React.Component

This page contains a detailed API reference for the React component class definition. It assumes you're familiar with fundamental React concepts, such as Components and Props, as well as State and Lifecycle. If you're not, read them first.

Overview

React lets you define components as classes or functions. Components defined as classes currently provide more features which are described in detail on this page. To define a React component class, you need to extend `React.Component`:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

The only method you *must* define in a `React.Component` subclass is called `render()`. All the other methods described on this page are optional.

We strongly recommend against creating your own base component classes. In React components, code reuse is primarily achieved through composition rather than inheritance.

> **Note:**
> React doesn't force you to use the ES6 class syntax. If you prefer to avoid it, you may use the `create-react-class` module or a similar custom abstraction instead. Take a look at Using React without ES6 to learn more.

The Component Lifecycle

Each component has several “lifecycle methods” that you can override to run code at particular times in the process. **You can use this lifecycle diagram as a cheat sheet.** In the list below, commonly used lifecycle methods are marked as **bold**. The rest of them exist for relatively rare use cases.

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- **`constructor()`**
- `static getDerivedStateFromProps()`
- **`render()`**
- **`componentDidMount()`**

> **Note:**
> These methods are considered legacy and you should avoid them in new code:
>
> - `UNSAFE_componentWillMount()`

Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- **`render()`**

- `getSnapshotBeforeUpdate()`
- **`componentDidUpdate()`**

> **Note:**
> These methods are considered legacy and you should avoid them in new code:
>
> - `UNSAFE_componentWillUpdate()`
> - `UNSAFE_componentWillReceiveProps()`

Unmounting

This method is called when a component is being removed from the DOM:

- **`componentWillUnmount()`**

Error Handling

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

Other APIs

Each component also provides some other APIs:

- `setState()`
- `forceUpdate()`

- `defaultProps`
- `displayName`

Instance Properties

- `props`
- `state`

Reference

Commonly Used Lifecycle Methods

The methods in this section cover the vast majority of use cases you'll encounter creating React components. **For a visual reference, check out this lifecycle diagram.**

`render()`

```
render()
```

The `render()` method is the only required method in a class component.

When called, it should examine `this.props` and `this.state` and return one of the following types:

- **React elements.** Typically created via JSX. For example, `<div />` and `<MyComponent />` are React elements that instruct React to render a DOM node, or another user-defined component, respectively.
- **Arrays and fragments.** Let you return multiple elements from render. See the documentation on fragments for more details.

- **Portals**. Let you render children into a different DOM subtree. See the documentation on portals for more details.
- **String and numbers.** These are rendered as text nodes in the DOM.
- **Booleans or `null`**. Render nothing. (Mostly exists to support `return test && <Child />` pattern, where `test` is boolean.)

The `render()` function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.

If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes components easier to think about.

> **Note**
>
> `render()` will not be invoked if `shouldComponentUpdate()` returns false.

`constructor()`

```
constructor(props)
```

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

The constructor for a React component is called before it is mounted. When implementing the constructor for a `React.Component` subclass, you should call `super(props)` before any other statement. Otherwise, `this.props` will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

- Initializing local state by assigning an object to `this.state`.
- Binding event handler methods to an instance.

You **should not call `setState()`** in the `constructor()`. Instead, if your component needs to use local state, **assign the initial state to `this.state`** directly in the constructor:

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Constructor is the only place where you should assign `this.state` directly. In all other methods, you need to use `this.setState()` instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use `componentDidMount()` instead.

Note

Avoid copying props into state! This is a common mistake:

```
constructor(props) {
 super(props);
 // Don't do this!
 this.state = { color: props.color };
}
```

The problem is that it's both unnecessary (you can use `this.props.color` directly instead), and creates bugs (updates to the `color` prop won't be reflected in the state).

Only use this pattern if you intentionally want to ignore prop updates. In that case, it makes sense to rename the prop to be called `initialColor` or `defaultColor`. You can then force a component to "reset" its internal state by changing its `key` when necessary.

> Read our blog post on avoiding derived state to learn about what to do if you think you need some state to depend on the props.

`componentDidMount()`

```
componentDidMount()
```

`componentDidMount()` is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in `componentWillUnmount()`.

