

React Crash Course

January 2019

Library

Libraries can be defined as **publicly available** and **easily reusable fragments of code** that we may want to use in our projects.

React

JavaScript **view library** for building user interfaces.

React is declarative

It means that using React, we **describe effects** we want to achieve,
not how to achieve them.

React is component based

Every element in React's world is a **component**. Components are **independent**, **reusable** pieces, which connected together build a fully-blown user interface.

Separation of components

It's a good practice to keep **separation of components**. We build components as **self-contained elements** and design them taking into consideration **rules** given below.

- composability and flexibility
 - reusability
 - maintainability
- ability to be run and tested independently

React components

There exists two component types in React - **function-based components** and **class-based components**.

Functional component

The simplest React component - just a function. They may accept **props** as a parameter and return **JSX element**.

```
1 // just a function!
2 function Welcome(props) {
3   return <h1>Hello, {props.name}</h1>
4 }
5
6 // usage
7 <Welcome name='Janusz' />
```


Props

Props is an **object** which contains all the inputs passed to the component from the parent component. **Props** stands for **properties**. Component cannot modify its own props.

```
1  function Welcome(props) {  
2    return <h1>Hello, you are {props.age >= 18 ? 'adult' : 'underage'}</h1>  
3  }  
4  
5  <Welcome age='18' />
```

Components rendering components

Components can be rendered inside other components.

```
1  function Welcome(props) {  
2    return <h1>Hello, {props.name}</h1>  
3  }  
4  
5  function App(props) {  
6    return <Welcome name='Janusz' />  
7  }  
8  
9  <App />
```

JSX

JavaScript XML - syntax used inside return statement. It's a syntax which **looks like HTML** and allows to create **own tags (=== components)**.

```
1  function Welcome(props) {  
2    return <h1>Hello, {props.name}</h1>  
3  }
```

JSX assignments

JSX element can be assigned to any JavaScript variable. It's treated as another data type.

```
1  function Welcome(props) {  
2    const element = <h1>Hello, {props.name}</h1>  
3    return element  
4  }
```

NOTE: JSX type is not available in native JavaScript. More on that later.

JSX - differences from HTML

- We can use HTML attributes, exactly like in pure HTML, but some of them are different - e.g. `className` instead of `class`.
- Attributes and methods are camel case - `onclick` will become `onClick`.

JS inside JSX

It's possible to include JavaScript **expressions** (remember: `if` statement is not an expression :)) inside JSX using curly brackets.

```
1 function Welcome(props) {  
2   return <h1>Hello, you are {props.age >= 18 ? 'adult' : 'underage'}</h1>  
3 }
```

Children property

children property is a special one - it contains another component passed inside the JSX tag.

```
1  function Welcome(props) {  
2    return <div>  
3      <div>Hello, {props.name}</div>  
4      {props.children}  
5    </div>  
6  }  
7  
8  <Welcome name='Janusz'>  
9    <div>Nice to see you!</div>  
10 </Welcome>
```

PropTypes

What **types** of attributes should be passed to the component?

Sometimes it's difficult to find out and can raise some issues.

Fortunately, we can use a mechanism called **PropTypes**. It allows to define types of properties which should be passed to a component.

These types are then check during the **application runtime**.

PropTypes

In order to use **PropTypes** mechanism, we need to register a dependency of **prop-types package** created by Facebook in our project by executing:

```
yarn add prop-types
```

From the project's root directory.

PropTypes

```
1  import PropTypes from "prop-types"
2
3  class Welcome extends React.Component {
4    render() {
5      return <div>
6        <h1>Hello, {this.props.name}</h1>
7        {this.props.children}
8      <div>
9    }
10 }
11
12 Welcome.propTypes = {
13   name: PropTypes.string.isRequired,
14   children: PropTypes.element
15 }
```

Class based components

Class based components extend `React.Component` or `React.PureComponent` classes. They have to implement `render()` function which returns JSX element. **Only one JSX element** can be returned from render function.

```
1  class Welcome extends React.Component {  
2    render() {  
3      return <h1>Hello, {this.props.name}</h1>  
4    }  
5  }
```

Class based components

Class based components can have other, different than render, methods defined within their body (between curly braces).

```
1  class Welcome extends React.Component {
2    constructor(props) {
3      super(props)
4      this.handleClick = this.handleClick.bind(this)
5    }
6
7    handleClick() {
8      console.log('Click happened')
9    }
10
11   render() {
12     return <button onClick={this.handleClick}>Click Me</button>
13   }
14 }
```

State

Each **class component** (and only class component) can contain its own **state** that may affect the rendered output.

Component's state is maintained by React itself and can be changed using **setState()** method only.

Modifying the state

```
1  class Navbar extends React.Component {
2    constructor(props) {
3      super(props)
4      this.state = { clicks: 0 }
5      this.handleClick = this.handleClick.bind(this)
6    }
7
8    handleClick() {
9      const currentClicksCount = this.state.clicks
10     this.setState({ clicks: currentClicksCount + 1 })
11   }
12
13   render() {
14     return (
15       <div>
16         <button onClick={this.handleClick}>Click Me</button>
17         <p>Button was clicked {this.state.clicks} times!</p>
18       </div>
19     )
20   }
21 }
```

State

`setState()` method is **asynchronous** which means that we cannot rely on `this.state` immediately after calling `setState()`.

```
1      class StateSampe extends React.Component {  
2...    // some code there...  
9  
10     doSomethingWithTheState() {  
11       this.setState({ name: "Bernard" })  
12       console.warn(this.state.name) // prints name before "Bernard"  
13     }  
14  
15...   // some code there  
16     }
```

State

`setState()` method cannot be called inside render method. In such cases we'll end up in “infinite render loop”.

Lifting state up

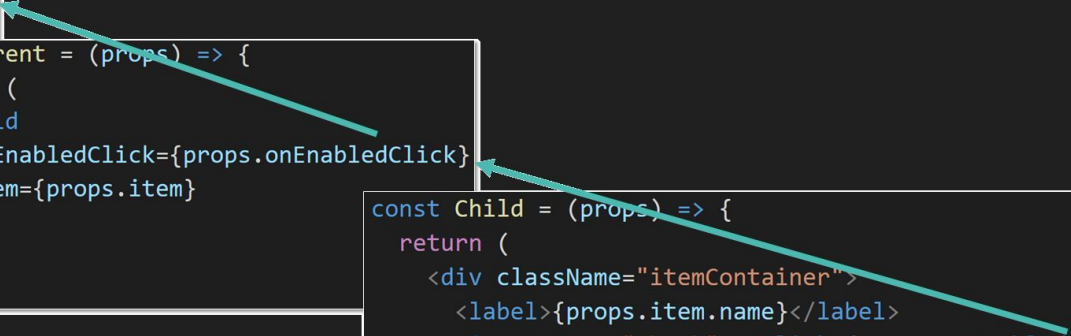
Sometimes the components very low down the hierarchy tree can affect the “global” state defined in the higher level of components tree. In such cases, we need to **lift the state up** through all levels of components tree.

Lifting state up

```
const Grandparent = React.createClass({
  // ...
  render: () => {
    return (
      <Parent
        onEnabledClick={this.handleItemClick}
        item={this.currentItem}
      />
    )
  },
})
```

```
const Parent = (props) => {
  return (
    <Child
      onEnabledClick={props.onEnabledClick}
      item={props.item}
    />
  )
}
```

```
const Child = (props) => {
  return (
    <div className="itemContainer">
      <label>{props.item.name}</label>
      <input type="checkbox" onClick={props.onEnabledClick} />
    </div>
  )
}
```



When React components are rerendered?

By default, React components are rerendered always whenever either **props** or **state** change.

Error handling

Whenever a **runtime errors** occur in any of our components, the **whole application will crash** by default.

We can, however, prevent the whole application from crashing by implementing specific methods (applicable only for **class components**) and render **fallback UI** instead.

Error handling

```
1  class Parent extends React.Component {  
2    constructor(props) {  
3      super(props)  
4    }  
5  
6    render() {  
7      return (  
8        <div className="wrapper">  
9          <Child />  
10         </div>  
11       )  
12     }  
13   }
```

```
1  function Child() {  
2    return <p>{person.getName()}</p>  
3  }
```

