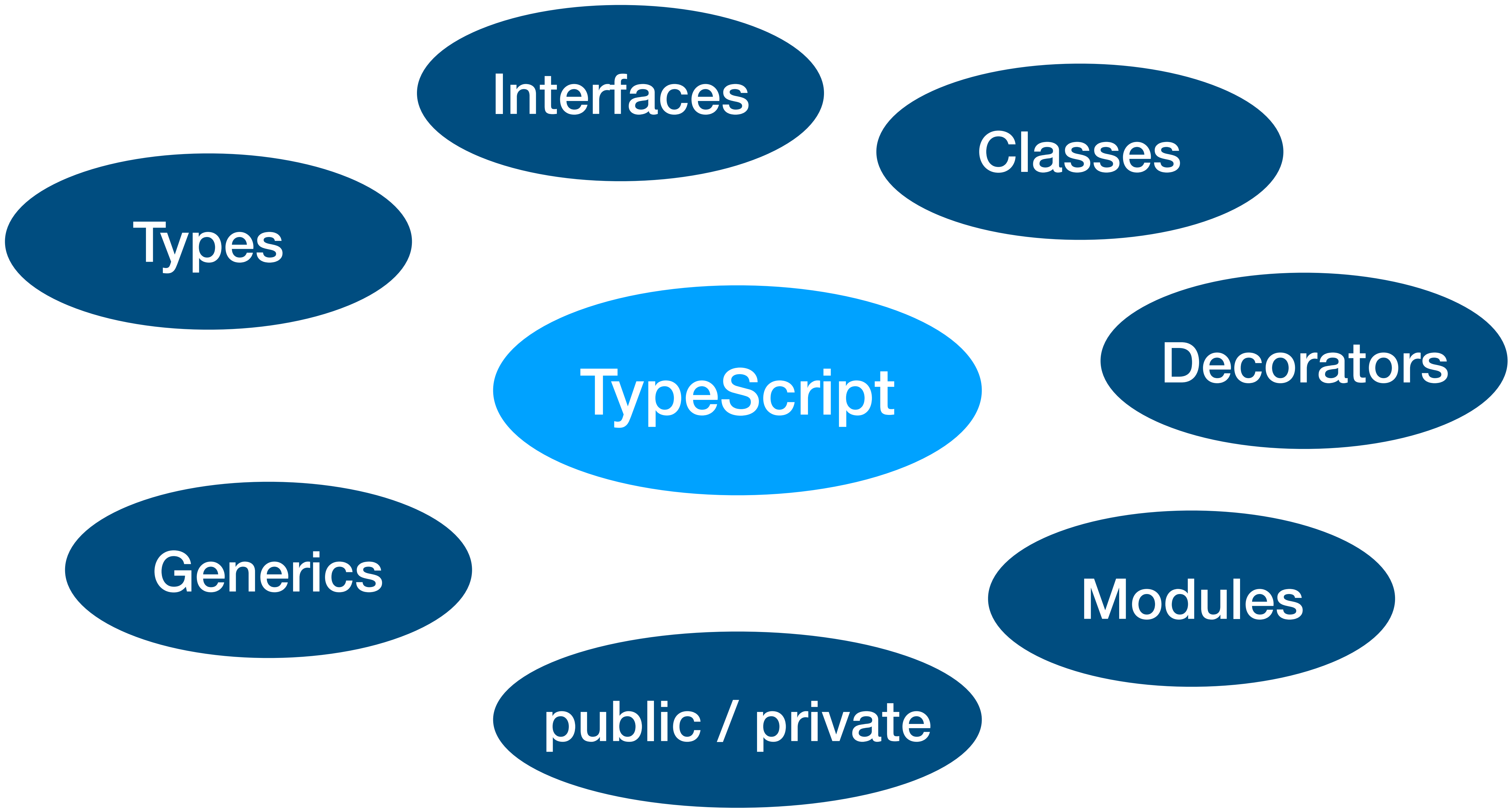
TypeScript

TypeScript is a **typed superset** of JavaScript that **compiles to plain JavaScript.**

TypeScript in a Nutshell



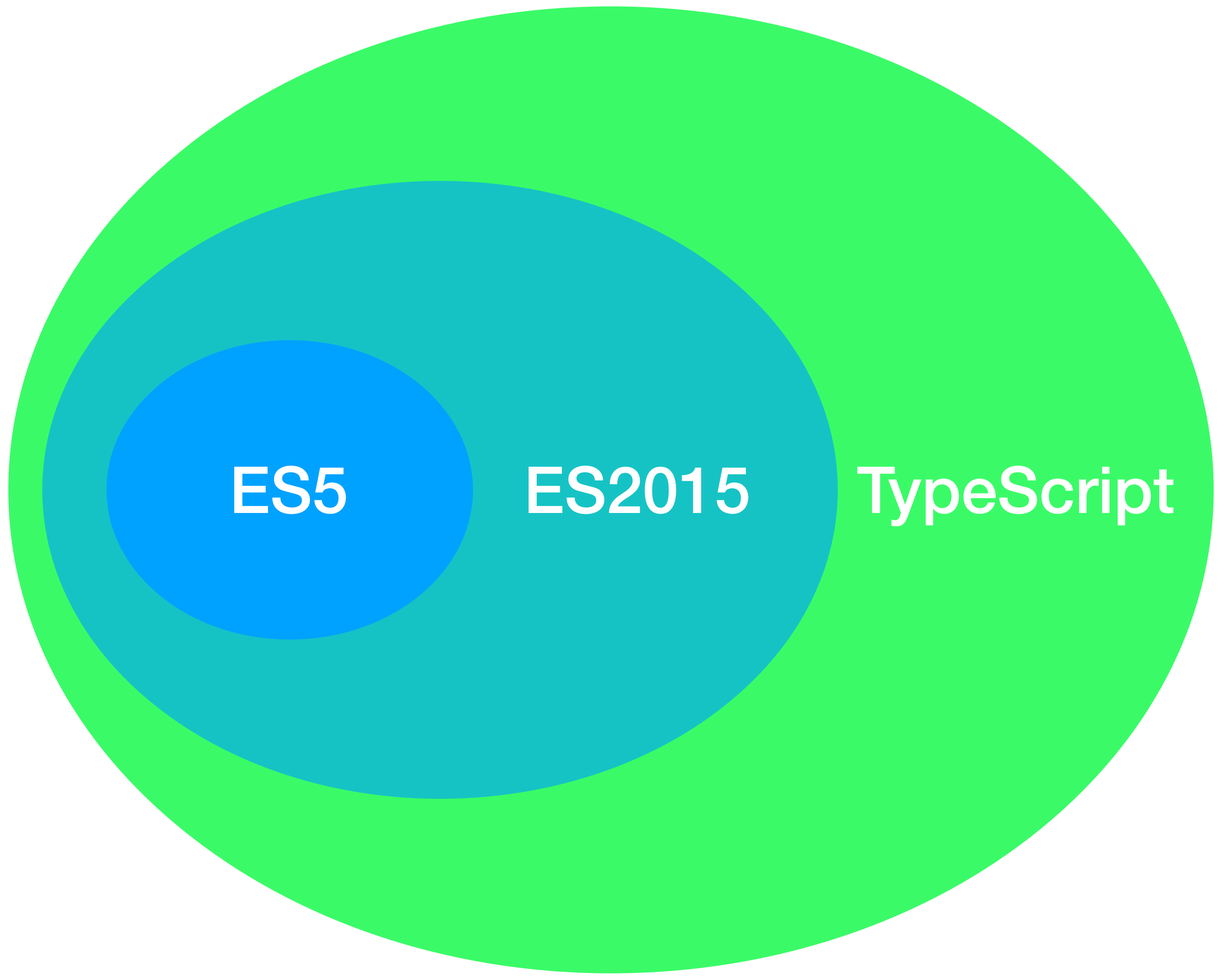
Why TypeScript?

