React: Crash Course

1. Thinking in React
2. React
3. State Management
4. Next Steps
5. Try it Yourself

We are going to be moving quickly, but feel free to stop me at anytime with questions you may have.

Thinking in React

1. **Cohesion**: Increase cohesion, decrease coupling.
2. **Data Flow**: All data flows in one direction.

These are going to be the two themes that we come back to at every point in this presentation, and in developing.
^ React can mean the UI library, or the whole ecosystem of libraries that create a loose framework.

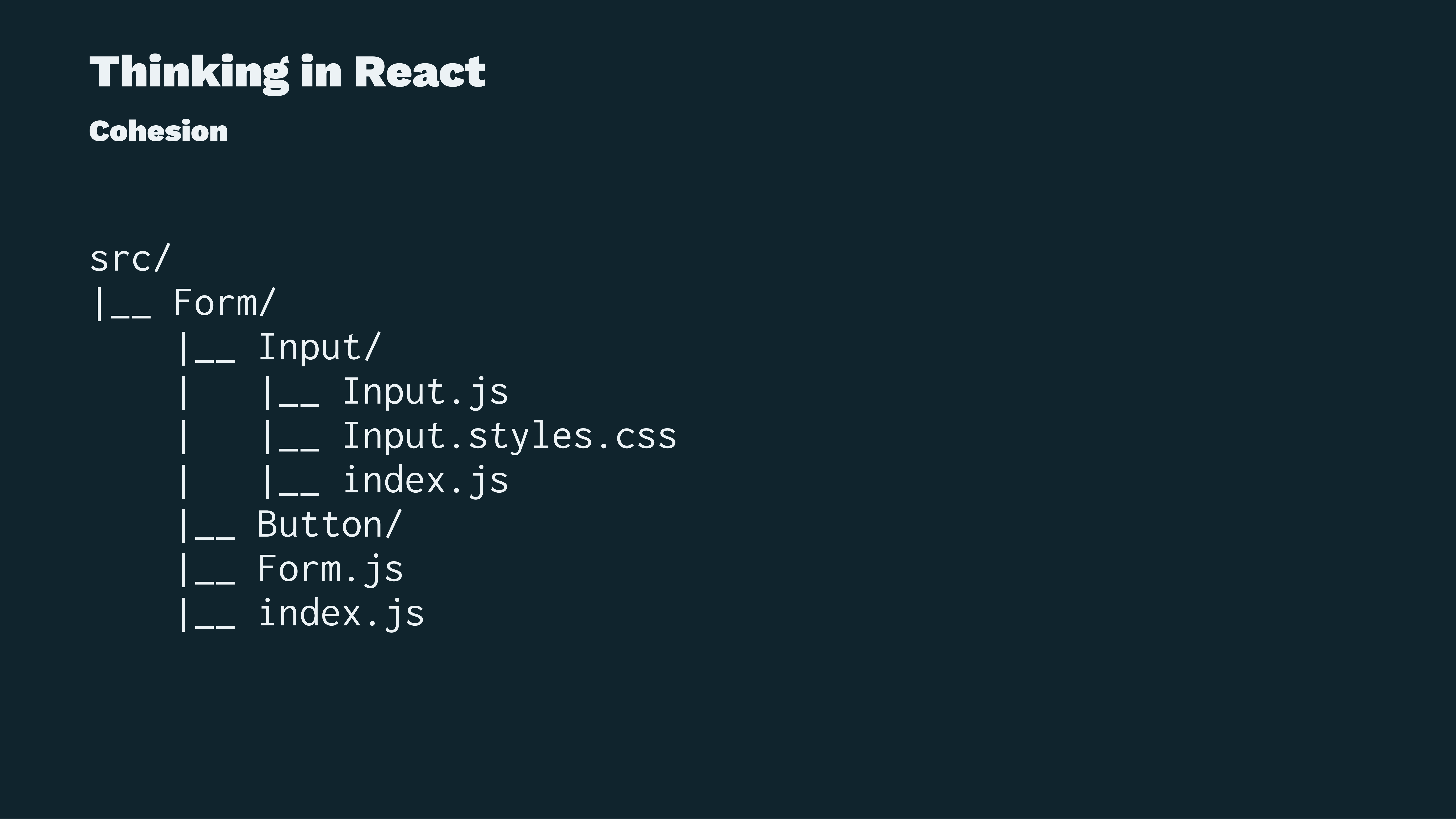
Thinking in React

Cohesion

— Features first, types second.

— Easy to package.

Two ways that cohesion can present itself is the above.



Thinking in React

Cohesion

```
src/
|__ Form/
    |__ Input/
    |   |__ Input.js
    |   |__ Input.styles.css
    |   |__ index.js
    |__ Button/
    |__ Form.js
    |__ index.js
```

This is organized by feature, everything that belongs to a feature is all in one place, the Form/ folder.
^ This is easy to package, you could just zip up the Form/ folder and take it with you to another repo.
^ Note the index.js file, we will come back to it later.

Thinking in React

Data Flow

— All data flows downwards.

