React

Getting Started

> 💡 **React** is a JavaScript library for building user interfaces.

React for Designers

If you’re coming from a design background, these resources are a great place to get started.

Practical Tutorial (my chose 🏁)

If you prefer to **learn by doing,** check out our practical tutorial. In this tutorial, we build a tic-tac-toe game in React. You might be tempted to skip it because you’re not into building games — but give it a chance. The techniques you’ll learn in the tutorial are fundamental to building *any* React apps, and mastering it will give you a much deeper understanding.

Document

▼ 1. **Introducing JSX** 🃏

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

It is called JSX, and it is a syntax extension to JavaScript. Using it with React is to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

▼ Why JSX? → *Contain both logic and markup in one called 'component', easy to see and find error, other can be use but mostly is JSX.*

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

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.

React doesn’t require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

▼ Embedding Expressions in JSX

In the example below, we declare a variable called `name` and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Phan Đại';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

even with function

```
function formatName(user) {
  return user.name + ', born in ' + user.birthYear;
}

const user = {
  name: 'Dai',
  birthYear: '1999'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

▼ JSX is an Expression Too

This means that JSX can be inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions.

```
function getGreeting(user){
  if(user)
    return <h1> Xin chào , {formatUser(user)}!</h1>
  else
    return <h1> Xin chào nguười lạ </h1>
```

▼ Specifying Attributes with JSX

Use quotes to specify string literals as attributes :

```
const element = <div tabIndex="0"></div>;
```

Also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

▼ Specifying Children with JSX

If a tag is empty, close it with />, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = {
  <div>
    <h1> Sao </h1>
    <h2> May nhin cai gi ? </h2>
  </div>
```

▼ JSX Prevents Injection Attacks

It is safe to embed user input in JSX

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that’s not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

▼ JSX Represents Objects

Babel compiles JSX down to `React.createElement()` calls.

Turn this

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

to this

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

▼ 2. **Rendering Elements** ⚡

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

We call this a “root” DOM node because everything inside it will be managed by React DOM.

If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, pass both to

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

▼ Updating the Rendered Element

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to `ReactDOM.render()`.

Consider this ticking clock example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>);
  ReactDOM.render(element, document.getElementById('root'));}
setInterval(tick, 1000);
```

▼ React Only Updates What’s Necessary

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM.

> In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

▼ 3. Components and Props 🅿️

▼ Function and Class Components → Each has pros

```
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.

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

▼ Rendering a Component → User-defined function

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 and children 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')
);
```

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>`.

▼ Composing Components → Many small functions in the big one

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')
);
```

💡 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 mind to crush it into many small

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>);
}
```

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} />
    );
}
```

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>
  );
}
```

▼ Props are Read-Only → Pure and not related props.

💡 Whether you declare a component as a function or a class, it must never modify its own props.

```
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;
}
```

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

▼ 4. **State and Lifecycle** 

In Rendering Elements, we have only learned one way to update the UI. We call `ReactDOM.render()` to change the rendered output:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

In this section, we will learn how to make the `Clock` component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

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 make this, we need 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>
    );
  }
}
```

▼ 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:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. Add a class constructor that assigns the initial `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
// the same render() { }
}
```

💡 Class components should always call the base constructor with props.

3. Remove the date prop from the <Clock /> element

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

4. The Result

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

▼ Adding Lifecycle Methods to a Class

Problem: In applications with many components, it's very important to free up resources taken by the components when they are destroyed.

We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called "mounting" in React.

We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called "unmounting" in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
  }
  componentWillUnmount() {
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

💡 These methods are called “lifecycle methods”.

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

```
// Like Constructors
componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

Note how we save the timer ID right on `this` (`this.timerID`).

While `this.props` is set up by React itself and `this.state` has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn’t participate in the data flow (like a timer ID).

We will tear down the timer in the `componentWillUnmount()` lifecycle method:

```
componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

Finally, we will implement a method called `tick()` that the `Clock` component will run every second.

It will use `this.setState()` to schedule updates to the component local state:

```
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>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

▼ Using State Correctly

There are three things you should know about `setState()`.

▼ Do Not Modify State Directly

For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
```

Instead, use `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

The only place where you can assign `this.state` is the constructor.

▼ State Updates May Be Asynchronous

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of `setState()` that accepts a function rather than an object. That function will receive **the previous state** as **the first argument**, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

▼ State Updates are Merged

When you call `setState()`, React merges the object you provide into the current state.

For example, your state may contain several independent variables:

```
constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

Then you can update them independently with separate `setState()` calls:

```
componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
```

```
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

▼ The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

```
<FormattedDate date={this.state.date} />
```

The `FormattedDate` component would receive the date in its props and wouldn't know *whether it came from the Clock's state, from the Clock's props, or was typed by hand:*

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an `App` component that renders three `<Clock>`s:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

▼ 5. **Handling Events** 🌲

There are some syntax differences:

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

Example:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

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 link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. For example, this Toggle component renders a button that lets the user toggle between "ON" and "OFF" states:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, this will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without `()` after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. If you are using the experimental public class fields syntax, you can use class fields to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in Create React App.

If you aren't using class fields syntax, you can use an arrow function in the callback

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the `LoggingButton` renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

▼ Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and `Function.prototype.bind` respectively.

In both cases, the `e` argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with `bind` any further arguments are automatically forwarded.

▼ 6. **Conditional Rendering** 🎉

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like `if` or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Consider these two components:

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

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

We'll create a `Greeting` component that displays either of these components depending on whether a user is logged in:

```
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')
);
```

▼ Element Variables

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn't change.

Consider these two new components representing Logout and Login buttons:

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

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

In the example below, we will create a stateful component called `LoginControl`.

It will render either `<LoginButton />` or `<LogoutButton />` depending on its current state. It will also render a `<Greeting />` from the previous example:

```
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>
    );
  }
}

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

While declaring a variable and using an if statement is a fine way to conditionally render a component, sometimes you might want to use a shorter syntax.

There are a few ways to inline conditions in JSX, explained below.

▼ Inline If with Logical && Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator `condition ? true : false`.

In the example below, we use it to conditionally render a small block of text.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

It can also be used for larger expressions although it is less obvious what's going on:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
```

```
  );
}
```

▼ Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this return `null` instead of its render output.

In the example below, the `<WarningBanner />` is rendered depending on the value of the prop called `warn`. If the value of the prop is `false`, then the component does not render:

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

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

▼ 7. **Lists and Keys** 📄

First, let's review how you transform lists in JavaScript.

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

This code logs `[2, 4, 6, 8, 10]` to the console.

In React, transforming arrays into lists of elements is nearly identical.

▼ Rendering Multiple Components

You can build collections of elements and include them in JSX using curly braces `{}`.

Below, we loop through the numbers array using the JavaScript `map()` function. We return a `<li>` element for each item. Finally, we assign the resulting array of elements to `listItems`:

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

We include the entire `listItems` array inside a `<ul>` element, and render it to the DOM:

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

This code displays a bullet list of numbers between 1 and 5.

▼ Basic List Component

Usually you would render lists inside a component.

We can refactor 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')
);
```

When you run this code, you'll be given a warning that a key should be provided for list items. A "key" is a special string attribute you need to include when creating lists of elements.

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

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

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

▼ 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:

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

The best way to pick a key is to use a unique string the same as IDs from your data as keys:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort:

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

⚠️ We don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state. If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

Here is an in-depth explanation about why keys are necessary if you’re interested in learning more.

▼ Extracting Components with Keys

Keys only make sense in the context of the surrounding array.

