**Introduction**

Welcome to the very start of your React Native journey! If you're looking for environment setup instructions, they've moved to [their own section](https://reactnative.dev/docs/environment-setup). Continue reading for an introduction to the documentation, Native Components, React, and more!

Many different kinds of people use React Native: from advanced iOS developers to React beginners, to people getting started programming for the first time in their career. These docs were written for all learners, no matter their experience level or background.

**How to use these docs**

You can start here and read through these docs linearly like a book; or you can read the specific sections you need. Already familiar with React? You can skip [that section](https://reactnative.dev/docs/intro-react)—or read it for a light refresher.

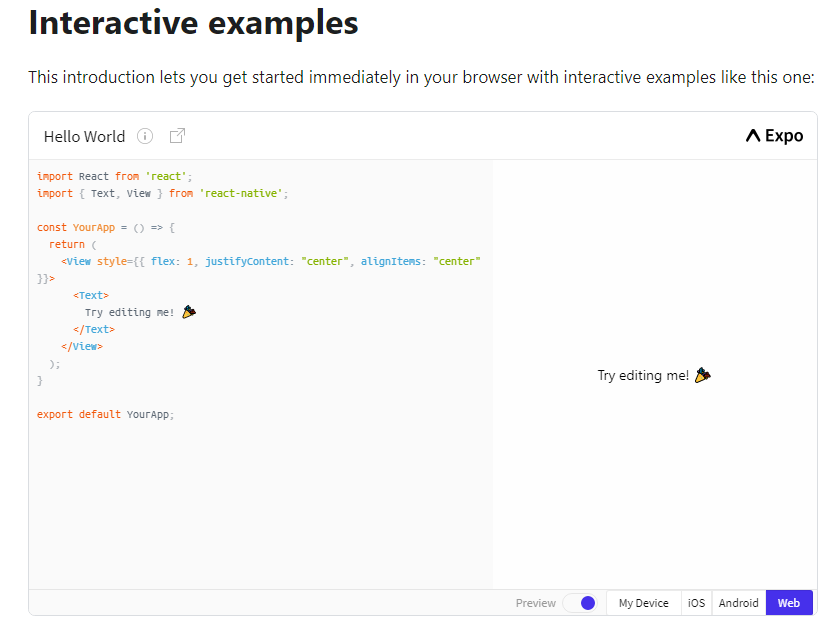
**Prerequisites**

To work with React Native, you will need to have an understanding of JavaScript fundamentals. If you’re new to JavaScript or need a refresher, you can [dive in](https://developer.mozilla.org/en-US/docs/Web/JavaScript) or [brush up](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript) at Mozilla Developer Network.

While we do our best to assume no prior knowledge of React, Android, or iOS development, these are valuable topics of study for the aspiring React Native developer. Where sensible, we have linked to resources and articles that go more in depth.

**Interactive examples**

This introduction lets you get started immediately in your browser with interactive examples like this one:



The above is a Snack Player. It’s a handy tool created by Expo to embed and run React Native projects and share how they render in platforms like Android and iOS. The code is live and editable, so you can play directly with it in your browser. Go ahead and try changing the "Try editing me!" text above to "Hello, world!"

Optionally, if you want to setup a local development environment, [you can follow our guide to setting up your environment on your local machine](https://reactnative.dev/docs/environment-setup) and paste the code examples into your App.js file there. (If you are a web developer, you may already have a local environment set up for mobile browser testing!)

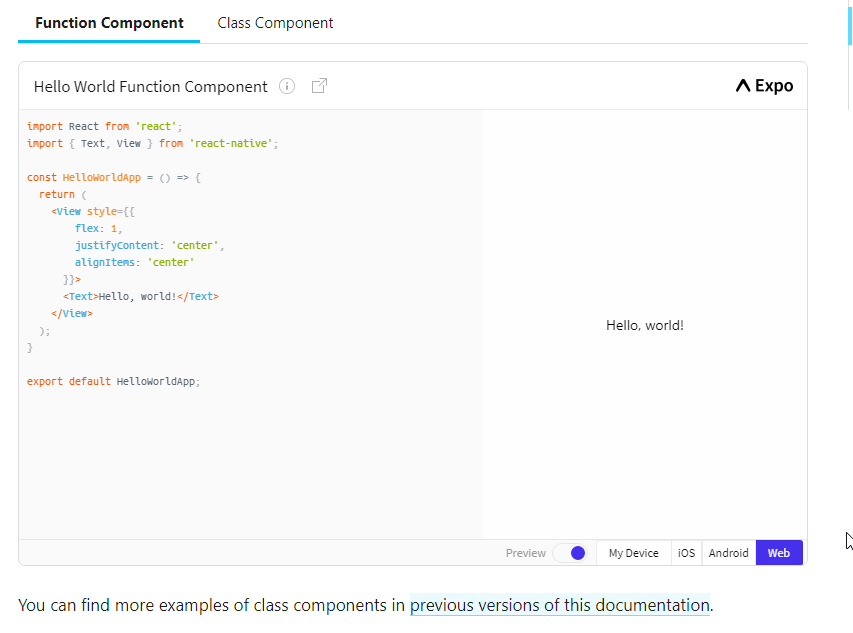
**Function Components and Class Components**[**#**](https://reactnative.dev/docs/getting-started#function-components-and-class-components)

