React

A JavaScript library for building user interfaces.

- Created & Maintained By Facebook
- Used to build dynamic user interfaces
- It component based which means every piece of your front end interface your application is going to use will be an individual component.
- Most popular framework in the industry.

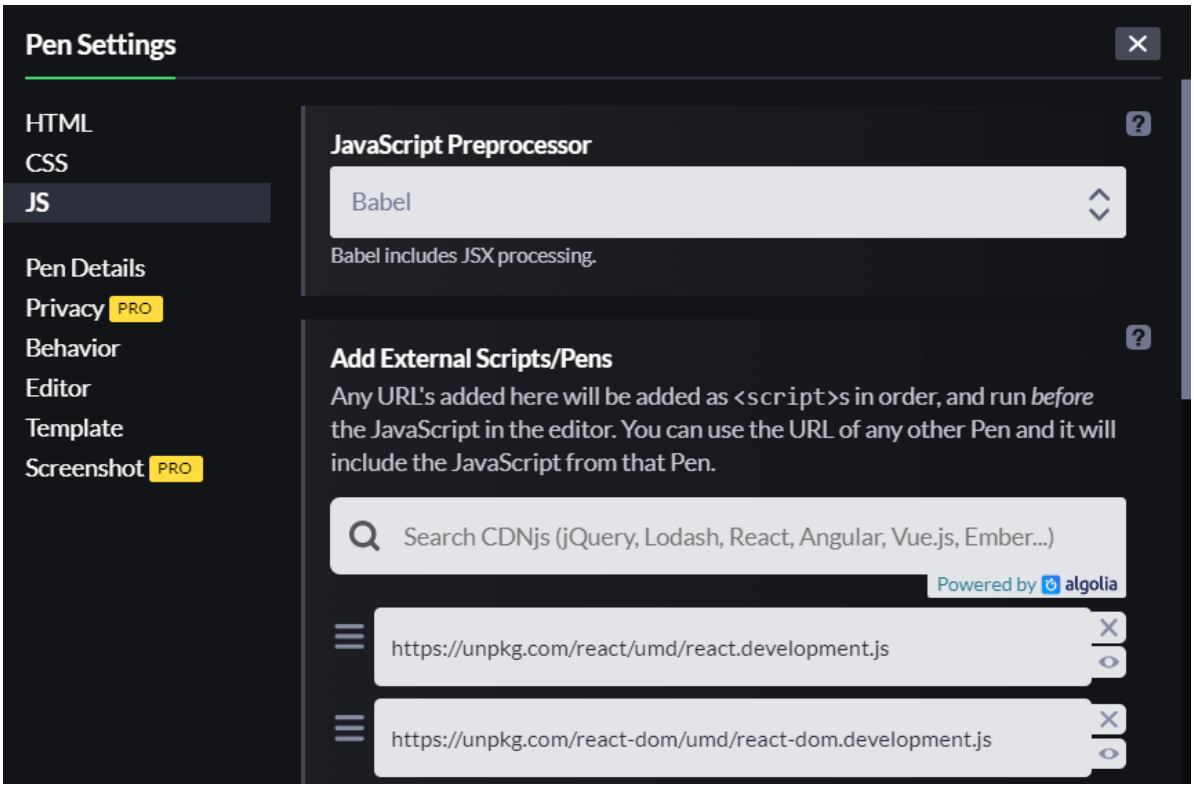
Configuration

All the coding is done on **CodePen**.

https://codepen.io/

Open Settings and these Configuration :

- Choose Babel as a JavaScript Preprocessor
- Add these link as External Scripts
    - https://unpkg.com/react/umd/react.development.js
    - https://unpkg.com/react-dom/umd/react-dom.development.js