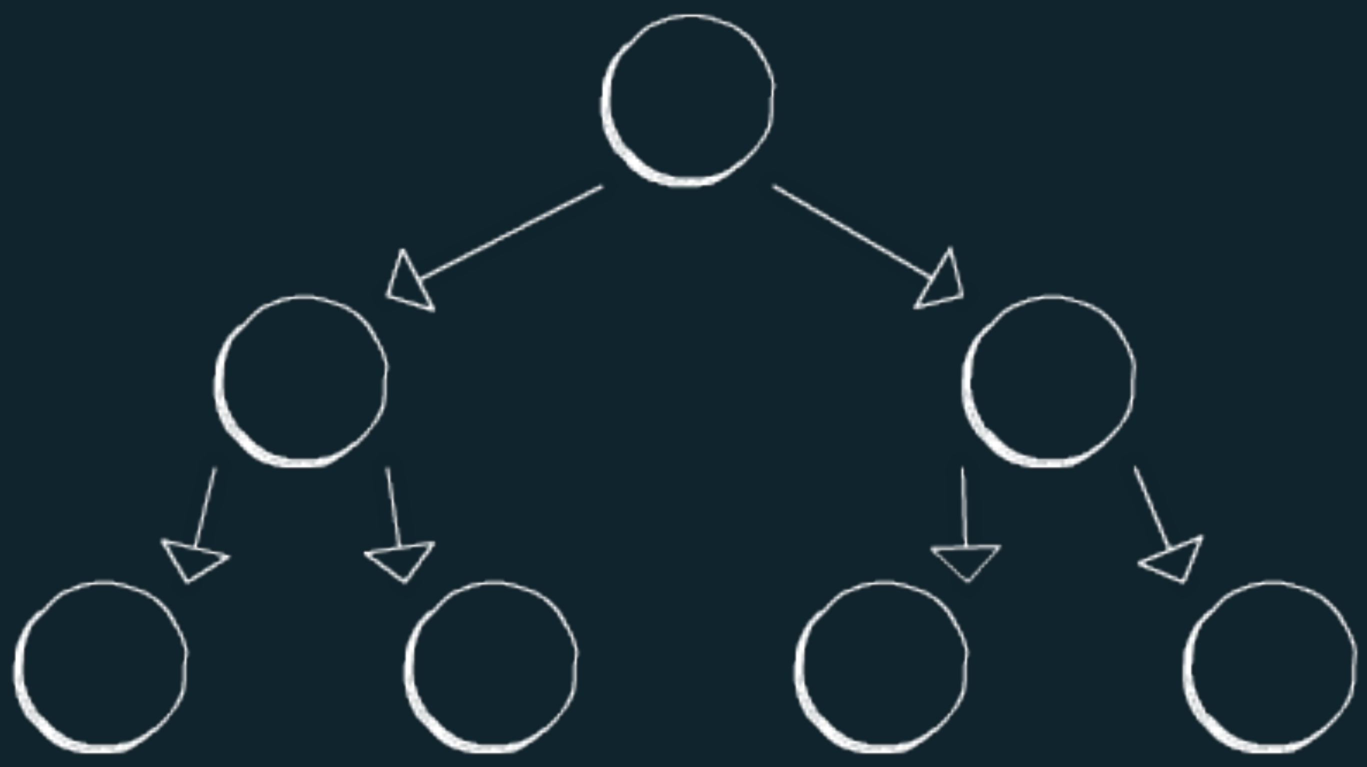
— There is no* way to pass information upwards.

There's no asterisk.

^ This is how we think of data flow in a React enviornment

Thinking in React

Data Flow

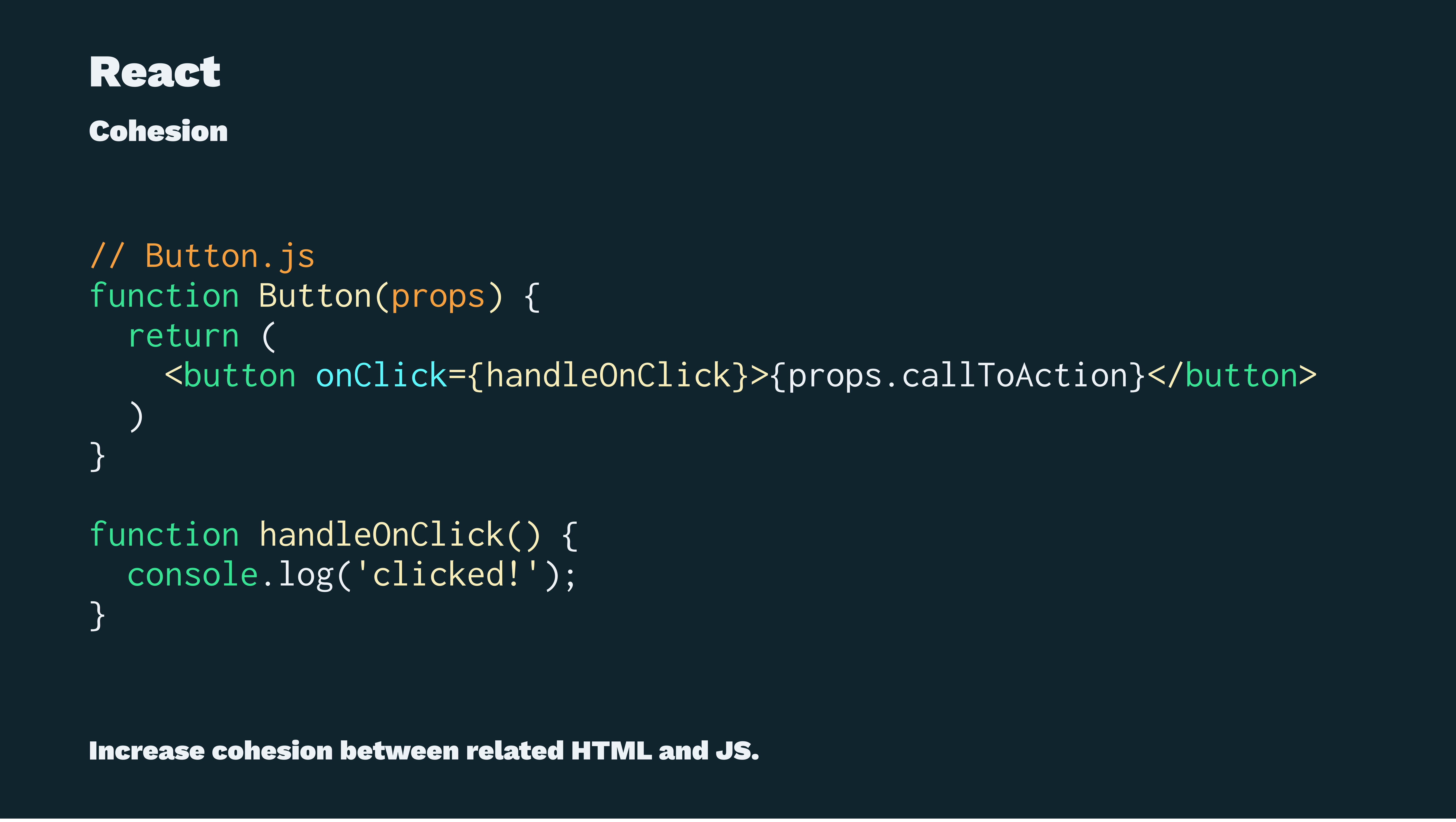


Component Tree

These are components, they only get data from the components above them. If you wanted to get new data into a component, you'd need to pass it in from above.
^ This might sound really slow, but it's actually incredibly fast because we only re-render what has updates.
^ Talk about how we don't mutate and performance

React

Now that we've talked about all the theory, let's see some examples.



React

Cohesion

```
// Button.js
function Button(props) {
  return (
    <button onClick={handleOnClick}>{props.callToAction}</button>
  )
}

function handleOnClick() {
  console.log('clicked!');
}
```

Increase cohesion between related HTML and JS.

This is the idea that lead to the creation of React. Putting the HTML and JS in different files is actually a false decoupling. These two things are actually deeply linked, so we want to increase cohesion.
^ JSX is how we get HTML in our JS

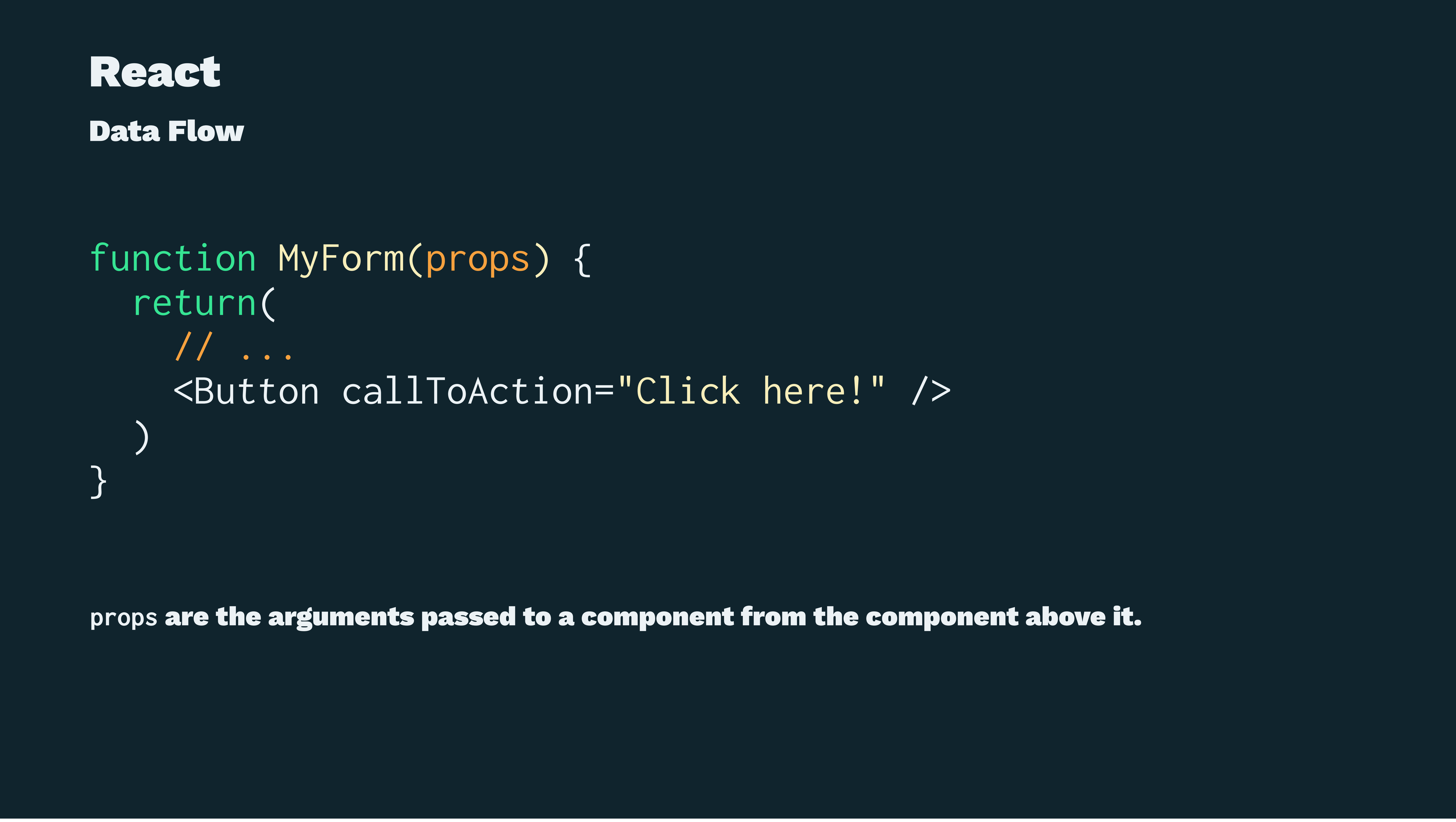
React

Cohesion

```
import btn from "./Button.styles";

function Button(props) {
  return (
    <button className={btn.primary}>{props.callToAction}</button>
  )
}
```

We can take it a step further and increase the cohesion between related CSS too.
^ CSS Modules help make sure that primary is unique no matter where you put this component.
^ POST CSS has a feature to automatically reset between components so that you never have to worry about inheritance.



