# **React Fundamentals**

React Native runs on React, a popular open source library for building user interfaces with JavaScript. To make the most of React Native, it helps to understand React itself. This section can get you started or can serve as a refresher course.

We're going to cover the core concepts behind React:

- components
- JSX
- props
- state

If you want to dig deeper, we encourage you to check out React's official documentation.

## Your first component

The rest of this introduction to React uses cats in its examples: friendly, approachable creatures that need names and a cafe to work in. Here is your very first Cat component:

```
Your Cat
                                                                                               ∧ Expo
import React from 'react';
import {Text} from 'react-native';
const Cat = () => {
 return <Text>Hello, I am your cat!</Text>;
};
export default Cat;
                                                            Preview
                                                                            My Device
                                                                                      iOS Android
                                                                                                    Web
```

Here is how you do it: To define your Cat component, first use JavaScript's <u>import</u> to import React and React Native's Text Core Component:

```
import React from 'react';
import {Text} from 'react-native';
```

Your component starts as a function:

```
const Cat = () => {};
```

You can think of components as blueprints. Whatever a function component returns is rendered as a **React element.** React elements let you describe what you want to see on the screen.

Here the Cat component will render a <Text> element:

```
const Cat = () => {
  return <Text>Hello, I am your cat!</Text>;
};
```

You can export your function component with JavaScript's export default for use throughout your app like so:

```
const Cat = () => {
  return <Text>Hello, I am your cat!</Text>;
};
export default Cat;
```

This is one of many ways to export your component. This kind of export works well with the Snack Player. However, depending on your app's file structure, you might need to use a different convention. This handy cheatsheet on JavaScript imports and exports can help.

Now take a closer look at that return statement. <Text>Hello, I am your cat!</Text> is using a kind of JavaScript syntax that makes writing elements convenient: JSX.

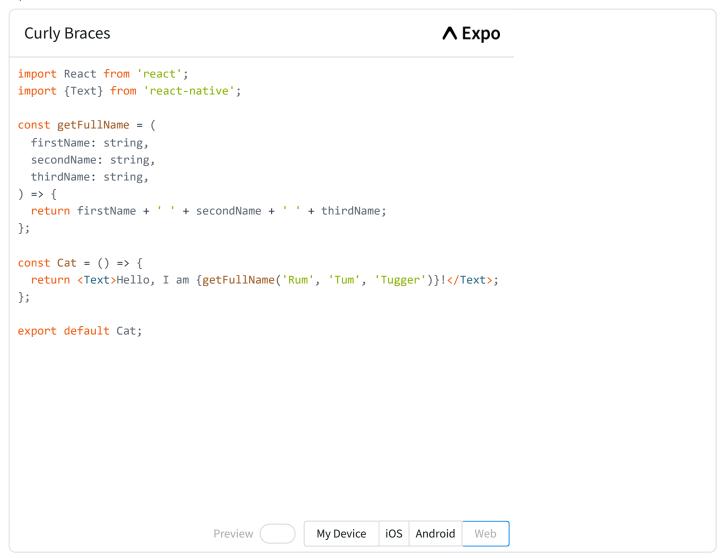
### **JSX**

React and React Native use **JSX**, a syntax that lets you write elements inside JavaScript like so: <Text>Hello, I am your cat!</Text>. The React docs have a comprehensive guide to JSX you can refer to learn even more. Because JSX is JavaScript, you can use variables inside it. Here you are declaring a name for the cat, name, and embedding it with curly braces inside <Text>.

```
Curly Braces
                                                                                                ∧ Expo
                                                               Hello, I am Maru!
import React from 'react';
import {Text} from 'react-native';
const Cat = () => {
 const name = 'Maru';
 return <Text>Hello, I am {name}!</Text>;
};
export default Cat;
                                                             Preview
                                                                            My Device
                                                                                       iOS Android
                                                                                                    Web
```

Any JavaScript expression will work between curly braces, including function calls like {getFullName("Rum", "Tum", "Tugger")}:

TypeScript JavaScript



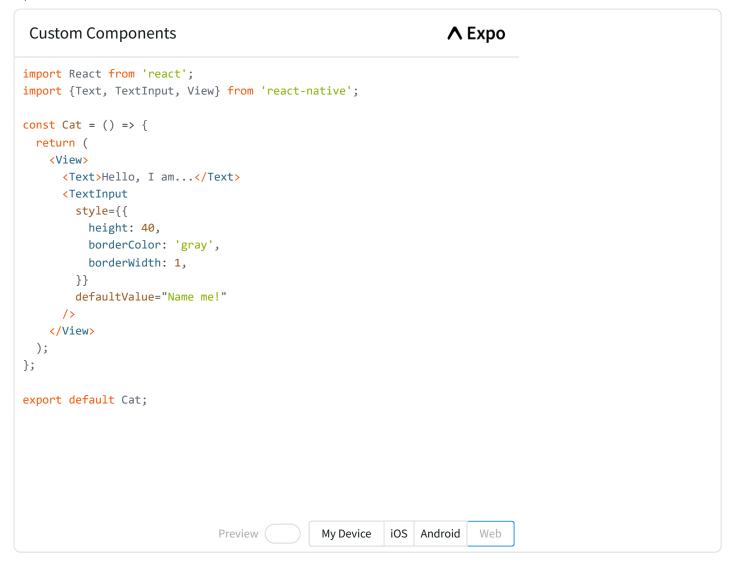
You can think of curly braces as creating a portal into JS functionality in your JSX!