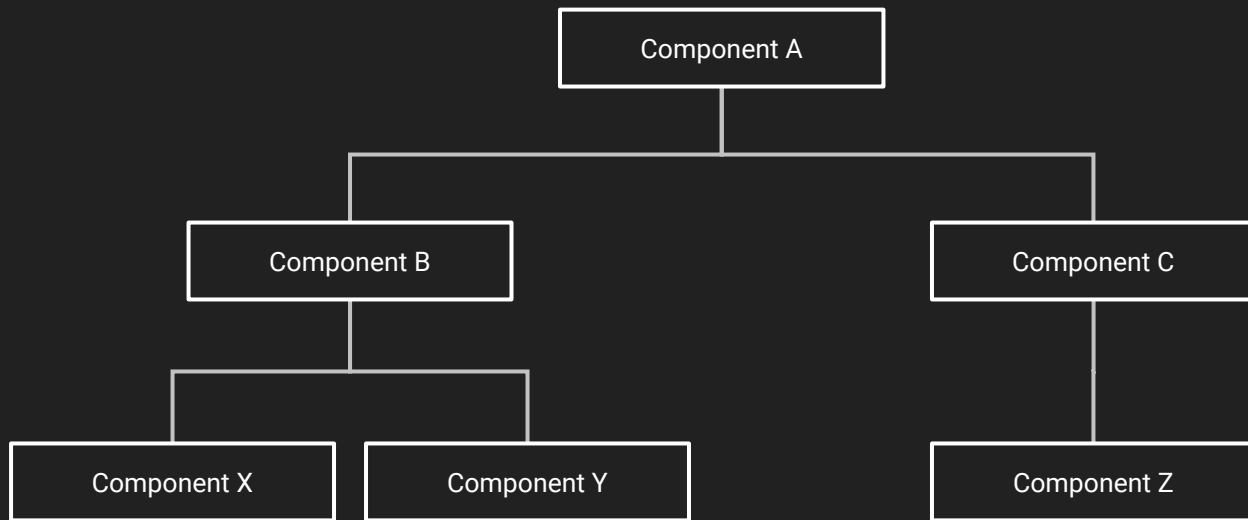
person in the above Child component code is not defined, so we're expecting an error.

The whole application will crash.

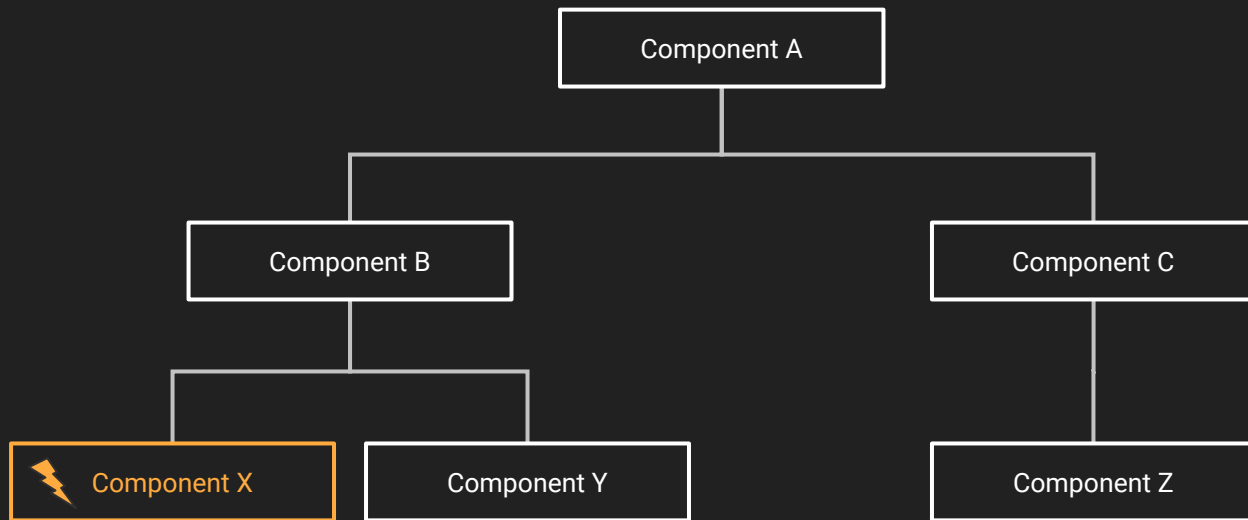
Error handling

```
1  class Parent extends React.Component {
2    constructor(props) {
3      super(props)
4      this.state = { hasErrors: false }
5    }
6
7    static getDerivedStateFromError() {
8      return { hasErrors: true }
9    }
10
11   render() {
12     return !this.state.hasErrors
13       ? (
14         <div className="wrapper">
15           <Child />
16         </div>
17       )
18       : <p>Error occurred</p>
19   }
```