- Find Errors and Flaws as soon as possible, at best: while developing or building.
- Refactoring based on Static Code Analysis.
- Code Completition and Symbol based navigation.
- Documentation
- Expressiveness and Software Engineering

Our Goal:

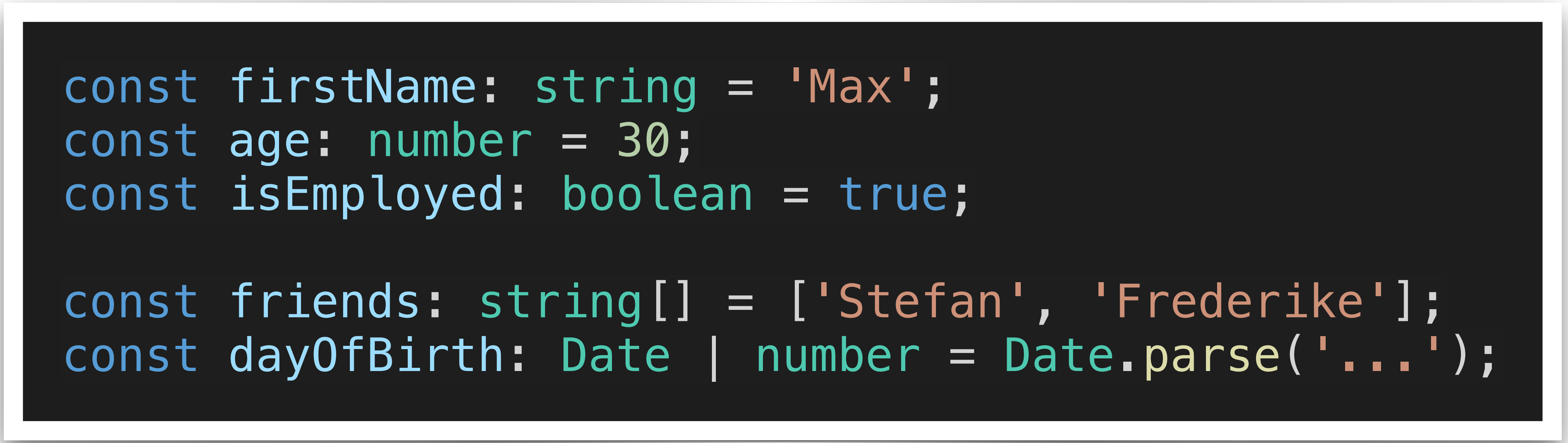
Better maintenance for long-living projects.

TypeScript as Superset



Primitive Types

TypeScript has all the JavaScript Primitive Types



```typescript
const firstName: string = 'Max';
const age: number = 30;
const isEmployed: boolean = true;

const friends: string[] = ['Stefan', 'Frederike'];
const dayOfBirth: Date | number = Date.parse('...');
```

Type inference

You don't have to state the type of a variable or constant if the compiler can *infer* it.

```
const firstName = 'Max'; // string
const age = 30; // number
const isEmployed = true; // boolean

const friends = ['Stefan', 'Frederike']; // string[]
const dayOfBirth = Date.parse('...'); // Date
```

Is this what you want?

null and undefined

By default each data type in JavaScript (and so in TypeScript) can accept ***null*** and ***undefined***.

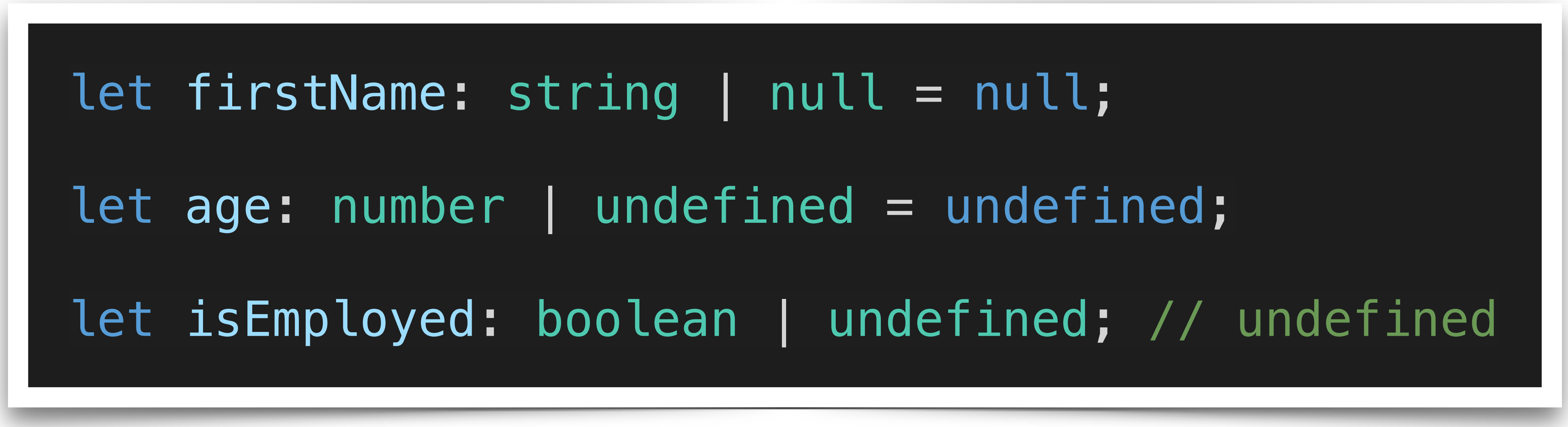
```
let firstName: string = null;

let age: number = undefined;

let isEmployed: boolean; // undefined
```

This is not recommended!

--strictNullChecks

In strict null checking mode, the null and undefined values are not in the domain of every type.

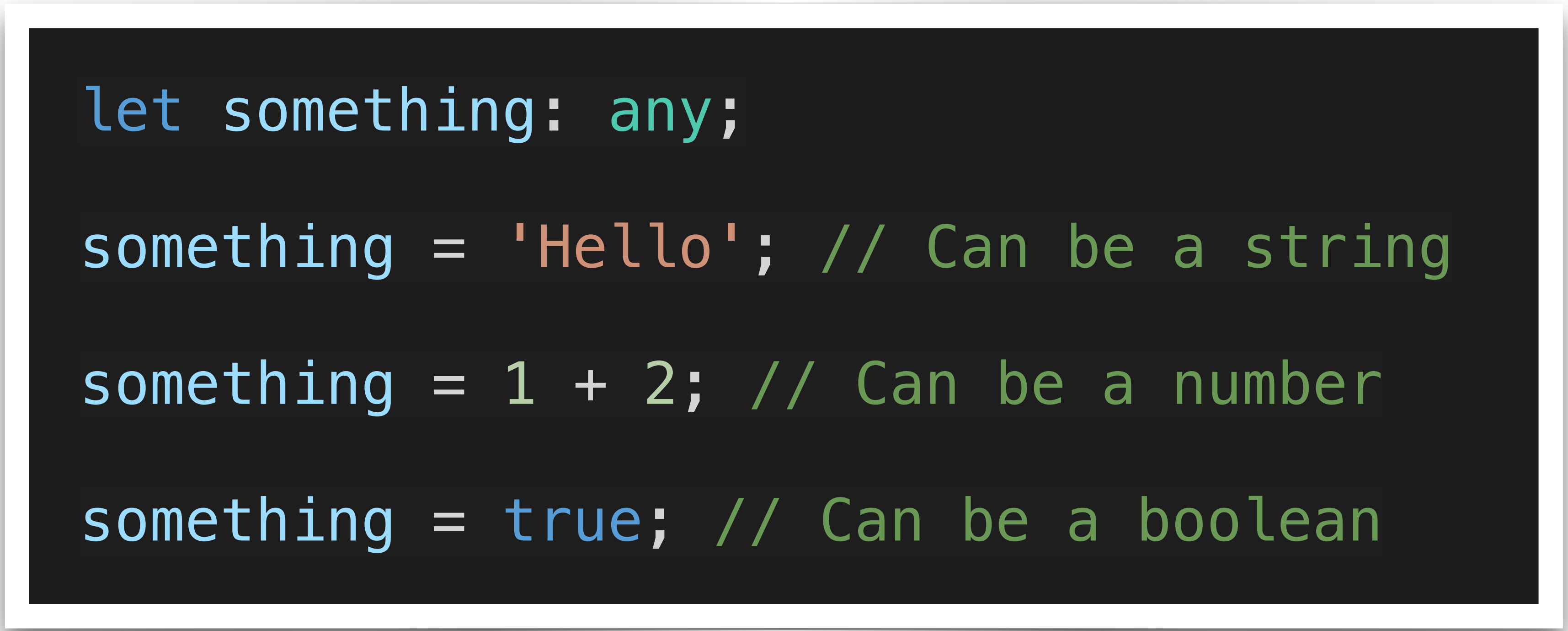
You have to state explicitly if undefined or null are accepted.



```
let firstName: string | null = null;

let age: number | undefined = undefined;

let isEmployed: boolean | undefined; // undefined
```

any

As the name suggests, ***any*** can be anything.