With React, you can make components using either classes or functions. Originally, class components were the only components that could have state. But since the introduction of React's Hooks API, you can add state and more to function components.

[Hooks were introduced in React Native 0.58.](https://reactnative.dev/blog/2019/03/12/releasing-react-native-059), and because Hooks are the future-facing way to write your React components, we wrote this introduction using function component examples. Where useful, we also cover class components under a toggle like so:

* **Function Component**
* Class Component

You can find more examples of class components in [previous versions of this documentation](https://reactnative.dev/versions).



**Developer Notes**

People from many different development backgrounds are learning React Native. You may have experience with a range of technologies, from web to Android to iOS and more. We try to write for developers from all backgrounds. Sometimes we provide explanations specific to one platform or another like so:

* Android
* iOS
* **Web**

Web developers may be familiar with this concept.

**Formatting**

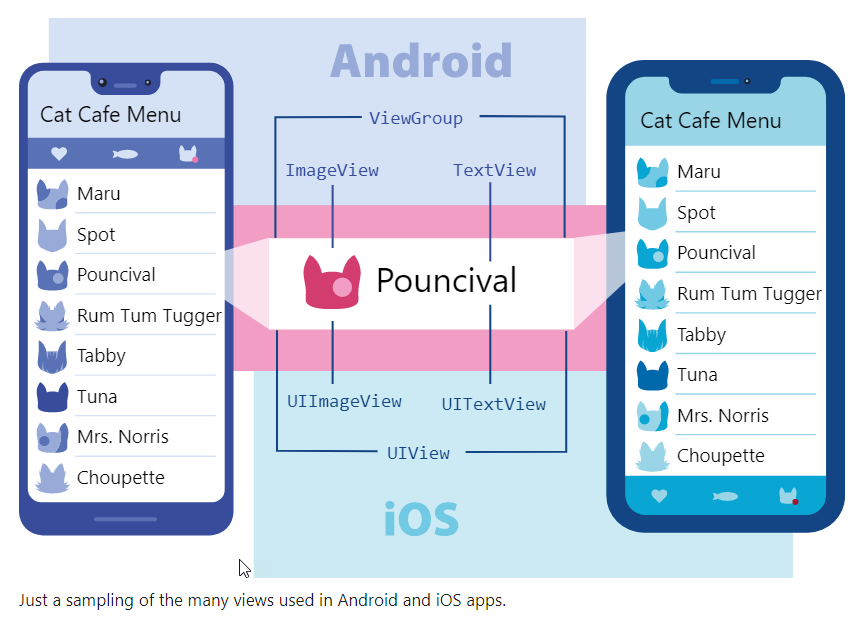
Menu paths are written in bold and use carets to navigate submenus. Example: **Android Studio > Preferences**

# Core Components and Native Components

React Native is an open source framework for building Android and iOS applications using [React](https://reactjs.org/) and the app platform’s native capabilities. With React Native, you use JavaScript to access your platform’s APIs as well as to describe the appearance and behavior of your UI using React components: bundles of reusable, nestable code. You can learn more about React in the next section. But first, let’s cover how components work in React Native.

## Views and mobile development[#](https://reactnative.dev/docs/intro-react-native-components#views-and-mobile-development)

In Android and iOS development, a **view** is the basic building block of UI: a small rectangular element on the screen which can be used to display text, images, or respond to user input. Even the smallest visual elements of an app, like a line of text or a button, are kinds of views. Some kinds of views can contain other views. It’s views all the way down!