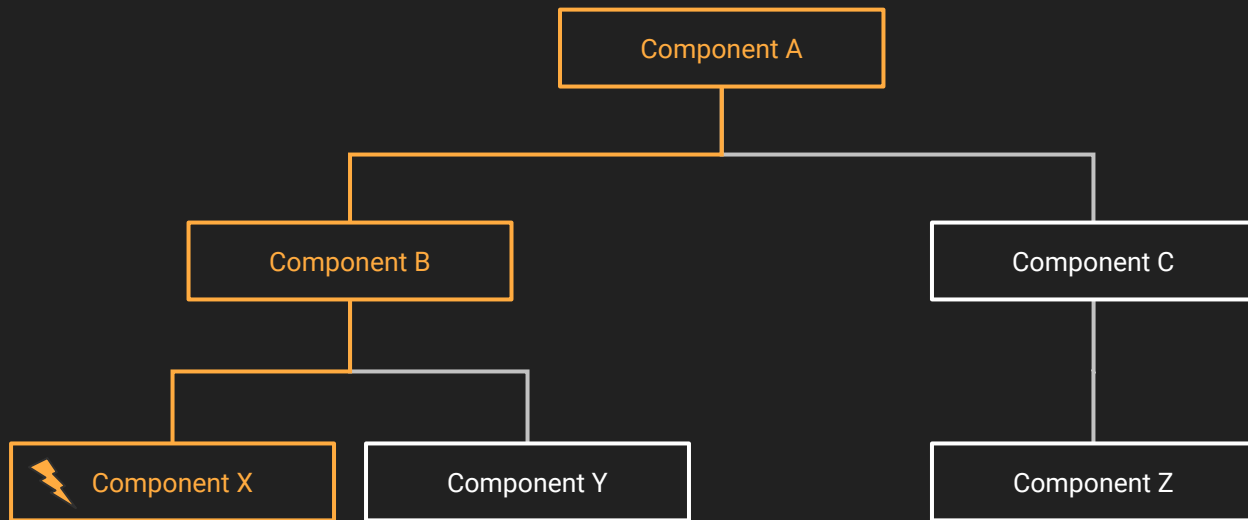
Error handling



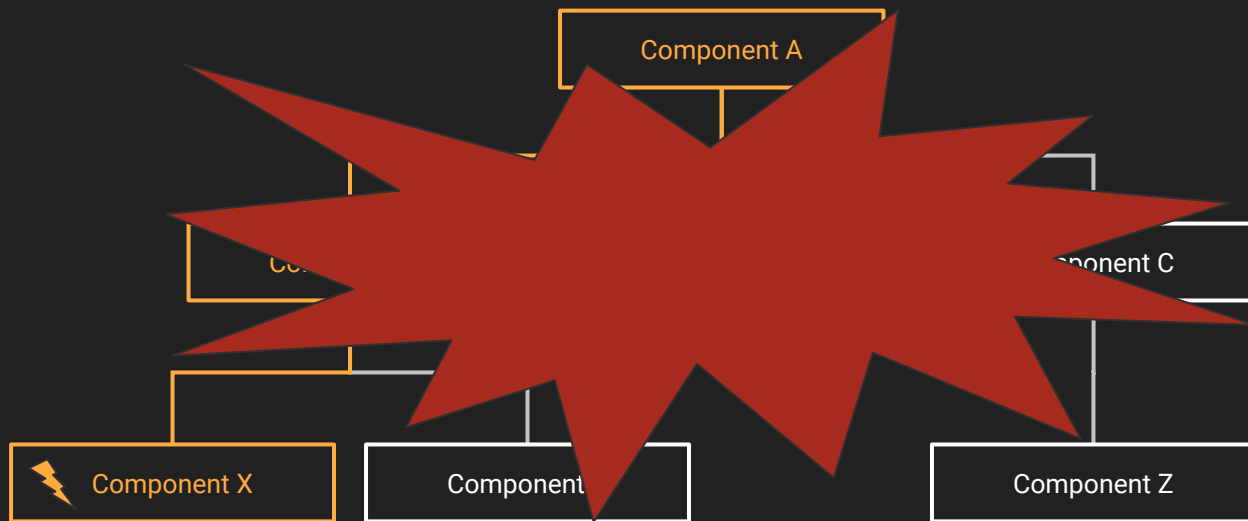
