1. **Function component**

[Function components](https://reactjs.org/docs/components-and-props.html#function-and-class-components) are the simplest way to declare reusable components.  
They're just functions.

function Greeting() {

return <div>Hi there!</div>;

}

1.1 Collect props from the first argument of your function.

function Greeting(props) {

return <div>Hi {props.name}!</div>;

}

1.2 Define any number of local variables to do what you need in your function components.  
**Always return your React Component at the end.**

function Greeting(props) {

let style = {

fontWeight: "bold",

color: context.color

};

return <div style={style}>Hi {props.name}!</div>;

}

1.3 Set defaults for any required props using defaultProps.

function Greeting(props) {

return <div>Hi {props.name}!</div>;

}

Greeting.defaultProps = {

name: "Guest"

};

1. **Destructuring props**

Destructuring assignment is a JavaScript feature.It was added to the language in ES2015.  
Think of it like the opposite of literal assignment.

let person = { name: "chantastic" };

let { name } = person;

Works with Arrays too.

let things = ["one", "two"];

let [first, second] = things;

2.1 Destructuring assignment is used a lot in [function components](https://reactpatterns.com/#function-component).  
These component declarations below are equivalent.

function Greeting(props) {

return <div>Hi {props.name}!</div>;

}

function Greeting({ name }) {

return <div>Hi {name}!</div>;

}

2.2 There's a syntax for collecting remaining props into an object.  
It's called [rest parameter syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) and looks like this.

function Greeting({ name, ...restProps }) {

return <div>Hi {name}!</div>;

}

Those three dots (...) take all the remaining properties and assign them to the object restProps.

So, what do you do with restProps once you have it?  
Keep reading...

1. **JSX spread attributes**

Spread Attributes is a feature of [JSX](https://reactjs.org/docs/introducing-jsx.html). It's a syntax for providing an object's properties as JSX attributes.

Following the example from [Destructuring props](https://reactpatterns.com/#destructuring-props). We can **spread** restProps over our <div>.

function Greeting({ name, ...restProps }) {

return <div {...restProps}>Hi {name}!</div>;

}

This makes Greeting super flexible.  
We can pass DOM attributes to Greeting and trust that they'll be passed through to div.

<Greeting name="Fancy pants" className="fancy-greeting" id="user-greeting" />

Avoid forwarding non-DOM props to components.  
Destructuring assignment is popular because it gives you a way to separate component-specific props from DOM/platform-specific attributes.

function Greeting({ name, ...platformProps }) {

return <div {...platformProps}>Hi {name}!</div>;

}

1. **Merge destructured props with other values**

Components are abstractions.Good abstractions allow for extension.

4.1 Consider this component that uses a class attribute for style a button.

function MyButton(props) {

return <button className="btn" {...props} />;

}

This works great until we try to extend it with another class.

In other words, we are using MyButton component and passing it a prop whose name is same as name of attribute used in the element (<button>) returned by MyButton component.

<MyButton className="delete-btn">Delete...</MyButton>

In this case, delete-btn replaces btn.

**Why?**

The props.className being spread is overriding the className in our component, because {...props}is coming after className="btn" {...props. Order matters for [JSX spread attributes](https://reactpatterns.com/#jsx-spread-attributes).

**Spread Operator Rule?**

We can override properties of object used with spread operator by redefining then in the end.

We can change the order but now the className will **never** be anything but btn.

function MyButton(props) {

return <button {...props} className="btn" />;

}

4.2 We need to use destructuring assignment to get the incoming className and merge with the base className. We can do this simply by adding all values to an array and joining them with a space.

function MyButton({ className, ...props }) {

let classNames = ["btn", className].join(" ");

return <button className={classNames} {...props} />;

}

**4.3 Use**[**default values**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Default_values_2)**.** To guard from undefined showing up as a className.

function MyButton({ className = "", ...props }) {

let classNames = ["btn", className].join(" ");

return <button className={classNames} {...props} />;

}

1. **Conditional rendering**

You can't use if/else statements inside a component declarations.So [conditional (ternary) operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) and [short-circuit evaluation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Short-circuit_evaluation) are your friends.