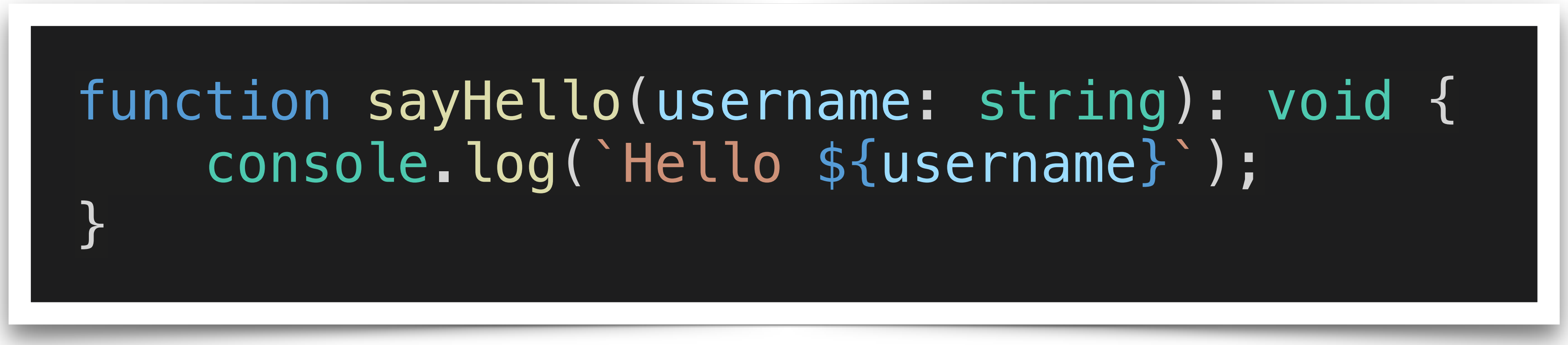
```
let something: any;

something = 'Hello'; // Can be a string

something = 1 + 2; // Can be a number

something = true; // Can be a boolean
```

Avoid the *any* data type.

Function

Function parameters as well as the return value can be typed.

If the function has no return value, use the special type ***void***.



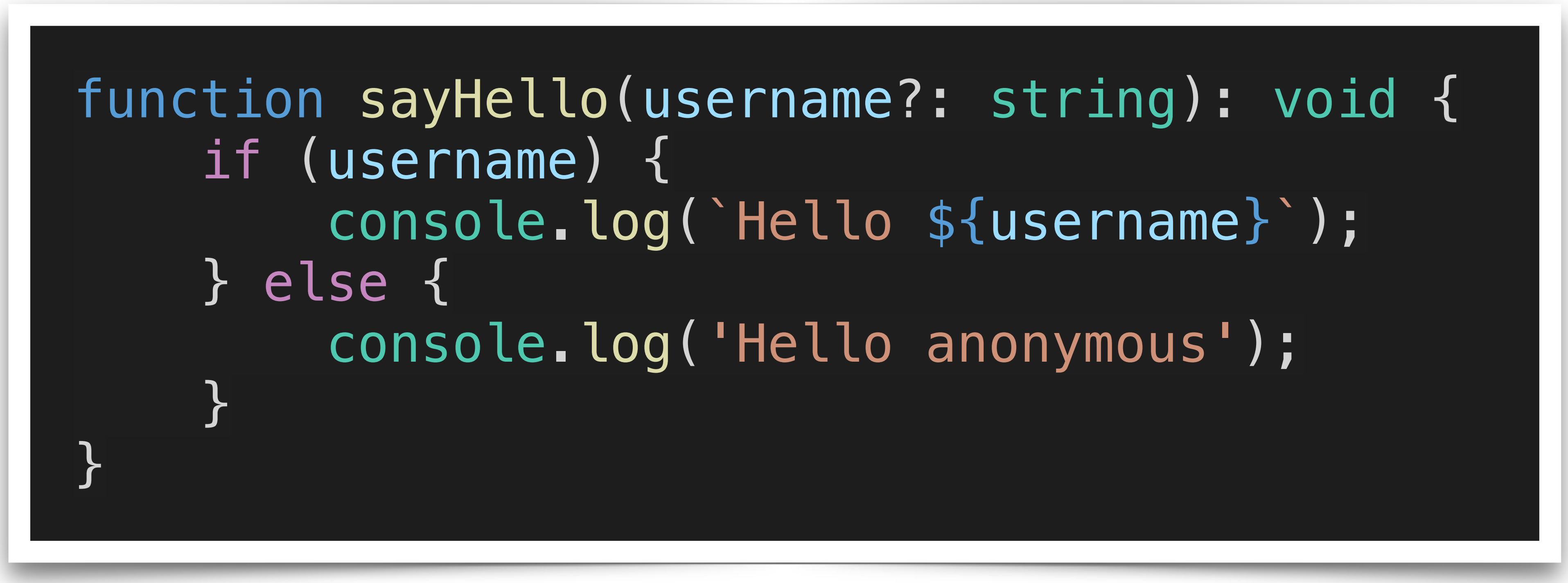
```
function sayHello(username: string): void {
    console.log(`Hello ${username}`);
}
```

What’s the return type if we omit it?

Optionals

We can use Optionals to state that a parameter might be undefined.

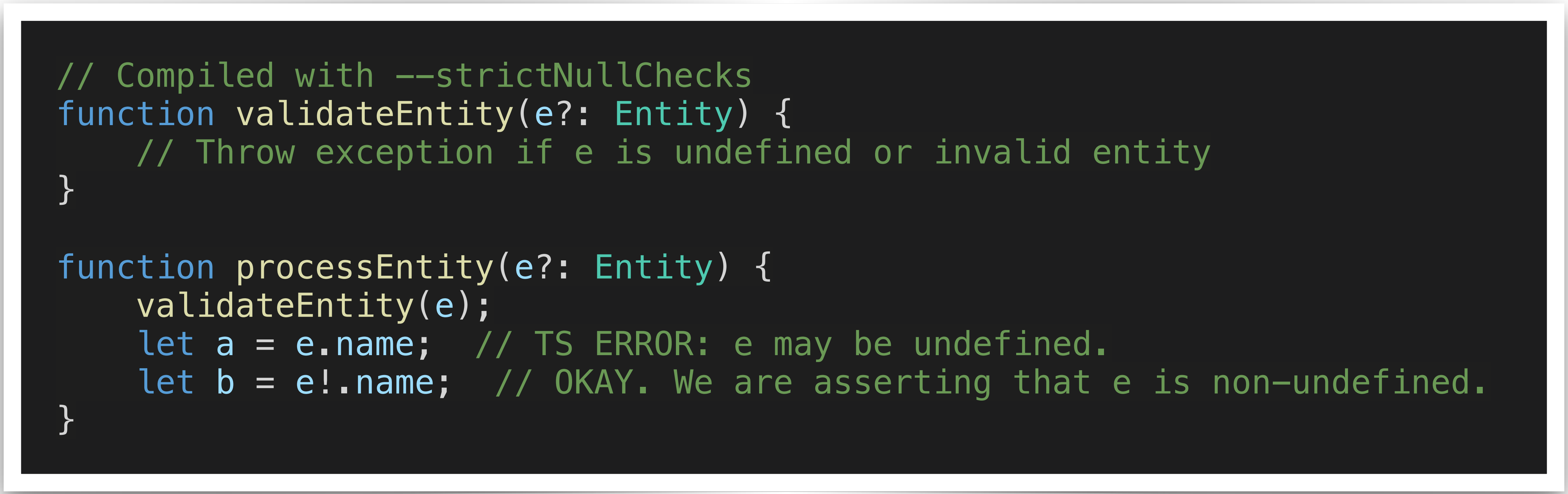
We have to guard our code in such case.



```typescript
function sayHello(username?: string): void {
    if (username) {
        console.log(`Hello ${username}`);
    } else {
        console.log('Hello anonymous');
    }
}
```