For example, if you extract a `ListItem` component, you should keep the key on the `<ListItem />` elements in the array rather than on the `<li>` element in the `ListItem` itself.

Example: Incorrect Key Usage

```
function ListItem(props) {
  const value = props.value;
  return (
    // Wrong! There is no need to specify the key here:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Wrong! The key should have been specified here:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

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

Example: Correct Key Usage

```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}
```

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

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

A good rule of thumb is that elements inside the `map()` call need keys.

▼ Keys Must Only Be Unique Among Siblings

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:

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
  {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
  {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
  <Blog posts={posts} />,
  document.getElementById('root')
);
```

Keys serve as a hint to React but they don't get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name:

```
const content = posts.map((post) =>
  <Post
    key={post.id}
       // `key` can't be listed in
          // { key : ... , title : ... }
       // so new another with 'id'
    id={post.id}
    title={post.title} />
);
```

With the example above, the Post component can read `props.id`, but not `props.key`.

▼ Embedding `map()` in JSX

In the examples above we declared a separate `listItems` variable and included it in JSX:

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

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

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

👉 Sometimes this results in clearer code, but this style can also be abused. Like in JavaScript, it is up to you to decide whether it is worth extracting a variable for readability. Keep in mind that if the `map()` body is too nested, it might be a good time to extract a component.

▼ 8. **Forms** 📰

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, 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".

▼ Controlled Components

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with `setState()`.

We can combine the two by making the React state be the "single source of truth". Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a "controlled component".

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }
```

```
  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Since the `value` attribute is set on our form element, the displayed value will always be `this.state.value`, making the React state the source of truth.
Since `handleChange` runs on every keystroke to update the React state, the displayed value will update as the user types.

With a controlled component, the input's value is always driven by the React state. While this means you have to type a bit more code, you can now pass the value to other UI elements too, or reset it from other event handlers.

▼ The `textarea` Tag

In HTML, a `<textarea>` element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

In React, a `<textarea>` uses a value attribute instead. This way, a form using a `<textarea>` can be written very similarly to a form that uses a single-line input:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }
```

```
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Notice that `this.state.value` is initialized in the constructor, so that the text area starts off with some text in it.

▼ The `select` Tag

In HTML, `<select>` creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Note that the Coconut option is initially selected, because of the selected attribute. React, instead of using this selected attribute, uses a value attribute on the root select tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
```

```
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Overall, this makes it so that `<input type="text">`, `<textarea>`, and `<select>` all work very similarly - they all accept a value attribute that you can use to implement a controlled component.

You can pass an array into the value attribute, allowing you to select multiple options in a select tag:

```
<select multiple={true} value={['B', 'C']}>
```

▼ The `file input` Tag

In HTML, an `<input type="file">` lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the File API.

```
<input type="file" />
```

Because its value is read-only, it is an uncontrolled component in React. It is discussed together with other uncontrolled components later in the documentation.

▼ Handling Multiple Inputs

When you need to handle multiple controlled `input` elements, you can add a `name` attribute to each element and let the handler function choose what to do based on the value of `event.target.name`.

For example:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
```

```
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

```
this.setState({
  [name]: value
});
```

It is equivalent to this ES5 code:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Also, since `setState()` automatically merges a partial state into the current state, we only needed to call it with the changed parts.

▼ Controlled Input Null Value

Specifying the value prop on a controlled component prevents the user from changing the input unless you desire so. If you've specified a `value` but the input is still editable, you may have accidentally set `value` to `undefined` or `null`.

The following code demonstrates this. (The input is locked at first but becomes editable after a short delay.)

```
ReactDOM.render(<input value="hi" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={null} />, mountNode);
}, 1000);
```

▼ Alternatives to Controlled Components

It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out uncontrolled components, an alternative technique for implementing input forms.

▼ Fully-Fledged Solutions

If you're looking for a complete solution including validation, keeping track of the visited fields, and handling form submission, `Formik` is one of the popular choices. However, it is built on the same principles of controlled components and managing state — so don't neglect to learn them.

▼ 9. **Lifting State Up** 🗽

Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor. Let's see how this works in action.

In this section, we will create a temperature calculator that calculates whether the water would boil at a given temperature.

We will start with a component called `BoilingVerdict`. It accepts the `celsius` temperature as a prop, and prints whether it is enough to boil the water:

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

Next, we will create a component called `Calculator`. It renders an `<input>` that lets you enter the temperature, and keeps its value in `this.state.temperature`.

Additionally, it renders the `BoilingVerdict` for the current input value.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
```

```
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

▼ Adding a Second Input

Our new requirement is that, in addition to a Celsius input, we provide a Fahrenheit input, and they are kept in sync.

We can start by extracting a `TemperatureInput` component from `Calculator`. We will add a new `scale` prop to it that can either be `"c"` or `"f"`:

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

We can now change the Calculator to render two separate temperature inputs:

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
```

```
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

We have two inputs now, but when you enter the temperature in one of them, the other doesn't update. This contradicts our requirement: we want to keep them in sync.

We also can't display the `BoilingVerdict` from `Calculator`. The `Calculator` doesn't know the current temperature because it is hidden inside the `TemperatureInput`.

▼ Writing Conversion Functions

First, we will write two functions to convert from Celsius to Fahrenheit and back:

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

We will write another function that takes a string `temperature` and a converter function as arguments and returns a string. We will use it to calculate the value of one input based on the other input.

It returns an empty string on an invalid `temperature`, and it keeps the output rounded to the third decimal place:

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

For example, `tryConvert('abc', toCelsius)` returns an empty string, and `tryConvert('10.22', toFahrenheit)` returns '50.396'.

▼ Lifting State Up

Currently, both `TemperatureInput` components independently keep their values in the local state:

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
```

```
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
```

When we update the Celsius input, the Fahrenheit input should reflect the converted temperature, and vice versa.

In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it. This is called “lifting state up”. We will remove the local state from the `TemperatureInput` and move it into the `Calculator` instead.

If the `Calculator` owns the shared state, it becomes the “source of truth” for the current temperature in both inputs. It can instruct them both to have values that are consistent with each other. Since the props of both `TemperatureInput` components are coming from the same parent `Calculator` component, the two inputs will always be in sync.

Let’s see how this works step by step.

First, we will replace `this.state.temperature` with `this.props.temperature` in the `TemperatureInput` component. For now, let’s pretend `this.props.temperature` already exists, although we will need to pass it from the `Calculator` in the future:

```
render() {
    // Before: const temperature = this.state.temperature;
    const temperature = this.props.temperature;
    // ...
```

We know that props are read-only. When the `temperature` was in the local state, the `TemperatureInput` could just call `this.setState()` to change it. However, now that the `temperature` is coming from the parent as a prop, the `TemperatureInput` has no control over it.

In React, this is usually solved by making a component “controlled”. Just like the DOM `<input>` accepts both a `value` and an `onChange` prop, so can the custom `TemperatureInput` accept both `temperature` and `onTemperatureChange` props from its parent `Calculator`.

Now, when the `TemperatureInput` wants to update its temperature, it calls `this.props.onTemperatureChange`:

```
handleChange(e) {
    // Before: this.setState({temperature: e.target.value});
    this.props.onTemperatureChange(e.target.value);
    // ...
```

The `onTemperatureChange` prop will be provided together with the `temperature` prop by the parent `Calculator` component. It will handle the change by modifying its own local state, thus re-rendering both inputs with the new values. We will look at the new `Calculator` implementation very soon.

Before diving into the changes in the `Calculator`, let's recap our changes to the `TemperatureInput` component. We have removed the local state from it, and instead of reading `this.state.temperature`, we now read `this.props.temperature`. Instead of calling `this.setState()` when we want to make a change, we now call `this.props.onTemperatureChange()`, which will be provided by the `Calculator`:

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Now let's turn to the `Calculator` component.

We will store the current input's `temperature` and `scale` in its local state. This is the state we "lifted up" from the inputs, and it will serve as the "source of truth" for both of them. It is the minimal representation of all the data we need to know in order to render both inputs.

For example, if we enter 37 into the Celsius input, the state of the `Calculator` component will be:

```
{
  temperature: '37',
  scale: 'c'
}
```

If we later edit the Fahrenheit field to be 212, the state of the `Calculator` will be:

```
{
  temperature: '212',
```

```
  scale: 'f'
}
```

We could have stored the value of both inputs but it turns out to be unnecessary. It is enough to store the value of the most recently changed input, and the scale that it represents. We can then infer the value of the other input based on the current `temperature` and `scale` alone.

The inputs stay in sync because their values are computed from the same state:

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

Now, no matter which input you edit, `this.state.temperature` and `this.state.scale` in the Calculator get updated. One of the inputs gets the value as is, so any user input is preserved, and the other input value is always recalculated based on it.

▼ Lessons Learned

There should be a single “source of truth” for any data that changes in a React application. Usually, the state is first added to the component that needs it for rendering. Then, if other components also need it, you can lift it up to their closest

common ancestor. Instead of trying to sync the state between different components, you should rely on the top-down data flow.

Lifting state involves writing more “boilerplate” code than two-way binding approaches, but as a benefit, it takes less work to find and isolate bugs. Since any state “lives” in some component and that component alone can change it, the surface area for bugs is greatly reduced. Additionally, you can implement any custom logic to reject or transform user input.

If something can be derived from either props or state, it probably shouldn’t be in the state. For example, instead of storing both `celsiusValue` and `fahrenheitValue`, we store just the last edited `temperature` and its `scale`. The value of the other input can always be calculated from them in the `render()` method. This lets us clear or apply rounding to the other field without losing any precision in the user input.

When you see something wrong in the UI, you can use React Developer Tools to inspect the props and move up the tree until you find the component responsible for updating the state. This lets you trace the bugs to their source:

▼ 10. **Composition vs Inheritance** 🔃

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

▼ Containment (recommended tip)

Some components don’t know their children ahead of time. This is especially common for components like `Sidebar` or `Dialog` that represent generic “boxes”.

We recommend that such components use the special `children` prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
```

```
  );
}
```

Anything inside the `<FancyBorder>` JSX tag gets passed into the `FancyBorder` component as a `children` prop. Since `FancyBorder` renders `{props.children}` inside a `<div>`, the passed elements appear in the final output.

While this is less common, sometimes you might need multiple “holes” in a component. In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

React elements like `<Contacts />` and `<Chat />` are just objects, so you can pass them as props like any other data. This approach may remind you of “slots” in other libraries but there are no limitations on what you can pass as props in React.

▼ Specialization

Sometimes we think about components as being “special cases” of other components. For example, we might say that a `WelcomeDialog` is a special case of `Dialog`.

In React, this is also achieved by composition, where a more “specific” component renders a more “generic” one and configures it with props:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
```