Error handling



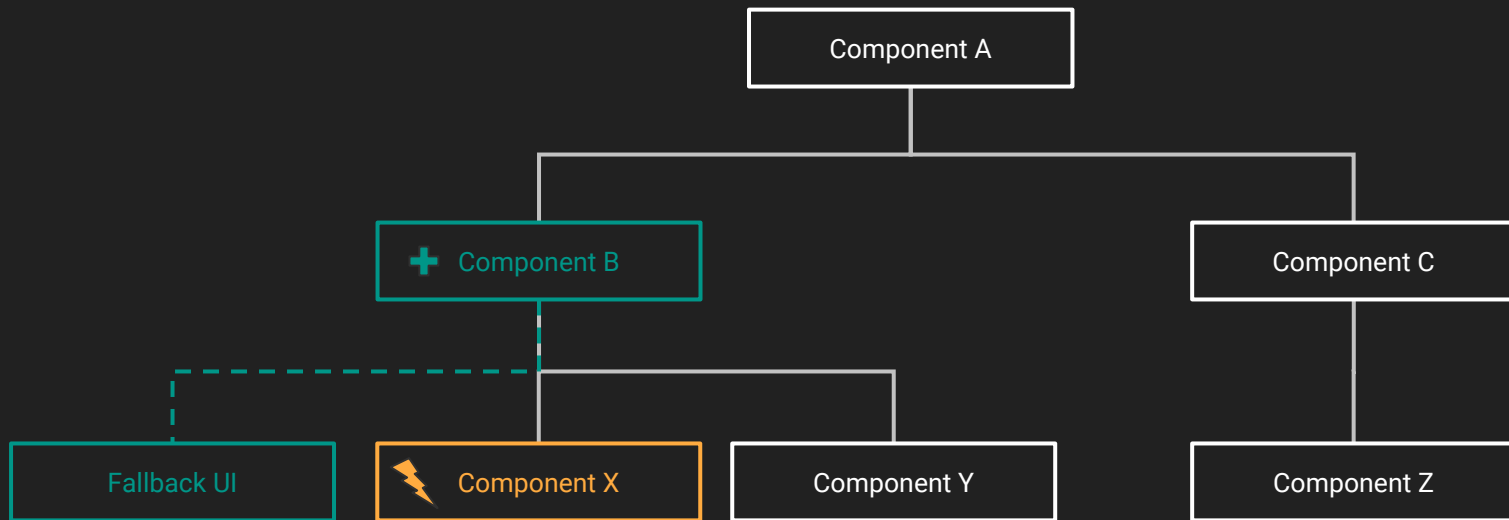
Error handling



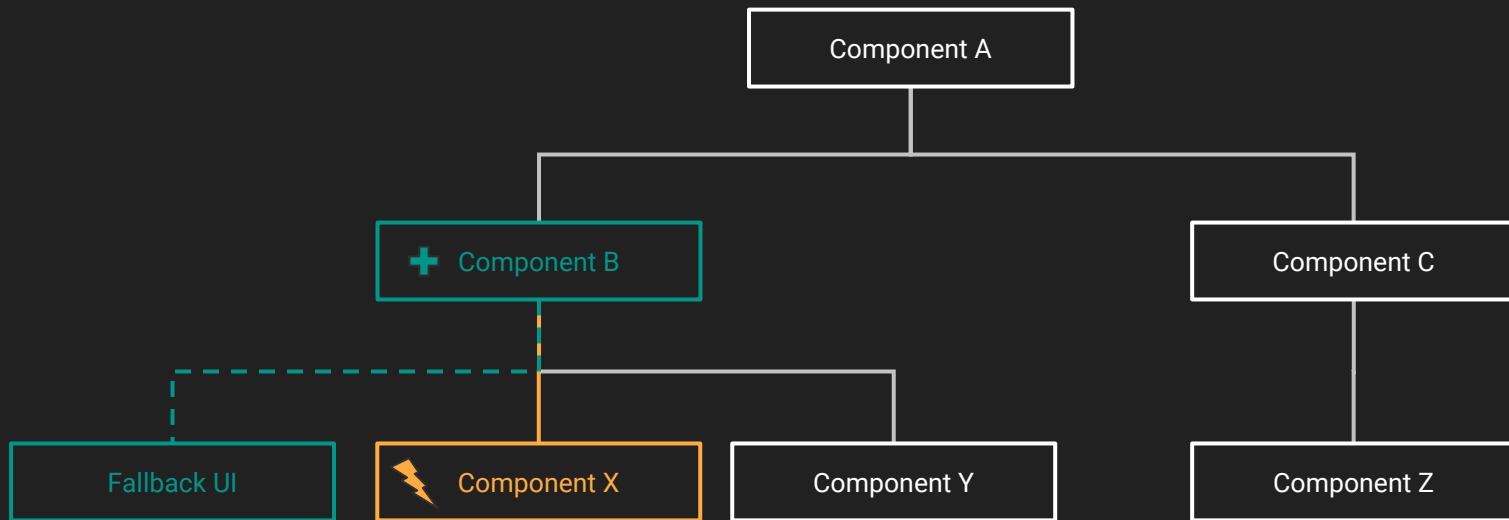