! - Non-Null Assertion Operator

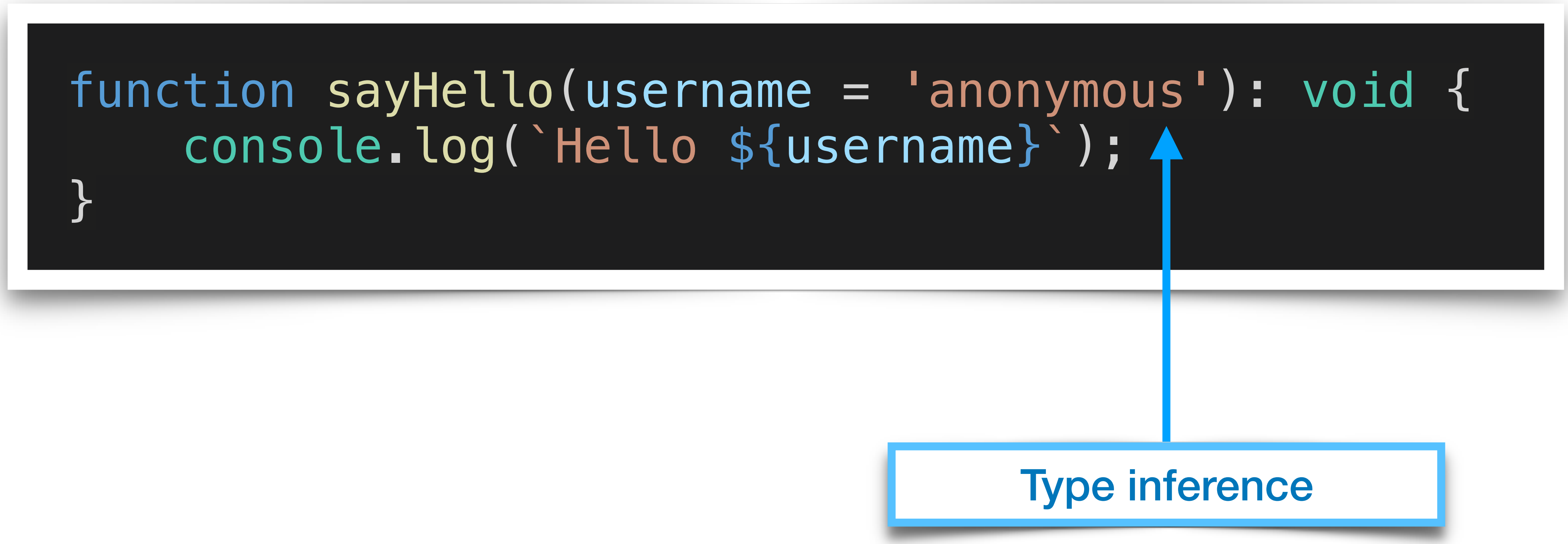
A new **!** post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact.



```typescript
// Compiled with --strictNullChecks
function validateEntity(e?: Entity) {
    // Throw exception if e is undefined or invalid entity
}

function processEntity(e?: Entity) {
    validateEntity(e);
    let a = e.name;  // TS ERROR: e may be undefined.
    let b = e!.name;  // OKAY. We are asserting that e is non-undefined.
}
```

default values

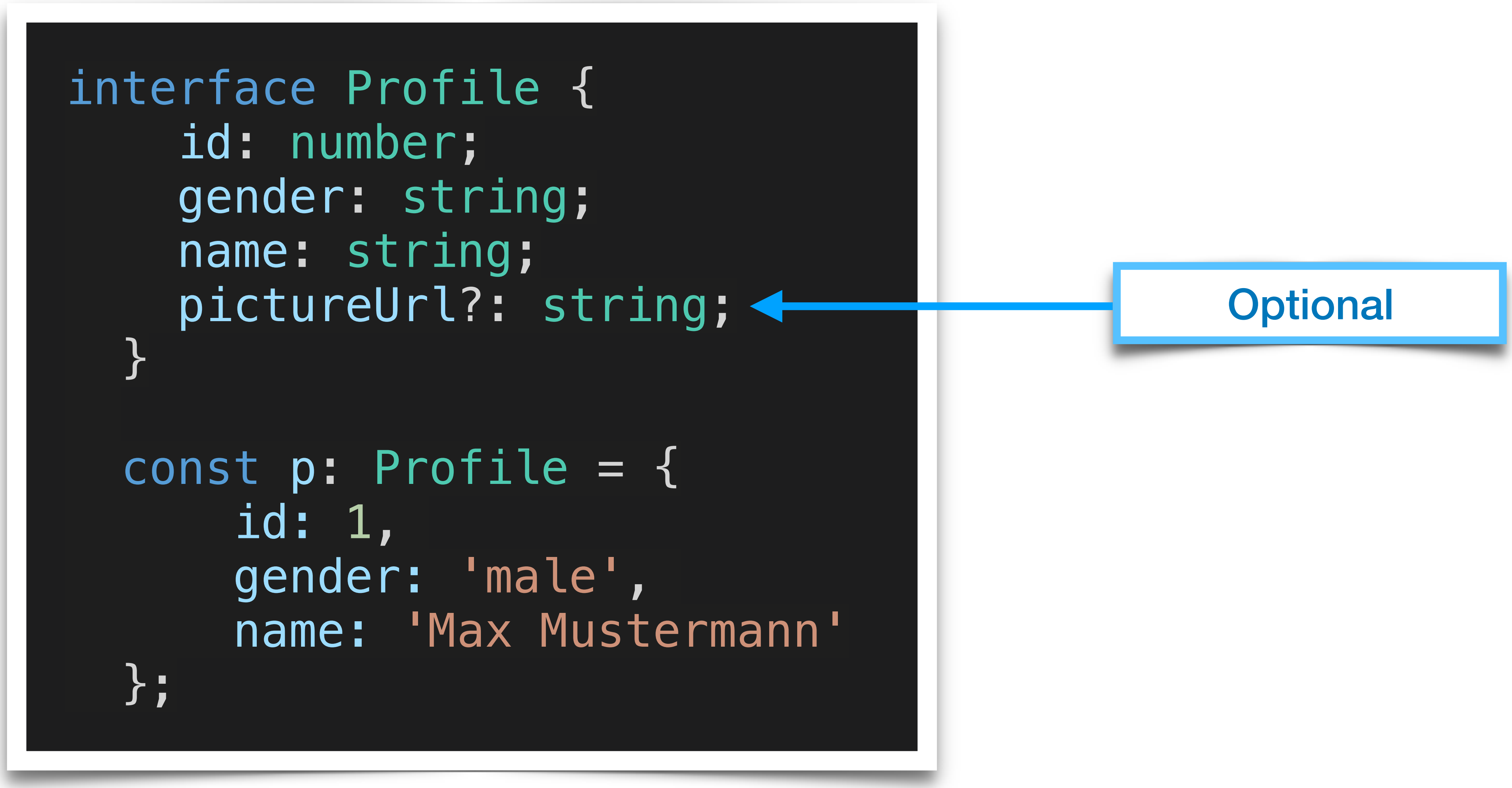
Quite often it's easier and safer to define default values for parameters instead of dealing with optionals.



```
function sayHello(username = 'anonymous'): void {
    console.log(`Hello ${username}`);
}
```

Interfaces

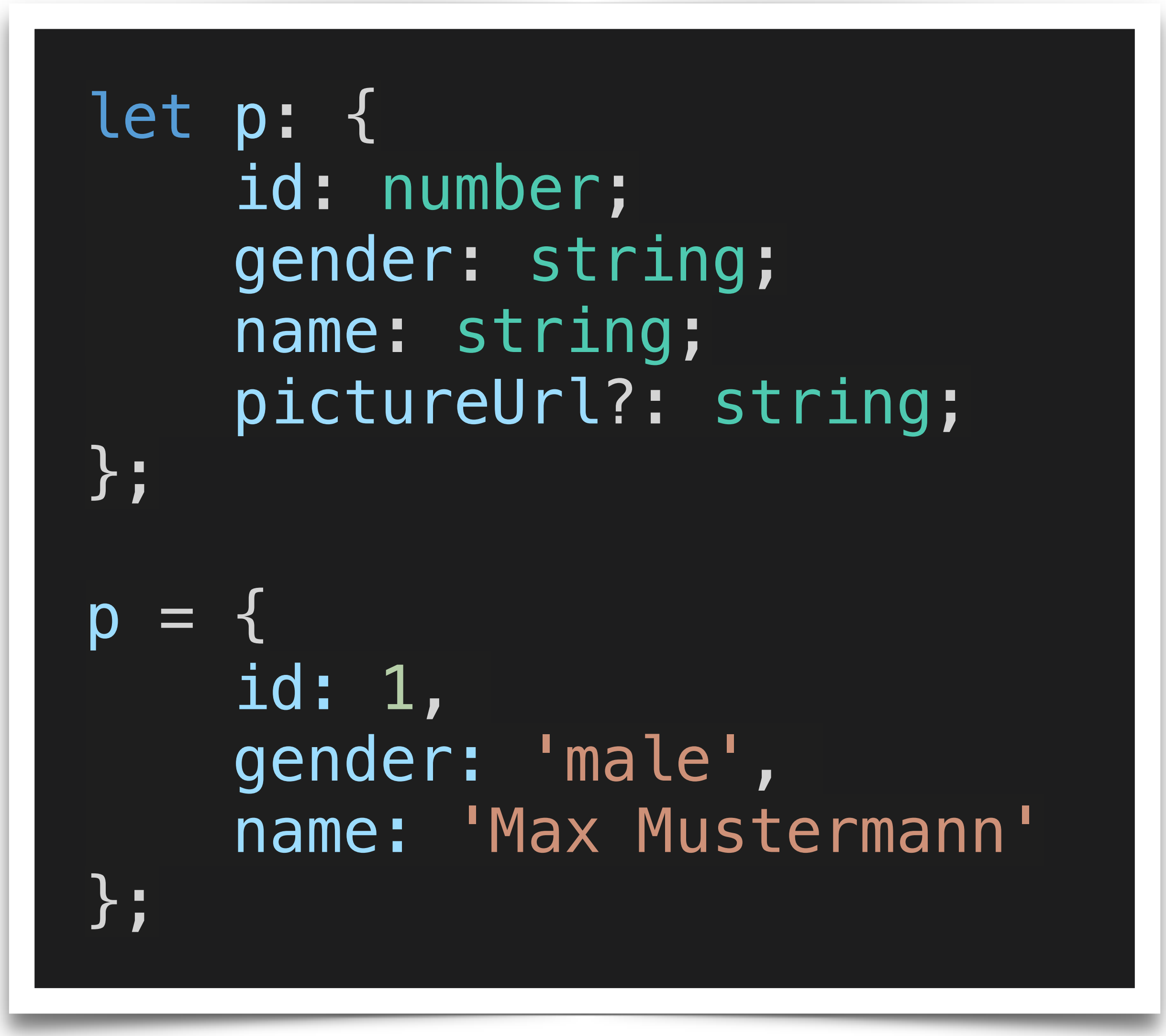
Interfaces give your data a ***shape***.



```
interface Profile {
    id: number;
    gender: string;
    name: string;
    pictureUrl?: string;
  }

  const p: Profile = {
      id: 1,
      gender: 'male',
      name: 'Max Mustermann'
  };
```

Interfaces „on the fly“

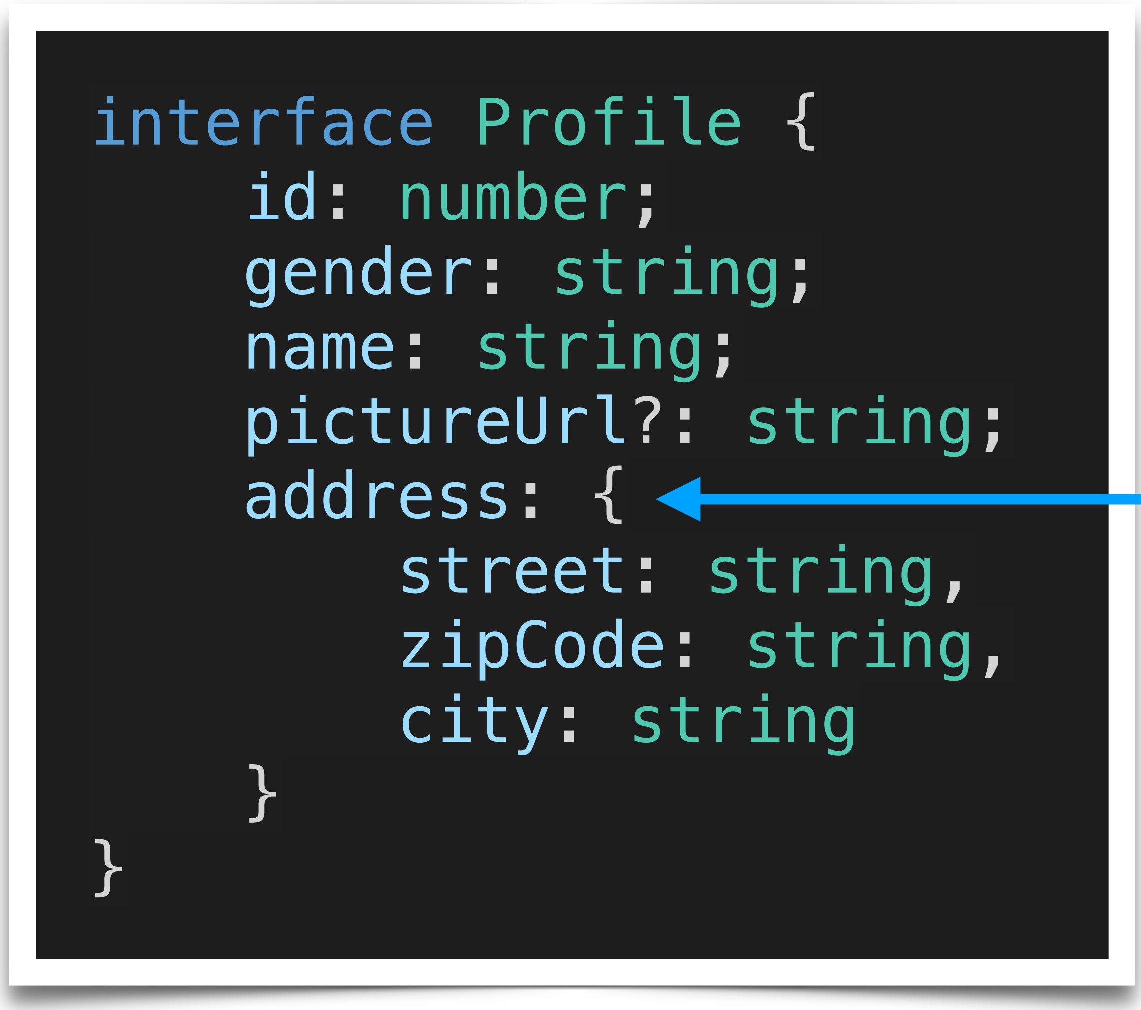
Interfaces can be declared „on the fly“.



```
let p: {
    id: number;
    gender: string;
    name: string;
    pictureUrl?: string;
};

p = {
    id: 1,
    gender: 'male',
    name: 'Max Mustermann'
};
```

Nested Interfaces

Interfaces can be declared nested.

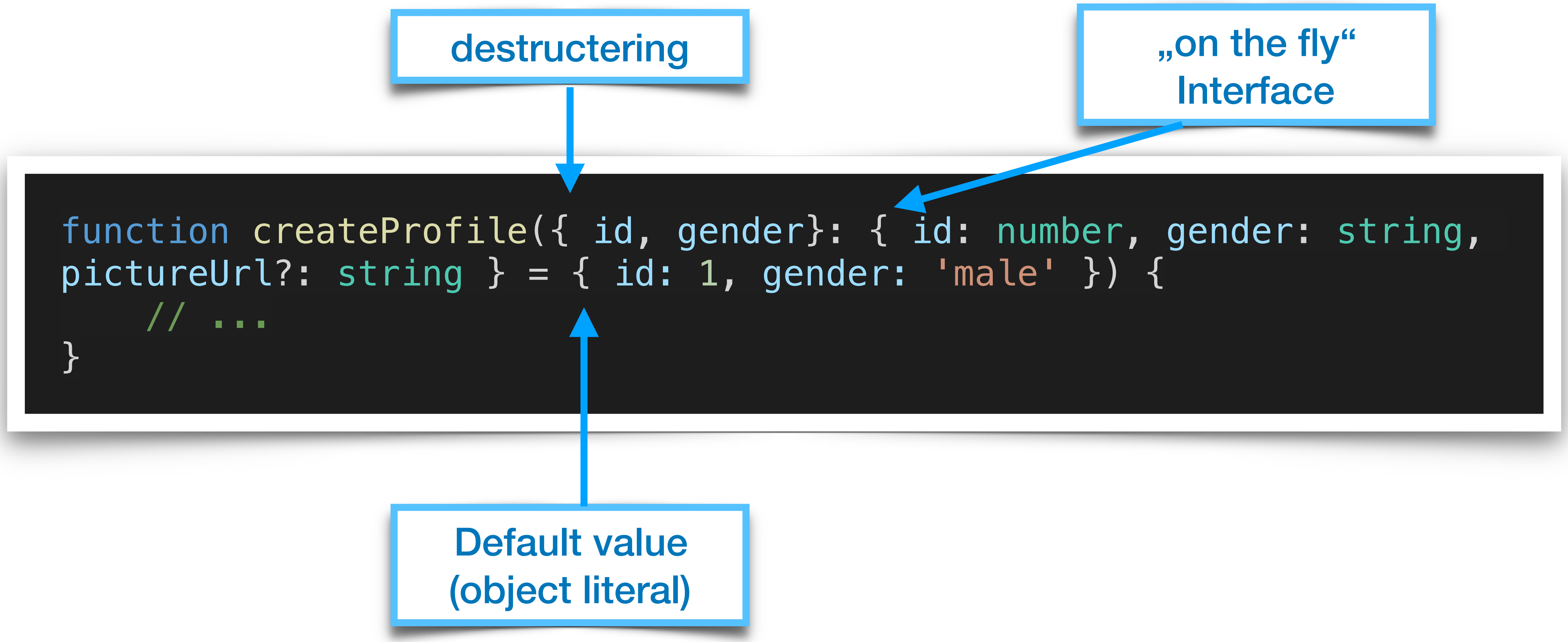


```
interface Profile {
    id: number;
    gender: string;
    name: string;
    pictureUrl?: string;
    address: {
        street: string,
        zipCode: string,
        city: string
    }
}
```

