooks are **THE ONLY** exception to [rule #1](#_t3tw8srz4sd7) of React hooks. Because React is generous, it lets us create our own hooks. These custom hooks allow us to recycle and reuse the logic of regular hooks to fit our own needs. Custom hooks follow a very specific set of rules:

1. Must be a function.
2. The name must start with the word use.
3. Must use at least one React hook.
4. Can return React hooks. **(this is the only non mandatory rule)**
5. The function must be called inside a component.
6. The function cannot be called in a condition (**if** statement).

****

### **2.6.9 useNavigate()**

The useNavigate() hook comes from "react-router-dom", not from "react". It allows you to change the current URL the user is on. It can take a second parameter which allows you to replace the current entry with the new one in the browser’s history.



## **2.7 Importing and Exporting**

Exporting is one of the main features of React. When something is exported from one file, we can import it into another when it is needed. Importing and exporting only works on functions, components, and data objects (objects and arrays). If you are trying to export state variables or other hooks you must use the useContext() hook. There are 2 types of import and export methods:

1. Default import/export
2. Named import/export

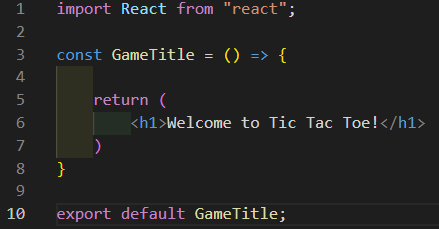
Both methods will make use of the **export** keyword, and have their own set of rules for renaming imported elements. We don’t usually rename what we import unless absolutely necessary.

### **2.7.1 Default Import/Export**

Default import/export is usually used for importing and exporting components from one file to another. It can also be used on data objects. **We can only have one default export per file! A named export always goes at the bottom of the file.**

#### **2.7.1.1 Exporting**

To do a default export, we simply use the code **export** **default** followed by the name of what we’re exporting.

Default export used with a component:

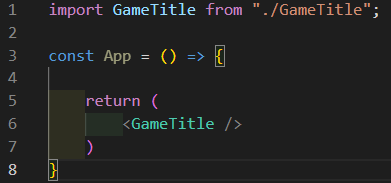
Default export used with a data object:



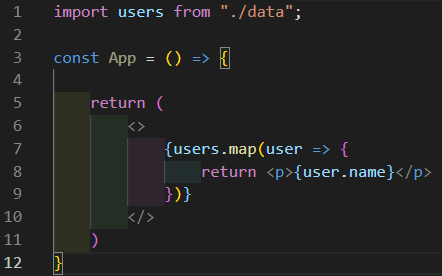
#### **2.7.1.2 Importing**

To do a default import, we simply need to give a name to the imported element from the destination file. It is important to know that when we do a default import, we are only importing what the file is exporting as a default export.

Default import used on a component. Here we are calling the element GameTitle:

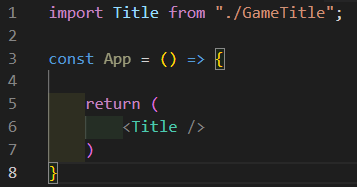


Default import used on a data object. Here we are calling the element users:



#### **2.7.1.3 Renaming**

To rename a default import, all we need to do is give it a different name on the **import** line. **The path does not change**:





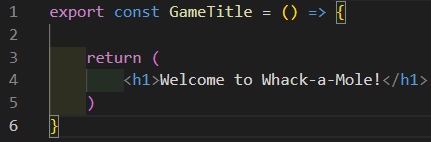
### **2.7.2 Named Import/Export**

Named import/export is usually used for importing and exporting data objects or utility functions. It can also be used on components. It is also generally used when a file has multiple exports. **A file can have an infinite number of named exports! A named export always goes on the same line as the declaration.**

#### **2.7.2.1 Exporting**

To do a named export, we simply put the word **export** in front of what we’re exporting.

Named export used with a component:



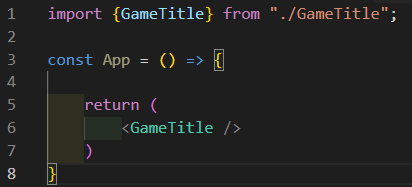
Named export used with a data object:



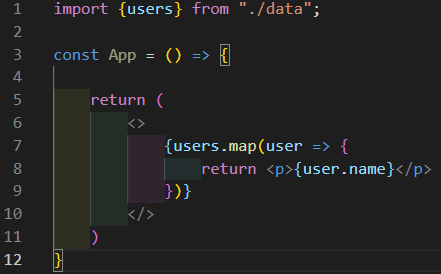
#### **2.7.2.2 Importing**

To do a named import, we need to [destructure](https://docs.google.com/document/d/1KURc9AZOmHga6UD4mCsNAW55r3x8es_x9L3kwdi0kQ0/edit#heading=h.62j1k1b9ohsd) the imported element from the destination file. It is important that the names match because you can have multiple elements exported from the same file.

Named import used on a component. Here we are calling the element GameTitle:



Named import used on a data object. Here we are calling the element users:



#### **2.7.2.3 Renaming**

To rename a named import, we need to put the name of the imported element, followed by the **as** keyword, and finally its new name . **The path does not change**:

