Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. static getDerivedStateFromProps()
3. render()
4. componentDidMount()

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The reason getDerivedStateFromProps is static is **to discourage any side-effects during the render phase**. For example, updating or using props on the instance. This isn't safe anymore with the upcoming async rendering. It is called when a component is created and each time it recieves a new props

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

* componentWillUnmount()

## Introducing Error Boundaries

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror) or [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch). Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown. 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;  }  } |

|  |
| --- |
| <ErrorBoundary>  <MyWidget />  </ErrorBoundary> |

**Note**

Error boundaries do **not** catch errors for:

* Event handlers ([learn more](https://reactjs.org/docs/error-boundaries.html#how-about-event-handlers))
* Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
* Server side rendering
* Errors thrown in the error boundary itself (rather than its children)

## How About Event Handlers? For error

|  |
| --- |
| 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 <div onClick={this.handleClick}>Click Me</div>  }  } |

# Code-Splitting

<https://reactjs.org/docs/code-splitting.html>

<https://create-react-app.dev/docs/code-splitting/>

## React.lazy

**Note:**

React.lazy and Suspense are not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we recommend [Loadable Components](https://github.com/gregberge/loadable-components). It has a nice [guide for bundle splitting with server-side rendering](https://loadable-components.com/docs/server-side-rendering/).

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.

|  |
| --- |
| **Before:**  import OtherComponent from './OtherComponent';  **After:**  const OtherComponent = React.lazy(() => import('./OtherComponent')); |

|  |
| --- |
| 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

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

# Context

<https://reactjs.org/docs/context.html>

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

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

|  |
| --- |
| // 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(props) {  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} />;  }  } |

### React.createContext

const MyContext = React.createContext(defaultValue);

### Context.Provider

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

### Class.contextType

It is used to subscribed to context into a class

|  |
| --- |
| 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; |

### 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 |

### Consuming Multiple Contexts

To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

|  |
| --- |
| // Theme context, default to light theme  const ThemeContext = React.createContext('light');  // Signed-in user context  const UserContext = React.createContext({  name: 'Guest',  });  class App extends React.Component {  render() {  const {signedInUser, theme} = this.props;  // App component that provides initial context values  return (  <ThemeContext.Provider value={theme}>  <UserContext.Provider value={signedInUser}>  <Layout />  </UserContext.Provider>  </ThemeContext.Provider>  );  }  }  function Layout() {  return (  <div>  <Sidebar />  <Content />  </div>  );  }  // A component may consume multiple contexts  function Content() {  return (  <ThemeContext.Consumer>  {theme => (  <UserContext.Consumer>  {user => (  <ProfilePage user={user} theme={theme} />  )}  </UserContext.Consumer>  )}  </ThemeContext.Consumer>  );  } |

Update Context API  
<https://medium.com/nerd-for-tech/using-context-api-in-react-with-functional-components-dbc653c7d485>

# Portals

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

Normally, when you return an element from a component’s render method, it’s mounted into the DOM as a child of the nearest parent node:

However, sometimes it’s useful to insert a child into a different location in the DOM:

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

return ReactDOM.createPortal(

this.props.children,

domNode

);

}

# Profiler API

The Profiler measures how often a React application renders and what the “cost” of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from [optimizations such as memoization](https://reactjs.org/docs/hooks-faq.html#how-to-memoize-calculations).

|  |
| --- |
| render(  <App>  <Profiler id="Navigation" onRender={callback}>  <Navigation {...props} />  </Profiler>  <Profiler id="Main" onRender={callback}>  <Main {...props} />  </Profiler>  </App>  ); |

# Render Props

The term [“render prop”](https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce) refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

<DataProvider render={data => (

<h1>Hello {data.target}</h1>

)}/>

# Static Type Checking

Static type checkers like [Flow](https://flow.org/) and [TypeScript](https://www.typescriptlang.org/) identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we recommend using Flow or TypeScript instead of PropTypes for larger code bases.

## TypeScript

[TypeScript](https://www.typescriptlang.org/) is a programming language developed by Microsoft. It is a typed superset of JavaScript, and includes its own compiler. Being a typed language, TypeScript can catch errors and bugs at build time, long before your app goes live. You can learn more about using TypeScript with React [here](https://github.com/Microsoft/TypeScript-React-Starter#typescript-react-starter).

# Strict Mode

StrictMode is a tool for highlighting potential problems in an application. Like Fragment, StrictMode does not render any visible UI. It activates additional checks and warnings for its descendants.

**Note:**

Strict mode checks are run in development mode only; they do not impact the production build.

|  |
| --- |
| import React from 'react';  function ExampleApplication() {  return (  <div>  <Header />  <React.StrictMode>  <div>  <ComponentOne />  <ComponentTwo />  </div>  </React.StrictMode>  <Footer />  </div>  );  } |

# Introducing Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

|  |
| --- |
| 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>  );  } |

|  |
| --- |
| function FriendStatusWithCounter(props) {  const [count, setCount] = useState(0);  useEffect(() => {  document.title = `You clicked ${count} times`;  });  const [isOnline, setIsOnline] = useState(null);  useEffect(() => {  function handleStatusChange(status) {  setIsOnline(status.isOnline);  }  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);  return () => {  ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);  };  });  // ...  } |

## [What is React.memo()?](https://scotch.io/tutorials/react-166-reactmemo-for-functional-components-rendering-control#toc-what-is-react-memo-)

[React.memo()](https://reactjs.org/docs/react-api.html#reactmemo) is similar to [PureComponent](https://reactjs.org/docs/react-api.html#reactpurecomponent) in that it will **help us control when our components rerender**.

export function Movie() {

return (

<div>

hello

</div>

);

}

export const MemoizedMovie = React.memo(Movie);

# React Fiber

React Fiber is an ongoing reimplementation of React's core algorithm. The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.