React

Data Flow

```
function MyForm(props) {
  return(
    // ...
    <Button callToAction="Click here!" />
  )
}
```

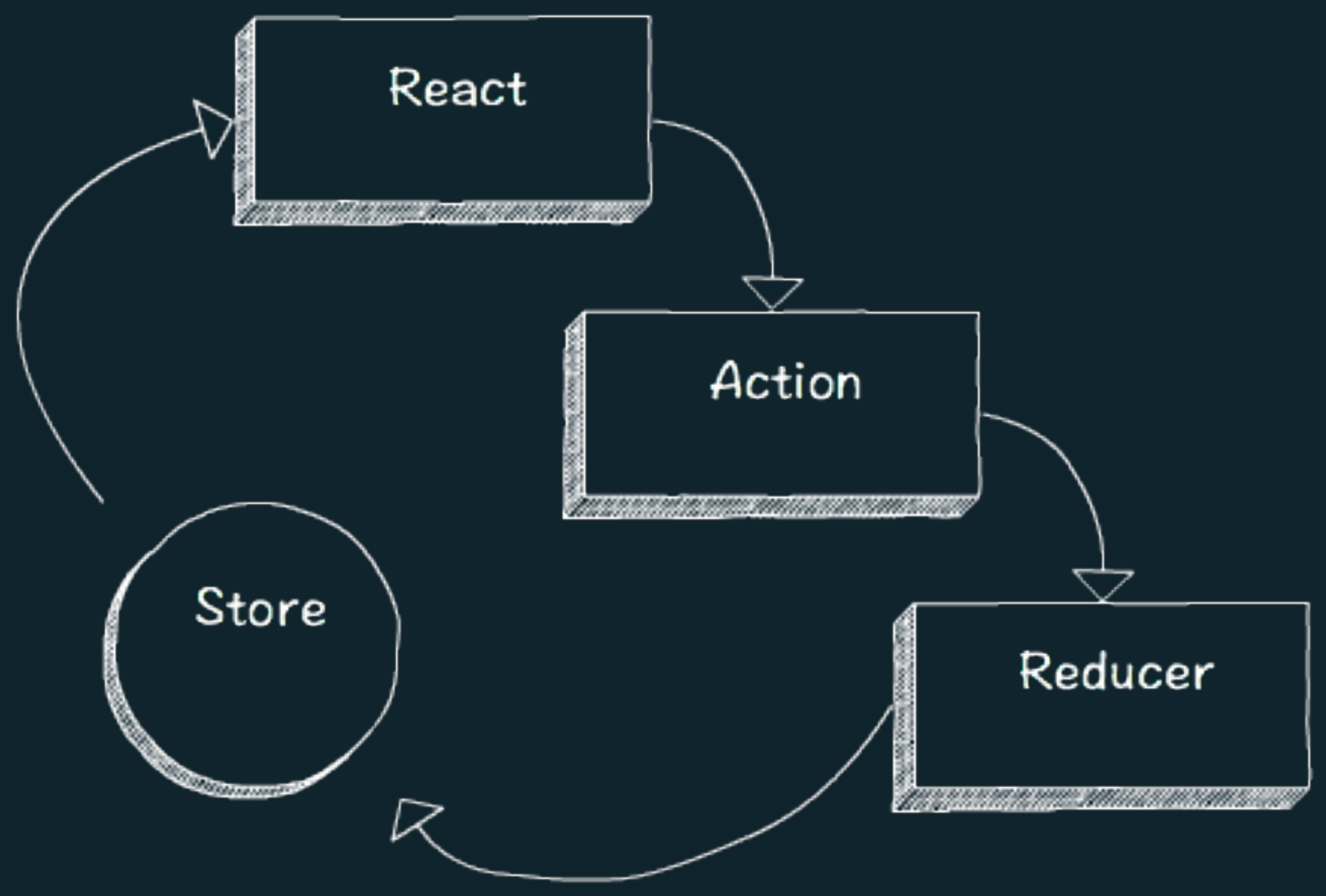
`props` **are the arguments passed to a component from the component above it.**

Relate this back to the component tree. Props are how we pass data between components.

State Management

State Management

Data Flow



Keeping with the one-way flow of data, this is how we communicate in a Redux environment.
^ This is basically the asterisk, this is how we can pass data upwards.
^ React, the UI, initiates an action, which is transformed by a reducer to create a new state in the store. The new store sends the new information to the top-level component for all the rest to re-render.

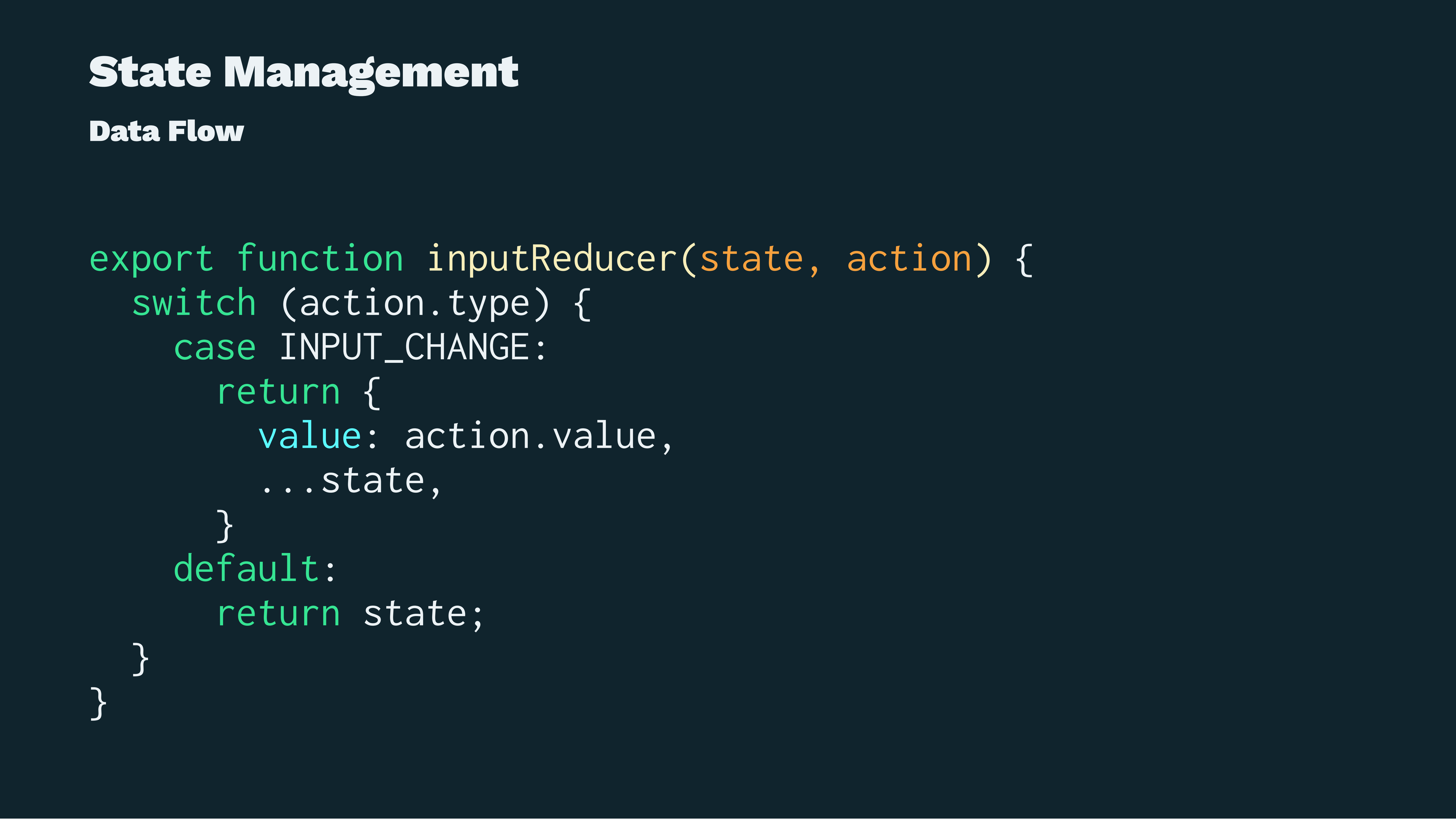
State Management

Data Flow

```
import { inputReducer } from 'components/Form'

const store = createStore({
  input: inputReducer,
})
```

The store is just a plain JavaScript object with special functions as values
^ Also: remember index.js? that's how we define a public API for our components.



State Management

Data Flow

```
export function inputReducer(state, action) {
  switch (action.type) {
    case INPUT_CHANGE:
      return {
        value: action.value,
        ...state,
      }
    default:
      return state;
  }
}
```

This is where we define what the new state will look like. That's what is happening in the INPUT_CHANGE case.

^ It takes in the state/store and an action.

^ Again, we're not mutating.

State Management

Data Flow

```js
// Input.actions.js
const INPUT_CHANGE = 'INPUT_CHANGE';

export function handleChange(event) {
  return {
    type: INPUT_CHANGE,
    value: event.target.value,
  }
}
```

An action is just a plain JS object with a special type key. This key is important because it's how we match it up with the correct reducer.

State Management

Data Flow

```
// Input.js
import { handleChange } from './actions';

function Input(props) {
  return (
    <input onChange={(event) => handleOnChange(event)}></input>
  )
}

function mapStateToProps(state) {
  return { input } = state;
}

function mapDispatchToProps(dispatch) {
  return({
    handleOnChange: (event) => {
      dispatch(handleChange(event));
    }
  })
}

export default connect(mapStateToProps, mapDispatchToProps)(Input)
```

Altogether now, you have a component, Input. It calls a function when you change the input. That function dispatches an action to the reducers/store. The store is updated. Input listens to the store, and gets the updated values in the form of props.

State Management

Cohesion

```
src/
|__ components/
|   |__ Form/
|       |__ Input/
|       |__ Form.js
|       |__ Form.actions.js
|       |__ Form.reducers.js
|       |__ index.js
|__ store.js
|__ index.js
```

We put everything together that is related to a feature.
^ Note the global store.

State Management

Cohesion

Q: How do components talk to each other?
A: Actions!