„on the fly“
Interface

Interfaces and destructering

„on the fly“ interfaces might be somehow confusing.



```
function createProfile({ id, gender}: { id: number, gender: string,
pictureUrl?: string } = { id: 1, gender: 'male' }) {
    // ...
}
```

Callbacks in TypeScript

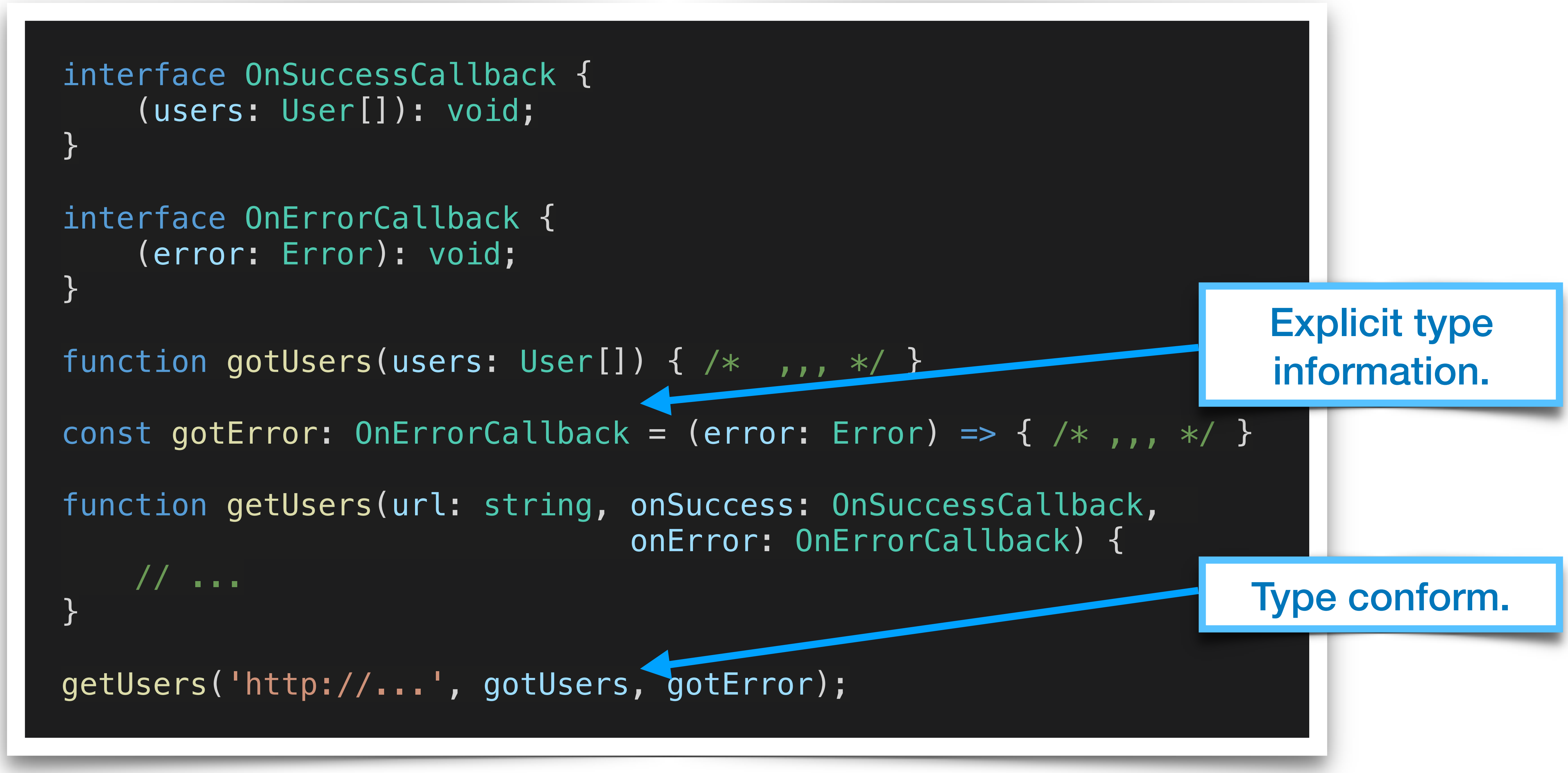
JavaScript lives in an asynchronous world and callbacks is one way to deal with it.

First approach with the data type ***Function***:

```typescript
function gotUsers(users: User[]) { /*  ,,, */ }

function gotError(error: Error) { /* ,,, */ }

function getUsers(url: string, onSuccess: Function, onError: Function) {
    // ...
}

getUsers('http://...', gotUsers, gotError);
```

Function interfaces

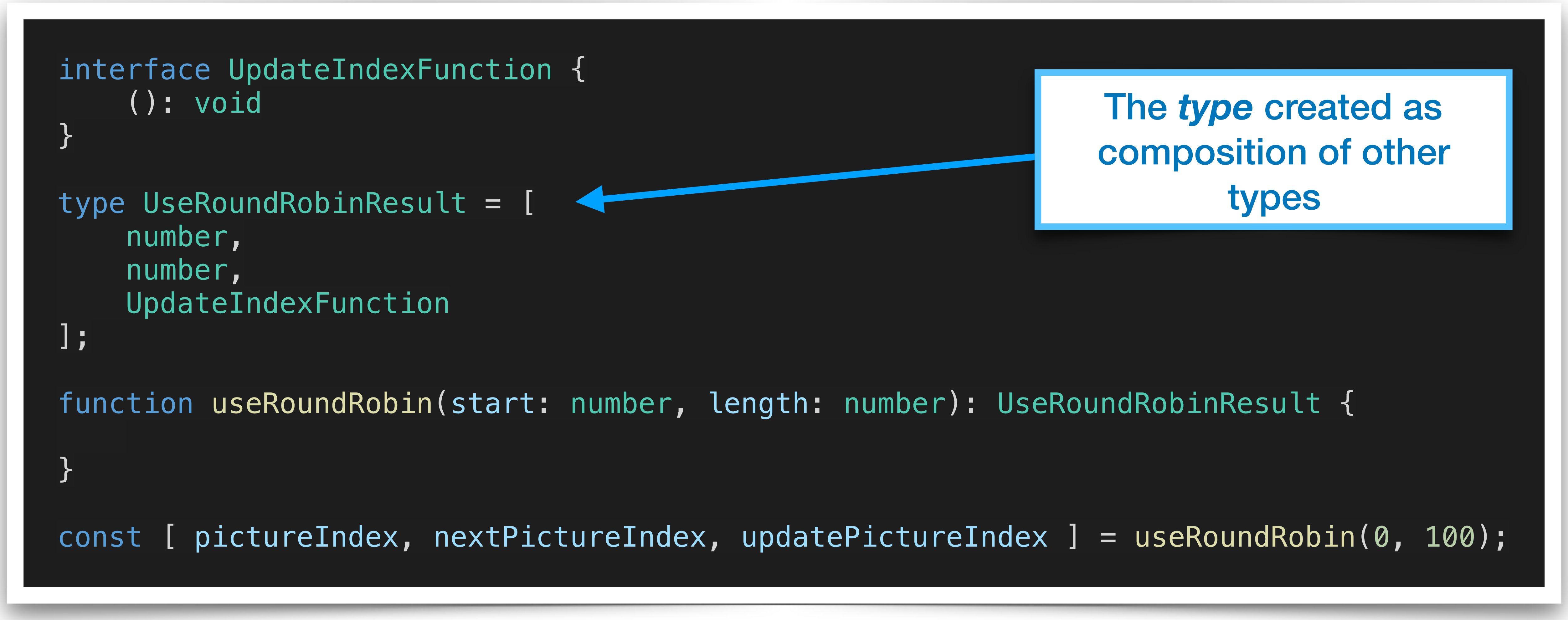
Let’s created an ***interface*** for a function:



```
interface OnSuccessCallback {
    (users: User[]): void;
}

interface OnErrorCallback {
    (error: Error): void;
}

function gotUsers(users: User[]) { /*  ,,, */ }

const gotError: OnErrorCallback = (error: Error) => { /* ,,, */ }

function getUsers(url: string, onSuccess: OnSuccessCallback,
                               onError: OnErrorCallback) {
    // ...
}

getUsers('http://...', gotUsers, gotError);
```

Typing Arrays

Let’s created a type for an array using the ***type*** keyword:



```
interface UpdateIndexFunction {
    (): void
}

type UseRoundRobinResult = [
    number,
    number,
    UpdateIndexFunction
];

function useRoundRobin(start: number, length: number): UseRoundRobinResult {

}

const [ pictureIndex, nextPictureIndex, updatePictureIndex ] = useRoundRobin(0, 100);
```

Typings for JavaScript

- Not all JavaScript libraries have TypeScript type informations „out of the box“.
- Type informations, so called ***typings*** contain only the type informations (interfaces, types, …) but not the implementation (no classes).
- They have a ****.d.ts*** suffix.
- Often managed by the JavaScript library vendor or by the community as part of the ***DefinitelyTyped*** project.