**if**

{ condition && <span>Rendered when `truthy`</span>;

}

**unless**

{ condition || <span>Rendered when `falsy`</span>;

}

**if-else**

{ condition ? (<span>Rendered when `truthy`</span>)

: ( <span>Rendered when `falsy`</span>);

}

1. **Children types**

React can render children from most types.  
In most cases it's either an array or a string.

**String**

<div>Hello World!</div>

**Array**

<div>{["Hello ", <span>World</span>, "!"]}</div>

1. **Array as children**

7.1 Providing an array as children is a very common.It's how lists are drawn in React.

We use map() to create an array of React Elements for every value in the array.

<ul>

{ ["first", "second"].map(item => (

<li>{item}</li>

))

}

</ul>

That's equivalent to providing a literal array.

<ul>{[<li>first</li>, <li>second</li>]}</ul>

7.2 This pattern can be combined with destructuring, JSX Spread Attributes, and other components, for some serious terseness.

<ul> {

arrayOfMessageObjects.map(({ id, ...message }) => (

<Message key={id} **{...message}** />

))

}

</ul>

1. **Function as children**

React components don't support functions as children. However, [render props](https://reactpatterns.com/#render-prop) is a pattern for creating components that take functions as children.

**Render prop**

Here's a component that uses a render callback.It's not useful, but it's an easy illustration to start with.

const Width = ({ children }) => children(500);

The component calls children as a function, with some number of arguments. Here, it's the number 500.

8.1 To use this component, we give it a [function as children](https://reactpatterns.com/#function-as-children).

<Width>{width => <div>window is {width}</div>}</Width>

We get this output.

<div>window is 500</div>

8.2 With this setup, we can use this width to make rendering decisions.

<Width>

{

width => (width > 600 **?** <div>min-width requirement met!</div> **:** null)

}

</Width>

If we plan to use this condition a lot, we can define another components to encapsulate the reused logic.

const MinWidth = ({ width: minWidth, children }) => (

<Width>{width => (width > minWidth ? children : null)}</Width>

);

8.3 Obviously a static Width component isn't useful but one that watches the browser window is. Here's a sample implementation.

class WindowWidth extends React.Component {

constructor() {

super();

this.state = { width: 0 };

}

componentDidMount() {

this.setState({ width: window.innerWidth }, () =>

window.addEventListener("resize", ({ target }) =>

this.setState({ width: target.innerWidth })

)

);

}

render() {

return **this.props.children(this.state.width);**

}

}

Many developers favor [Higher Order Components](https://reactpatterns.com/#higher-order-component) for this type of functionality. It's a matter of preference.

1. **Proxy component** (A good example of HOC)

*(I'm not sure if this name makes sense)*

Buttons are everywhere in web apps. And every one of them must have the type attribute set to "button".

<button type="button">

Writing this attribute hundreds of times is error prone. We can write a higher level component to proxy props to a lower-level button component.

const Button = props =>

<button type="button" {...props}>

We can use Button in place of button and ensure that the type attribute is consistently applied everywhere.

<Button />

*// <button type="button"><button>*

<Button className="CTA">Send Money</Button>

*// <button type="button" class="CTA">Send Money</button>*

**Application #1 Style component**

This is a [Proxy component](https://reactpatterns.com/#proxy-component) applied to the practices of style.

Say we have a button. It uses classes to be styled as a "primary" button.

<button type="button" className="btn btn-primary">

We can generate this output using a couple single-purpose components.

import classnames from "classnames";

const PrimaryBtn = props => <Btn {...props} primary />;

const Btn = ({ className, primary, ...props }) => (

<button

type="button"

className={classnames("btn", primary && "btn-primary", className)}

{...props}

/>);

It can help to visualize this.

PrimaryBtn()

↳ Btn({primary: true})

↳ Button({className: "btn btn-primary"}, type: "button"})

↳ '<button type="button" class="btn btn-primary"></button>'

Using these components, all of these result in the same output.

<PrimaryBtn />

<Btn primary />

<button type="button" className="btn btn-primary" />

This can be a huge boon to style maintenance. It isolates all concerns of style to a single component.

