Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page provides an introduction to the idea of components. You can find a detailed component API reference here.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Functional and Class Components

The 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 "functional" because they are literally JavaScript functions.

You can also use an ES6 class to define a component:

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

The above two components are equivalent from React's point of view.

Classes have some additional features that we will discuss in the next sections. Until then, we will use functional components for their conciseness.

Rendering a Component

Previously, we only encountered React elements that represent DOM tags:

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

However, elements can also represent user-defined components:

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

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

For example, this code renders “Hello, Sara” on the page:

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

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Try it on CodePen

Let’s recap what happens in this example:

1. We call `ReactDOM.render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.

3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</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 and requires `Welcome` to be in scope.
>
> You can read more about the reasoning behind this convention here.

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an `App` component that renders `Welcome` many times:

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

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

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

Try it on CodePen

Typically, new React apps have a single `App` component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like `Button` and gradually work your way to the top of the view hierarchy.

Extracting Components

Don't be afraid to split components into smaller components.

For example, consider this `Comment` component:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Try it on CodePen

It accepts `author` (an object), `text` (a string), and `date` (a date) as props, and describes a comment on a social media website.

This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let's extract a few components from it.

First, we will extract `Avatar`:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />

  );
}
```

The `Avatar` doesn't need to know that it is being rendered inside a `Comment`. This is why we have given its prop a more generic name: `user` rather than `author`.

We recommend naming props from the component's own point of view rather than the context in which it is being used.

We can now simplify `Comment` a tiny bit:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Next, we will extract a `UserInfo` component that renders an `Avatar` next to the user's name:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

This lets us simplify `Comment` even further:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Try it on CodePen

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. 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.

Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this `sum` function:

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

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

In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React is pretty flexible but it has a single strict rule:

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

Of course, application UIs are dynamic and change over time. In the next section, we will introduce a new concept of "state". State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Previous article
Rendering Elements

Next article
State and Lifecycle

DOCS

- Installation
- Main Concepts
- Advanced Guides
- API Reference
- Contributing
- FAQ

CHANNELS

- GitHub
- Stack Overflow
- Discussion Forum
- Reactiflux Chat
- DEV Community
- Facebook
- Twitter

COMMUNITY

- Community Resources
- Tools

MORE

- Tutorial
- Blog
- Acknowledgements
- React Native