Error handling



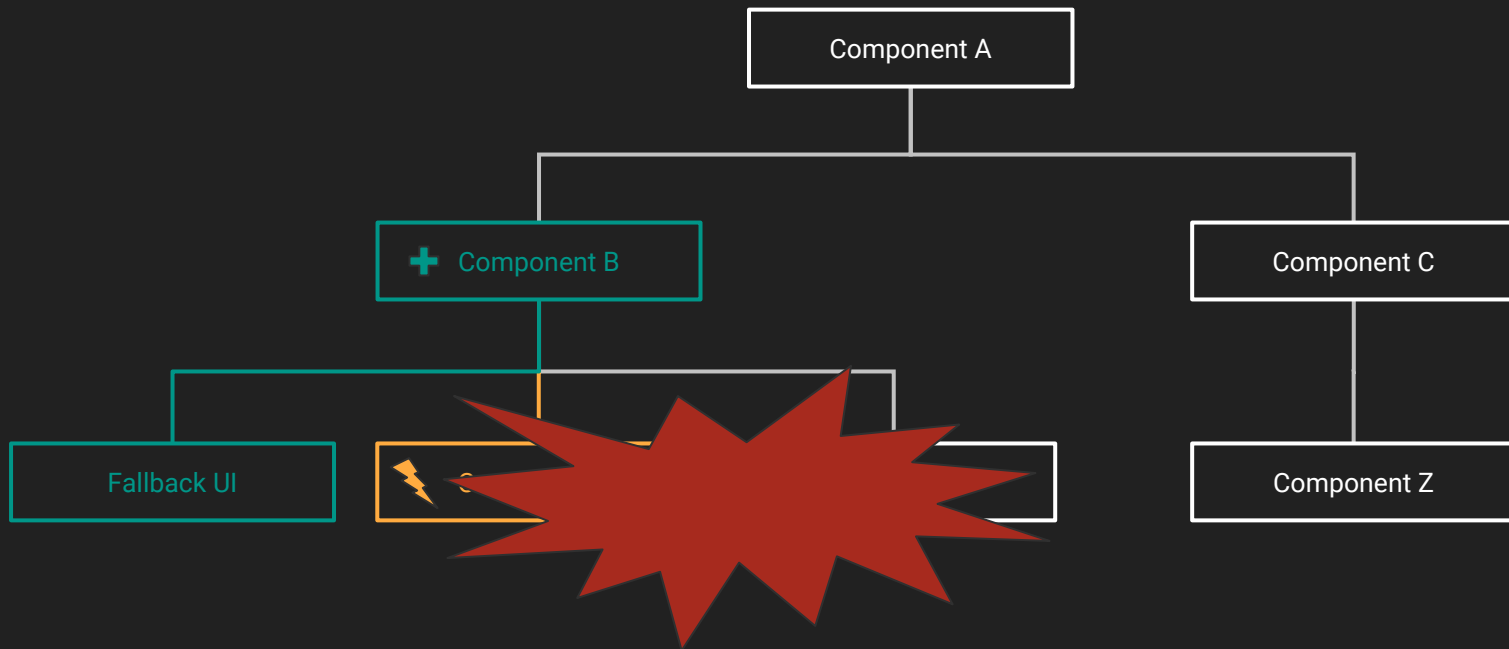
Error handling



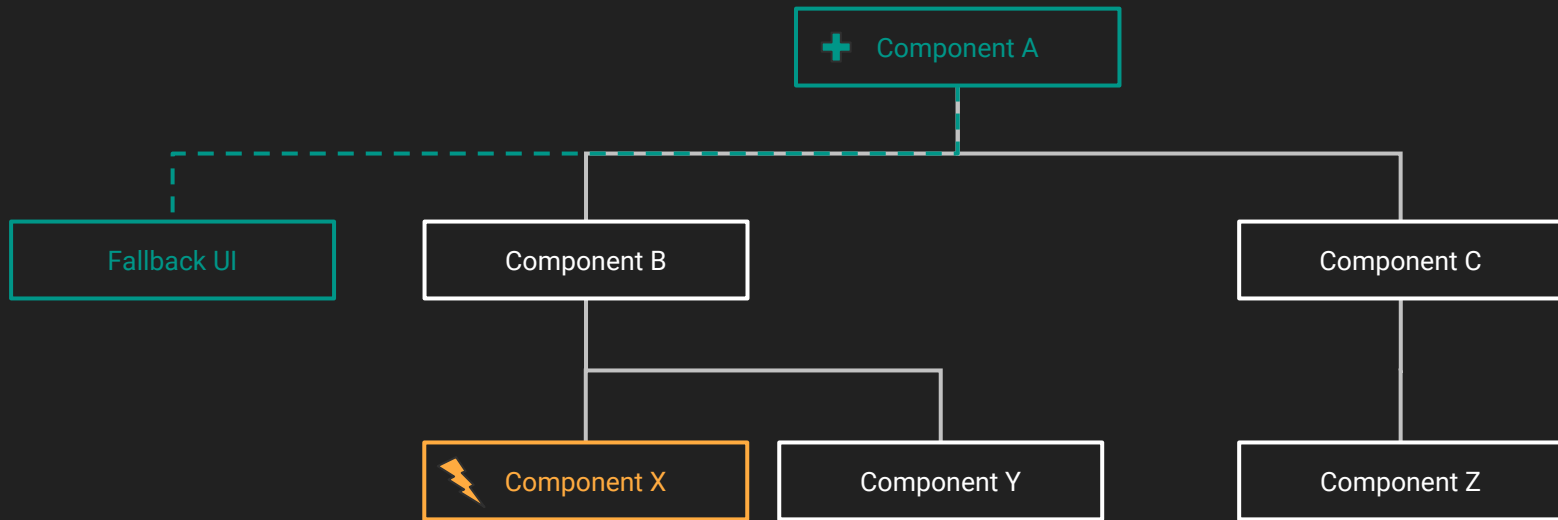