1. **Event switch**

When writing event handlers it's common to adopt the handle{eventName} naming convention.

handleClick(e) { /\* do something \*/ }

For components that handle several event types, these function names can be repetitive. The names themselves might not provide much value, as they simply proxy to other actions/functions.

handleClick() { require("./actions/doStuff")(/\* action stuff\*/) }

handleMouseEnter() { this.setState({ hovered: true }) }

handleMouseLeave() { this.setState({ hovered: false }) }

Consider writing a single event handler for your component and switching on event.type.

handleEvent({type}) {

switch(type) {

case "click":

return require("./actions/doStuff")(/\* action dates \*/)

case "mouseenter":

return this.setState({ hovered: true })

case "mouseleave":

return this.setState({ hovered: false })

default:

return console.warn(`No case for event type "${type}"`)

}

}

Alternatively, for simple components, you can call imported actions/functions directly from components, using arrow functions.

<div onClick={() => someImportedAction({ action: "DO\_STUFF" })}

Don't fret about performance optimizations until you have problems. Seriously don't.

1. **Layout component**

Layout components result in some form of static DOM element. It might not need to update frequently, if ever.

Consider a component that renders two children side-by-side.

<HorizontalSplit

leftSide={<SomeSmartComponent />}

rightSide={<AnotherSmartComponent />}

/>

We can aggressively optimize this component.

While HorizontalSplit will be parent to both components, it will never be their owner. We can tell it to update never, without interrupting the lifecycle of the components inside.

class HorizontalSplit extends React.Component {

shouldComponentUpdate() {

return false;

}

render() {

return (

<FlexContainer>

<div>{this.props.leftSide}</div>

<div>{this.props.rightSide}</div>

</FlexContainer>

);

}

}

1. **Container component**

"A container does data fetching and then renders its corresponding sub-component. That’s it."—[Jason Bonta](https://twitter.com/jasonbonta)

Given this reusable CommentList component.

const CommentList = ({ comments }) => (

<ul>

{comments.map(comment => (

<li>

{comment.body}-{comment.author}

</li>

))}

</ul>

);

We can create a new component responsible for fetching data and rendering the CommentList function component.

class CommentListContainer extends React.Component {

constructor() {

super()

this.state = { comments: [] }

}

componentDidMount() {

$.ajax({

url: "/my-comments.json",

dataType: 'json',

success: comments =>

this.setState({comments: comments});

})

}

render() {

return **<CommentList comments={this.state.comments} />**

}

}

We can write different containers for different application contexts.

1. **State hoisting** (passing child component data to parent component)

[function-component](https://reactpatterns.com/#function-component) don't hold state (as the name implies).Events are changes in state. Their data needs to be **passed to** stateful container components parents.

This is called "state hoisting". It's accomplished by passing a callback from a container component to a child component.

class NameContainer extends React.Component {

render() {

return <Name onChange={newName => alert(newName)} />;

}

}

const Name = ({ onChange }) => (

<input onChange={e => onChange(e.target.value)} />

);

Name receives an onChange callback from NameContainer and calls on events.

The alert above makes for a terse demo but it's not changing state. Let's change the internal state of NameContainer.

class NameContainer extends React.Component {

constructor() {

super();

this.state = { name: "" };

}

render() {

return <Name onChange={newName => this.setState({ name: newName })} />;

}

}

The state is *hoisted* to the container, by the provided callback, where it's used to update local state. This sets a nice clear boundary and maximizes the re-usability of function component.

This pattern isn't limited to function components. Because function components don't have lifecycle events, you'll use this pattern with component classes as well.

[*Controlled input*](https://reactpatterns.com/#controlled-input)*is an important pattern to know for use with state hoisting*

*(It's best to process the event object on the stateful component)*

1. **Controlled input**

It's hard to talk about controlled inputs in the abstract. Let's start with an uncontrolled (normal) input and go from there.

<input type="text" />

When you fiddle with this input in the browser, you see your changes. This is normal.

A controlled input disallows the DOM mutations that make this possible. You set the value of the input in component-land and it doesn't change in DOM-land.

<input type="text" value="This won't change. Try it." />

Obviously static inputs aren't very useful to your users. So, we derive a value from state.

class ControlledNameInput extends React.Component {

constructor() {

super();

this.state = { name: "" };

}

render() {

return <input type="text" value={this.state.name} />;

}

}

Then, changing the input is a matter of changing component state.

return (

<input

value={this.state.name}

onChange={e => this.setState({ name: e.target.value })}

/>

);

This is a controlled input. It only updates the DOM when state has changed in our component. This is invaluable when creating consistent UIs.