# React Error Boundaries

In the past, if we get any JavaScript errors inside components, it corrupts the React?s internal state and put React in a broken state on next renders. There are no ways to handle these errors in React components, nor it provides any methods to recover from them. But, **React 16** introduces a new concept to handle the errors by using the **error boundaries**. Now, if any JavaScript error found in a part of the UI, it does not break the whole app.

Error boundaries are React components which catch JavaScript errors anywhere in our app, log those errors, and display a fallback UI. It does not break the whole app component tree and only renders the fallback UI whenever an error occurred in a component. Error boundaries catch errors during rendering in component lifecycle methods, and constructors of the whole tree below them.

Note:

Sometimes, it is not possible to catch Error boundaries in React application. These are:

- Event handlers
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server-side rendering
- Errors are thrown in the error boundary itself rather than its children.

For simple React app, we can declare an error boundary once and can use it for the whole application. For a complex application which have multiple components, we can declare multiple error boundaries to recover each part of the entire application.

We can also report the error to an error monitoring service like **Rollbar**. This monitoring service provides the ability to track how many users are affected by errors, find causes of them, and improve the user experience.

## Error boundary in class

A class component can becomes an error boundary if it defines a new lifecycle methods either static getDerivedStateFromError() or componentDidCatch(error, info). We can use static getDerivedStateFromError() to render a fallback UI when an error has been thrown, and can use componentDidCatch() to log error information.

An error boundary can?t catch the error within itself. If the error boundary fails to render the error message, the error will go to the closest error boundary above it. It is similar to the catch {} block in JavaScript.

## How to implement error boundaries

**Step-1** Create a class which extends React component and passes the props inside it.

**Step-2** Now, add componentDidCatch() method which allows you to catch error in the components below them in the tree.

**Step-3** Next add render() method, which is responsible for how the component should be rendered. For example, it will display the error message like "Something is wrong."

### Example

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    // It will update the state so the next render shows the fallback UI.
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    // It will catch error in any component below. We can also log the error to an error reporting service.
    logErrorToMyService(error, info);
  }
  render() {
    if (this.state.hasError) {
        return (
        <div>Something is wrong.</div>;
    );
    }
    return this.props.children;
  }
}
```

**Step-4** Now, we can use it as a regular component. Add the new component in HTML, which you want to include in the error boundary. In this example, we are adding an error boundary around a MyWidgetCounter component.

```
<ErrorBoundary>
      <MyWidgetCounter/>
</ErrorBoundary>
```

## Where to Place Error Boundaries

An error boundary entirely depends on you. You can use error boundaries on the top-level of the app components or wrap it on the individual components to protect them from breaking the other parts of the app.

Let us see an example.

```
import React from 'react';
import './App.css'

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: false, errorInfo: null };
  }

  componentDidCatch(error, errorInfo) {
    // Catch errors in any components below and re-render with error message
    this.setState({
      error: error,
      errorInfo: errorInfo
    })
  }

  render() {
    if (this.state.errorInfo) {
      return (
        <div>
          <h2>Something went wrong.</h2>
          <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
            <br />
            {this.state.errorInfo.componentStack}
          </details>
        </div>
      );
    }
    return this.props.children;
  }
}

class BuggyCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(({counter}) => ({
      counter: counter + 1
    }));
  }

  render() {
    if (this.state.counter === 3) {
      throw new Error('I crashed!');
    }
    return <h1 onClick={this.handleClick}>{this.state.counter}</h1>;
  }
}

