Better React Components

What are we going to cover

Prettier

Splitting components

Container components with render props

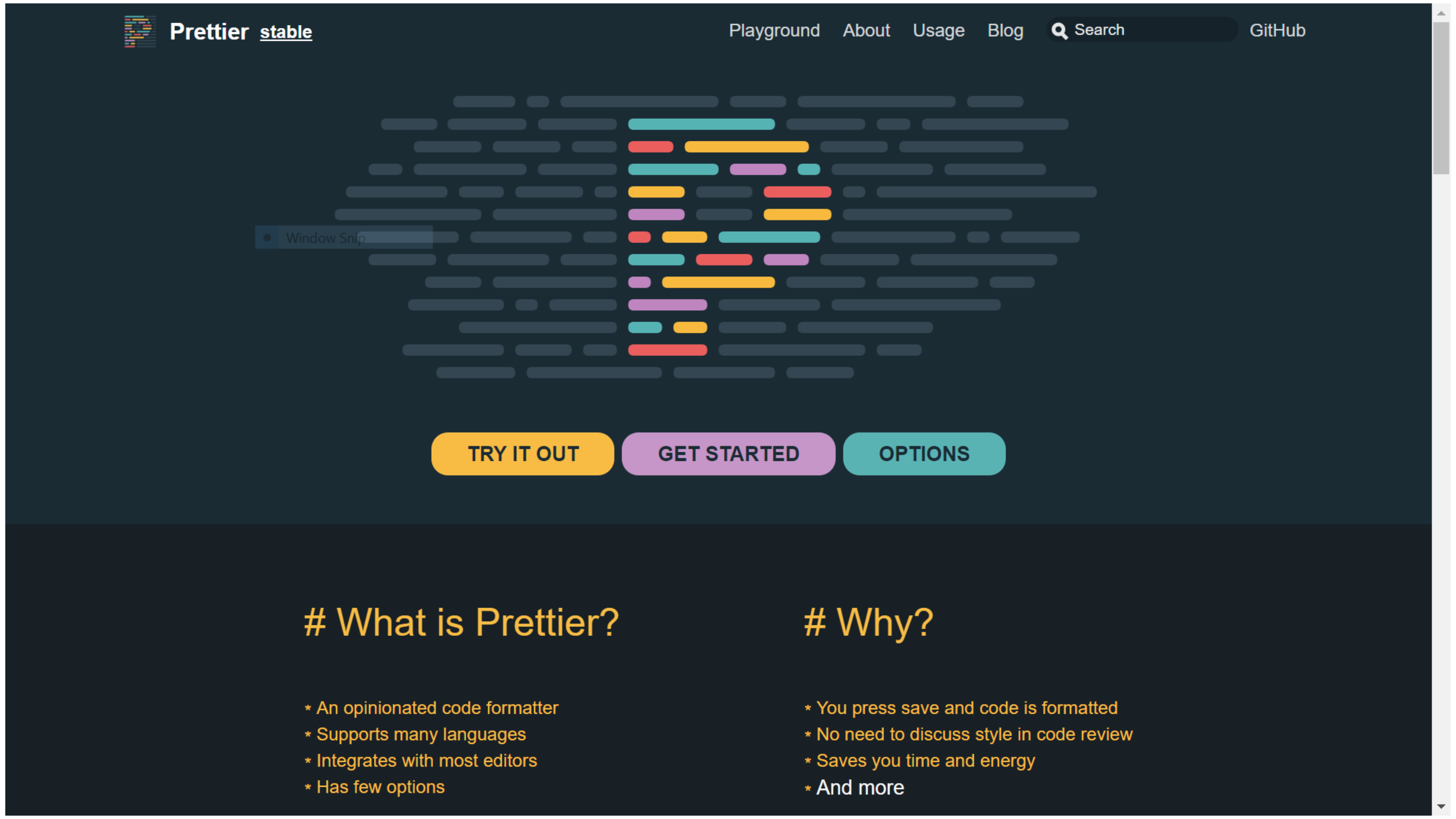
Error boundaries

Using **key** to reset state

StrictMode

Lazy and Suspense

Prettier



Prettier stable

What is Prettier?

* An opinionated code formatter
* Supports many languages
* Integrates with most editors
* Has few options

Why?