```
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}
```

Composition works equally well for components defined as classes:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
              message="How should we refer to you?">
        <input value={this.state.login}
               onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

▼ So What About Inheritance?

At Facebook, we use React in thousands of components, and we haven’t found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

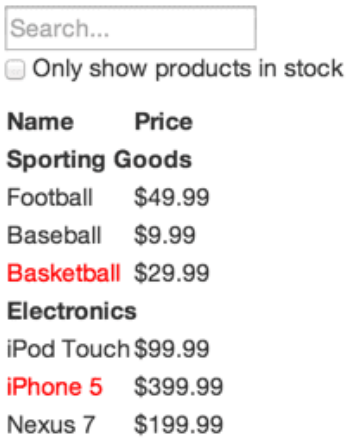
▼ 11. **Thinking In React**

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this document, we'll walk you through the thought process of building a searchable product data table using React.

▼ Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:



Our JSON API returns some data that looks like this:

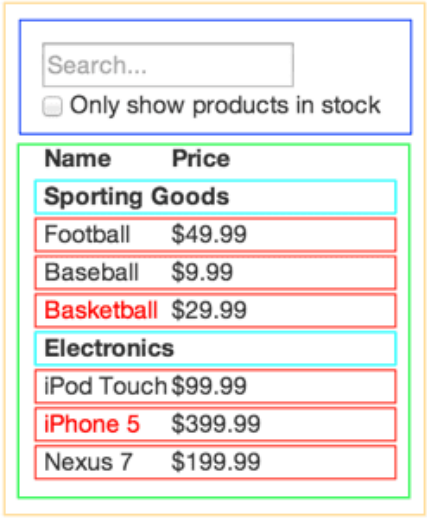
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

▼ Step 1: Break The UI Into A Component Hierarchy

The first thing you'll want to do is to draw boxes around every component (and subcomponent) in the mock and give them all names. If you're working with a designer, they may have already done this, so go talk to them! Their Photoshop layer names may end up being the names of your React components!

But how do you know what should be its own component? Use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

Since you're often displaying a JSON data model to a user, you'll find that if your model was built correctly, your UI (and therefore your component structure) will map nicely. That's because UI and data models tend to adhere to the same *information architecture*. Separate your UI into components, where each component matches one piece of your data model.



You'll see here that we have five components in our app. We've italicized the data each component represents.

1. `FilterableProductTable` **(orange):** contains the entirety of the example
2. `SearchBar` **(blue):** receives all *user input*
3. `ProductTable` **(green):** displays and filters the *data collection* based on *user input*
4. `ProductCategoryRow` **(turquoise):** displays a heading for each *category*
5. `ProductRow` **(red):** displays a row for each *product*

If you look at `ProductTable`, you'll see that the table header (containing the "Name" and "Price" labels) isn't its own component. This is a matter of preference, and there's an argument to be made either way. For this example, we left it as part of `ProductTable` because it is part of rendering the *data collection* which

is `ProductTable`'s responsibility. However, if this header grows to be complex (e.g., if we were to add affordances for sorting), it would certainly make sense to make this its own `ProductTableHeader` component.

Now that we've identified the components in our mock, let's arrange them into a hierarchy. Components that appear within another component in the mock should appear as a child in the hierarchy:

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

▼ Step 2: Build A Static Version in React

See the Pen Thinking In React: Step 2 on `CodePen`.

Now that you have your component hierarchy, it's time to implement your app. The easiest way is to build a version that takes your data model and renders the UI but has no interactivity. It's best to decouple these processes because building a static version requires a lot of typing and no thinking, and adding interactivity requires a lot of thinking and not a lot of typing. We'll see why.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using *props*. *props* are a way of passing data from parent to child. If you're familiar with the concept of *state*, **don't use state at all** to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.

You can build top-down or bottom-up. That is, you can either start with building the components higher up in the hierarchy (i.e. starting with `FilterableProductTable`) or with the ones lower in it (`ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up and write tests as you build.

At the end of this step, you'll have a library of reusable components that render your data model. The components will only have `render()` methods since this is a static version of your app. The component at the top of the hierarchy (`FilterableProductTable`) will take your data model as a prop. If you make a change to your underlying data model and call `ReactDOM.render()` again, the UI will be updated. You can see how your UI is updated and where to make changes. React's **one-way data flow** (also called *one-way binding*) keeps everything modular and fast.

Refer to the React docs if you need help executing this step.

▼ A Brief Interlude: Props vs State

There are two types of “model” data in React: props and state. It’s important to understand the distinction between the two; skim the official React docs if you aren’t sure what the difference is. See also FAQ: What is the difference between state and props?

▼ Step 3: Identify The Minimal (but complete) Representation Of UI State

To make your UI interactive, you need to be able to trigger changes to your underlying data model. React achieves this with **state**.

To build your app correctly, you first need to think of the minimal set of mutable state that your app needs. The key here is DRY: *Don’t Repeat Yourself*. Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand. For example, if you’re building a TODO list, keep an array of the TODO items around; don’t keep a separate state variable for the count. Instead, when you want to render the TODO count, take the length of the TODO items array.

Think of all of the pieces of data in our example application. We have:

- The original list of products
- The search text the user has entered
- The value of the checkbox
- The filtered list of products

Let’s go through each one and figure out which one is state. Ask three questions about each piece of data:

1. Is it passed in from a parent via props? If so, it probably isn’t state.
2. Does it remain unchanged over time? If so, it probably isn’t state.
3. Can you compute it based on any other state or props in your component? If so, it isn’t state.

The original list of products is passed in as props, so that’s not state. The search text and the checkbox seem to be state since they change over time and can’t be computed from anything. And finally, the filtered list of products isn’t state because it can be computed by combining the original list of products with the search text and value of the checkbox.

So finally, our state is:

- The search text the user has entered
- The value of the checkbox

▼ Step 4: Identify Where Your State Should Live

See the Pen Thinking In React: Step 4 on CodePen.

OK, so we’ve identified what the minimal set of app state is. Next, we need to identify which component mutates, or *owns*, this state.

Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand,** so follow these steps to figure it out:

For each piece of state in your application:

- Identify every components that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's run through this strategy for our application:

- `ProductTable` needs to filter the product list based on state and `SearchBar` needs to display the search text and checked state.
- The common owner component is `FilterableProductTable`.
- It conceptually makes sense for the filter text and checked value to live in `FilterableProductTable`

Cool, so we've decided that our state lives in `FilterableProductTable`. First, add an instance property `this.state = {filterText: '', inStockOnly: false}` to `FilterableProductTable`'s `constructor` to reflect the initial state of your application. Then,
pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as a prop. Finally, use these props to filter the rows in `ProductTable` and set the values of the form fields in `SearchBar`.

You can start seeing how your application will behave:
set `filterText` to `"ball"` and refresh your app. You'll see that the data table is updated correctly.

▼ Step 5: Add Inverse Data Flow

See the Pen Thinking In React: Step 5 on CodePen.

So far, we've built an app that renders correctly as a function of props and state flowing down the hierarchy. Now it's time to support data flowing the other way: the form components deep in the hierarchy need to update the state in `FilterableProductTable`.

React makes this data flow explicit to help you understand how your program works, but it does require a little more typing than traditional two-way data binding.

If you try to type or check the box in the current version of the example, you'll see that React ignores your input. This is intentional, as we've set the `value` prop of

the `input` to always be equal to the `state` passed in from `FilterableProductTable`.

Let's think about what we want to happen. We want to make sure that whenever the user changes the form, we update the state to reflect the user input. Since components should only update their own state, `FilterableProductTable` will pass callbacks to `SearchBar` that will fire whenever the state should be updated. We can use the `onChange` event on the inputs to be notified of it. The callbacks passed by `FilterableProductTable` will call `setState()`, and the app will be updated.

▼ And That's It

Hopefully, this gives you an idea of how to think about building components and applications with React. While it may be a little more typing than you're used to, remember that code is read far more than it's written, and it's less difficult to read this modular, explicit code. As you start to build large libraries of components, you'll appreciate this explicitness and modularity, and with code reuse, your lines of code will start to shrink.

Hook

▼ **1. Introducing Hooks**

▼ **2. Hooks at a Glance**

State Hook

This example renders a counter. When you click the button, it increments the value:

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

React will preserve this state between re-renders. `useState` returns a pair: the *current* state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It's similar to `this.setState` in a class, except it doesn't merge the old and new state together. (We'll show an example comparing `useState` to `this.state` in Using the State Hook.)

The only argument to `useState` is the initial state. In the example above, it is `0` because our counter starts from zero. Note that unlike `this.state`, the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.

Declaring multiple state variables

You can use the State Hook more than once in a single component:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

The array destructuring syntax lets us give different names to the state variables we declared by calling `useState`. These names aren't a part of the `useState` API. Instead, React assumes that if you call `useState` many times, you do it in the same order during every render. We'll come back to why this works and when this is useful later.

But what is a Hook?

Hooks are functions that let you "hook into" React state and lifecycle features from function components. Hooks don't work inside classes — they let you use React without classes. (We don't recommend rewriting your existing components overnight but you can start using Hooks in the new ones if you'd like.)

React provides a few built-in Hooks like `useState`. You can also create your own Hooks to reuse `stateful` behavior between different components. We'll look at the built-in Hooks first.

🔫 Effect Hook

You've likely performed data fetching, subscriptions, or manually changing the DOM from React components before. We call these operations "side effects" (or "effects" for short) because they can affect other components and can't be done during rendering.

The Effect Hook, `useEffect`, adds the ability to perform side effects from a function component. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes, but unified into a single API. (We'll show examples comparing `useEffect` to these methods in Using the Effect Hook.)

For example, this component sets the document title after React updates the DOM:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);
```

```
  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

When you call `useEffect`, you're telling React to run your "effect" function after flushing changes to the DOM. Effects are declared inside the component so they have access to its props and state. By default, React runs the effects after every render — *including* the first render. (We'll talk more about how this compares to class lifecycles in Using the Effect Hook.)

Effects may also optionally specify how to "clean up" after them by returning a function. For example, this component uses an effect to subscribe to a friend's online status, and cleans up by unsubscribing from it:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

In this example, React would unsubscribe from our `ChatAPI` when the component unmounts, as well as before re-running the effect due to a subsequent render. (If you want, there's a way to tell React to skip re-subscribing if the `props.friend.id` we passed to `ChatAPI` didn't change.)

Just like with `useState`, you can use more than a single effect in a component:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
```

```
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
```

Hooks let you organize side effects in a component by what pieces are related (such as adding and removing a subscription), rather than forcing a split based on lifecycle methods. (`Didmount` and `UnMount`)

🚦 Rules of Hooks

Hooks are JavaScript functions, but they impose two additional rules:

- Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — your own custom Hooks. We'll learn about them in a moment.)

We provide a linter plugin to enforce these rules automatically. We understand these rules might seem limiting or confusing at first, but they are essential to making Hooks work well.

💡 Building Your Own Hooks

Traditionally, there were two popular solutions to this problem: higher-order components and render props. Custom Hooks let you do this, but without adding more components to your tree.

Earlier on this page, we introduced a `FriendStatus` component that calls the `useState` and `useEffect` Hooks to subscribe to a friend's online status. Let's say we also want to reuse this subscription logic in another component.

First, we'll extract this logic into a custom Hook called `useFriendStatus`:

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
```

```
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

It takes `friendID` as an argument, and returns whether our friend is online.

Now we can use it from both components:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

The state of each component is completely independent. Hooks are a way to reuse *stateful logic*, not state itself. In fact, each *call* to a Hook has a completely isolated state — so you can even use the same custom Hook twice in one component.

Custom Hooks are more of a convention than a feature. If a function's name starts with "`use`" and it calls other Hooks, we say it is a custom Hook. The `useSomething` naming convention is how our linter plugin is able to find bugs in the code using Hooks.