## Native Components[#](https://reactnative.dev/docs/intro-react-native-components#native-components)

In Android development, you write views in Kotlin or Java; in iOS development, you use Swift or Objective-C. With React Native, you can invoke these views with JavaScript using React components. At runtime, React Native creates the corresponding Android and iOS views for those components. Because React Native components are backed by the same views as Android and iOS, React Native apps look, feel, and perform like any other apps. We call these platform-backed components **Native Components.**

React Native lets you to build your own Native Components for [Android](https://reactnative.dev/docs/native-components-android) and [iOS](https://reactnative.dev/docs/native-components-ios) to suit your app’s unique needs. We also have a thriving ecosystem of these **community-contributed components.** Check out [Native Directory](https://reactnative.directory/) to find what the community has been creating.

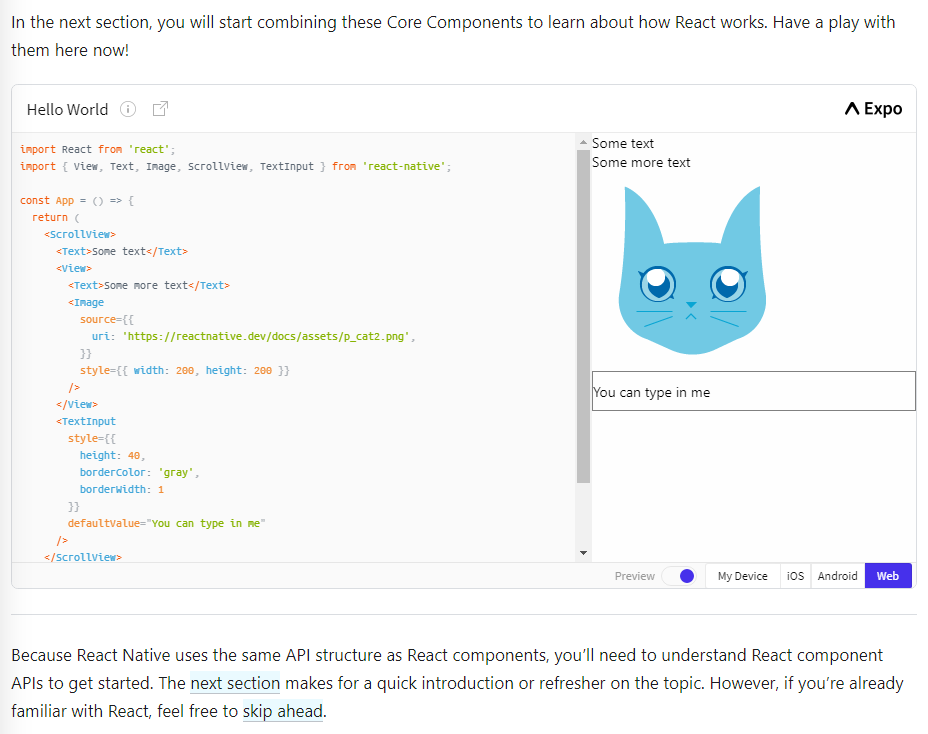
React Native also includes a set of essential, ready-to-use Native Components you can use to start building your app today. These are React Native's **Core Components**.

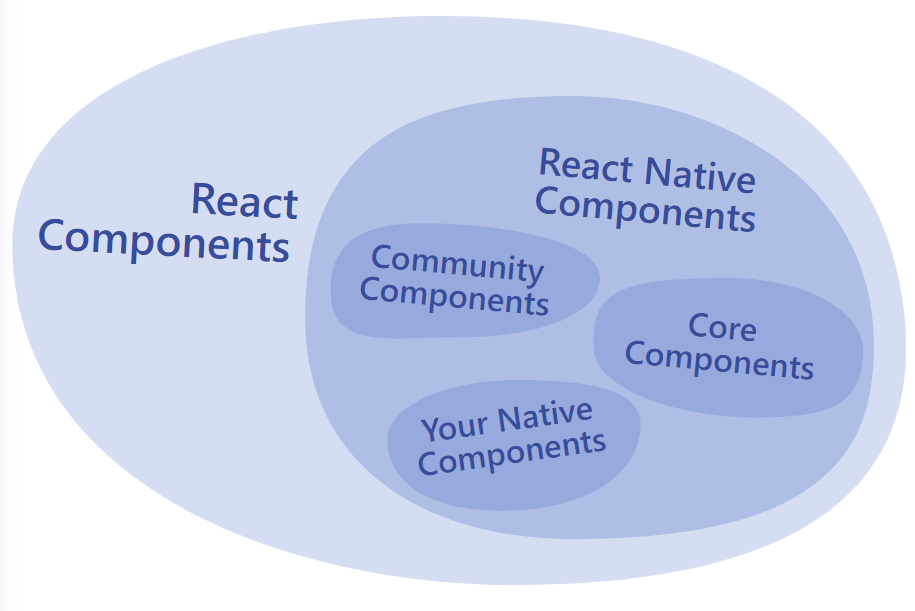
## Core Components[#](https://reactnative.dev/docs/intro-react-native-components#core-components)