You **may call `setState()` immediately** in `componentDidMount()`. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the `render()` will be called twice in this case, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, you should be able to assign the initial state in the `constructor()` instead. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

`componentDidUpdate()`

```
componentDidUpdate(prevProps, prevState, snapshot)
```

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

You **may call `setState()` immediately** in `componentDidUpdate()` but note that **it must be wrapped in a condition** like in the example above, or you'll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance. If you're trying to "mirror" some state to a prop coming from above, consider using the prop directly instead. Read more about why copying props into state causes bugs.

If your component implements the `getSnapshotBeforeUpdate()` lifecycle (which is rare), the value it returns will be passed as a third "snapshot" parameter to `componentDidUpdate()`. Otherwise this parameter will be undefined.

> **Note**
>
> `componentDidUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

`componentWillUnmount()`

```
componentWillUnmount()
```

`componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers

canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.

You **should not call `setState()`** in `componentWillUnmount()` because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

Rarely Used Lifecycle Methods

The methods in this section correspond to uncommon use cases. They're handy once in a while, but most of your components probably don't need any of them. **You can see most of the methods below on this lifecycle diagram if you click the "Show less common lifecycles" checkbox at the top of it.**

`shouldComponentUpdate()`

```
shouldComponentUpdate(nextProps, nextState)
```

Use `shouldComponentUpdate()` to let React know if a component's output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

`shouldComponentUpdate()` is invoked before rendering when new props or state are being received. Defaults to `true`. This method is not called for the initial render or when `forceUpdate()` is used.

This method only exists as a **performance optimization.** Do not rely on it to "prevent" a rendering, as this can lead to bugs. **Consider using the built-in `PureComponent`** instead of writing `shouldComponentUpdate()` by hand. `PureComponent` performs a shallow comparison of props and state, and reduces the chance that you'll skip a necessary update.

If you are confident you want to write it by hand, you may compare `this.props` with `nextProps` and `this.state` with `nextState` and return `false` to tell React the update can be skipped. Note that returning `false` does not prevent child components from re-rendering when *their* state changes.

We do not recommend doing deep equality checks or using `JSON.stringify()` in `shouldComponentUpdate()`. It is very inefficient and will harm performance.

Currently, if `shouldComponentUpdate()` returns `false`, then `UNSAFE_componentWillUpdate()`, `render()`, and `componentDidUpdate()` will not be invoked. In the future React may treat `shouldComponentUpdate()` as a hint rather than a strict directive, and returning `false` may still result in a re-rendering of the component.

`static getDerivedStateFromProps()`

```
static getDerivedStateFromProps(props, state)
```

`getDerivedStateFromProps` is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

This method exists for rare use cases where the state depends on changes in props over time. For example, it might be handy for implementing a `<Transition>` component that compares its previous and next children to decide which of them to animate in and out.

Deriving state leads to verbose code and makes your components difficult to think about. Make sure you're familiar with simpler alternatives:

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use `componentDidUpdate` lifecycle instead.
- If you want to **re-compute some data only when a prop changes**, use a memoization helper instead.

- If you want to **"reset" some state when a prop changes**, consider either making a component fully controlled or fully uncontrolled with a `key` instead.

This method doesn't have access to the component instance. If you'd like, you can reuse some code between `getDerivedStateFromProps()` and the other class methods by extracting pure functions of the component props and state outside the class definition.

Note that this method is fired on *every* render, regardless of the cause. This is in contrast to `UNSAFE_componentWillReceiveProps`, which only fires when the parent causes a re-render and not as a result of a local `setState`.

`getSnapshotBeforeUpdate()`

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

`getSnapshotBeforeUpdate()` is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to `componentDidUpdate()`.

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

A snapshot value (or `null`) should be returned.

For example:

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }
```

```
    // Capture the scroll position so we can adjust scroll later.
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // If we have a snapshot value, we've just added new items.
    // Adjust scroll so these new items don't push the old ones out of view.
    // (snapshot here is the value returned from getSnapshotBeforeUpdate)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

In the above examples, it is important to read the `scrollHeight` property in `getSnapshotBeforeUpdate` because there may be delays between "render" phase lifecycles (like `render`) and "commit" phase lifecycles (like `getSnapshotBeforeUpdate` and `componentDidUpdate`).