🏗 Other Hooks

There are a few less commonly used built-in Hooks that you might find useful. For example, `useContext` lets you subscribe to React context without introducing nesting:

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

And `useReducer` lets you manage local state of complex components with a reducer:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
```

Advanced - 22

▼ **1. Accessibility** 💻

▼ Why Accessibility?

Web accessibility (also referred to as **a11y**) is the design and creation of websites that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages.

React fully supports building accessible websites, often by using standard HTML techniques.

▼ Standards and Guidelines

WCAG : Web Content Accessibility Guidelines

WAI-ARIA : Web Accessibility Initiative - Accessible Rich Internet Applications

Note that all aria-* HTML attributes are fully supported in JSX.

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

Don't care much cause it was helpful for HTML 4

Semantic HTML : is the foundation of accessibility in a web application. Using the various HTML elements to reinforce the meaning of information in our websites will often give us accessibility for free.

Sometimes we break HTML semantics when we add `<div>` elements to our JSX to make our React code work, especially when working with lists (`<ol>, <ul> and <dl>`) and the HTML `<table>`. In these cases we should rather use React `Fragments` to group together multiple elements.

```
import React, { Fragment } from 'react';

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
```

```
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Fragments should also have a `key` prop when mapping collections
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

When you don’t need any props on the Fragment tag you can use the short syntax, if your tooling supports it:

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

▼ Accessible Forms

▼ Labeling

Although these standard HTML practices can be directly used in React, note that the for attribute is written as `htmlFor` in JSX:

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

▼ Notifying the user of errors

Error situations need to be understood by all users.

```
<h1>3 Errors – Billing Address</h1>
<title>3 Errors – Billing Address</title>
<!-- or use Dialog -->
<script>
document.getElementById('alertconfirm')
  .addEventListener('click', function() {
    /* [… code saving data …] */
    alert('Thanks for submitting the form!');
  });
</script>
```

▼ Focus Control

Ensure that your web application can be fully operated with the keyboard only:

There is a table including many of the most common online interactions, the standard keystrokes for the interaction, and additional information on things to consider during testing.

Access here : Table

▼ Keyboard focus and focus outline

Keyboard focus refers to the current element in the DOM that is selected to accept input from the keyboard. We see it everywhere as a focus outline similar to that shown in the following image:

Only ever use CSS that removes this outline, for example by setting `outline: 0`, if you are replacing it with another focus outline implementation.

▼ Mechanisms to skip to desired content

Provide a mechanism to allow users to skip past navigation sections in your application as this assists and speeds up keyboard navigation.

Skiplinks or Skip Navigation Links are hidden navigation links that only become visible when keyboard users interact with the page. They are very easy to implement with internal page anchors and some styling:

- WebAIM - Skip Navigation Links

```
<a href="#gotomain"/> Skip to main content
<main id="gotomain">
  Bla Bla bla
</main>
```

Also use landmark elements and roles, such as `<main>` and `<aside>`, to demarcate page regions as assistive technology allow the user to quickly navigate to these sections.

Read more about the use of these elements to enhance accessibility here:

- Accessible Landmarks

▼ Programmatically managing focus

Our React applications continuously modify the HTML DOM during runtime, sometimes leading to keyboard focus being lost or set to an unexpected element. In order to repair this, we need to programmatically nudge the keyboard focus in the right direction. For example, by resetting keyboard focus to a button that opened a modal window after that modal window is closed.

MDN Web Docs takes a look at this and describes how we can build keyboard-navigable JavaScript widgets.

To set focus in React, we can use Refs to DOM elements.

Using this, we first create a ref to an element in the JSX of a component class:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Create a ref to store the textInput DOM element
    this.textInput = React.createRef();
  }
  render() {
  // Use the `ref` callback to store a reference to the text input DOM
  // element in an instance field (for example, this.textInput).
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

Then we can focus it elsewhere in our component when needed:

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  // Note: we're accessing "current" to get the DOM node
  this.textInput.current.focus();
}
```

Sometimes a parent component needs to set focus to an element in a child component. We can do this by exposing DOM refs to parent components through a special prop on the child component that forwards the parent's ref to the child's DOM node.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
```

```
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// Now you can set focus when required.
this.inputElement.current.focus();
```

When using a HOC to extend components, it is recommended to forward the ref to the wrapped component using the `forwardRef` function of React. If a third party HOC does not implement ref forwarding, the above pattern can still be used as a fallback.

A great focus management example is the react-aria-modal. This is a relatively rare example of a fully accessible modal window. Not only does it set initial focus on the cancel button (preventing the keyboard user from accidentally activating the success action) and trap keyboard focus inside the modal, it also resets focus back to the element that initially triggered the modal.

While this is a very important accessibility feature, it is also a technique that should be used judiciously. Use it to repair the keyboard focus flow when it is disturbed, not to try and anticipate how users want to use applications.

▼ Mouse and pointer events

Ensure that all functionality exposed through a mouse or pointer event can also be accessed using the keyboard alone. Depending only on the pointer device will lead to many cases where keyboard users cannot use your application.

To illustrate this, let's look at a prolific example of broken accessibility caused by click events. This is the outside click pattern, where a user can disable an opened popover by clicking outside the element.

Image GIF here → Example

This is typically implemented by attaching a click event to the window object that closes the popover:

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

```
    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target)) {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

This may work fine for users with pointer devices, such as a mouse, but operating this with the keyboard alone leads to broken functionality when tabbing to the next element as the window object never receives a click event. This can lead to obscured functionality which blocks users from using your application.

Image GIF here → Example

Click here for more information about Blur and Focus event

The same functionality can be achieved by using appropriate event handlers instead, such as `onBlur` and `onFocus`:

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

    this.state = { isOpen: false };
    this.timeOutId = null;
```

```
    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // We close the popover on the next tick by using setTimeout.
  // This is necessary because we need to first check if
  // another child of the element has received focus as
  // the blur event fires prior to the new focus event.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // If a child receives focus, do not close the popover.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React assists us by bubbling the blur and
    // focus events to the parent.
    return (
      <div onBlur={this.onBlurHandler}
           onFocus={this.onFocusHandler}>
        <button onClick={this.onClickHandler}
                aria-haspopup="true"
                aria-expanded={this.state.isOpen}>
          Select an option
        </button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

This code exposes the functionality to both pointer device and keyboard users. Also note the added aria-* props to support screen-reader users. For simplicity's sake the keyboard events to enable arrow key interaction of the popover options have not been implemented.

This is one example of many cases where depending on only pointer and mouse events will break functionality for keyboard users. Always testing with the keyboard will immediately highlight the problem areas which can then be fixed by using keyboard aware event handlers.

▼ Other Points for Consideration

Setting the language

Indicate the human language of page texts as screen reader software uses this to select the correct voice settings:

- WebAIM - Document Language

Setting the document title

Set the document `<title>` to correctly describe the current page content as this ensures that the user remains aware of the current page context:

- WCAG - Understanding the Document Title Requirement

We can set this in React using the React Document Title Component.

Color contrast

Ensure that all readable text on your website has sufficient color contrast to remain maximally readable by users with low vision:

- WCAG - Understanding the Color Contrast Requirement
- Everything About Color Contrast And Why You Should Rethink It
- A11yProject - What is Color Contrast

It can be tedious to manually calculate the proper color combinations for all cases in your website so instead, you can calculate an entire accessible color palette with Colorable.

Both the aXe and WAVE tools mentioned below also include color contrast tests and will report on contrast errors.

If you want to extend your contrast testing abilities you can use these tools:

- WebAIM - Color Contrast Checker
- The Paciello Group - Color Contrast Analyzer

Development and Testing Tools

▼ 2. **Code-Splitting** 🎞️

▼ Bundling

Most React apps will have their files “bundled” using tools like `Webpack`, `Rollup` or `Browserify`. Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle:

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

console.log(add(16, 26)); // 42
```

▼ Code Splitting

Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don't accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it's good to get ahead of the problem and start "splitting" your bundle. Code-Splitting is a feature supported by bundlers like `Webpack`, `Rollup` and `Browserify` (via factor-bundle) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you "lazy-load" just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven't reduced the overall amount of code in your app, you've avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

`import()`

The best way to introduce code-splitting into your app is through the dynamic `import()` syntax.

