# State and Lifecycle

Consider the ticking clock example from [one of the previous sections](https://facebook.github.io/react/docs/rendering-elements.html#updating-the-rendered-element).

So far we have only learned one way to update the UI.

We call ReactDOM.render() to change the rendered output:

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(

element,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[Try it on CodePen.](http://codepen.io/gaearon/pen/gwoJZk?editors=0010)

In this section, we will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

function Clock(props) {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {props.date.toLocaleTimeString()}.</h2>

</div>

);

}

function tick() {

ReactDOM.render(

<Clock date={new Date()} />,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[Try it on CodePen.](http://codepen.io/gaearon/pen/dpdoYR?editors=0010)

However, it misses a crucial requirement: the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock.

Ideally we want to write this once and have the Clock update itself:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

To implement this, we need to add "state" to the Clock component.

State is similar to props, but it is private and fully controlled by the component.

We [mentioned before](https://facebook.github.io/react/docs/components-and-props.html#functional-and-class-components) that components defined as classes have some additional features. Local state is exactly that: a feature available only to classes.

## Converting a Function to a Class

You can convert a functional component like Clock to a class in five steps:

1. Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes) with the same name that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.props.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

[Try it on CodePen.](http://codepen.io/gaearon/pen/zKRGpo?editors=0010)

Clock is now defined as a class rather than a function.

This lets us use additional features such as local state and lifecycle hooks.

## Adding Local State to a Class

We will move the date from props to state in three steps:

1) Replace this.props.date with this.state.date in the render() method:

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

2) Add a [class constructor](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor) that assigns the initial this.state:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

Note how we pass props to the base constructor:

constructor(props) {

super(props);

this.state = {date: new Date()};

}

Class components should always call the base constructor with props.

3) Remove the date prop from the <Clock /> element:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

We will later add the timer code back to the component itself.

The result looks like this:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[Try it on CodePen.](http://codepen.io/gaearon/pen/KgQpJd?editors=0010)

Next, we'll make the Clock set up its own timer and update itself every second.

## Adding Lifecycle Methods to a Class

In applications with many components, it's very important to free up resources taken by the components when they are destroyed.

We want to [set up a timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) whenever the Clock is rendered to the DOM for the first time. This is called "mounting" in React.

We also want to [clear that timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) whenever the DOM produced by the Clock is removed. This is called "unmounting" in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

}

componentWillUnmount() {

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

These methods are called "lifecycle hooks".

The componentDidMount() hook runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

Note how we save the timer ID right on this.

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output.

If you don't use something in render(), it shouldn't be in the state.

We will tear down the timer in the componentWillUnmount() lifecycle hook:

componentWillUnmount() {

clearInterval(this.timerID);

}

Finally, we will implement the tick() method that runs every second.

It will use this.setState() to schedule updates to the component local state:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({

date: new Date()

});

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[Try it on CodePen.](http://codepen.io/gaearon/pen/amqdNA?editors=0010)

Now the clock ticks every second.

Let's quickly recap what's going on and the order in which the methods are called:

1) When <Clock /> is passed to ReactDOM.render(), React calls the constructor of theClock component. Since Clock needs to display the current time, it initializes this.statewith an object including the current time. We will later update this state.

2) React then calls the Clock component's render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.

3) When the Clock output is inserted in the DOM, React calls the componentDidMount()lifecycle hook. Inside it, the Clock component asks the browser to set up a timer to calltick() once a second.

4) Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls render()method again to learn what should be on the screen. This time, this.state.date in therender() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.

5) If the Clock component is ever removed from the DOM, React calls thecomponentWillUnmount() lifecycle hook so the timer is stopped.

## Using State Correctly

There are three things you should know about setState().

### Do Not Modify State Directly

For example, this will not re-render a component:

// Wrong

this.state.comment = 'Hello';

Instead, use setState():

// Correct

this.setState({comment: 'Hello'});

The only place where you can assign this.state is the constructor.

### State Updates May Be Asynchronous

React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

});

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

We used an [arrow function](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) above, but it also works with regular functions:

// Correct

this.setState(function(prevState, props) {

return {

counter: prevState.counter + props.increment

};

});

### State Updates are Merged

When you call setState(), React merges the object you provide into the current state.

For example, your state may contain several independent variables:

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

Then you can update them independently with separate setState() calls:

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

## The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

This also works for user-defined components:

<FormattedDate date={this.state.date} />

The FormattedDate component would receive the date in its props and wouldn't know whether it came from the Clock's state, from the Clock's props, or was typed by hand:

function FormattedDate(props) {

return <h2>It is {props.date.toLocaleTimeString()}.</h2>;

}

[Try it on CodePen.](http://codepen.io/gaearon/pen/zKRqNB?editors=0010)

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an App component that renders three <Clock>s:

function App() {

return (

<div>

<Clock />

<Clock />

<Clock />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

[Try it on CodePen.](http://codepen.io/gaearon/pen/vXdGmd?editors=0010)

Each Clock sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.