React Native has many Core Components for everything from form controls to activity indicators. You can find them all [documented in the API section](https://reactnative.dev/docs/components-and-apis). You will mostly work with the following Core Components:

| **REACT NATIVE UI COMPONENT** | **ANDROID VIEW** | **IOS VIEW** | **WEB ANALOG** | **DESCRIPTION** |
| --- | --- | --- | --- | --- |
| <View> | <ViewGroup> | <UIView> | A non-scrollling <div> | A container that supports layout with flexbox, style, some touch handling, and accessibility controls |
| <Text> | <TextView> | <UITextView> | <p> | Displays, styles, and nests strings of text and even handles touch events |
| <Image> | <ImageView> | <UIImageView> | <img> | Displays different types of images |
| <ScrollView> | <ScrollView> | <UIScrollView> | <div> | A generic scrolling container that can contain multiple components and views |
| <TextInput> | <EditText> | <UITextField> | <input type="text"> | Allows the user to enter text |

In the next section, you will start combining these Core Components to learn about how React works. Have a play with them here now!

.



**React Fundamentals**

React Native runs on [React](https://reactjs.org/), 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:

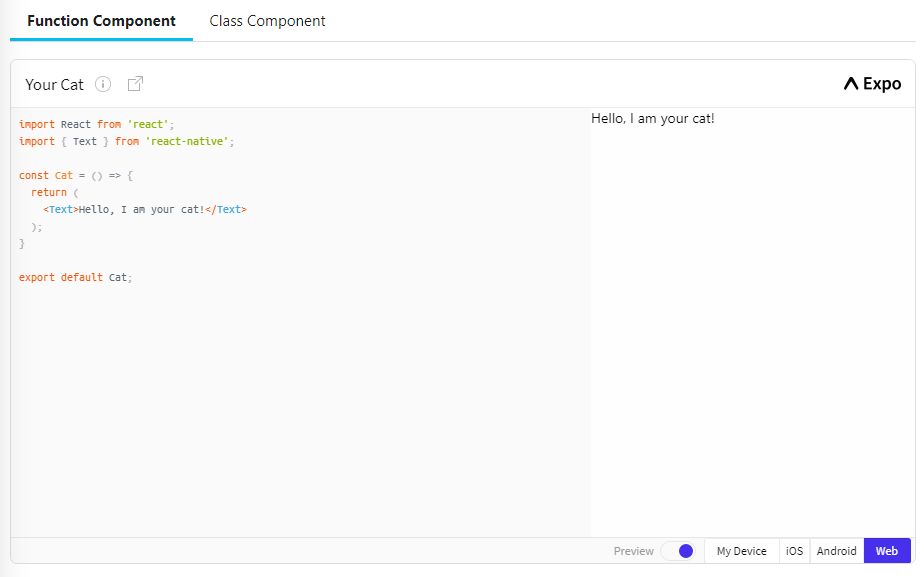
* components
* JSX
* props
* state

If you want to dig deeper, we encourage you to check out [React’s official documentation](https://reactjs.org/docs/getting-started.html" \t "_blank).

**Your first component**[**#**](https://reactnative.dev/docs/intro-react#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:

* **Function Component**
* Class Component



Here is how you do it: To define your Cat component, first use JavaScript’s [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) to import React and React Native’s [Text](https://reactnative.dev/docs/next/text) Core Component:

Copy

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:

Copy

const Cat = () => {

return <Text>Hello, I am your cat!</Text>;

};

export default Cat;

You can export your function component with JavaScript’s [export default](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) for use throughout your app like so:

Copy

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](https://medium.com/dailyjs/javascript-module-cheatsheet-7bd474f1d829) 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](https://reactjs.org/docs/jsx-in-depth.html) you can reference 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>.