TypeScript in React

TypeScript in React

```
> create-react-app myApp --typescript
```

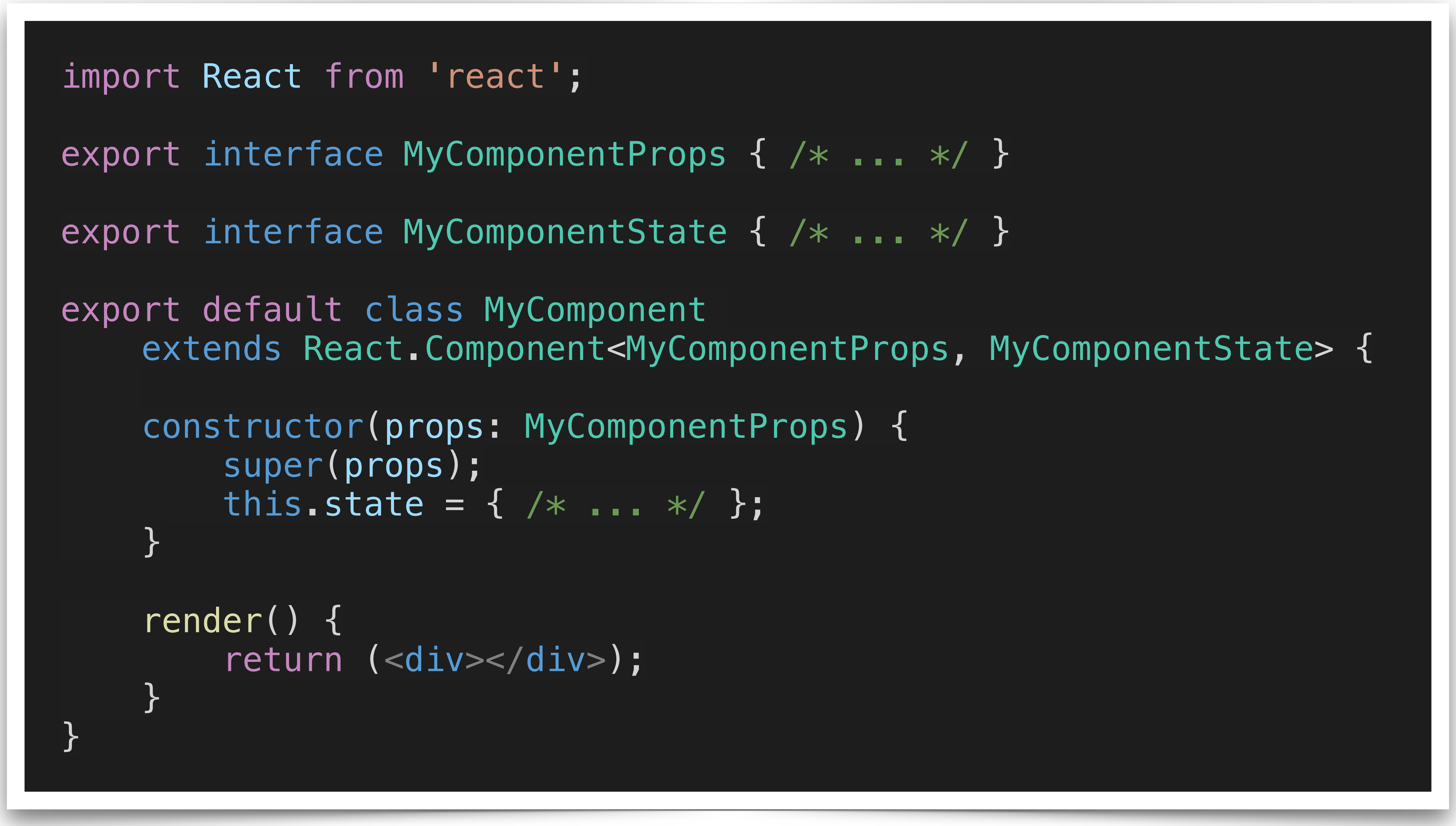
- Included now is a ***tsconfig.json*** file configuring the Typescript compiler options.

- ***.js* files are now **.tsx* files**. The Typescript compiler will pick up all .tsx files at compile time.

- ***package.json* contains dependencies for @*types* packages**, including support for node, jest, react and react-dom out of the box.

Class Component with TypeScript

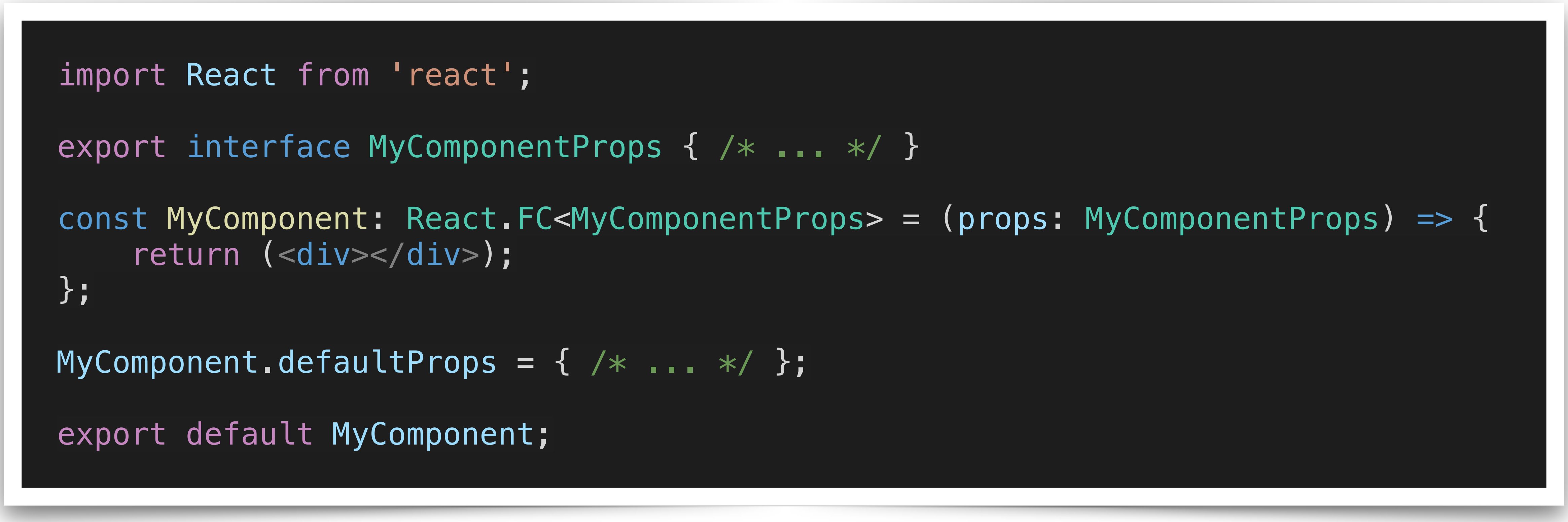
Extend ***React.Component<TProps, TState>***



```
import React from 'react';

export interface MyComponentProps { /* ... */ }

export interface MyComponentState { /* ... */ }

export default class MyComponent
    extends React.Component<MyComponentProps, MyComponentState> {

    constructor(props: MyComponentProps) {
        super(props);
        this.state = { /* ... */ };
    }

    render() {
        return (<div></div>);
    }
}
```

Function Component with TypeScript

Extend ***React.FC<TProps>***



```
import React from 'react';

export interface MyComponentProps { /* ... */ }

const MyComponent: React.FC<MyComponentProps> = (props: MyComponentProps) => {
    return (<div></div>);
};

MyComponent.defaultProps = { /* ... */ };

export default MyComponent;
```