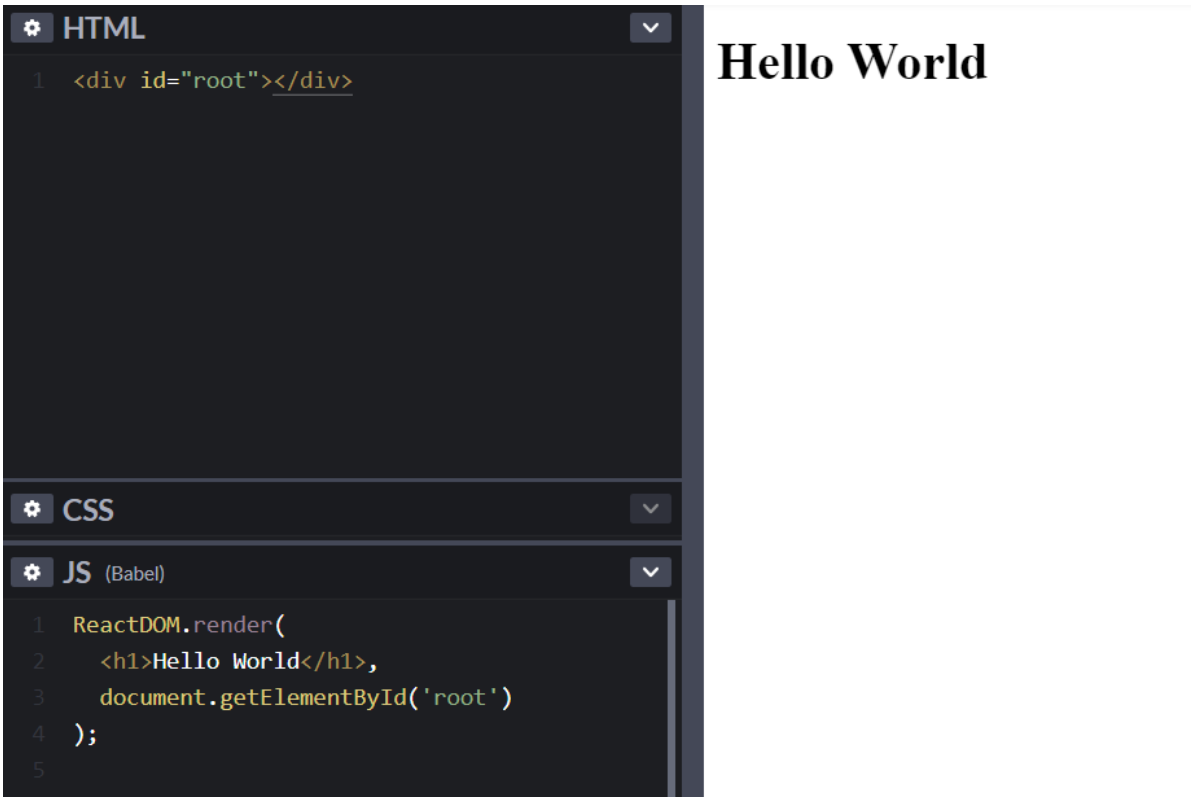


Main Concepts

1. Hello World

It displays a heading saying “Hello, world on the page”.

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```



2. Introducing JSX

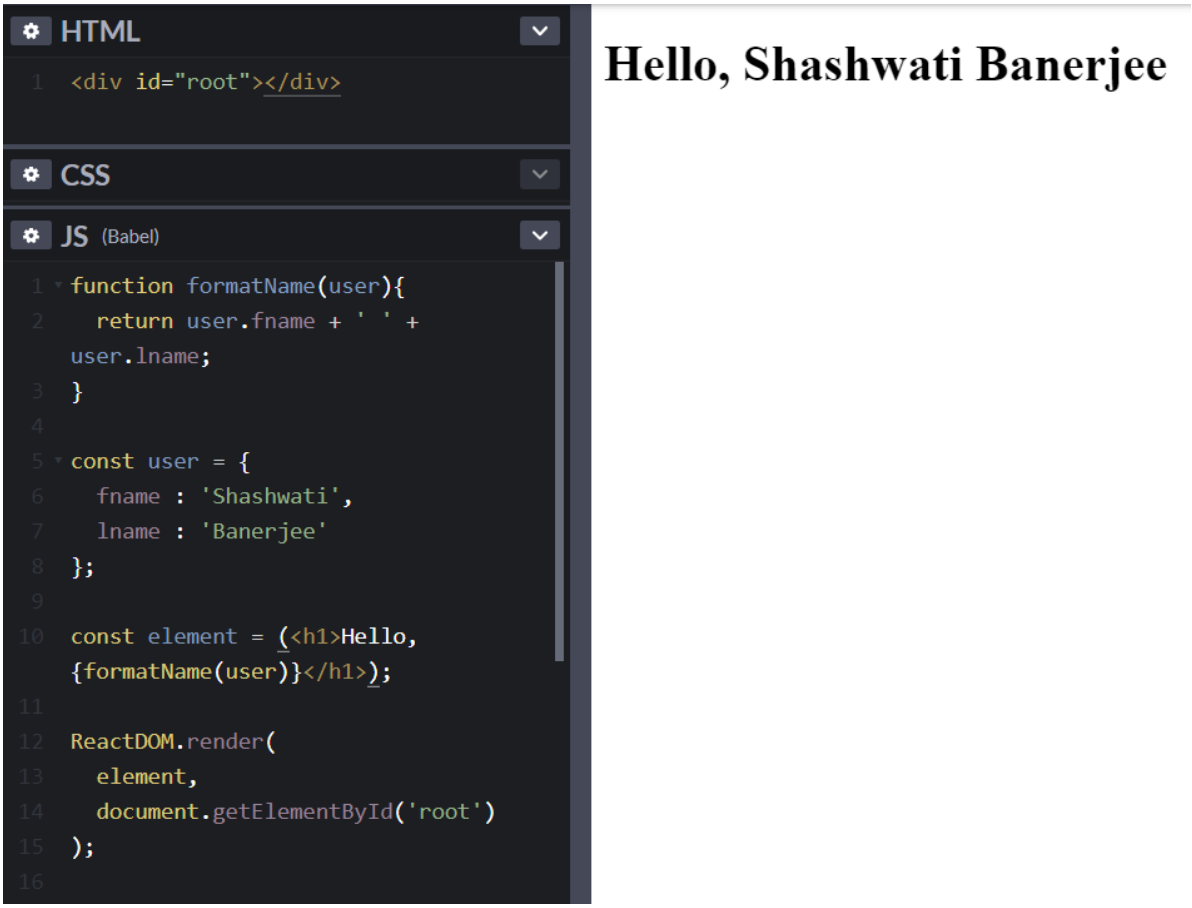
The process of using HTML in react is known as JSX. When XML code and JavaScript code combine it's JSX Code.