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



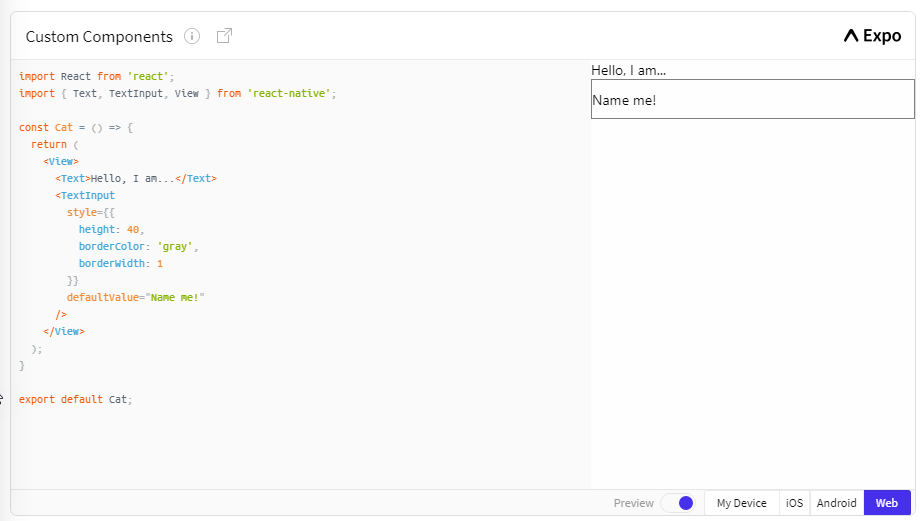
Because JSX is included in the React library, it won’t work if you don’t have import React from 'react' at the top of your file!

**Custom Components**

You’ve already met [React Native’s Core Components](https://reactnative.dev/docs/intro-react-native-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.

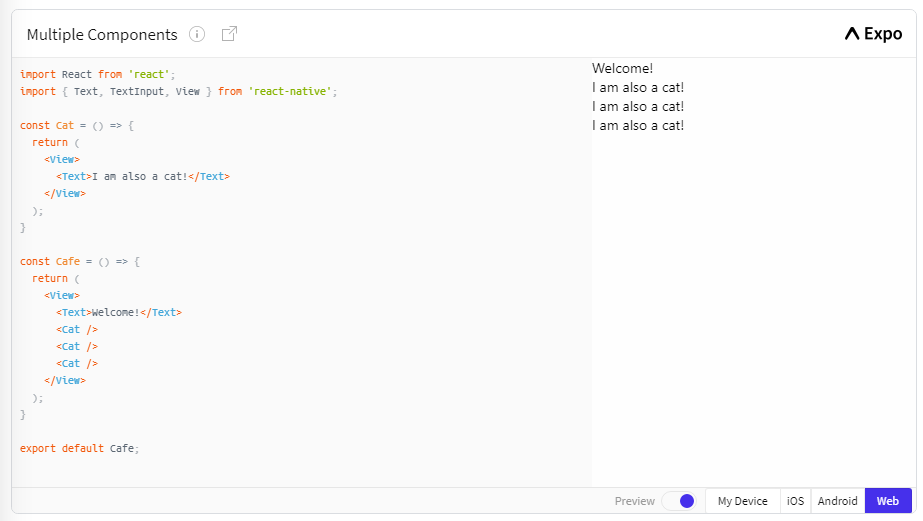
For example, you can nest [Text](https://reactnative.dev/docs/text) and [TextInput](https://reactnative.dev/docs/textinput) inside a [View](https://reactnative.dev/docs/view) below, and React Native will render them together:

* 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 <p> tags of application development.

You can render this component multiple times and 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:



Most of React Native’s Core Components can be customized with props, too. For example, when using [Image](https://reactnative.dev/docs/image), you pass it a prop named [source](https://reactnative.dev/docs/image#source) to define what image it shows:

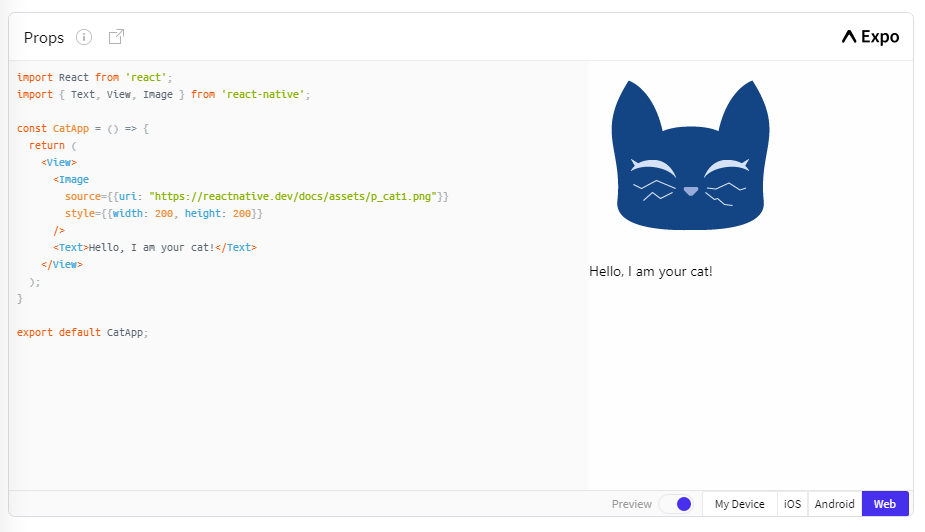


Image has [many different props](https://reactnative.dev/docs/image#props), including [style](https://reactnative.dev/docs/image#style), which accepts a JS object of design and layout related property-value pairs.

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](https://reactnative.dev/docs/text), [Image](https://reactnative.dev/docs/image), and [View](https://reactnative.dev/docs/view)! But to build something interactive, you’ll need state.

**State**

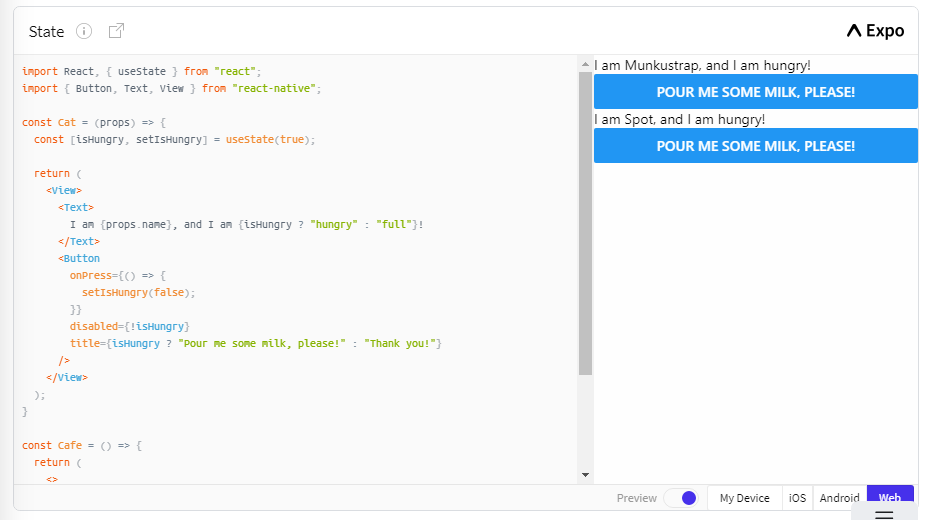
While you can think of props as arguments you use to configure how components render, **state** is like a component’s personal data storage. State is useful for handling data that changes over time or that comes from user interaction. State gives your components memory!

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.

* **Function Component**
* Class Component

You can add state to a component by calling [React’s useState Hook](https://reactjs.org/docs/hooks-state.html" \t "_blank). 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.](https://reactjs.org/docs/hooks-intro.html)



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

Copy

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:

Copy

const Cat = (props) => {

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](https://reactnative.dev/docs/button) Core Component and give it an onPress prop:

Copy

<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:

Copy

<Button

//..

disabled={!isHungry}

title={isHungry ? 'Pour me some milk, please!' : 'Thank you!'}

/>

You might’ve noticed that although isHungry is a [const](https://developer.mozilla.org/Web/JavaScript/Reference/Statements/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:

Copy

const Cafe = () => {

return (

<>

<Cat name="Munkustrap" />

<Cat name="Spot" />

</>

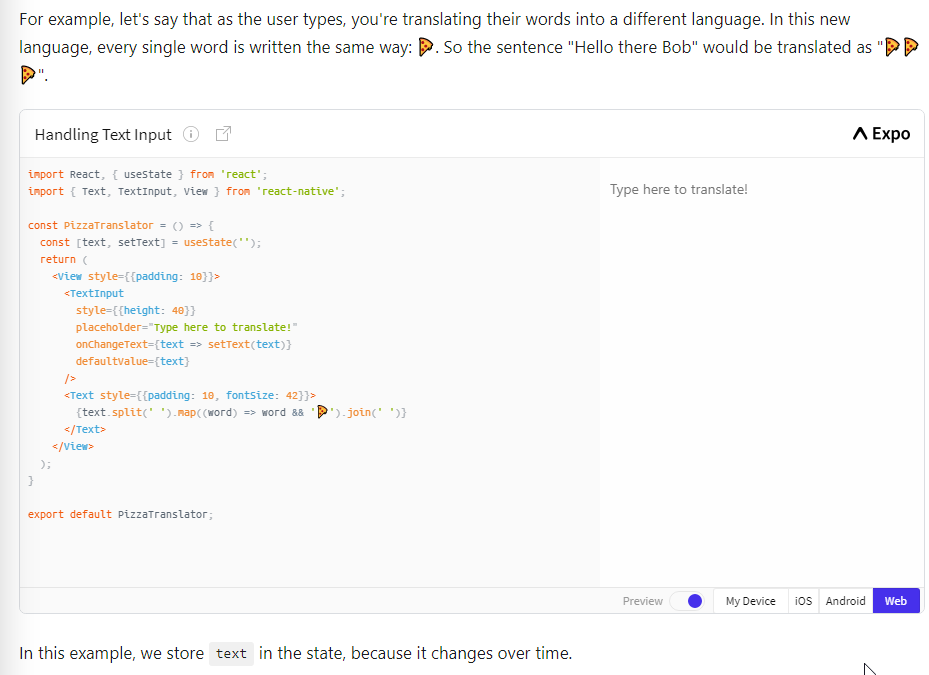
);

};

See the <> and </> above? These bits of JSX are [fragments](https://reactjs.org/docs/fragments.html). 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>](https://reactnative.dev/docs/handling-text-input).

# Handling Text Input

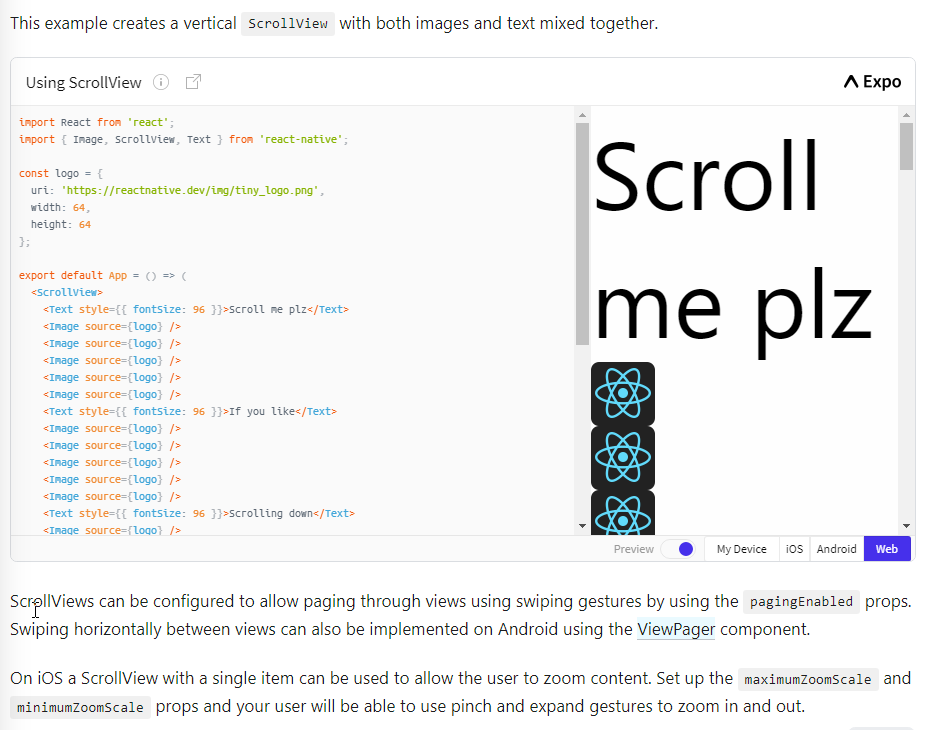
[TextInput](https://reactnative.dev/docs/textinput#content) is a [Core Component](https://reactnative.dev/docs/intro-react-native-components) that allows the user to enter text. It has an onChangeText prop that takes a function to be called every time the text changed, and an onSubmitEditing prop that takes a function to be called when the text is submitted.