State Management

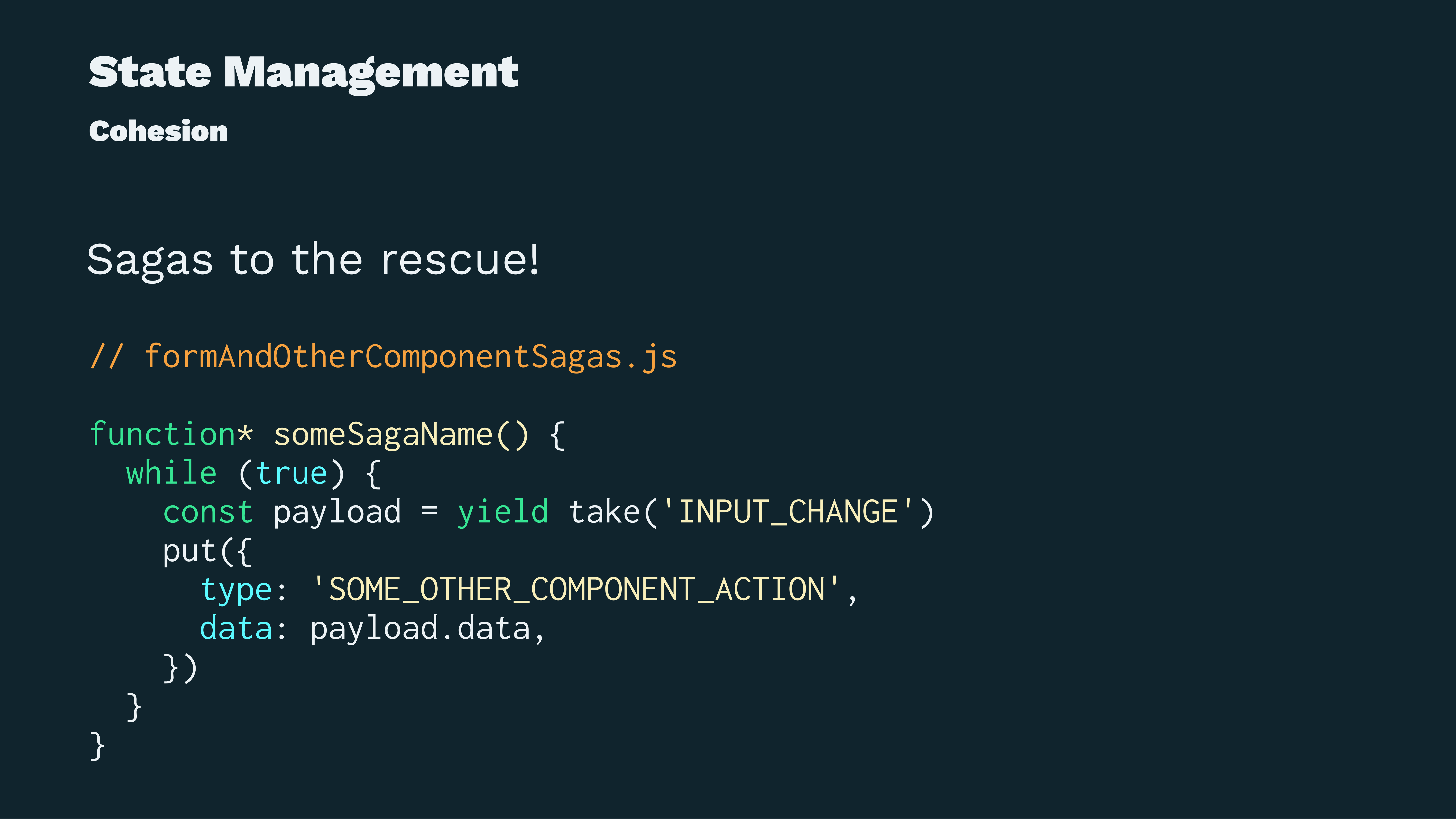
Cohesion

```
// Input.js
import { handleChange } from '../components/OtherComponent';

function Input(props) {
  // ...
}

function mapDispatchToProps(dispatch) {
  return({
    hadleOnChange: (event) => {
      dispatch(handleChange(event));
    }
  })
}
```

This time we're importing an action from another component to get something to happen.
^ Uh oh! Now these two unrelated components are coupled together.