JSX stands for **JavaScript XML**. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code)

Why JSX ?

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called "components" that contain both.

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into an `<h1>` element.



3. Rendering Elements

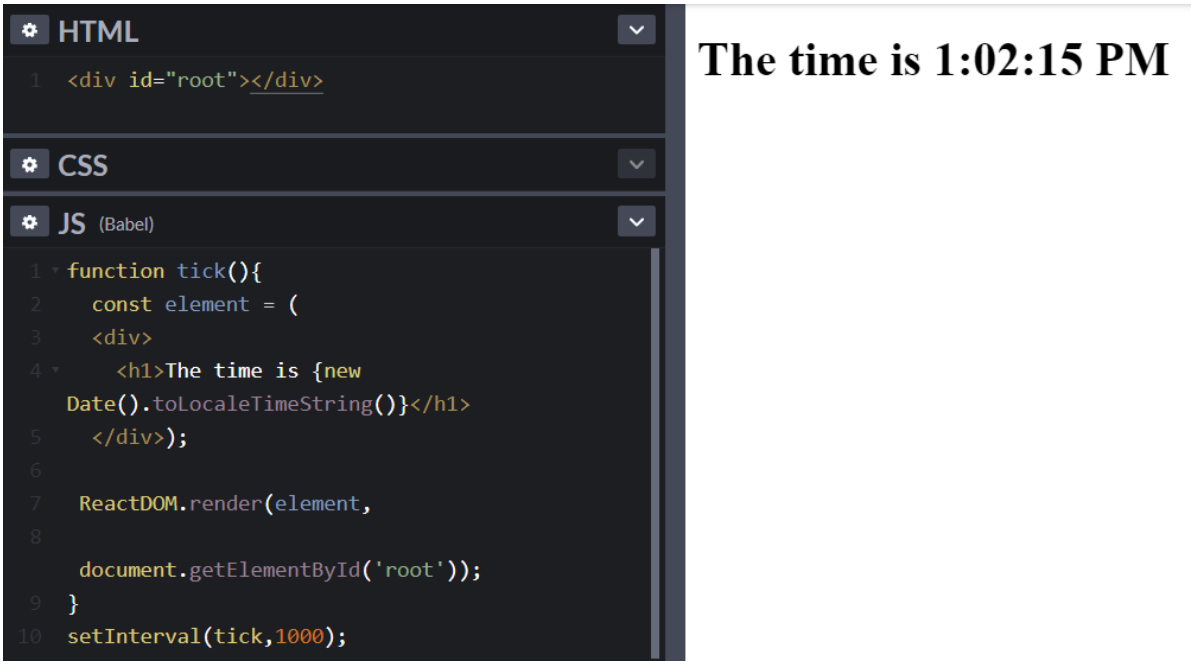
Elements are the smallest building blocks of React apps. An element describes what you want to see on the screen.

To render a React element into a root DOM node, pass both to `ReactDOM.render()`:

```
const element = <h1>Hello</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Updating the Rendered Element

React elements are [immutable](). Once you create an element, you can't change its children or attributes. So the only way to update the UI is to create a new element.



4. Components

Components let you split the UI into independent, reusable pieces and think about each piece in isolation. Conceptually component are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

Function and Class Components

Simplest way to define a component is to write a JavaScript function :

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

`ES6 class` can be used to define a component.

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

Rendering a Component

React Element that represent DOM tag :

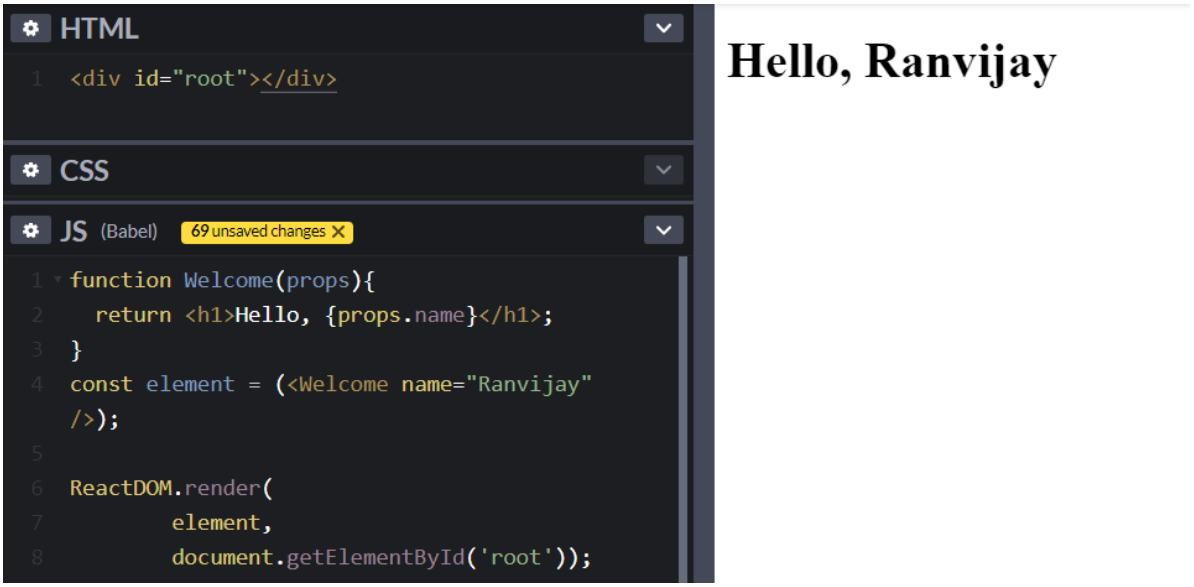
```
const element = <div />
```

React Element also represent user-defined components :