Because JSX is included in the React library, it won't work if you don't have import React from 'react' at the top of your file!

### **Custom Components**

You've already met React Native's Core Components. React lets you nest these components inside each other to create new components. These nestable, reusable components are at the heart of the React paradigm.

For example, you can nest <u>Text</u> and <u>TextInput</u> inside a <u>View</u> below, and React Native will render them together:



#### **Developer notes**

Android Web

If you're familiar with web development, <View> and <Text> might remind you of HTML! You can think of them as the <div> and tags of application development.

You can render this component multiple times and in multiple places without repeating your code by using <Cat>:

Any component that renders other components is a **parent component.** Here, Cafe is the parent component and each Cat is a **child component.** 

You can put as many cats in your cafe as you like. Each <Cat> renders a unique element— which you can customize with props.

### **Props**

**Props** is short for "properties". Props let you customize React components. For example, here you pass each <Cat> a different name for Cat to render:

TypeScript

**JavaScript** 

Most of React Native's Core Components can be customized with props, too. For example, when using <a href="Image">Image</a>, you pass it a prop named <a href="source">source</a> to define what image it shows:

Image has many different props, including <u>style</u>, which accepts a JS object of design and layout related property-value pairs.

Notice the double curly braces {{ }} surrounding style 's width and height. In JSX, JavaScript values are referenced with {}. This is handy if you are passing something other than a string as props, like an array or number: <Cat food={["fish", "kibble"]} age={2} />. However, JS objects are *also* denoted with curly braces: {width: 200, height: 200}. Therefore, to pass a JS object in JSX, you must wrap the object in another pair of curly braces: {{width: 200, height: 200}}

You can build many things with props and the Core Components \_Text , \_Image , and View ! But to build something interactive, you'll need state.

#### **State**

While you can think of props as arguments you use to configure how components render, **state** is like a component's personal data storage. State is useful for handling data that changes over time or that comes from user interaction. State gives your components memory!

As a general rule, use props to configure a component when it renders. Use state to keep track of any component data that you expect to change over time.

The following example takes place in a cat cafe where two hungry cats are waiting to be fed. Their hunger, which we expect to change over time (unlike their names), is stored as state. To feed the cats, press their buttons—which will update their state.

You can add state to a component by calling React's useState Hook. A Hook is a kind of function that lets you "hook into" React features. For example, useState is a Hook that lets you add state to function components. You can learn more about other kinds of Hooks in the React documentation.

TypeScript JavaScript

First, you will want to import useState from React like so:

```
import React, {useState} from 'react';
```

Then you declare the component's state by calling useState inside its function. In this example, useState creates an isHungry state variable:

```
const Cat = (props: CatProps) => {
  const [isHungry, setIsHungry] = useState(true);
  // ...
};
```

You can use useState to track any kind of data: strings, numbers, Booleans, arrays, objects. For example, you can track the number of times a cat has been petted with

```
const [timesPetted, setTimesPetted] = useState(0)!
```

Calling useState does two things:

- it creates a "state variable" with an initial value—in this case the state variable is isHungry and its initial value is true
- it creates a function to set that state variable's value—setIsHungry

It doesn't matter what names you use. But it can be handy to think of the pattern as [<getter>, <setter>] = useState(<initialValue>).

Next you add the Button Core Component and give it an onPress prop:

```
<Button
onPress={() => {
    setIsHungry(false);
}}
//..
/>
```

Now, when someone presses the button, onPress will fire, calling the setIsHungry(false). This sets the state variable isHungry to false. When isHungry is false, the Button's disabled prop is set to true and its title also changes:

```
<Button
//..
disabled={!isHungry}
title={isHungry ? 'Pour me some milk, please!' : 'Thank you!'}
/>
```

You might've noticed that although isHungry is a const, it is seemingly reassignable! What is happening is when a state-setting function like setIsHungry is called, its component will re-render. In this case the Cat function will run again—and this time, useState will give us the next value of isHungry.

Finally, put your cats inside a Cafe component:

```
const Cafe = () => {
  return (
    <>
      <Cat name="Munkustrap" />
      <Cat name="Spot" />
    </>
  );
};
```

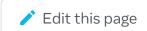
See the <> and </> above? These bits of JSX are fragments. Adjacent JSX elements must be wrapped in an enclosing tag. Fragments let you do that without nesting an extra, unnecessary wrapping element like View.

Now that you've covered both React and React Native's Core Components, let's dive deeper on some of these core components by looking at handling <TextInput> .

## Is this page useful?







Last updated on Jun 21, 2023