State Management

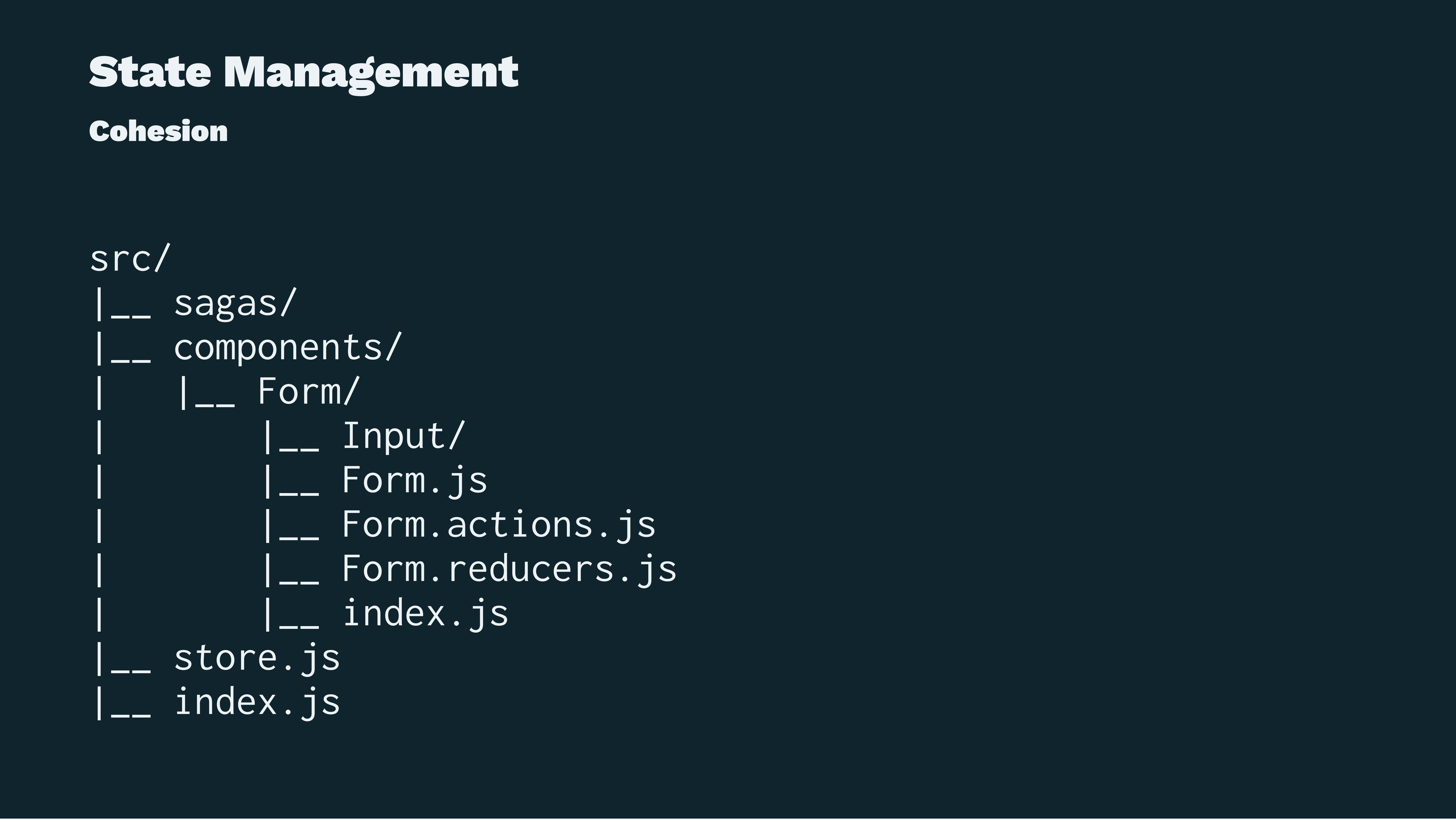
Cohesion

Sagas to the rescue!

```
// formAndOtherComponentSagas.js

function* someSagaName() {
  while (true) {
    const payload = yield take('INPUT_CHANGE')
    put({
      type: 'SOME_OTHER_COMPONENT_ACTION',
      data: payload.data,
    })
  }
}
```

Without getting too far into it, Sagas are these generator functions that pause on yields inside of them.
^ There's a Saga middleware that sits in the middle of the redux action/reducer flow that can listen for specific actions.
^ We've reduced coupling between the two components, because now all that our original one has to do is emit the same plain action that it always would have. It has no knowledge of how OtherComponent works.



State Management

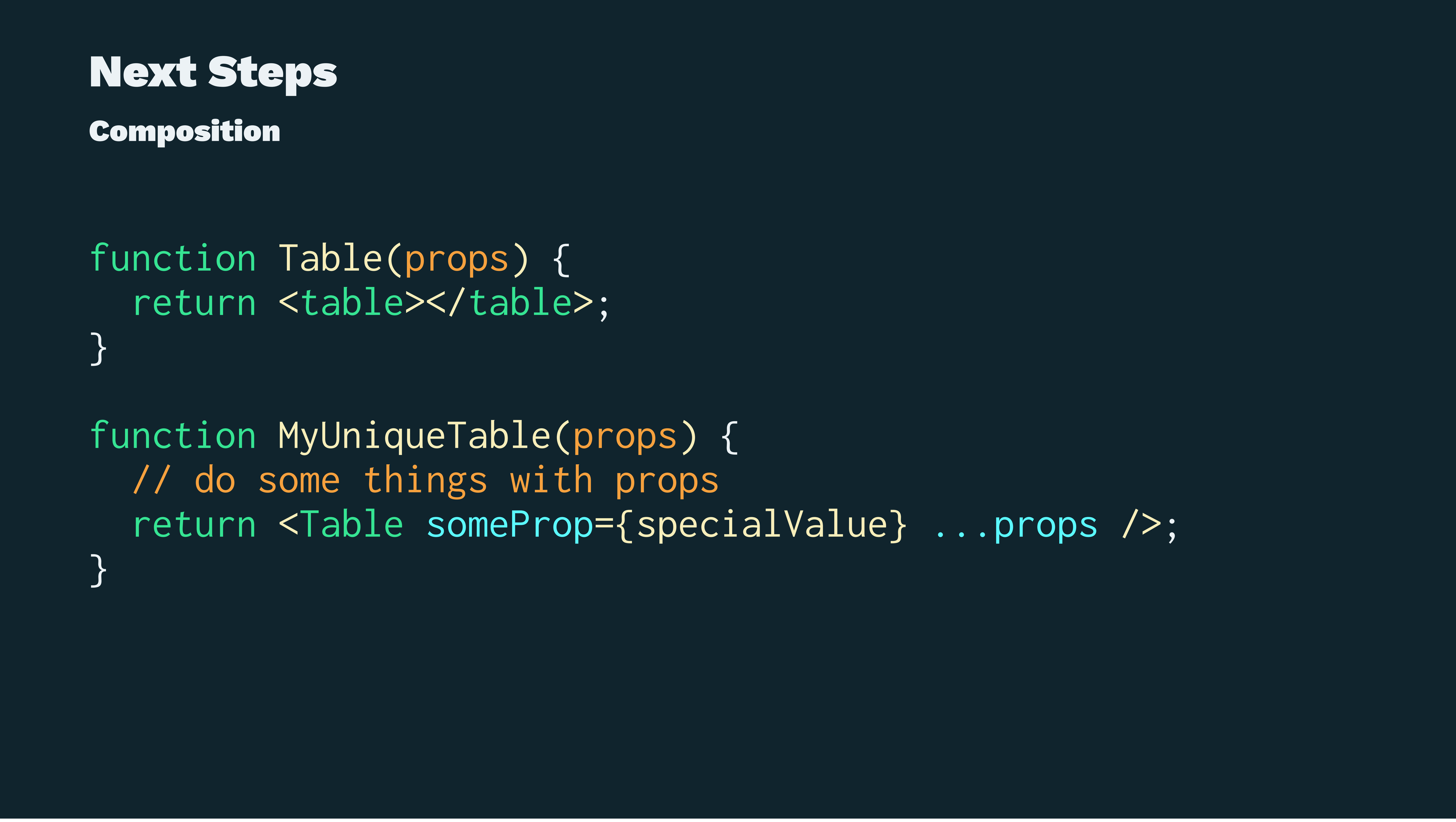
Cohesion

```
src/
|__ sagas/
|__ components/
|    |__ Form/
|         |__ Input/
|         |__ Form.js
|         |__ Form.actions.js
|         |__ Form.reducers.js
|         |__ index.js
|__ store.js
|__ index.js
```

This is something I need to learn more about. Where do the 'glue' sagas live? Right now, the best place I have for them is outside of the components/ directory.
^ React/Redux is a constant state of not knowing what to do.