```
const element = <Welcome name="Ranvijay" />;
```

When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

This code renders “Hello, Ranvijay” on the page:



Let’s recap what happens in this example :

1. We call `ReactDOM.render()` with the `<Welcome name="Ranvijay" />` element.
2. React calls the `Welcome` component with `{name: 'Ranvijay'}` as the props.
3. Our `Welcome` component returns a `<h1>Hello, Ranvijay</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Ranvijay</h1>`.

> **Note:** Always start component names with a capital letter.
>
> React treats components starting with lowercase letters as DOM tags. For example, `<div />` represents an HTML div tag, but `<Welcome />` represents a component

Props are Read-Only

React is flexible but it has a single strict rule:

All React components must act like pure functions with respect to their props.

```
function sum(a, b) {
  return a + b;
}
```

Such function are called pure because they do not attempt to change their inputs and always return the same result for the same input.

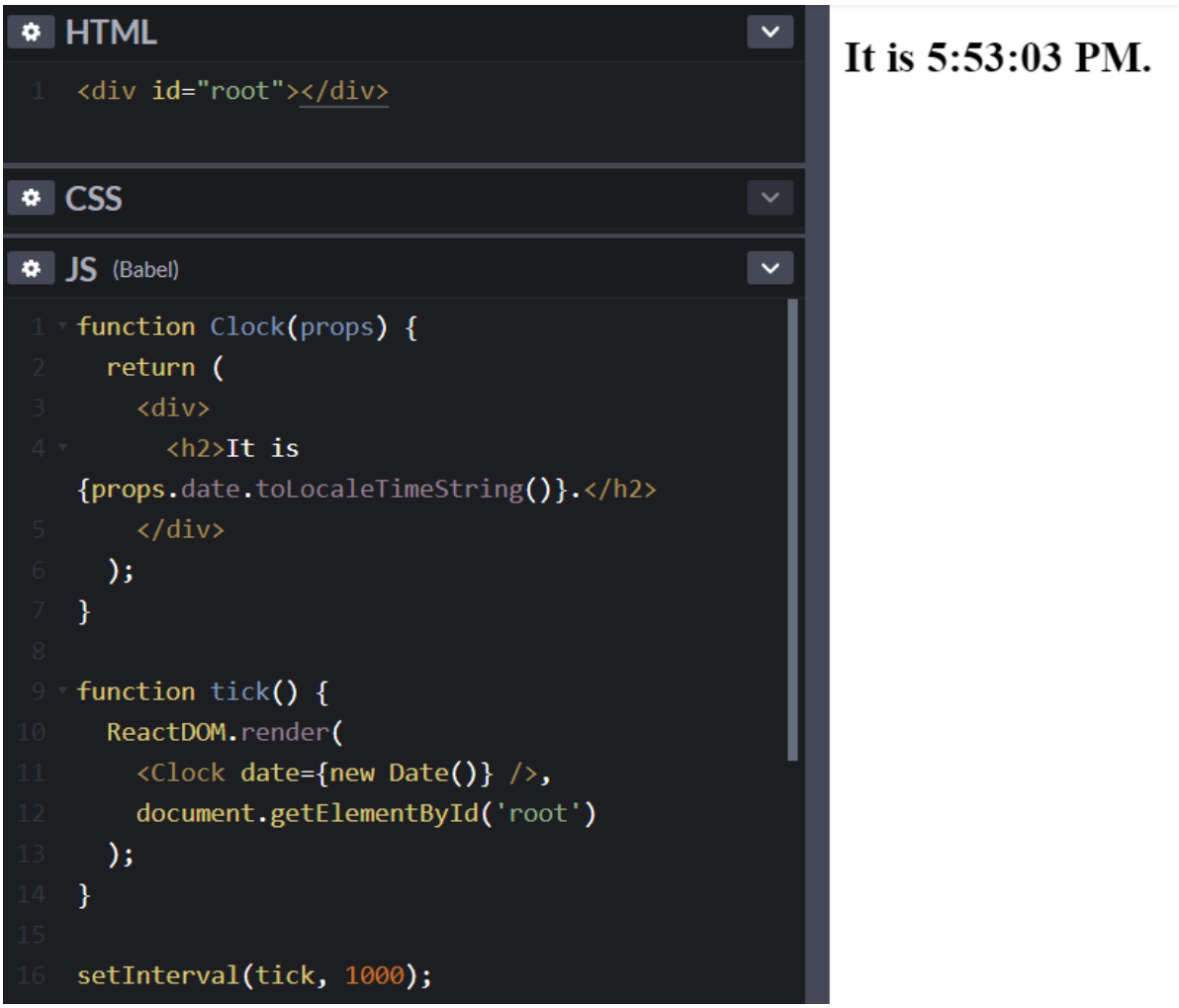
5. State and Lifecycle

State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Here , we make the `Clock` component truly reusable and encapsulated. It will set up its own timer and update itself every second.

Why State ?

Consider the ticking clock example from one of the previous sections. To update the UI we are calling ReactDOM.render() each second to change the rendered output.



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.

Converting a Function to a Class

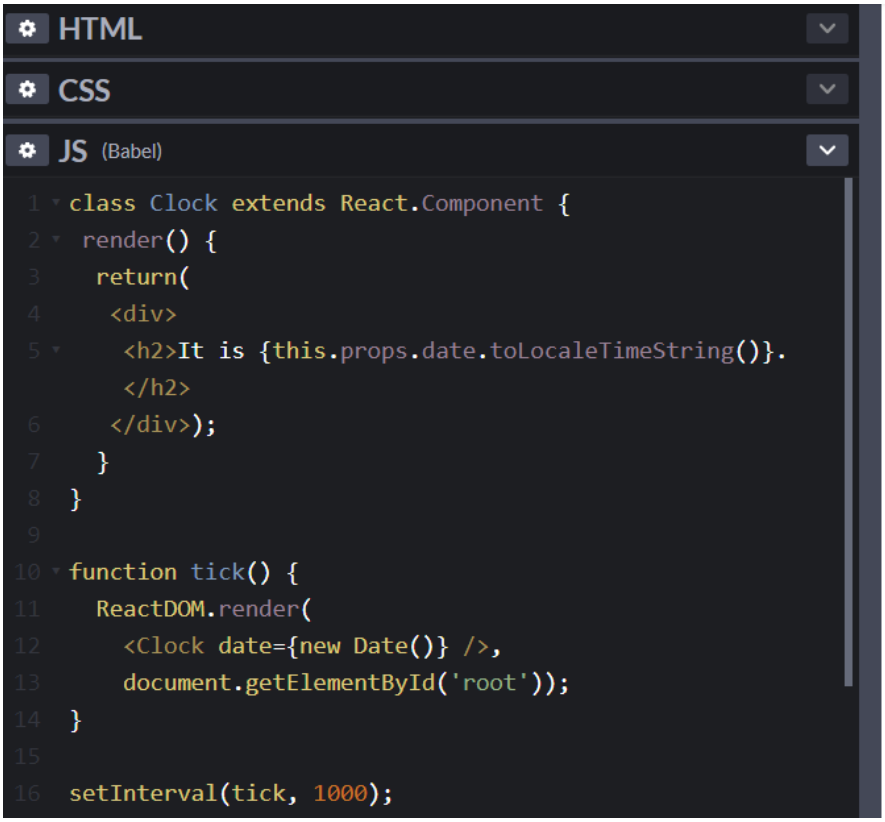
You can convert a function component like `Clock` to a class in five steps:

1. Create an ES6 class, 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>
    );
  }
}
```

Now Clock is a class rather than a function.

