React Top-Level API

`React` is the entry point to the React library. If you load React from a `<script>` tag, these top-level APIs are available on the `React` global. If you use ES6 with npm, you can write `import React from 'react'`. If you use ES5 with npm, you can write `var React = require('react')`.

Overview

Components

React components let you split the UI into independent, reusable pieces, and think about each piece in isolation. React components can be defined by subclassing `React.Component` or `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

If you don't use ES6 classes, you may use the `create-react-class` module instead. See Using React without ES6 for more information.

React components can also be defined as functions which can be wrapped:

- `React.memo`

Creating React Elements

We recommend using JSX to describe what your UI should look like. Each JSX element is just syntactic sugar for calling `React.createElement()`. You will not typically invoke the following methods directly if you are using JSX.

- `createElement()`
- `createFactory()`

See Using React without JSX for more information.

Transforming Elements

`React` provides several APIs for manipulating elements:

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

`React` also provides a component for rendering multiple elements without a wrapper.

`React` also provides a component for rendering multiple elements without a wrapper.

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

Suspense lets components "wait" for something before rendering. Today, Suspense only supports one use case: loading components dynamically with `React.lazy`. In the future, it will support other use cases like data fetching.

- `React.lazy`
- `React.Suspense`

Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class. Hooks have a dedicated docs section and a separate API reference:

- Basic Hooks
  - `useState`
  - `useEffect`
  - `useContext`
- Additional Hooks
  - `useReducer`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`
  - `useDebugValue`

Reference

`React.Component`

`React.Component` is the base class for React components when they are defined using ES6 classes:

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

See the React.Component API Reference for a list of methods and properties related to the base `React.Component` class.

`React.PureComponent`

`React.PureComponent` is similar to `React.Component`. The difference between them is that `React.Component` doesn't implement `shouldComponentUpdate()`, but `React.PureComponent` implements it with a shallow prop and state comparison.

If your React component's `render()` function renders the same result given the same props and state, you can use `React.PureComponent` for a performance boost in some cases.

Note

`React.PureComponent`'s `shouldComponentUpdate()` only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only extend `PureComponent` when you expect to have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data.

Furthermore, `React.PureComponent`'s `shouldComponentUpdate()` skips prop updates for the whole component subtree. Make sure all the children components are also "pure".

`React.memo`

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

`React.memo` is a higher order component. It's similar to `React.PureComponent` but for function components instead of classes.

If your function component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

`React.memo` only checks for prop changes. If your function component wrapped in `React.memo` has a `useState` or `useContext` Hook in its implementation, it will still rerender when state or context change.

By default it will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.

```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render.
```

```
  otherwise return false
  */
}
export default React.memo(MyComponent, areEqual);
```

This method only exists as a **performance optimization.** Do not rely on it to "prevent" a render, as this can lead to bugs.

Note
Unlike the `shouldComponentUpdate()` method on class components, the `areEqual` function returns `true` if the props are equal and `false` if the props are not equal. This is the inverse from `shouldComponentUpdate`.

`createElement()`

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Create and return a new React element of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), a React component type (a class or a function), or a React fragment type.

Code written with JSX will be converted to use `React.createElement()`. You will not typically invoke `React.createElement()` directly if you are using JSX. See React Without JSX to learn more.

`cloneElement()`

```
React.cloneElement(
  element,
  [props],
  [...children]
)
```

Clone and return a new React element using `element` as the starting point. The resulting element will have the original element's props with the new props merged in shallowly. New children will replace existing children. `key` and `ref` from the original element will be preserved.

`React.cloneElement()` is almost equivalent to:

```
<element.type {...element.props} {...props}>{children}</element.type>
```

However, it also preserves `ref`s. This means that if you get a child with a `ref` on it, you won't accidentally steal it from your ancestor. You will get the same `ref` attached to your new element.

This API was introduced as a replacement of the deprecated `React.addons.cloneWithProps()`.

`createFactory()`

```
React.createFactory(type)
```

Return a function that produces React elements of a given type. Like `React.createElement()`, the type argument can be either a tag name string (such as `'div'` or `'span'`), a React component type (a class or a function), or a React fragment type.

This helper is considered legacy, and we encourage you to either use JSX or use `React.createElement()` directly instead.

