**React Hooks**

React.js is an open-source JavaScript-based user interface library. It is hugely popular for web and mobile app development.

React follows the principle of component-based architecture. A component in React is an isolated and reusable piece of code. The components can be of two types – class components and functional components.

Before React version 16.8, developers could handle state and other React features only using class components. But with version 16.8, React introduced a new pattern called Hooks.

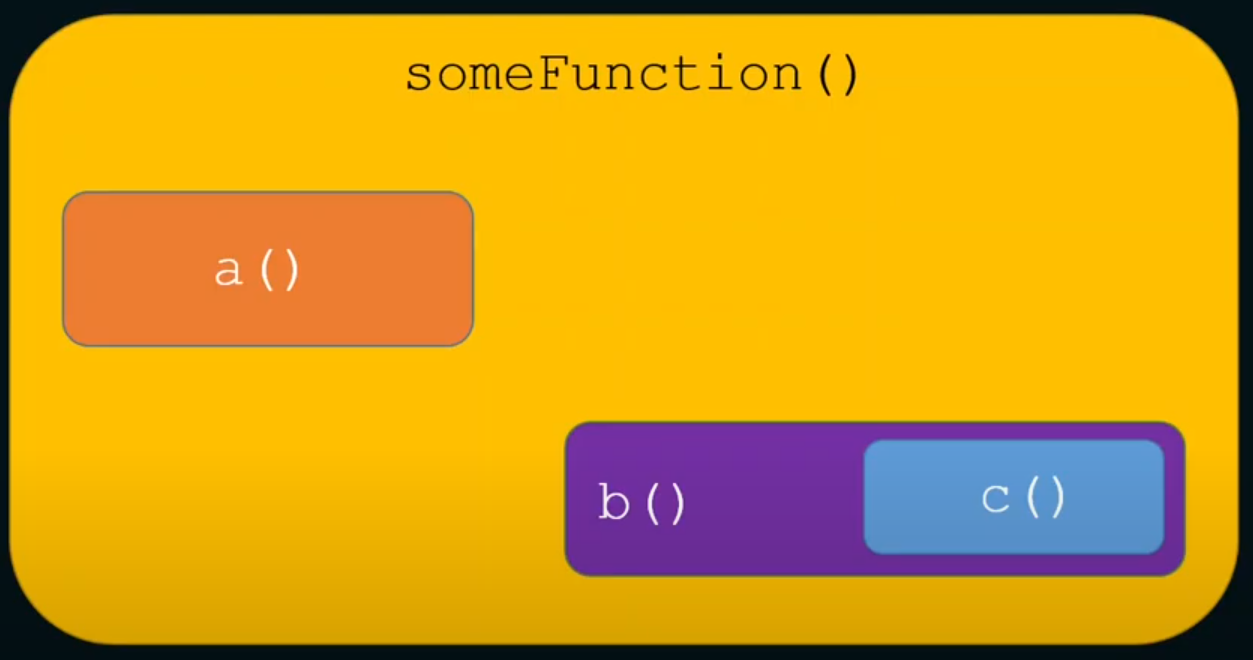
With React Hooks, we can use state, and other React features, in a functional component. It empowers devs to do functional programming in React.

## Before You Learn About Hooks...

Before you think of hooks, think of plain-old (aka vanilla) JavaScript functions.

In the JavaScript programming language, functions are reusable code logic to perform repeated tasks. Functions are composable. This means you can invoke a function in another function and use its output.

In the image below, the someFunction() function composes (uses) functions a() and b(). The b() function uses the function c().

Function Composability

If we write this in code, it will be like this:

function a() {

// some code

}

function c() {

// some code

}

function b() {

// some code

c();

// some code

}

function someFunction() {

// some code

a();

b();

// some code

}

It is not a secret that functional components in React are just plain old JavaScript functions! So if functions have composability, React components can also have composability. This means we can use (compose) one or more components into another component as shown in the image below:

Components Composability

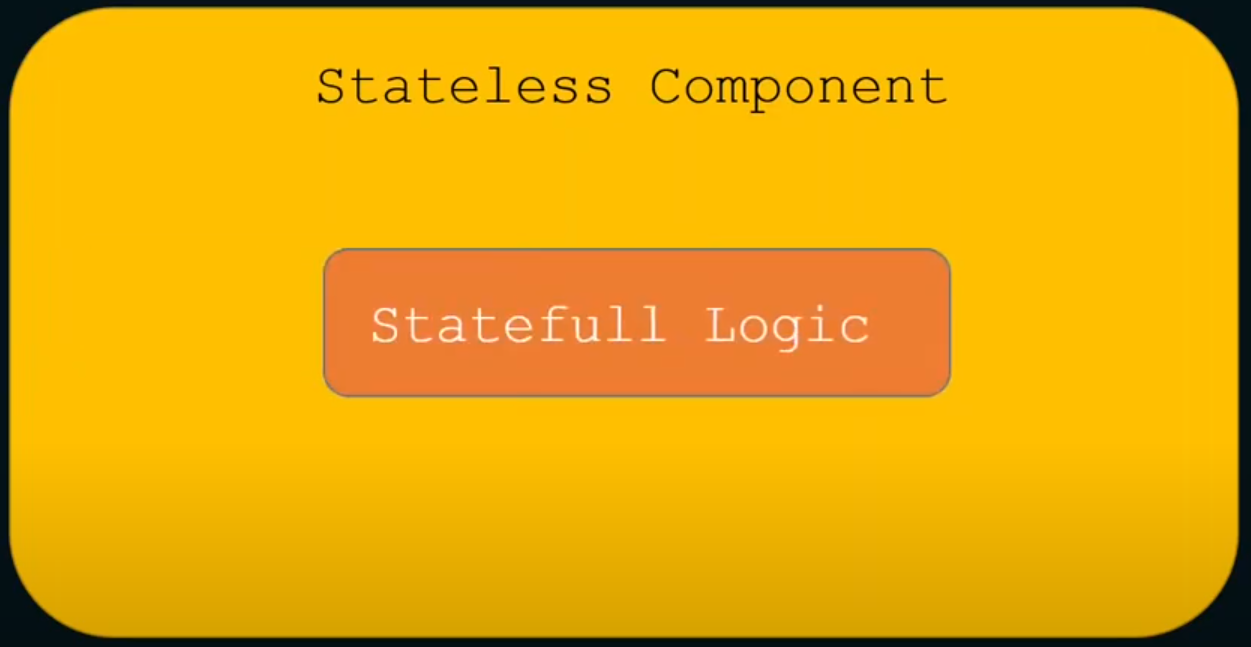
## Stateful vs. Stateless Components

Components in React can be stateful or stateless.

* A stateful component declares and manages local state in it.
* A stateless component is a pure function that doesn't have a local state and side-effects to manage.

A [pure function](https://blog.greenroots.info/what-are-pure-functions-and-side-effects-in-javascript) is a function without any side-effects. This means that a function always returns the same output for the same input.