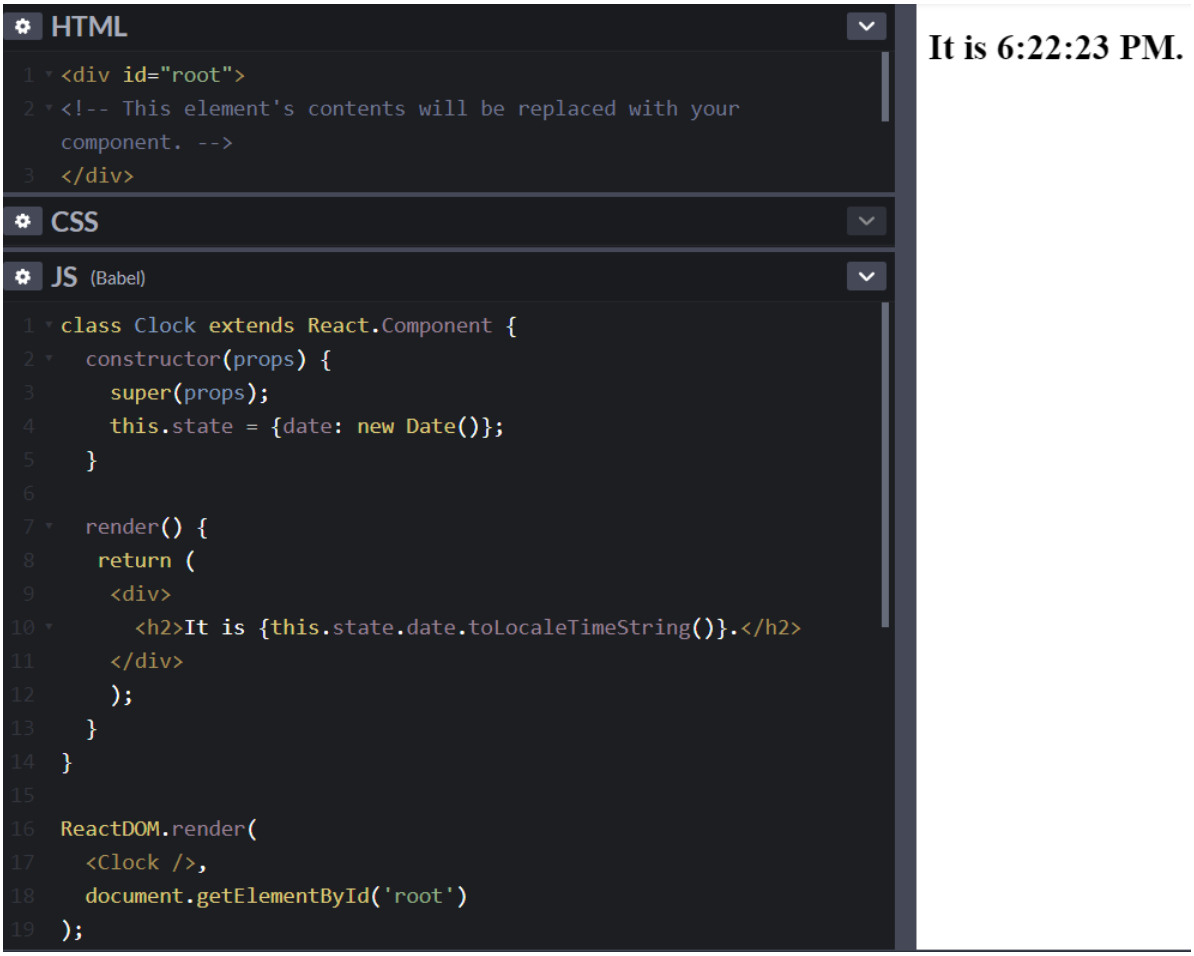
The render method will be called each time an update happens.



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.
2. Add a class constructor that assigns the initial `this.state`.
3. Remove the `date` prop from the `<Clock />` element



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

Adding Lifecycle Methods to a Class

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

Mounting

Mounting means putting elements into the DOM. The `componentDidMount()` method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`.

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

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

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

6. Handling Events

Handling events with React elements is very similar to handling events on DOM elements. React has the same events as HTML: click, change, mouseover etc.

Adding Events

React events are written in camelCase syntax:

`onClick` instead of `onclick`.