There are a lot more things you might want to do with a text input. For example, you could validate the text inside while the user types. For more detailed examples, see the [React docs on controlled components](https://reactjs.org/docs/forms.html#controlled-components), or the [reference docs for TextInput](https://reactnative.dev/docs/textinput).

Text input is one of the ways the user interacts with the app. Next, let's look at another type of input and [learn how to handle touches](https://reactnative.dev/docs/handling-touches).

# Using a ScrollView

The [ScrollView](https://reactnative.dev/docs/scrollview) is a generic scrolling container that can contain multiple components and views. The scrollable items need not be homogeneous, and you can scroll both vertically and horizontally (by setting the horizontal property).



ScrollViews can be configured to allow paging through views using swiping gestures by using the pagingEnabled props. Swiping horizontally between views can also be implemented on Android using the [ViewPager](https://github.com/react-native-community/react-native-viewpager" \t "_blank) component.

On iOS a ScrollView with a single item can be used to allow the user to zoom content. Set up the maximumZoomScale and minimumZoomScale props and your user will be able to use pinch and expand gestures to zoom in and out.

The ScrollView works best to present a small amount of things of a limited size. All the elements and views of a ScrollView are rendered, even if they are not currently shown on the screen. If you have a long list of more items than can fit on the screen, you should use a FlatList instead. So let's [learn about list views](https://reactnative.dev/docs/using-a-listview) next.

# Using List Views

React Native provides a suite of components for presenting lists of data. Generally, you'll want to use either [FlatList](https://reactnative.dev/docs/flatlist) or [SectionList](https://reactnative.dev/docs/sectionlist).

The FlatList component displays a scrolling list of changing, but similarly structured, data. FlatList works well for long lists of data, where the number of items might change over time. Unlike the more generic [ScrollView](https://reactnative.dev/docs/using-a-scrollview), the FlatList only renders elements that are currently showing on the screen, not all the elements at once.

The FlatList component requires two props: data and renderItem. data is the source of information for the list. renderItem takes one item from the source and returns a formatted component to render.

This example creates a basic FlatList of hardcoded data. Each item in the data props is rendered as a Text component. The FlatListBasics component then renders the FlatList and all Text components.

import React from 'react';

import { FlatList, StyleSheet, Text, View } from 'react-native';

const styles = StyleSheet.create({

container: {

flex: 1,

paddingTop: 22

},

item: {

padding: 10,

fontSize: 18,

height: 44,

},

});

const FlatListBasics = () => {

return (

<View style={styles.container}>

<FlatList

data={[

{key: 'Devin'},

{key: 'Dan'},

{key: 'Dominic'},

{key: 'Jackson'},

{key: 'James'},

{key: 'Joel'},

{key: 'John'},

{key: 'Jillian'},

{key: 'Jimmy'},

{key: 'Julie'},

]}

renderItem={({item}) => <Text style={styles.item}>{item.key}</Text>}

/>

</View>

);

}

export default FlatListBasics;



If you want to render a set of data broken into logical sections, maybe with section headers, similar to UITableViews on iOS, then a [SectionList](https://reactnative.dev/docs/sectionlist) is the way to go.

import React from 'react';

import { SectionList, StyleSheet, Text, View } from 'react-native';

const styles = StyleSheet.create({

container: {

flex: 1,

paddingTop: 22

},

sectionHeader: {

paddingTop: 2,

paddingLeft: 10,

paddingRight: 10,

paddingBottom: 2,

fontSize: 14,

fontWeight: 'bold',

backgroundColor: 'rgba(247,247,247,1.0)',

},

item: {

padding: 10,

fontSize: 18,

height: 44,

},

})

const SectionListBasics = () => {

return (

<View style={styles.container}>

<SectionList

sections={[

{title: 'D', data: ['Devin', 'Dan', 'Dominic']},

{title: 'J', data: ['Jackson', 'James', 'Jillian', 'Jimmy', 'Joel', 'John', 'Julie']},

]}

renderItem={({item}) => <Text style={styles.item}>{item}</Text>}

renderSectionHeader={({section}) => <Text style={styles.sectionHeader}>{section.title}</Text>}

keyExtractor={(item, index) => index}

/>

</View>

);

}

export default SectionListBasics;