Before:

```
import { add } from './math';

console.log(add(16, 26));
```

After:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

`React.lazy`

The `React.lazy` function lets you render a dynamic import as a regular component.

Before:

```
import OtherComponent from './OtherComponent';
```

After:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

This will automatically load the bundle containing the `OtherComponent` when this component is first rendered.

`React.lazy` takes a function that must call a dynamic `import()`. This must return a `Promise` which resolves to a module with a default export containing a React component.

The lazy component should then be rendered inside a `Suspense` component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

The fallback prop accepts any React elements that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap multiple lazy components with a single Suspense component.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
```

```
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

Error boundaries

If the other module fails to load (for example, due to network failure), it will trigger an error. You can handle these errors to show a nice user experience and manage recovery with Error Boundaries. Once you've created your Error Boundary, you can use it anywhere above your lazy components to display an error state when there's a network error.

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

Route-based code splitting

Most people on the web are used to page transitions taking some amount of time to load. You also tend to be re-rendering the entire page at once so your users are unlikely to be interacting with other elements on the page at the same time.

Here's an example of how to setup route-based code splitting into your app using libraries like React Router with `React.lazy`.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
```

```
    </Suspense>
  </Router>
);
```

Named Exports

`React.lazy` currently only supports default exports. If the module you want to import uses named exports, you can create an intermediate module that reexports it as the default. This ensures that tree shaking keeps working and that you don't pull in unused components.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

▼ 3. **Context** 📑 - You tube recommended 🚨

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

When to Use Context

Context is designed to share data that can be considered "global" for a tree of React components such as the current authenticated user, theme, or preferred language.

For example, in the code below we manually thread through a "theme" prop in order to style the Button component:

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // The Toolbar component must take an extra "theme" prop
  // and pass it to the ThemedButton. This can become painful
  // if every single button in the app needs to know the theme
  // because it would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
```

```
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

Using context, we can avoid passing props through intermediate elements:

```
// Context lets us pass a value deep into the component tree
// without explicitly threading it through every component.
// Create a context for the current theme (with "light" as the default).
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// A component in the middle doesn't have to
// pass the theme down explicitly anymore.
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Assign a contextType to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Before You Use Context

Context is primarily used when some data needs to be accessible by many components at different nesting levels.

⚠️ *Apply it sparingly because it makes component reuse more difficult.*

The simpler way to avoid passing some props through many levels, component composition is recommended because it can store some default value for props in new version when created.

For example, consider a `Page` component that passes a `user` and `avatarSize` prop several levels down so that deeply nested Link and Avatar components can read it:

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
  <PageLayout user={user} avatarSize={avatarSize} />
  // ... which renders ...
    <NavigationBar user={user} avatarSize={avatarSize} />
    // ... which renders ...
      <Link href={user.permalink}>
        <Avatar user={user} size={avatarSize} />
      </Link>
```

One way to solve this issue without context is to pass down the Avatar component itself so that the intermediate components don't need to know about the user or `avatarSize` props:

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Now, we have:
<Page user={user} avatarSize={avatarSize} />
  // ... which renders ...
  <PageLayout userLink={...} />
    // ... which renders ...
    <NavigationBar userLink={...} />
      // ... which renders ...
      {props.userLink}
```

With this change, only the top-most Page component needs to know about the Link and Avatar components' use of `user` and `avatarSize`.

This isn't the right choice in every case: moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

You're not limited to a single child for a component. You may pass multiple children, or even have multiple separate "slots" for children, as documented here:

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
```

```
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}
```

However, sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you "broadcast" such data, and changes to it, to all components below. Common examples where using context might be simpler than the alternatives include managing the current locale, theme, or a data cache.

API

`React.createContext`

```
const MyContext = React.createContext(defaultValue);
```

> ✔️ *When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.*

The `defaultValue` argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them.

> 📌 Note: passing `undefined` as a Provider value does not cause consuming components to use `defaultValue`.

`Context.Provider`

```
<MyContext.Provider value={/* some value */}>
```

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.

The Provider component accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

> ⚠️ *All consumers that are descendants of a Provider will re-render whenever the Provider's value prop changes.*

The propagation from Provider to its descendant consumers (including `.contextType` and `useContext`) is follow to the `shouldComponentUpdate` method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as Object.is.

`Class.contextType`

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. This lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

Or you can use a static class field to initialize your `contextType`.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

`Context.Consumer`

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

A React component that subscribes to context changes. This lets you subscribe to a context within a function component.

Requires a function as a child. The function receives the current context value and returns a React node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the `defaultValue` that was passed to `createContext()`.

`Context.displayName`

Context object accepts a `displayName` string property. `React DevTools` uses this string to determine what to display for the context.

For example, the following component will appear as `MyDisplayName` in the `DevTools`:

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

Base to this examples and try your best.

Caveats

Because context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider's parent re-renders. For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for value:

```
class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{something: 'something'}}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

To get around this, lift the value into the parent's state:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <Provider value={this.state.value}>
```

```
        <Toolbar />
      </Provider>
    );
  }
}
```

▼ 4. Error Boundaries 🚧

In the past, JavaScript errors inside components used to corrupt React's internal state and cause it to emit (alert) cryptic errors on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.

Introducing Error Boundaries

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

> ⚠️ *Note*
> *Error boundaries do not catch errors for:*
> *- Event handlers*
> *- Asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks)*
> *- Server side rendering*
> *- Errors thrown in the error boundary itself (rather than its children)*

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods `static`, `getDerivedStateFromError()` or `componentDidCatch()`.

> ℹ️ Use static `getDerivedStateFromError()` to render a fallback UI after an error has been thrown.

> ℹ️ Use `componentDidCatch()` to log error information.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
```

```
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Then you can use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

📌 Error boundaries work like a JavaScript catch {} block, but for components. Only class components can be error boundaries. In practice, most of the time you'll want to declare an error boundary component once and use it throughout your application.
Note that error boundaries only catch errors in the components below them in the tree. An error boundary can't catch an error within itself. If an error boundary fails trying to render the error message, the error will propagate to the closest error boundary above it. This, too, is similar to how catch {} block works in JavaScript.

Where to Place Error Boundaries

The granularity of error boundaries is up to you. You may wrap top-level route components to display a "Something went wrong" message to the user, just like server-side frameworks often handle crashes. You may also wrap individual widgets in an error boundary to protect them from crashing the rest of the application.

New Behavior for Uncaught Errors

This change has an important implication. As of React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree. (clear page : nothing is displayed)

For example, in a product like Messenger leaving the broken UI visible could lead to somebody sending a message to the wrong person. Similarly, it is worse for a payments app to display a wrong amount than to render nothing.

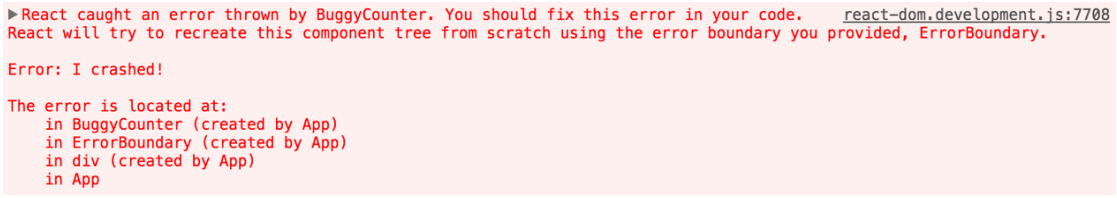
This change means that as you migrate to React 16, you will likely uncover existing crashes in your application that have been unnoticed before. Adding error boundaries lets you provide better user experience when something goes wrong.

For example, Facebook Messenger wraps content of the sidebar, the info panel, the conversation log, and the message input into separate error boundaries. If some component in one of these UI areas crashes, the rest of them remain interactive.

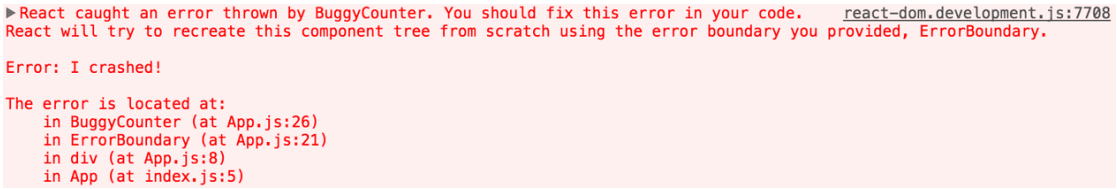
> 📣 We also encourage you to **use JS error reporting services** (or build your own) so that you can learn about unhandled exceptions as they happen in production, and fix them.

Component Stack Traces

Now you can see where exactly in the component tree the failure has happened:



You can also see the filenames and line numbers in the component stack trace. This works by default in Create React App projects:



How About try/catch?

try / catch is great but it only works for imperative code