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

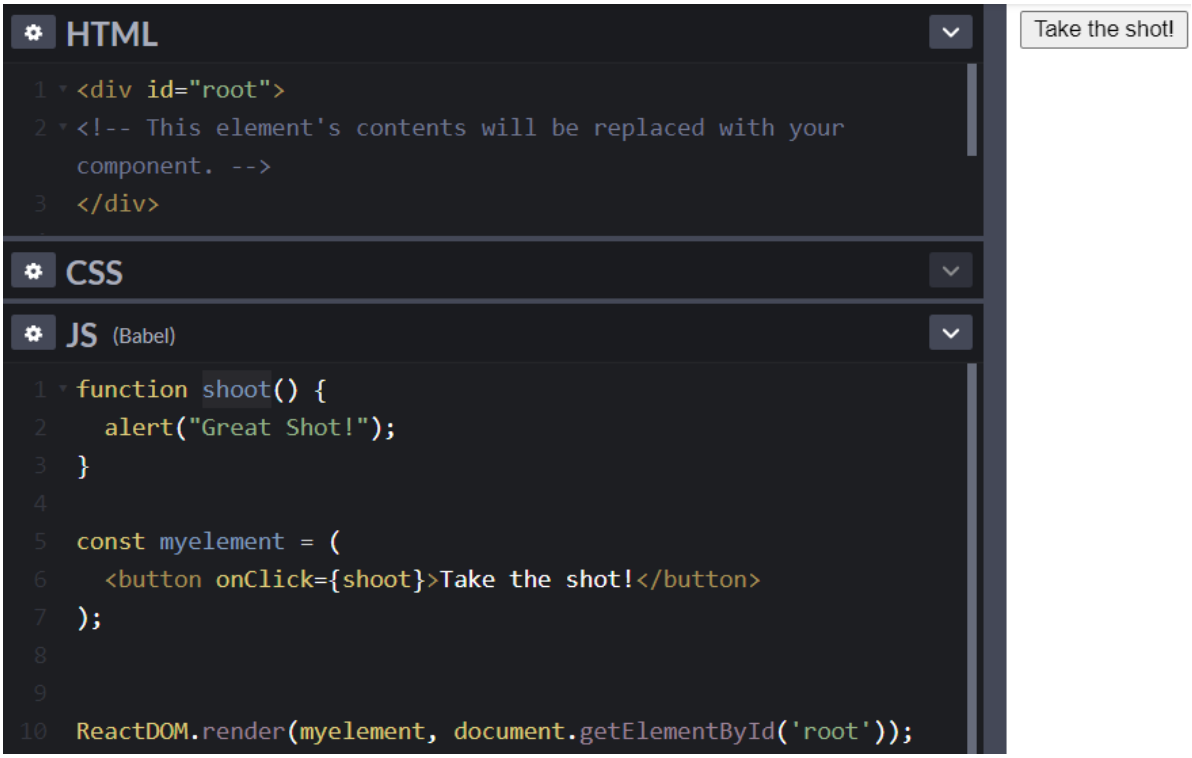
```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

React event handlers are written inside curly braces:

`onClick={shoot}` instead of `onClick="shoot()"`.



A good practice is to put the event handler as a method in the component class.

For methods in React, the `this` keyword should represent the component that owns the method.

That is why you should use arrow functions. With arrow functions, `this` will always represent the object that defined the arrow function.

```
class Football extends React.Component {
  shoot = () => {
    alert(this);
    /*
    The 'this' keyword refers to the component object
    */
  }
  render() {
    return (
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
}

ReactDOM.render(<Football />, document.getElementById('root'));
```

If you use regular functions instead of arrow functions you have to bind `this` to the component instance using the `bind()` method:

```
class Football extends React.Component {
  constructor(props) {
    super(props)
    this.shoot = this.shoot.bind(this)
  }
  shoot() {
    alert(this);
    /*
```

```
    Thanks to the binding in the constructor function,
    the 'this' keyword now refers to the component object
    */
  }
  render() {
    return (
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
}

ReactDOM.render(<Football />, document.getElementById('root'));
```

Passing Arguments

If you want to send parameters into an event handler, you have two options:

1. Make an anonymous arrow function:
2. Bind the event handler to `this`.

7. Conditional Rendering

Conditional rendering in React works the same way conditions work in JavaScript.

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

In this example we'll create a Greeting component that displays either of these component depending on whether a user is logged in:

HTML

```
<div id="root"></div>
```

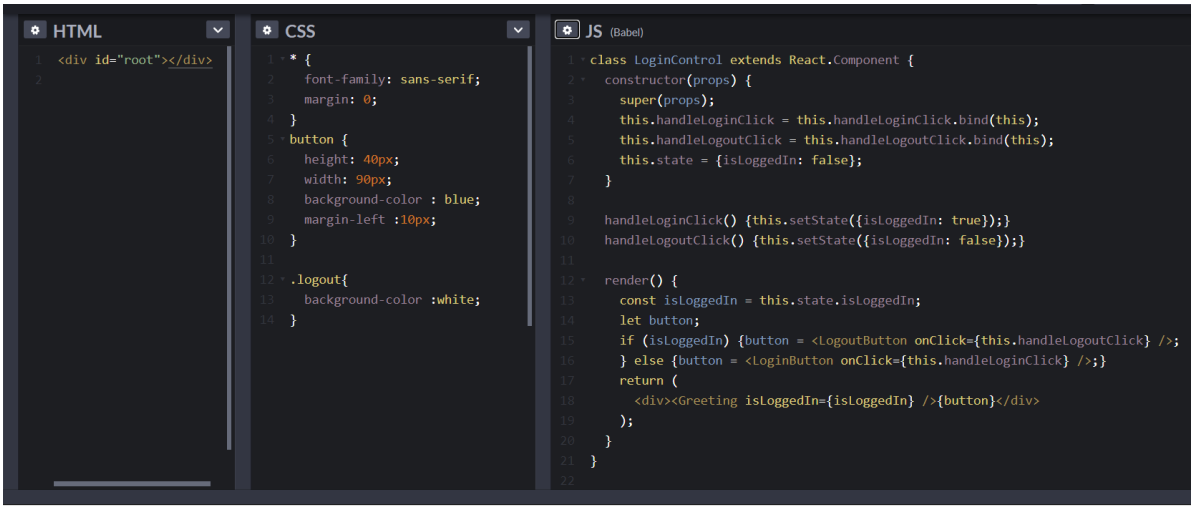
CSS

JS (Babel)

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

Please sign up.

In the example below, we will create a stateful component called `LoginControl` . It will render either `<LoginButton />` or `<LogoutButton />` depending on its current state.



Please sign up.

Login

JS Code:

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
```

```
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
```

```
      Logout
    </button>
  );
}

ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);
```

8. Lists & Keys

Lists

In JavaScript ,

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
```

In React, transforming arrays into lists of elements is nearly identical. You can build collections of elements and include them in JSX using curly braces `{}`.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

If we want to render lists inside a component.

We can reconstruct the previous example into a component that accepts an array of `numbers` and outputs a list of elements.