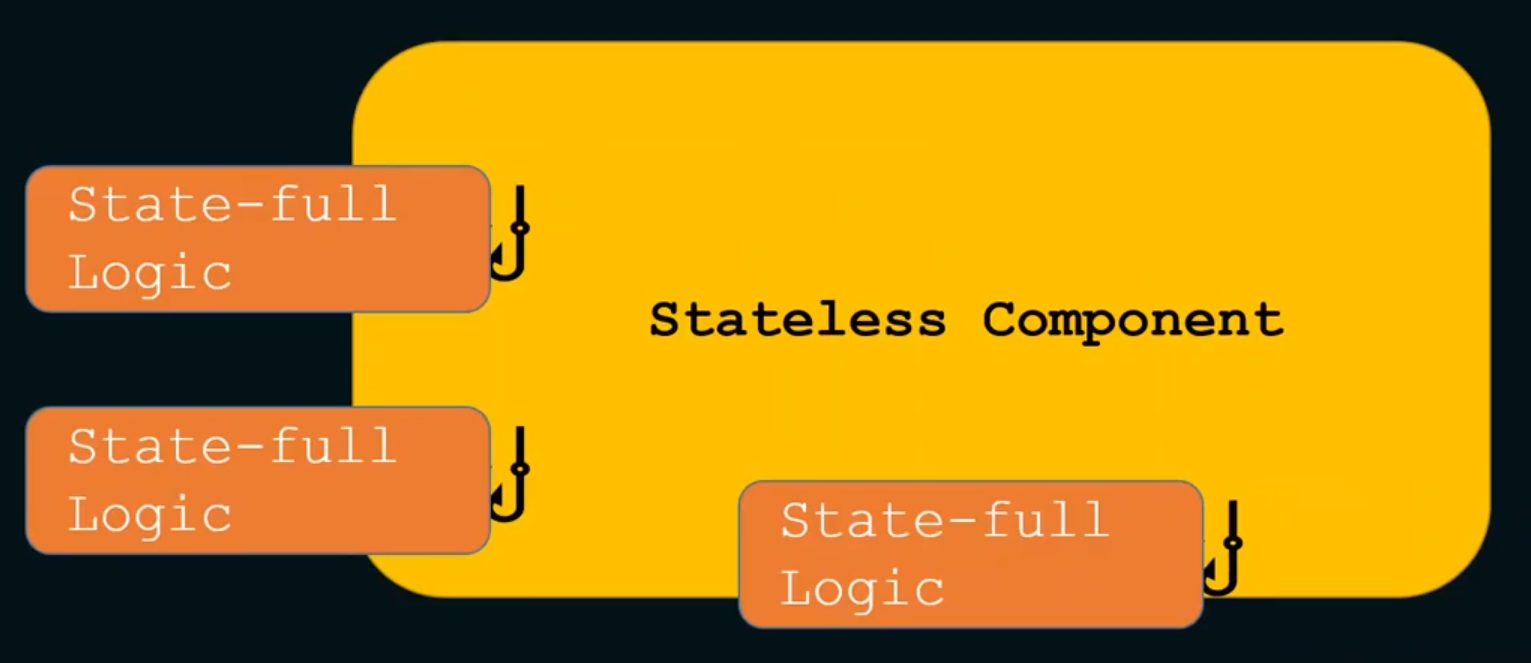
If we take out the stateful and side-effects logic from a functional component, we have a stateless component. Also, the stateful and side-effects logic can be reusable elsewhere in the app. So it makes sense to isolate them from a component as much as possible.