* You press save and code is formatted
* No need to discuss style in code review
* Saves you time and energy
* And more

Splitting components

Create **separate components** for each piece of functionality

- Use single responsibility principals

A lot of behavior is done by **wrapping components**

- Redux connect()
- React router withRouter()
- Error boundaries

Components with render props

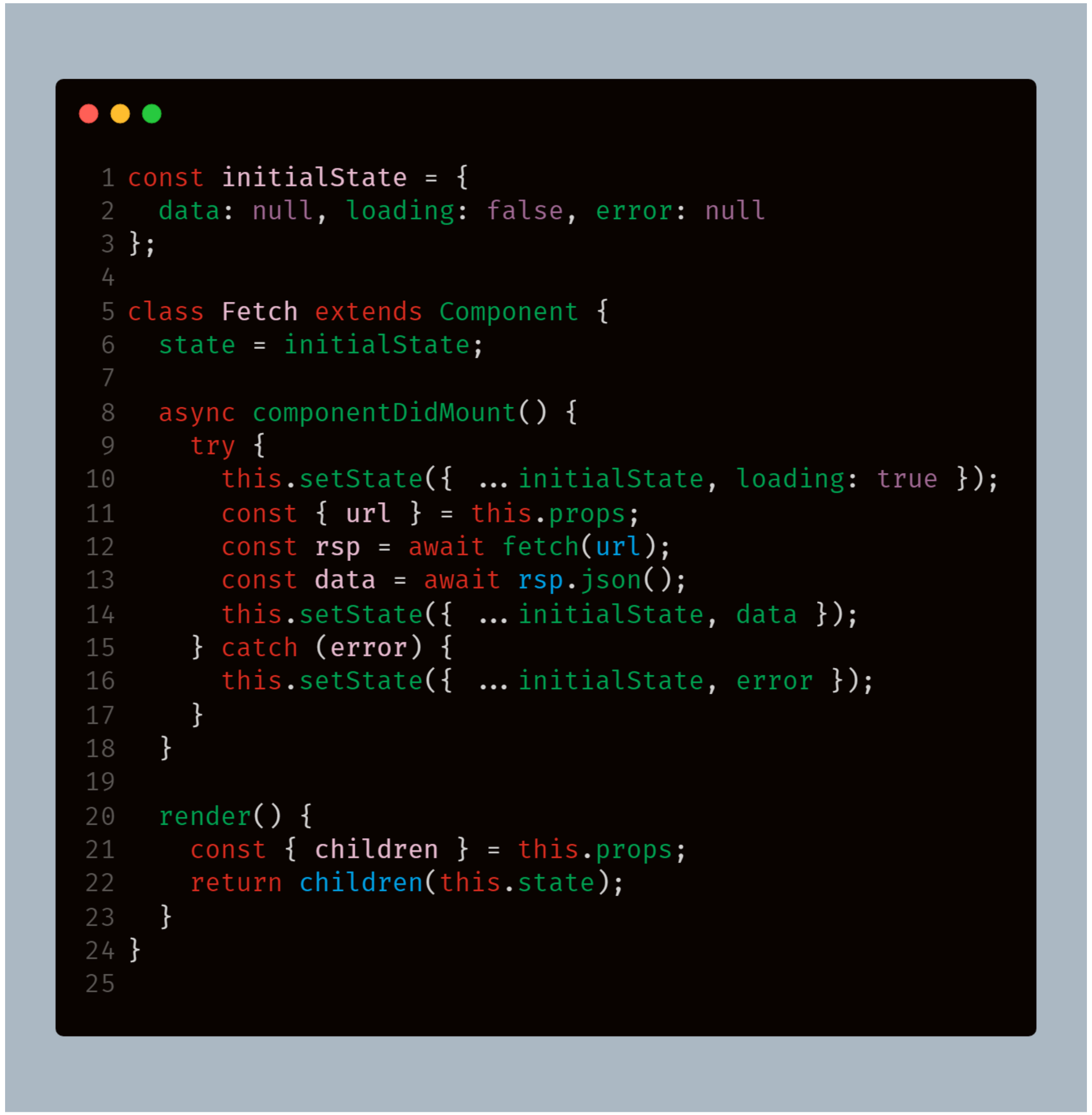
Render props is **more flexible** then regular higher order functions