function App() {
  return (
    <div>
      <p><b>Example of Error Boundaries</b></p>
      <hr />
      <ErrorBoundary>
        <p>These two counters are inside the same error boundary.</p>
          <BuggyCounter />
        <BuggyCounter />
      </ErrorBoundary>
      <hr />
      <p>These two counters are inside of their individual error boundary.</p>
        <ErrorBoundary><BuggyCounter /></ErrorBoundary>
        <ErrorBoundary><BuggyCounter /></ErrorBoundary>
    </div>
  );
}
export default App
```

In the above code snippet, when we click on the **numbers**, it increases the **counters**. The counter is programmed to **throw** an error when it reaches **3**. It simulates a JavaScript error in a component. Here, we used an error boundary in **two ways**, which are given below.

**First**, these two counters are inside the same error boundary. If anyone crashes, the error boundary will replace both of them.

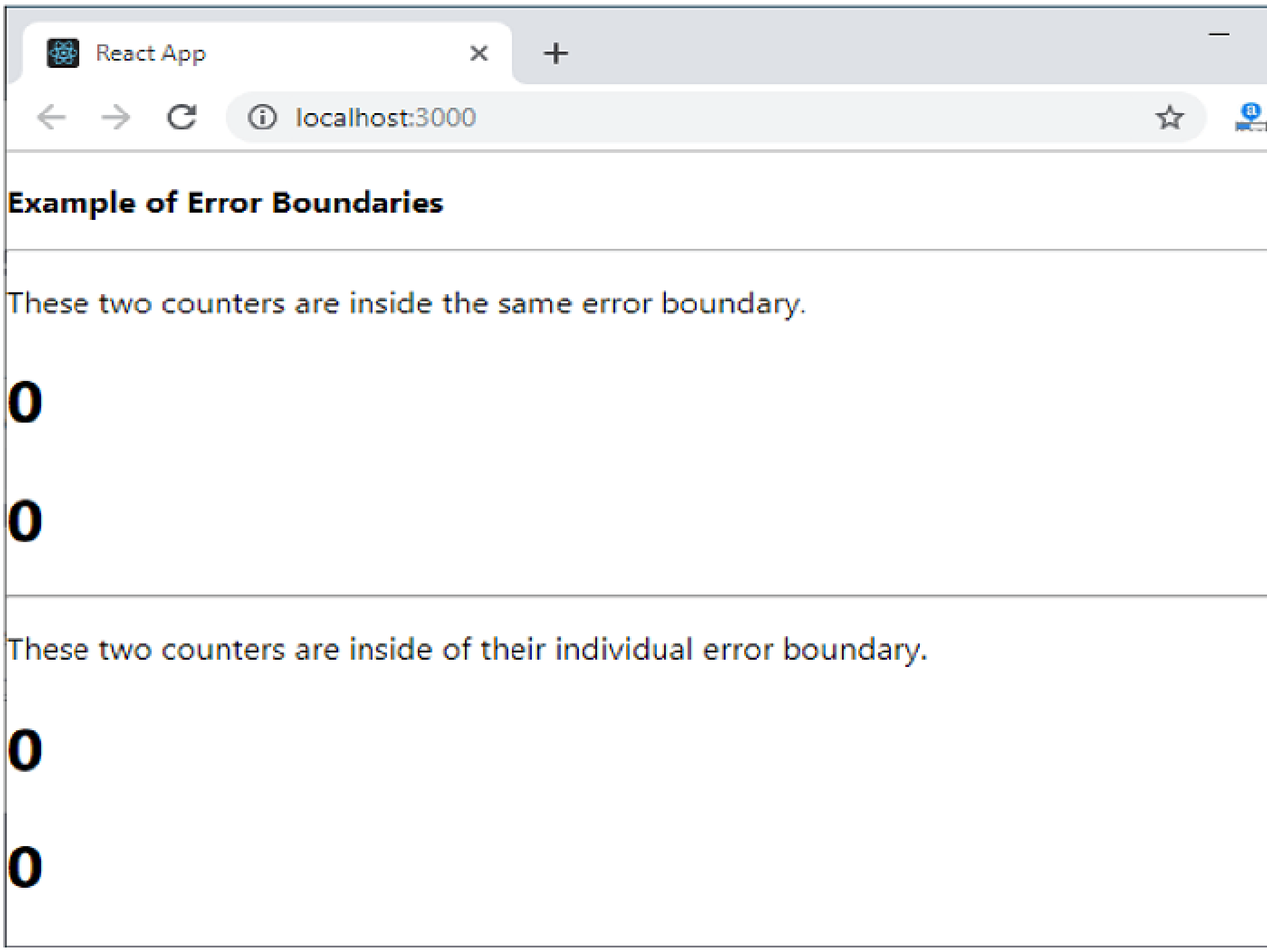
```
<ErrorBoundary>
        <BuggyCounter />
        <BuggyCounter />
</ErrorBoundary>
```

**Second**, these two counters are inside of their individual error boundary. So if anyone crashes, the other is not affected.

