# Glossary of React Terms

## Single-page Application

A single-page application is an application that loads a single HTML page and all the necessary assets (such as JavaScript and CSS) required for the application to run. Any interactions with the page or subsequent pages do not require a round trip to the server which means the page is not reloaded.

Though you may build a single-page application in React, it is not a requirement. React can also be used for enhancing small parts of existing websites with additional interactivity. Code written in React can coexist peacefully with markup rendered on the server by something like PHP, or with other client-side libraries. In fact, this is exactly how React is being used at Facebook.

## ES6, ES2015, ES2016, etc

These acronyms all refer to the most recent versions of the ECMAScript Language Specification standard, which the JavaScript language is an implementation of. The ES6 version (also known as ES2015) includes many additions to the previous versions such as: arrow functions, classes, template literals, `let` and `const` statements. You can learn more about specific versions here.

## Compilers

A JavaScript compiler takes JavaScript code, transforms it and returns JavaScript code in a different format. The most common use case is to take ES6 syntax and transform it into synt that older browsers are capable of interpreting. Babel is the compiler most commonly used with React.

## Bundlers

Bundlers take JavaScript and CSS code written as separate modules (often hundreds of them), and combine them together into a few files better optimized for the browsers. Some bundlers commonly used in React applications include Webpack and Browserify.

## Package Managers

Package managers are tools that allow you to manage dependencies in your project. npm and Yarn are two package managers commonly used in React applications. Both of them are clients for the same npm package registry.

## CDN

CDN stands for Content Delivery Network. CDNs deliver cached, static content from a network of servers across the globe.

## JSX

JSX is a syntax extension to JavaScript. It is similar to a template language, but it has full power of JavaScript. JSX gets compiled to `React.createElement()` calls which return plain JavaScript objects called “React elements”. To get a basic introduction to JSX see the docs here and find a more in-depth tutorial on JSX here.

React DOM uses camelCase property naming convention instead of HTML attribute names. For example, `tabindex` becomes `tabIndex` in JSX. The attribute `class` is also written as `className` since `class` is a reserved word in JavaScript:

```
const name = 'Clementine';
ReactDOM.render(
  <h1 className="hello">My name is {name}!</h1>,
  document.getElementById('root')
);
```

## Elements

React elements are the building blocks of React applications. One might confuse elements with a more widely known concept of "components". An element describes what you want to see on the screen. React elements are immutable.

```
const element = <h1>Hello, world</h1>;
```

Typically, elements are not used directly, but get returned from components.

## Components

React components are small, reusable pieces of code that return a React element to be rendered to the page. The simplest version of React component is a plain JavaScript function that returns a React element:

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

Components can also be ES6 classes:

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

Components can be broken down into distinct pieces of functionality and used within other components. Components can return other components, arrays, strings and numbers. A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component. Component names should also always start with a capital letter (`<Wrapper/>` **not** `<wrapper/>`). See this documentation for more information on rendering components.

## `props`

`props` are inputs to a React component. They are data passed down from a parent component to a child component.

Remember that `props` are readonly. They should not be modified in any way:

```
// Wrong!
props.number = 42;
```

If you need to modify some value in response to user input or a network response, use `state` instead.

## `props.children`

`props.children` is available on every component. It contains the content between the opening and closing tags of a component. For example:

```
<Welcome>Hello world!</Welcome>
```

The string `Hello world!` is available in `props.children` in the `Welcome` component:

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

For components defined as classes, use `this.props.children`:

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

## `state`

A component needs `state` when some data associated with it changes over time. For example, a `Checkbox` component might need `isChecked` in its state, and a `NewsFeed` component might want to keep track of `fetchedPosts` in its state.

The most important difference between `state` and `props` is that `props` are passed from a parent component, but `state` is managed by the component itself. A component cannot change its `props`, but it can change its `state`. To do so, it must call `this.setState()`. Only components defined as classes can have state.

For each particular piece of changing data, there should be just one component that "owns" it in its state. Don't try to synchronize states of two different components. Instead, lift it up to their closest shared ancestor, and pass it down as props to both of them.

## Lifecycle Methods

Lifecycle methods are custom functionality that gets executed during the different phases of a component. There are methods available when the component gets created and inserted into the DOM (mounting), when the component updates, and when the component gets unmounted or removed from the DOM.

## Controlled vs. Uncontrolled Components

React has two different approaches to dealing with form inputs.

An input form element whose value is controlled by React is called a *controlled component*. When a user enters data into a controlled component a change event handler is triggered and your code decides whether the input is valid (by re-rendering with the updated value). If you do not re-render then the form element will remain unchanged.

An *uncontrolled component* works like form elements do outside of React. When a user inputs data into a form field (an input box, dropdown, etc) the updated information is reflected without React needing to do anything. However, this also means that you can't force the field to have a certain value.

In most cases you should use controlled components.

## Keys

A "key" is a special string attribute you need to include when creating arrays of elements. Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside an array to give the elements a stable identity.

Keys only need to be unique among sibling elements in the same array. They don't need to be unique across the whole application or even a single component.

Don't pass something like `Math.random()` to keys. It is important that keys have a "stable identity" across re-renders so that React can determine when items are added, removed, or re-ordered. Ideally, keys should correspond to unique and stable identifiers coming from your data, such as `post.id`.

## Refs

React supports a special attribute that you can attach to any component. The `ref` attribute can be an object created by `React.createRef()` function or a callback function, or a string (in legacy API). When the `ref` attribute is a callback function, the function receives the underlying DOM element or class instance (depending on the type of element) as its argument. This allows you to have direct access to the DOM element or component instance.

Use refs sparingly. If you find yourself often using refs to “make things happen” in your app, consider getting more familiar with top-down data flow.

## Events

Handling events with React elements has some syntactic differences:

- React event handlers are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

## Reconciliation

When a component’s props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called “reconciliation”.

API Reference

Contributing

FAQ

Reactiflux Chat

DEV Community

Facebook

Twitter

**COMMUNITY**

Community Resources

Tools

**MORE**

Tutorial

Blog

Acknowledgements

React Native