Error boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle

state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.

Only use error boundaries for recovering from unexpected exceptions; **don't try to use them for control flow.**

For more details, see *Error Handling in React 16*.

> **Note**
> Error boundaries only catch errors in the components **below** them in the tree. An error boundary can't catch an error within itself.

`static getDerivedStateFromError()`

```
static getDerivedStateFromError(error)
```

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
```

```
    return this.props.children;
  }
}
```

> **Note**
>
> `getDerivedStateFromError()` is called during the "render" phase, so side-effects are not permitted. For those use cases, use `componentDidCatch()` instead.

`componentDidCatch()`

```
componentDidCatch(error, info)
```

This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

1. `error` - The error that was thrown.
2. `info` - An object with a `componentStack` key containing information about which component threw the error.

`componentDidCatch()` is called during the "commit" phase, so side-effects are permitted. It should be used for things like logging errors:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
```

```
  }

  componentDidCatch(error, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Note

In the event of an error, you can render a fallback UI with `componentDidCatch()` by calling `setState`, but this will be deprecated in a future release. Use `static getDerivedStateFromError()` to handle fallback rendering instead.

Legacy Lifecycle Methods

The lifecycle methods below are marked as “legacy”. They still work, but we don’t recommend using them in the new code. You can learn more about migrating away from legacy lifecycle methods in this blog post.

`UNSAFE_componentWillMount()`

> **Note**
> This lifecycle was previously named `componentWillMount`. That name will continue to work until version 17. Use the `rename-unsafe-lifecycles` codemod to automatically update your components.

`UNSAFE_componentWillMount()` is invoked just before mounting occurs. It is called before `render()`, therefore calling `setState()` synchronously in this method will not trigger an extra rendering. Generally, we recommend using the `constructor()` instead for initializing state.

Avoid introducing any side-effects or subscriptions in this method. For those use cases, use `componentDidMount()` instead.

This is the only lifecycle method called on server rendering.

`UNSAFE_componentWillReceiveProps()`

```
UNSAFE_componentWillReceiveProps(nextProps)
```

> **Note**
> This lifecycle was previously named `componentWillReceiveProps`. That name will continue to work until version 17. Use the `rename-unsafe-lifecycles` codemod to automatically update your components.

> **Note:**

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use `componentDidUpdate` lifecycle instead.
- If you used `componentWillReceiveProps` for **re-computing some data only when a prop changes**, use a memoization helper instead.
- If you used `componentWillReceiveProps` to **"reset" some state when a prop changes**, consider either making a component fully controlled or fully uncontrolled with a `key` instead.

For other use cases, follow the recommendations in this blog post about derived state.

`UNSAFE_componentWillReceiveProps()` is invoked before a mounted component receives new props. If you need to update the state in response to prop changes (for example, to reset it), you may compare `this.props` and `nextProps` and perform state transitions using `this.setState()` in this method.

Note that if a parent component causes your component to re-render, this method will be called even if props have not changed. Make sure to compare the current and next values if you only want to handle changes.

React doesn't call `UNSAFE_componentWillReceiveProps()` with initial props during mounting. It only calls this method if some of component's props may update. Calling `this.setState()` generally doesn't trigger `UNSAFE_componentWillReceiveProps()`.

`UNSAFE_componentWillUpdate()`

```
UNSAFE_componentWillUpdate(nextProps, nextState)
```

Note

This lifecycle was previously named `componentWillUpdate`. That name will continue to

update your components.

`UNSAFE_componentWillUpdate()` is invoked just before rendering when new props or state are being received. Use this as an opportunity to perform preparation before an update occurs. This method is not called for the initial render.

Note that you cannot call `this.setState()` here; nor should you do anything else (e.g. dispatch a Redux action) that would trigger an update to a React component before `UNSAFE_componentWillUpdate()` returns.

Typically, this method can be replaced by `componentDidUpdate()`. If you were reading from the DOM in this method (e.g. to save a scroll position), you can move that logic to `getSnapshotBeforeUpdate()`.

Note

`UNSAFE_componentWillUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

Other APIs

Unlike the lifecycle methods above (which React calls for you), the methods below are the methods *you* can call from your components.

There are just two of them: `setState()` and `forceUpdate()`.

`setState()`