Error handling



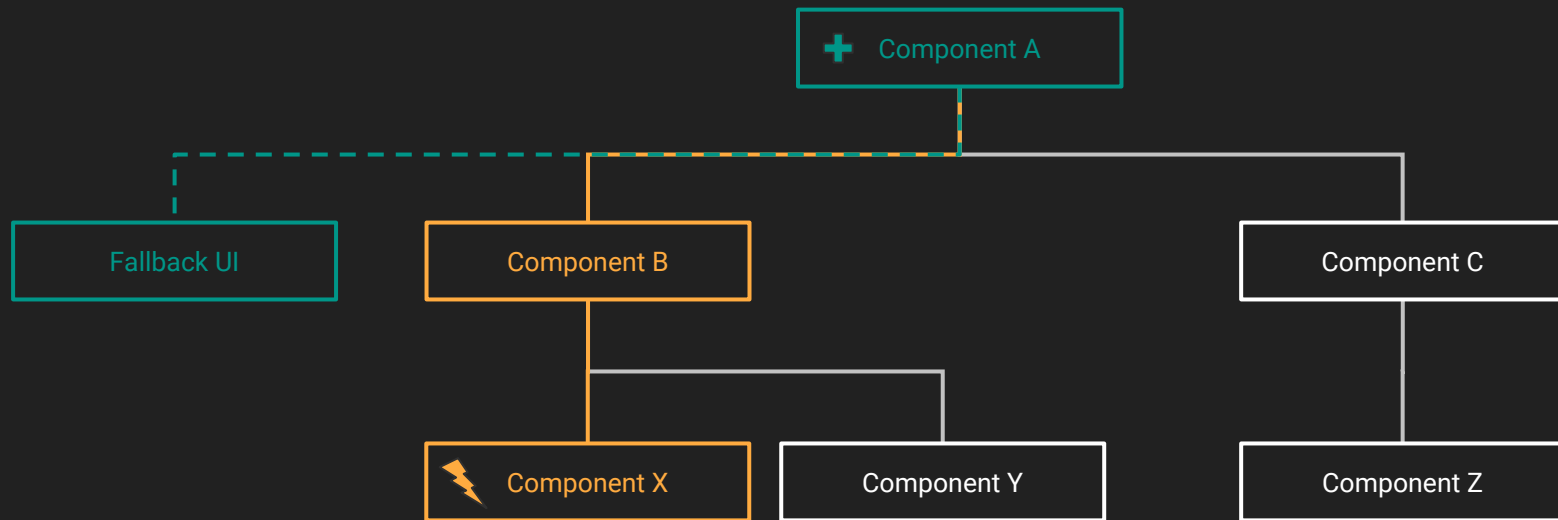
Error handling