Stateful Component as the component has Stateful Logic

## React Hooks and Stateful Logic

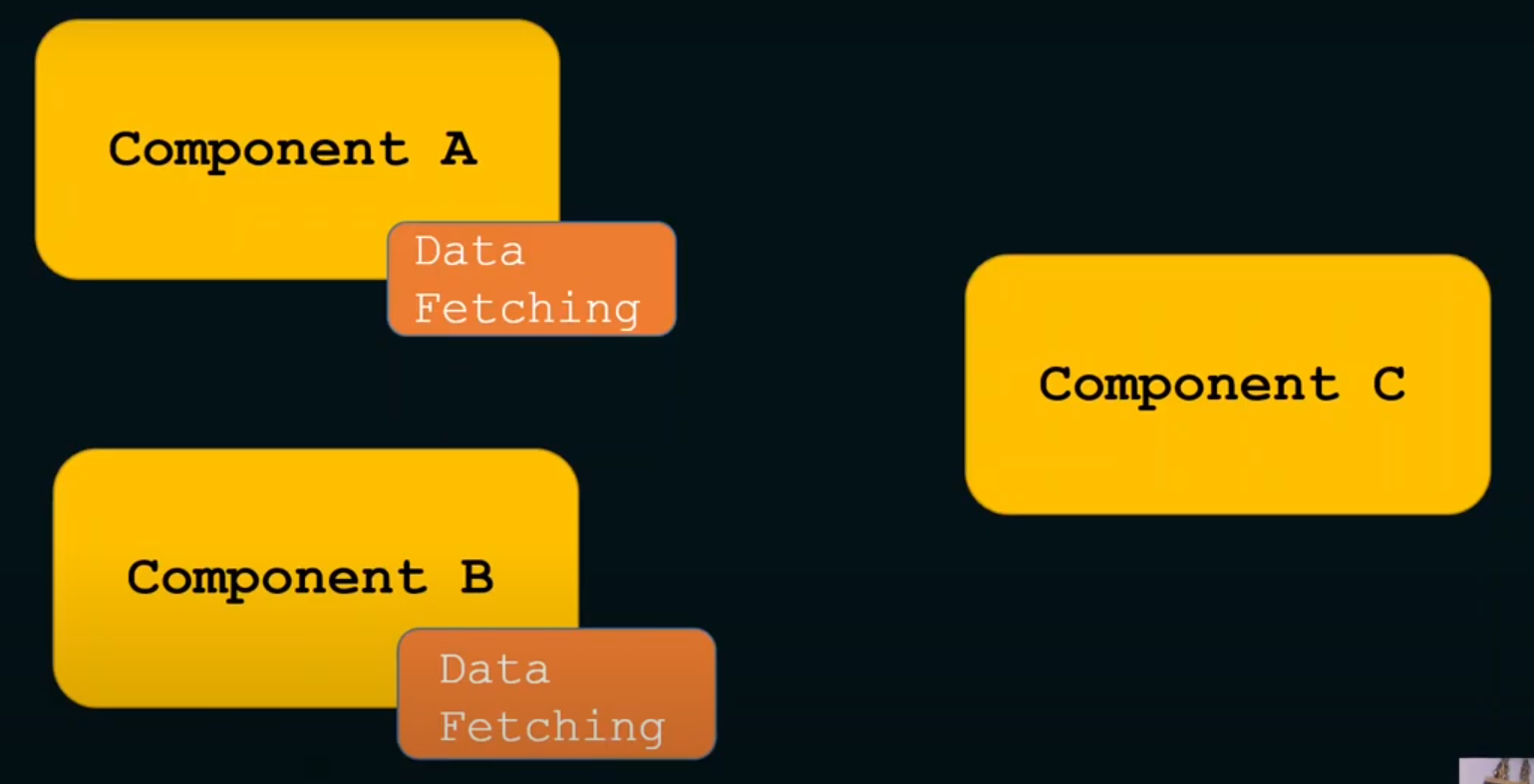
With React Hooks, we can isolate stateful logic and side-effects from a functional component. Hooks are JavaScript functions that manage the state's behaviour and side effects by isolating them from a component.

So, we can now isolate all the stateful logic in hooks and use (compose them, as hooks are functions, too) into the components.

Isolated Stateful Logic into Hooks

The question is, what is this stateful logic? It can be anything that needs to declare and manage a state variable locally.

For example, the logic to fetch data and manage the data in a local variable is stateful. We may also want to reuse the fetching logic in multiple components.



## So, What Exactly Are React Hooks?

So, how can we define React Hooks in plain English? Now that we understand functions, composability, components, states, and side-effects, here goes a definition of React Hooks:

React Hooks are simple JavaScript functions that we can use to isolate the reusable part from a functional component. Hooks can be stateful and can manage side-effects.

React provides a bunch of standard in-built hooks:

* useState: To manage states. Returns a stateful value and an updater function to update it.
* useEffect: To manage side-effects like API calls, subscriptions, timers, mutations, and more.
* useContext: To return the current value for a context.
* useReducer: A useState alternative to help with complex state management.
* useCallback: It returns a memorized version of a callback to help a child component not re-render unnecessarily.
* useMemo: It returns a memoized value that helps in performance optimizations.
* useRef: It returns a ref object with a .current property. The ref object is mutable. It is mainly used to access a child component imperatively.
* useLayoutEffect: It fires at the end of all DOM mutations. It's best to use useEffect as much as possible over this one as the useLayoutEffect fires synchronously.
* useDebugValue: Helps to display a label in React DevTools for custom hooks.

Note:

**useEffect** is a React **hook** for side effects. Side effects or effects are the operations that can affect other components and can’t be done during rendering. That’s why **useEffect** accepts a callback function that will run every time there’s a re-render in the component’s state by default. But we can set a dependency array to affect just when our component’s intended state changed, we can put the dependency array as the second argument.

**useEffect** runs once our component mounts, which means we’re telling React to run our **effect** function after applying the changes to the DOM.

Where do we need **useEffect**?

The most popular use case for useEffect is when we need to fetch data from an API.