You will not typically invoke `React.createFactory()` directly if you are using JSX. See React Without JSX to learn more.

`isValidElement()`

```
React.isValidElement(object)
```

Verifies the object is a React element. Returns `true` or `false`.

`React.Children`

`React.Children` provides utilities for dealing with the `this.props.children` opaque data structure.

`React.Children.map`

```
React.Children.map(children, function[(thisArg)])
```

Invokes a function on every immediate child contained within `children` with `this` set to `thisArg`. If `children` is an array it will be traversed and the function will be called for each child in the array. If children is `null` or `undefined`, this method will return `null` or `undefined` rather than an array.

> **Note**
> If `children` is a `Fragment` it will be treated as a single child and not traversed.

`React.Children.forEach`

```
React.Children.forEach(children, function[(thisArg)])
```

Like `React.Children.map()` but does not return an array.

`React.Children.count`

```
React.Children.count(children)
```

Returns the total number of components in `children`, equal to the number of times that a callback passed to `map` or `forEach` would be invoked.

`React.Children.only`

```
React.Children.only(children)
```

Verifies that `children` has only one child (a React element) and returns it. Otherwise this method throws an error.

Note:
`React.Children.only()` does not accept the return value of `React.Children.map()` because it is an array rather than a React element.

`React.Children.toArray`

```
React.Children.toArray(children)
```

Returns the `children` opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice `this.props.children` before passing it down.

Note:
`React.Children.toArray()` changes keys to preserve the semantics of nested arrays when flattening lists of children. That is, `toArray` prefixes each key in the returned array so that each element's key is scoped to the input array containing it.

`React.Fragment`

The `React.Fragment` component lets you return multiple elements in a `render()` method without creating an additional DOM element:

```
render() {
  return (
    <React.Fragment>
```

```
      Some text.

      <h2>A heading</h2>
    </React.Fragment>
  );
}
```

You can also use it with the shorthand `<></>` syntax. For more information, see React v16.2.0: Improved Support for Fragments.

`React.createRef`

`React.createRef` creates a ref that can be attached to React elements via the ref attribute.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }

  componentDidMount() {
    this.inputRef.current.focus();
  }
}
```

`React.forwardRef`

`React.forwardRef` creates a React component that forwards the ref attribute it receives to another component below in the tree. This technique is not very common but is particularly useful in two scenarios:

- Forwarding refs to DOM components
- Forwarding refs in higher-order-components

`React.forwardRef` accepts a rendering function as an argument. React will call this function with `props` and `ref` as two arguments. This function should return a React node.

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

In the above example, React passes a `ref` given to `<FancyButton ref={ref}>` element as a second argument to the rendering function inside the `React.forwardRef` call. This rendering function passes the `ref` to the `<button ref={ref}>` element.

As a result, after React attaches the ref, `ref.current` will point directly to the `<button>` DOM element instance.

For more information, see forwarding refs.

`React.lazy`

`React.lazy()` lets you define a component that is loaded dynamically. This helps reduce the bundle size to delay loading components that aren't used during the initial render.

You can learn how to use it from our code splitting documentation. You might also want to check out this article explaining how to use it in more detail.

```
// This component is loaded dynamically
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Note that rendering `lazy` components requires that there's a `<React.Suspense>` component higher in the rendering tree. This is how you specify a loading indicator.

Note
Using `React.lazy` with dynamic import requires Promises to be available in the JS environment. This requires a polyfill on IE11 and below.

`React.Suspense`

`React.Suspense` lets you specify the loading indicator in case some components in the tree below it are not yet ready to render. Today, lazy loading components is the **only** use case supported by `<React.Suspense>`:

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

It is documented in our code splitting guide. Note that `lazy` components can be deep inside the `Suspense` tree — it doesn't have to wrap every one of them. The best practice is to place `<Suspense>` where you want to see a loading indicator, but to use `lazy()` wherever you want to do code splitting.

While this is not supported today, in the future we plan to let `Suspense` handle more scenarios such as data fetching. You can read about this in our roadmap.

Note:
`React.lazy()` and `<React.Suspense>` are not yet supported by `ReactDOMServer`. This is a known limitation that will be resolved in the future.

Edit this page