Next Steps

These are some ideas that I wanted to touch on to either explain some more advanced techniques or introduce what is possible at the extremes.



Next Steps

Composition

```
function Table(props) {
  return <table></table>;
}

function MyUniqueTable(props) {
  // do some things with props
  return <Table someProp={specialValue} ...props />;
}
```

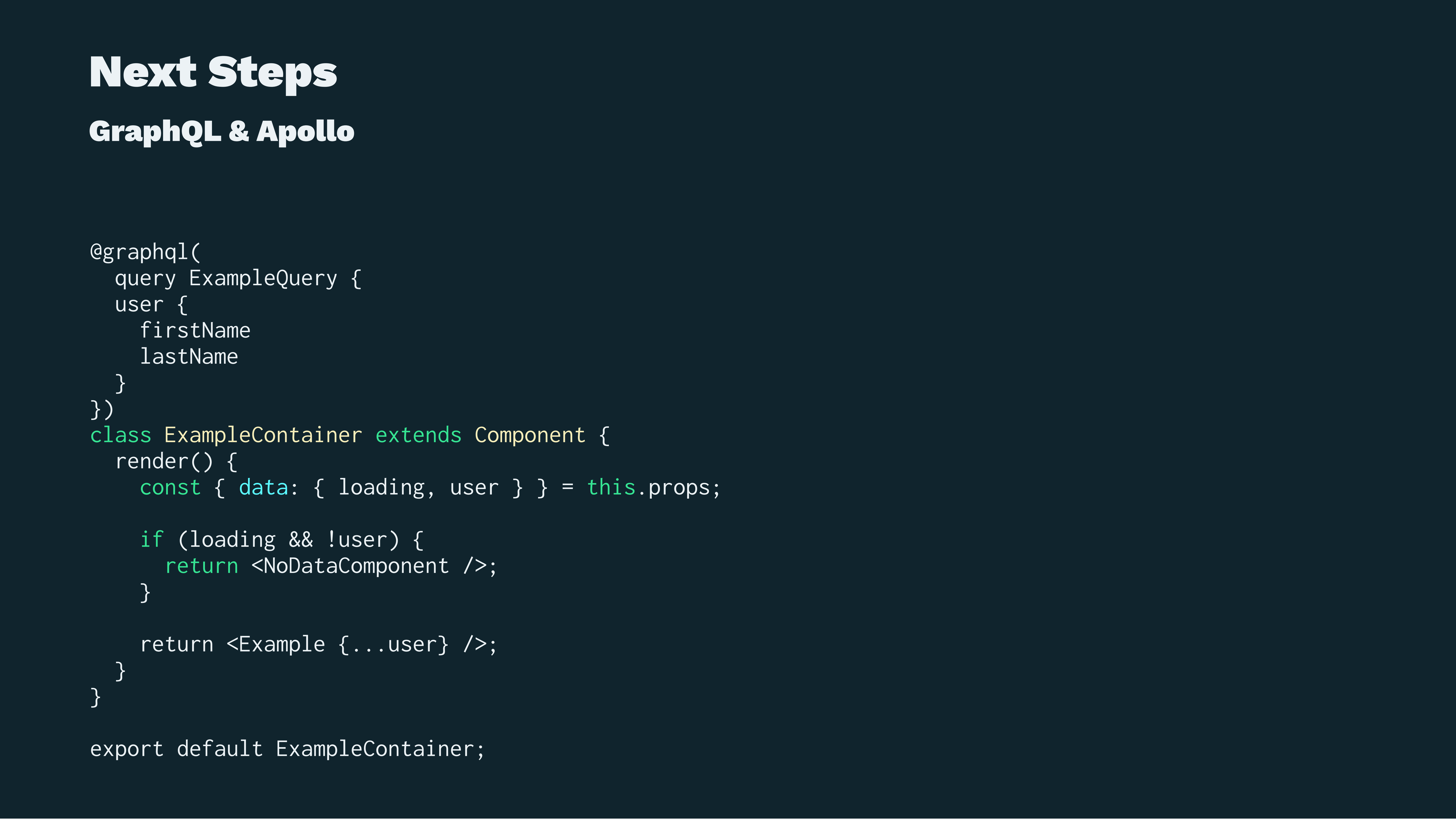
Instead of inheritance with OOP, we have composition with functional programming.
^ We should focus initially on building generic components that we can easily specialize to create full experiences.
^ https://reactjs.org/docs/composition-vs-inheritance.html

Next Steps

Async Actions

```
// Input.sagas.js
function* inputSaga() {
  while (true) {
    const payload = yield take('INPUT_CHANGE');
    const response = yield fetch('/some/api/endpoing', {value: payload.value});
    if (response.ok?) {
      put({type: 'REQUEST_SUCCESS', newValue: response.value});
    } else {
      put({type: 'REQUEST_FAILED', error: response.error});
    }
  }
}
```

Here's a quick example (with some pseudo code) to illustrate how to do async requests with Sagas.
^ This example exists with the rest of the Input component because it is specific to that component.
^ Now this is cool, but this can easily breakdown when you have multiple components all initiating their own fetches.



Next Steps

GraphQL & Apollo

```
@graphql(
  query ExampleQuery {
  user {
    firstName
    lastName
  }
})
class ExampleContainer extends Component {
  render() {
    const { data: { loading, user } } = this.props;

    if (loading && !user) {
      return <NoDataComponent />;
    }

    return <Example {...user} />;
  }
}

export default ExampleContainer;
```

Components can declare what data they want. GraphQL will handle the fetching. Apollo will make sure it only fetches data that we don't alreay have locally.
^ Apollo query batching
^ https://blog.apollographql.com/reducing-our-redux-code-with-react-apollo-5091b9de9c2a

Try it Yourself

1. Clone the repo (github.com/benortiz/talk-react-crash-course).
2. Create a `<HelloWorld />` component.
3. Use JSON Placeholder to create your own async component.