```
function NumberList(props) {
```

```
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

On executing this code, we'll be given a warning that a key should be provided for list items. A "key" is a special string attribute when creating list of elements.

Let's assign a `key` to our list items inside `numbers.map()` and fix the missing key issue.

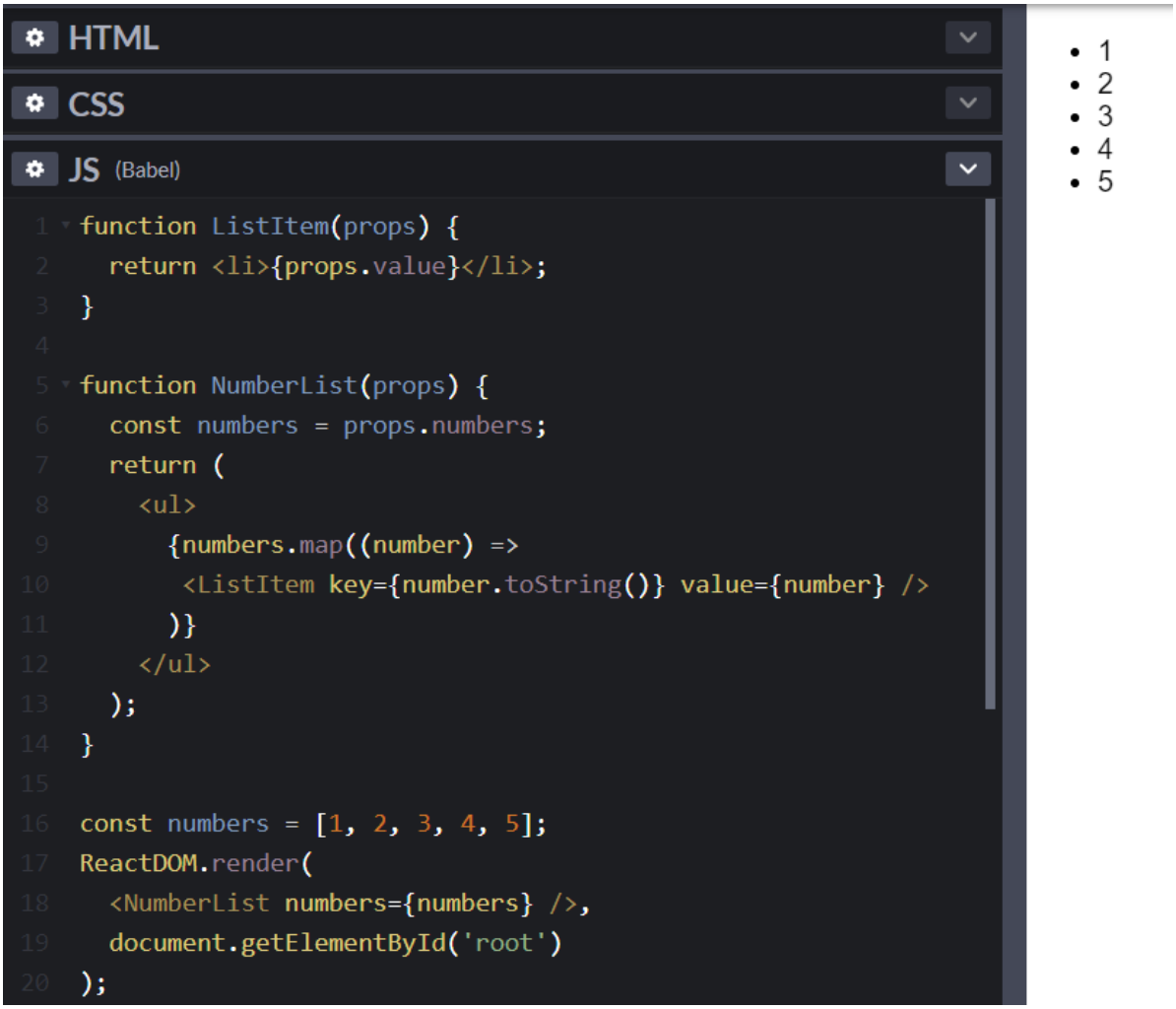
```
<li key={number.toString()}>
```

Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.

Keys used within arrays should be unique among their siblings. However, they don't need to be globally unique. We can use the same keys when we produce two different arrays.

JSX allows embedding any expression in curly braces so we could inline the map() result:



9. Forms

HTML form elements work a bit differently from other DOM elements in React, because form elements naturally keep some internal state.

It's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".