- With an HOC the HOC developer has full control
- With render props the developer using it has full control

Downside of render props is the more complex syntax

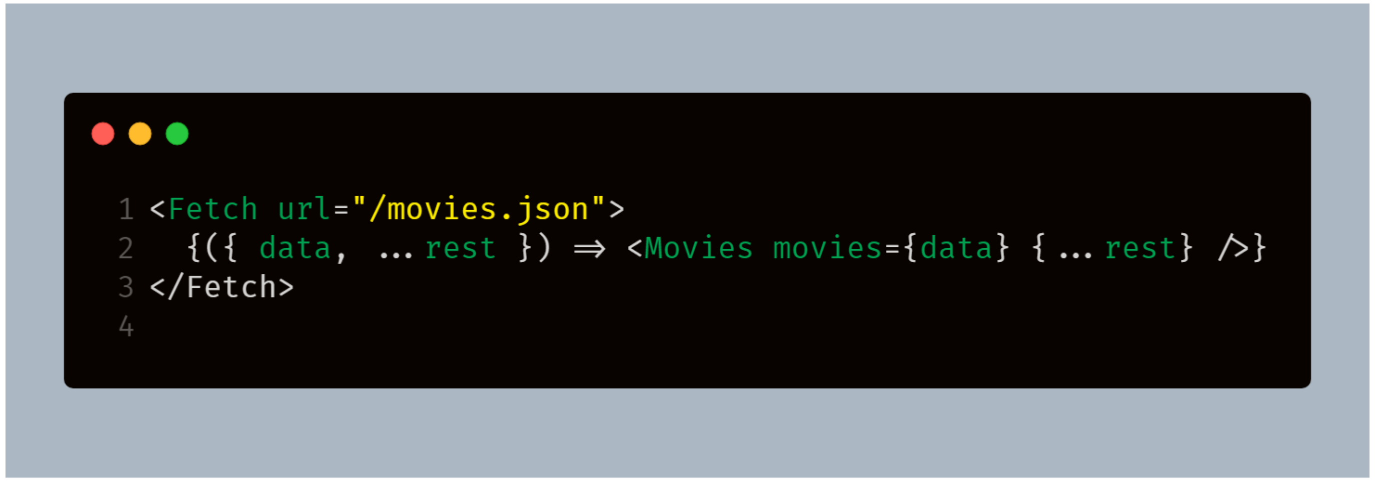
- Can be solved by calling into a presentational component to render the UI

A render prop component



```
const initialState = {
  data: null, loading: false, error: null
};

class Fetch extends Component {
  state = initialState;

  async componentDidMount() {
    try {
      this.setState({ ...initialState, loading: true });
      const { url } = this.props;
      const rsp = await fetch(url);
      const data = await rsp.json();
      this.setState({ ...initialState, data });
    } catch (error) {
      this.setState({ ...initialState, error });
    }
  }

  render() {
    const { children } = this.props;
    return children(this.state);
  }
}

```

Using the component



```
<Fetch url="/movies.json">
  {({ data, ...rest }) => <Movies movies={data} {...rest} />}
</Fetch>
```