```
try {
  showButton();
} catch (error) {
  // ...
}
```

However, React components are declarative and specify what should be rendered:

```
<Button />
```

Error boundaries preserve the declarative nature of React, and behave as you would expect. For example, even if an error occurs in a `componentDidUpdate` method caused by a `setState` somewhere deep in the tree, it will still correctly propagate to the closest error boundary.

How About Event Handlers?

⛔ Error boundaries do not catch errors inside event handlers.

React doesn't need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle methods, the event handlers don't happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside event handler, use the regular JavaScript try / catch statement:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

Naming Changes from React 15

▼ 5. Forwarding Refs 🚩 - You tube recommended 🚨

📌 *Technique for automatically passing a ref through a component to one of its children.*

Forwarding refs to DOM components

```
function FancyButton(props) {
  return (
    <button className="FancyButton">
      {props.children}
    </button>
  );
}
```

React components hide their implementation details, including their rendered output. Other components using `FancyButton` usually will not need to obtain a ref to the inner button DOM element. This is good because it prevents components from relying on each other's DOM structure too much.

Although such encapsulation is desirable for application-level components like `FeedStory` or `Comment`, it can be inconvenient for highly reusable "leaf" components like `FancyButton` or `MyTextInput`. These components tend to be used throughout the application in a similar manner as a regular DOM button and input, and accessing their DOM nodes may be unavoidable for managing focus, selection, or animations.

Ref forwarding is an opt-in feature that lets some components take a ref they receive, and pass it further down (in other words, "forward" it) to a child.

In the example below, `FancyButton` uses `React.forwardRef` to obtain the ref passed to it, and then forward it to the DOM button that it renders:

```
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>;
```

This way, components using `FancyButton` can get a ref to the underlying button DOM node and access it if necessary—just like if they used a DOM button directly.

Here is a step-by-step explanation of what happens in the above example:

1. We create a React ref by calling `React.createRef` and assign it to a `ref` variable.
2. We pass our `ref` down to `<FancyButton ref={ref}>` by specifying it as a JSX attribute.
3. React passes the `ref` to the `(props, ref) => ...` function inside `forwardRef` as a second argument.
4. We forward this `ref` argument down to `<button ref={ref}>` by specifying it as a JSX attribute.
5. When the ref is attached, `ref.current` will point to the `<button>` DOM node.

Note for component library maintainers

✔️ When you start using `forwardRef` in a component library, you should treat it as a breaking change and release a new major version of your library.

This is because your library likely has an observably different behavior (such as what refs get assigned to, and what types are exported), and this can break apps and other libraries that depend on the old behavior.

Conditionally applying `React.forwardRef` when it exists is also not recommended for the same reasons: it changes how your library behaves and can break your users' apps when they upgrade React itself.

Forwarding refs in higher-order components

This technique can also be particularly useful with higher-order components (also known as HOCs). Let's start with an example HOC that logs component props to the console:

```
function logProps(WrappedComponent) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  }

  return LogProps;
}
```

The "`logProps`" HOC passes all props through to the component it wraps, so the rendered output will be the same. For example, we can use this HOC to log all props that get passed to our "fancy button" component:

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}

// Rather than exporting FancyButton, we export LogProps.
// It will render a FancyButton though.
export default logProps(FancyButton);
```

There is one caveat to the above example: refs will not get passed through. That's because `ref` is not a prop. Like `key`, it's handled differently by React. If you add a ref to a HOC, the ref will refer to the outermost container component, not the wrapped component.

This means that refs intended for our `FancyButton` component will actually be attached to the `LogProps` component:

```
import FancyButton from './FancyButton';

const ref = React.createRef();

// The FancyButton component we imported is the LogProps HOC.
// Even though the rendered output will be the same,
// Our ref will point to LogProps instead of the inner FancyButton component!
// This means we can't call e.g. ref.current.focus()
<FancyButton
  label="Click Me"
  handleClick={handleClick}
  ref={ref}
/>;
```

Fortunately, we can explicitly forward refs to the inner `FancyButton` component using the `React.forwardRef` API. `React.forwardRef` accepts a render function that receives props and ref parameters and returns a React node. For example:

```
function logProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      const {forwardedRef, ...rest} = this.props;

      // Assign the custom prop "forwardedRef" as a ref
      return <Component ref={forwardedRef} {...rest} />;
    }
  }

  // Note the second param "ref" provided by React.forwardRef.
  // We can pass it along to LogProps as a regular prop, e.g. "forwardedRef"
  // And it can then be attached to the Component.
  return React.forwardRef((props, ref) => {
    return <LogProps {...props} forwardedRef={ref} />;
  });
}
```

Displaying a custom name in DevTools

▼ 6. Fragments 🧱

> ✔️ *A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.*

There is also a new short syntax for declaring them.

Motivation

A common pattern is for a component to return a list of children. Take this example React snippet:

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

`<Columns />` would need to return multiple `<td>` elements in order for the rendered HTML to be valid. If a parent `div` was used inside the `render()` of `<Columns />`, then the resulting HTML will be invalid.

Results in a `<Table />` output of:

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
    </div>
  </tr>
</table>
```

Fragments solve this problem.

Usage

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

which results in a correct `<Table />` output of:

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
```

```
  </tr>
</table>
```

Short Syntax

There is a new, shorter syntax you can use for declaring fragments. It looks like empty tags:

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

Keyed Fragments

Fragments declared with the explicit `<React.Fragment>` syntax may have keys. A use case for this is mapping a collection to an array of fragments — for example, to create a description list:

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

> ⚠️ ***Key is the only attribute that can be passed to Fragment. In the future, we may add support for additional attributes, such as event handlers.***

Live Demo

▼ 7. Higher-Order Components 🧮 - Template 👥 - Complex implement 😟 - You tube recommended 🚨

✔️ A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from `React`'s compositional nature.