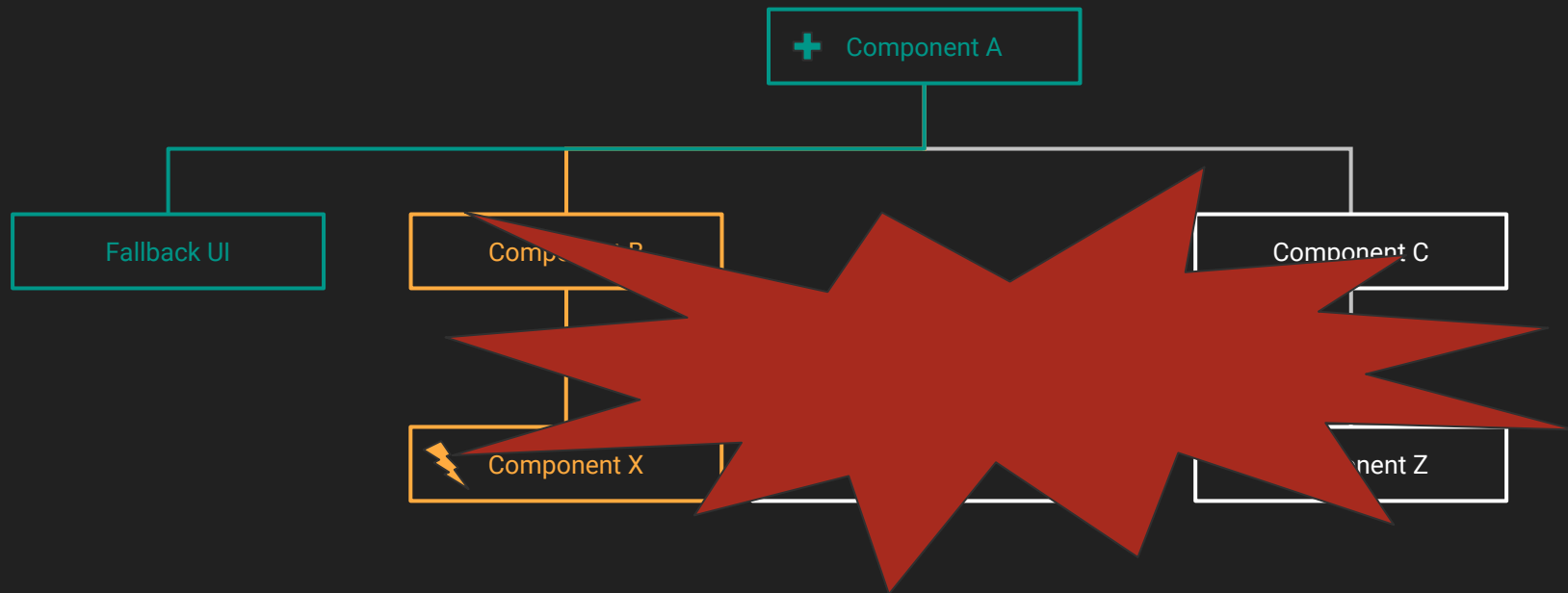
Error handling



Error handling



Error handling



Questions?



Under the hood