Error boundaries

Errors in React lifecycle functions **destroy** the React component tree

- Use an error boundary to only destroy part of the tree

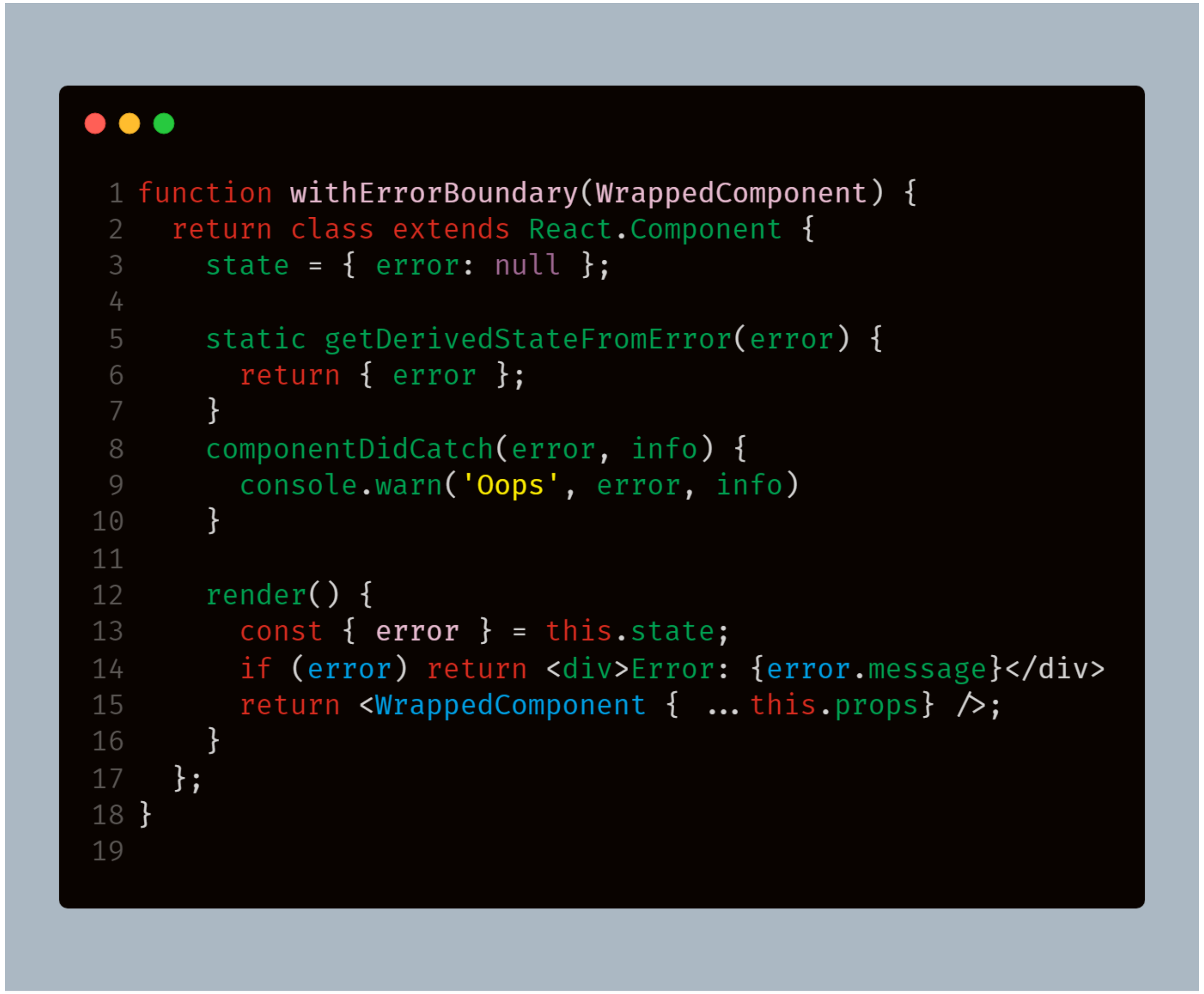
Use the **componentDidCatch()** to **log** the error

Use the static **getDerivedStateFromError()** to update the **state** based on the error

Errors in **event handlers** are **not caught** in error boundaries

- Use a global error handler

Error boundaries



```
function withErrorBoundary(WrappedComponent) {
  return class extends React.Component {
    state = { error: null };

    static getDerivedStateFromError(error) {
      return { error };
    }
    componentDidCatch(error, info) {
      console.warn('Oops', error, info)
    }

    render() {
      const { error } = this.state;
      if (error) return <div>Error: {error.message}</div>
      return <WrappedComponent { ...this.props} />;
    }
  };
}

```

Resetting state

A **key** prop can be used on any component

- Useful to **reset** a components internal **state**

When the value changes

- The **old** component will be **unmounted**
- A **new** component will be **mounted**

StrictMode

Detects problematic usage of API’s that will be deprecated because of possible side effects

- Unsafe lifecycle functions
- Unexpected side effects
- String ref usage
- findDOMNode usage
- legacy context