```
setState(updater[, callback])
```

`setState()` enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state. This is the primary method you use to update the user interface in response to event handlers and server responses.

Think of `setState()` as a *request* rather than an immediate command to update the component. For better perceived performance, React may delay it, and then update several components in a single pass. React does not guarantee that the state changes are applied immediately.

`setState()` does not always immediately update the component. It may batch or defer the update until later. This makes reading `this.state` right after calling `setState()` a potential pitfall. Instead, use `componentDidUpdate` or a `setState` callback (`setState(updater, callback)`), either of which are guaranteed to fire after the update has been applied. If you need to set the state based on the previous state, read about the `updater` argument below.

`setState()` will always lead to a re-render unless `shouldComponentUpdate()` returns `false`. If mutable objects are being used and conditional rendering logic cannot be implemented in `shouldComponentUpdate()`, calling `setState()` only when the new state differs from the previous state will avoid unnecessary re-renders.

The first argument is an `updater` function with the signature:

```
(state, props) => stateChange
```

`state` is a reference to the component state at the time the change is being applied. It should not be directly mutated. Instead, changes should be represented by building a new object based on the input from `state` and `props`. For instance, suppose we wanted to increment a value in state by `props.step`:

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

Both `state` and `props` received by the updater function are guaranteed to be up-to-date. The output of the updater is shallowly merged with `state`.

The second parameter to `setState()` is an optional callback function that will be executed once `setState` is completed and the component is re-rendered. Generally we recommend using `componentDidUpdate()` for such logic instead.

You may optionally pass an object as the first argument to `setState()` instead of a function:

```
setState(stateChange[, callback])
```

This performs a shallow merge of `stateChange` into the new state, e.g., to adjust a shopping cart item quantity:

```
this.setState({quantity: 2})
```

This form of `setState()` is also asynchronous, and multiple calls during the same cycle may be batched together. For example, if you attempt to increment an item quantity more than once in the same cycle, that will result in the equivalent of:

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

Subsequent calls will override values from previous calls in the same cycle, so the quantity will only be incremented once. If the next state depends on the current state, we recommend using the updater function form, instead:

```
this.setState((state) => {
```

For more detail, see:

- State and Lifecycle guide
- In depth: When and why are `setState()` calls batched?
- In depth: Why isn't `this.state` updated immediately?

`forceUpdate()`

```
component.forceUpdate(callback)
```

By default, when your component's state or props change, your component will re-render. If your `render()` method depends on some other data, you can tell React that the component needs re-rendering by calling `forceUpdate()`.

Calling `forceUpdate()` will cause `render()` to be called on the component, skipping `shouldComponentUpdate()`. This will trigger the normal lifecycle methods for child components, including the `shouldComponentUpdate()` method of each child. React will still only update the DOM if the markup changes.

Normally you should try to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`.

Class Properties

`defaultProps`

`defaultProps` can be defined as a property on the component class itself, to set the default props for the class. This is used for undefined props, but not for null props. For example:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

If `props.color` is not provided, it will be set by default to `'blue'`:

```
  render() {
    return <CustomButton /> ; // props.color will be set to blue
  }
```

If `props.color` is set to null, it will remain null:

```
  render() {
    return <CustomButton color={null} /> ; // props.color will remain null
  }
```

---

displayName

The `displayName` string is used in debugging messages. Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component, see Wrap the Display Name for Easy Debugging for details.

Instance Properties

`props`

`this.props` contains the props that were defined by the caller of this component. See Components and Props for an introduction to props.

In particular, `this.props.children` is a special prop, typically defined by the child tags in the JSX expression rather than in the tag itself.

`state`

The state contains data specific to this component that may change over time. The state is user-defined, and it should be a plain JavaScript object.

If some value isn't used for rendering or data flow (for example, a timer ID), you don't have to put it in the state. Such values can be defined as fields on the component instance.

See State and Lifecycle for more information about the state.

Never mutate `this.state` directly, as calling `setState()` afterwards may replace the mutation you made. Treat `this.state` as if it were immutable.

COMMUNITY

MORE