Concretely, a higher-order component is a function that takes a component and returns a new component.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as `Redux`'s connect and Relay's `createFragmentContainer`.

Use HOCs For Cross-Cutting Concerns

✔️ We previously recommended mixins as a way to handle cross-cutting concerns. We've since realized that mixins create more trouble than they are worth. Read more about why we've moved away from mixins and how you can transition your existing components.

Components are the primary unit of code reuse in React. However, you'll find that some patterns aren't a straightforward fit for traditional components.

For example, say you have a `CommentList` component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
```

```
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}
```

Later, you write a component for subscribing to a single blog post, which follows a similar pattern:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` and `BlogPost` aren't identical — they call different methods on `DataSource`, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to `DataSource`.
- Inside the listener, call `setState` whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to `DataSource` and calling `setState` will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share it across many components. This is where higher-order components excel.

We can write a function that creates components, like `CommentList` and `BlogPost`, that subscribe to `DataSource`. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a `DataSource` and the current props.

When `CommentListWithSubscription` and `BlogPostWithSubscription` are rendered, `CommentList` and `BlogPost` will be passed a data prop with the most current data retrieved from `DataSource`:

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC *composes* the original component by *wrapping* it in a container component. A HOC is a pure function with zero side-effects.

And that's it! The wrapped component receives all the props of the container, along with a new prop, `data`, which it uses to render its output. The HOC isn't concerned with how or why the data is used, and the wrapped component isn't concerned with where the data came from.

Because `withSubscription` is a normal function, you can add as many or as few arguments as you like. For example, you may want to make the name of the `data` prop configurable, to further isolate the HOC from the wrapped component. Or you could accept an argument that configures `shouldComponentUpdate`, or one that configures the data source. These are all possible because the HOC has full control over how the component is defined.

Like components, the contract between `withSubscription` and the wrapped component is entirely props-based. This makes it easy to swap one HOC for a different one, as long as they provide the same props to the wrapped component. This may be useful if you change data-fetching libraries, for example.

Don't Mutate the Original Component. Use Composition.

Resist the temptation to modify a component's prototype (or otherwise mutate it) inside a HOC.

```
function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

There are a few problems with this. One is that the input component cannot be reused separately from the enhanced component. More crucially, if you apply another HOC to `EnhancedComponent` that also mutates `componentDidUpdate`, the first HOC's functionality will be overridden! This HOC also won't work with function components, which do not have lifecycle methods.

Mutating HOCs are a leaky abstraction—the consumer must know how they are implemented in order to avoid conflicts with other HOCs.

Instead of mutation, HOCs should use composition, by wrapping the input component in a container component:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
```

```
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

This HOC has the same functionality as the mutating version while avoiding the potential for clashes. It works equally well with class and function components. And because it's a pure function, it's composable with other HOCs, or even with itself.

You may have noticed similarities between HOCs and a pattern called **container components**. Container components are part of a strategy of separating responsibility between high-level and low-level concerns. Containers manage things like subscriptions and state, and pass props to components that handle things like rendering UI. HOCs use containers as part of their implementation. You can think of HOCs as parameterized container component definitions.

📌 Container is " < " + component + " /> "

Convention: Pass Unrelated Props Through to the Wrapped Component

HOCs add features to a component. They shouldn't drastically alter its contract. It's expected that the component returned from a HOC has a similar interface to the wrapped component.

HOCs should pass through props that are unrelated to its specific concern. Most HOCs contain a render method that looks something like this:

```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

🔥 This convention helps ensure that HOCs are as flexible and reusable as possible.

Convention: Maximizing Composability

🔥 Not all HOCs look the same. Sometimes they accept only a single argument, the wrapped component:

```
const NavbarWithRouter = withRouter(Navbar);
```

Usually, HOCs accept additional arguments. In this example from Relay, a config object is used to specify a component's data dependencies:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

The most common signature for HOCs looks like this:

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

What?! If you break it apart, it's easier to see what's going on.

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

In other words, `connect` is a higher-order function that returns a higher-order component!

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the `connect` function have the signature `Component => Component`. Functions whose output type is the same as its input type are really easy to compose together.

```
// Instead of doing this...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector)
```

```
)
const EnhancedComponent = enhance(WrappedComponent)
```

(This same property also allows `connect` and other enhancer-style HOCs to be used as decorators, an experimental JavaScript proposal.)

The `compose` utility function is provided by many third-party libraries including `lodash` (as `lodash.flowRight`), `Redux`, and `Ramda`.

Convention: Wrap the Display Name for Easy Debugging

The container components created by HOCs show up in the React Developer Tools like any other component. To ease debugging, choose a display name that communicates that it's the result of a HOC.

The most common technique is to wrap the display name of the wrapped component. So if your higher-order component is named `withSubscription`, and the wrapped component's display name is `CommentList`, use the display name `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

Caveats

Higher-order components come with a few caveats that aren't immediately obvious if you're new to React.

Don't Use HOCs Inside the render Method

`React`'s diffing algorithm (called Reconciliation) uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. If the component returned from `render` is identical (`===`) to the component from the previous render, React recursively updates the subtree by diffing it with the new one. If they're not equal, the previous subtree is unmounted completely.

Normally, you shouldn't need to think about this. But it matters for HOCs because it means you can't apply a HOC to a component within the render method of a component:

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

The problem here isn't just about performance — remounting a component causes the state of that component and all of its children to be lost.

Instead, apply HOCs outside the component definition so that the resulting component is created only once. Then, its identity will be consistent across renders. This is usually what you want, anyway.

In those rare cases where you need to apply a HOC dynamically, you can also do it inside a component's lifecycle methods or its constructor.

Static Methods Must Be Copied Over

Sometimes it's useful to define a static method on a React component. For example, Relay containers expose a static method `getFragment` to facilitate the composition of `GraphQL` fragments.

When you apply a HOC to a component, though, the original component is wrapped with a container component. That means the new component does not have any of the static methods of the original component.

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

To solve this, you could copy the methods onto the container before returning it:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

However, this requires you to know exactly which methods need to be copied. You can use hoist-non-react-statics to automatically copy all non-React static methods:

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

Another possible solution is to export the static method separately from the component itself.

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs Aren't Passed Through

While the convention for higher-order components is to pass through all props to the wrapped component, this does not work for refs. That's because `ref` is not really a prop — like `key`, it's handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

The solution for this problem is to use the `React.forwardRef` API (introduced with React 16.3). Learn more about it in the forwarding refs section.

Practices

- **State and Lifecycle 🧬 - Clock 🕑**

  Try to remake Clock with seeing Tips and focus to make styled one

  - Learn `webpack` work ;
  - How CRA (create-react-app) do their job ?
  - Fixing `webpack server` new version problem ;

- **Handling Events 🎄 - Switch 💊**

  On and Off option to change background-color

  - Listener ;
  - Bind, Apply, Call in Function Prototype ;
  - `this` LOCAL and `this` GLOBAL ;
  - Feel more free to do JS ;

- **Lists and Keys 📃 - Table 📚**

  Simple table lists all the name inside an object.

  - `Object.keys()` and it ' values

- Forms 📰 - **Form & Login 🔑**

  Try with small step with form

  All the form components too.

▼ **Lifting State Up 🗽 - Temperature Calculator 🌡️**

Done with basic step

▼ Composition vs Inheritance 🔃 - Composition one 📲

▼ Accessibility - Todo list

Try 2 times to finish

Convention: Wrap the Display Name for Easy Debugging