The StrictMode component can be used on a part of the component tree

- The goal is to make migration to async rendering with React 17 easier

Note: StrictMode does nothing in a **production** build

Lazy and Suspense

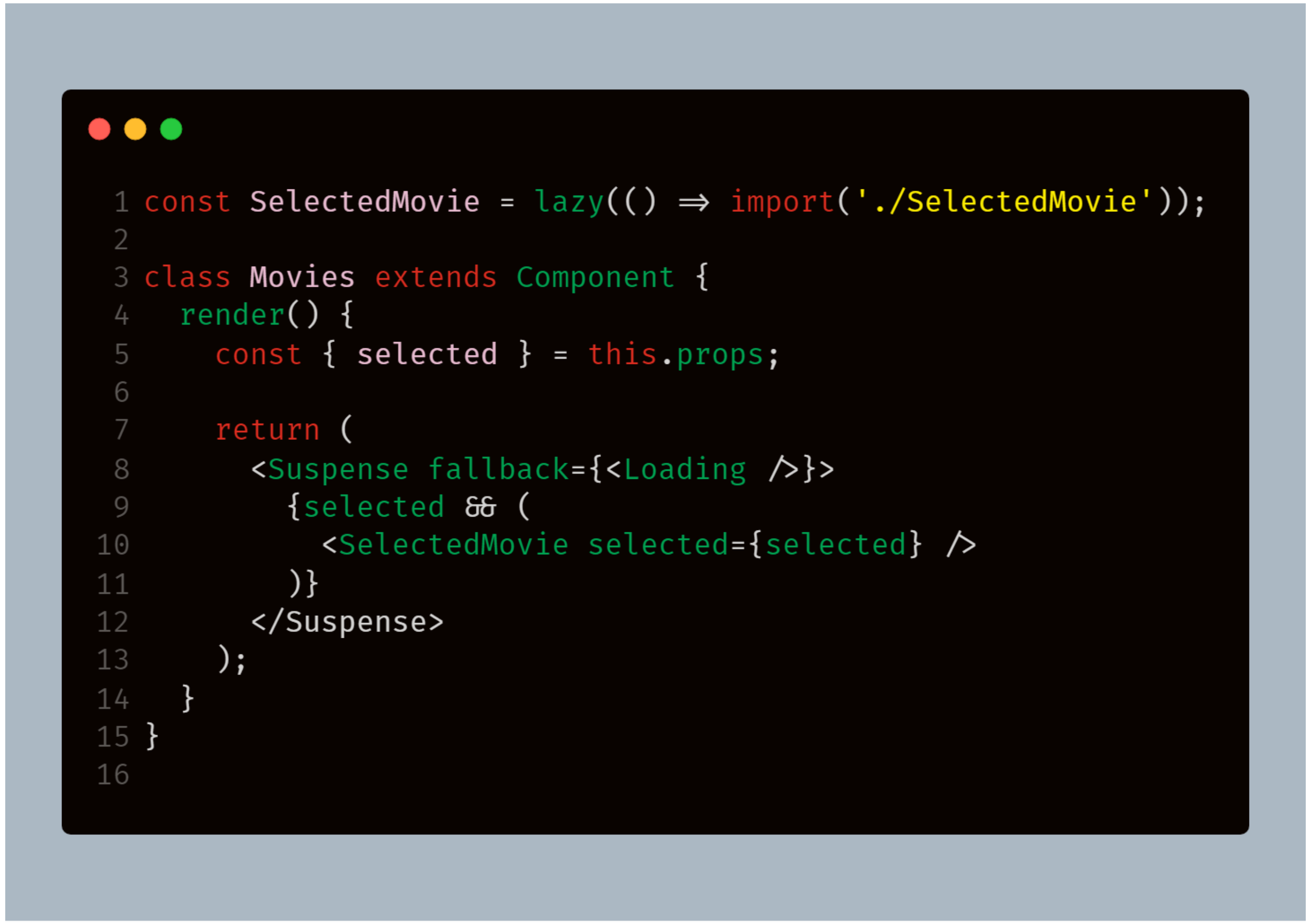
The **lazy()** function creates a component around a **dynamic import**

- The component chunk will not be loaded until the component is rendered

Must be inside of a **Suspense** component

- The fallback will be shown while the component chunk is loaded
- Doesn’t need to be the direct child

Using a lazy component



```
const SelectedMovie = lazy(() ⇒ import('./SelectedMovie'));

class Movies extends Component {
  render() {
    const { selected } = this.props;

    return (
      <Suspense fallback={<Loading />}>
        {selected && (
          <SelectedMovie selected={selected} />
        )}
      </Suspense>
    );
  }
}

```

Thank You!