```
<ErrorBoundary><BuggyCounter /></ErrorBoundary>
<ErrorBoundary><BuggyCounter /></ErrorBoundary>
```

**Output:**

When we execute the above code, we will get the following output.



**Example of Error Boundaries**

These two counters are inside the same error boundary.

# 0

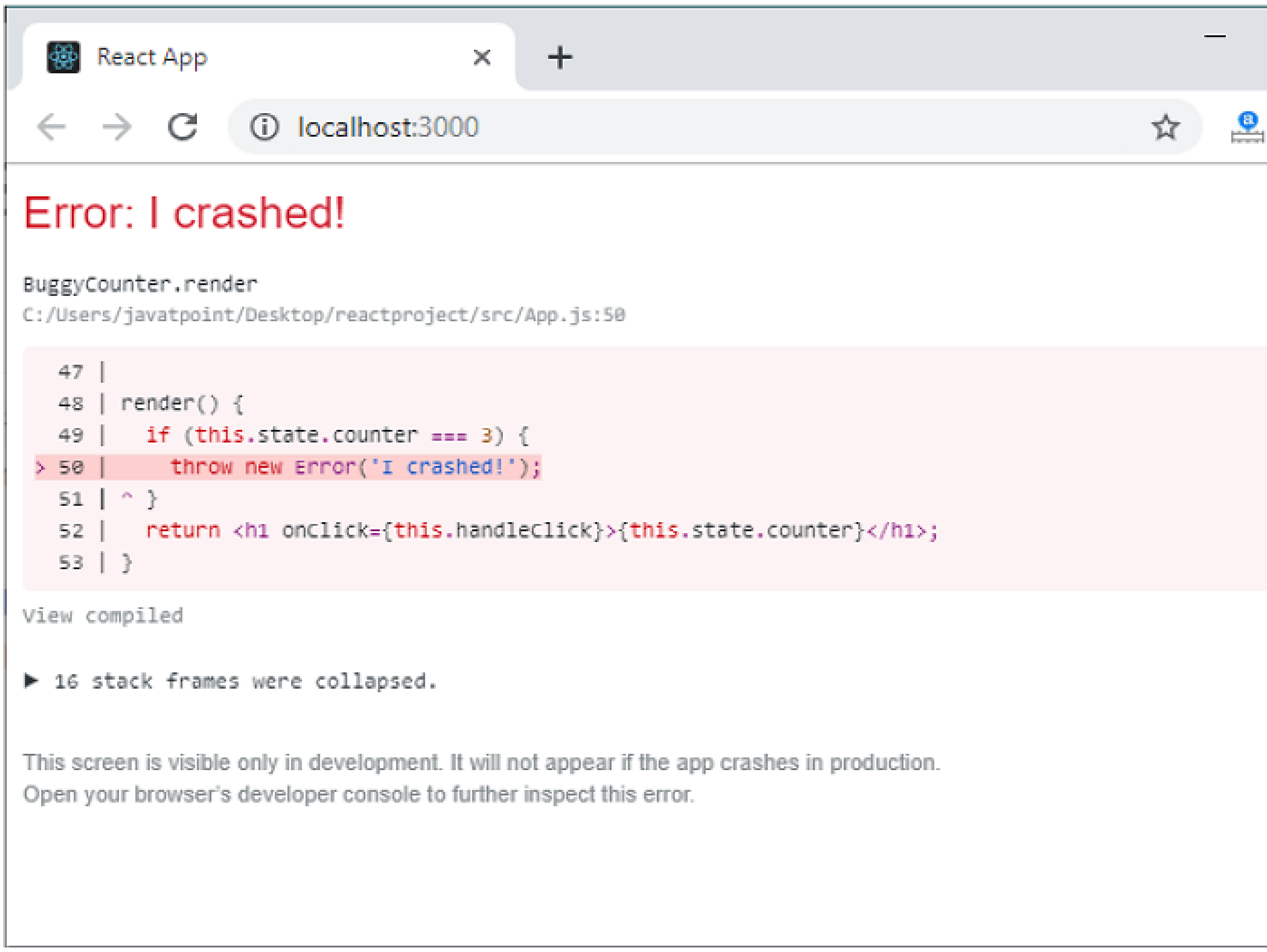
# 0

These two counters are inside of their individual error boundary.

# 0

# 0

When the counter has reached at 3, it gives the following output.



Error: I crashed!

```
BuggyCounter.render
C:/Users/javatpoint/Desktop/reactproject/src/App.js:50

  47 |
  48 | render() {
  49 |   if (this.state.counter === 3) {
> 50 |     throw new Error('I crashed!');
  51 | ^ }
  52 |   return <h1 onClick={this.handleClick}>{this.state.counter}</h1>;
  53 | }
View compiled

▶ 16 stack frames were collapsed.
```

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.

## New Behavior for Uncaught error

It is an important implication related to error boundaries. If the error does not catch by any error boundary, it will result in **unmounting** of the whole React application.

## Error Boundary in Event Handler

Error boundaries do not allow catching errors inside event handlers. React does not need any error boundary to recover from errors in the event handler. If there is a need to catch errors in the event handler, you can use JavaScript **try-catch** statement.

In the below example, you can see how an event handler will handle the errors.

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

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

  render() {
    if (this.state.error) {
      return
          <h2>It caught an error.</h2>
    }
    return <div onClick={this.handleClick}>Click Me</div>
  }
}
```