Hooks

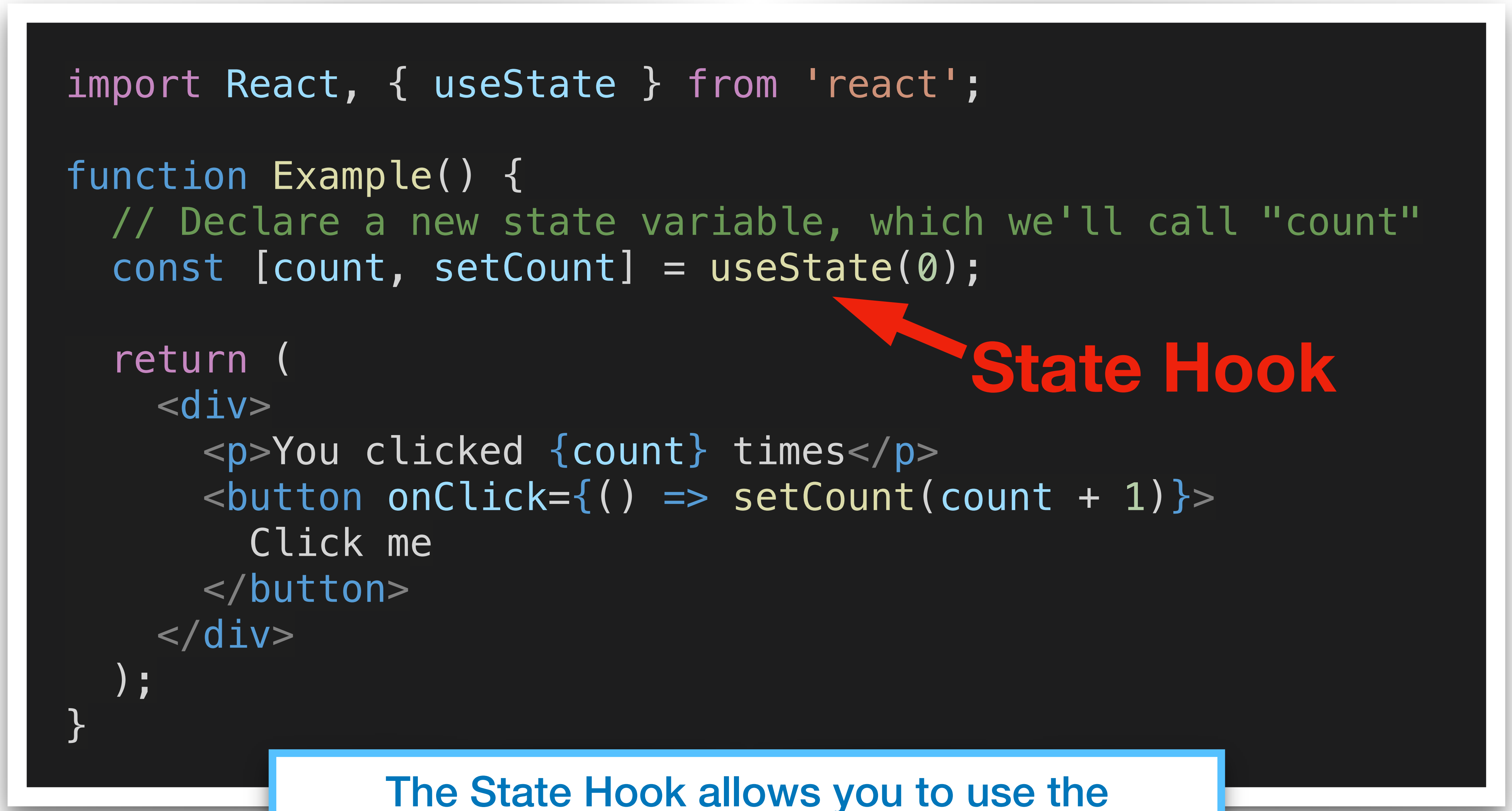
„*Hooks* are a new addition in React 16.8. They let you use state and other React features without writing a class.“

– reactjs.org

Why Hooks?

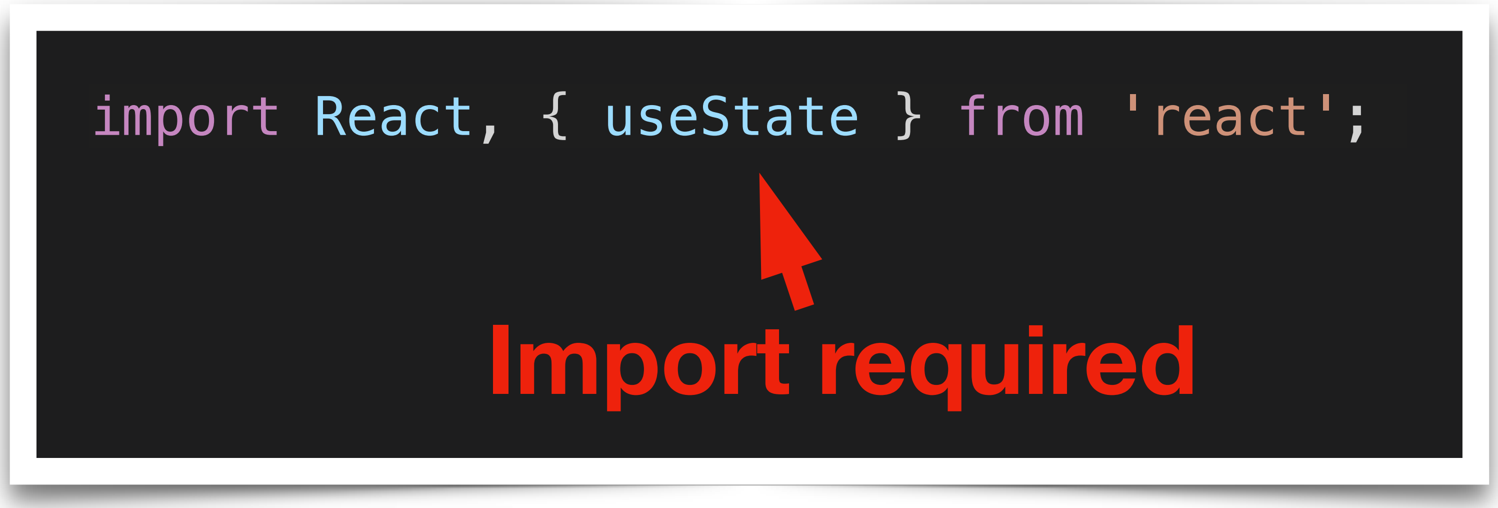
- With Hooks, you can extract stateful logic from a component so it can be tested independently and reused.
- Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data), rather than forcing a split based on lifecycle methods
- Classes confuse both people and machines - Hooks let you use more of React's features without classes

State Hook

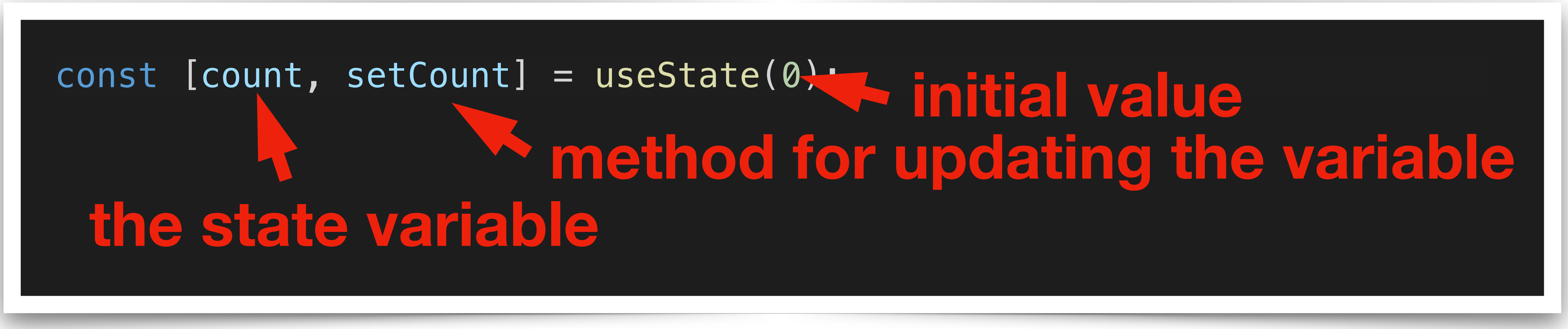


```
import React, { useState } from 'react';

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>
  );
}
```

The State Hook allows you to use the component State within a Function Component

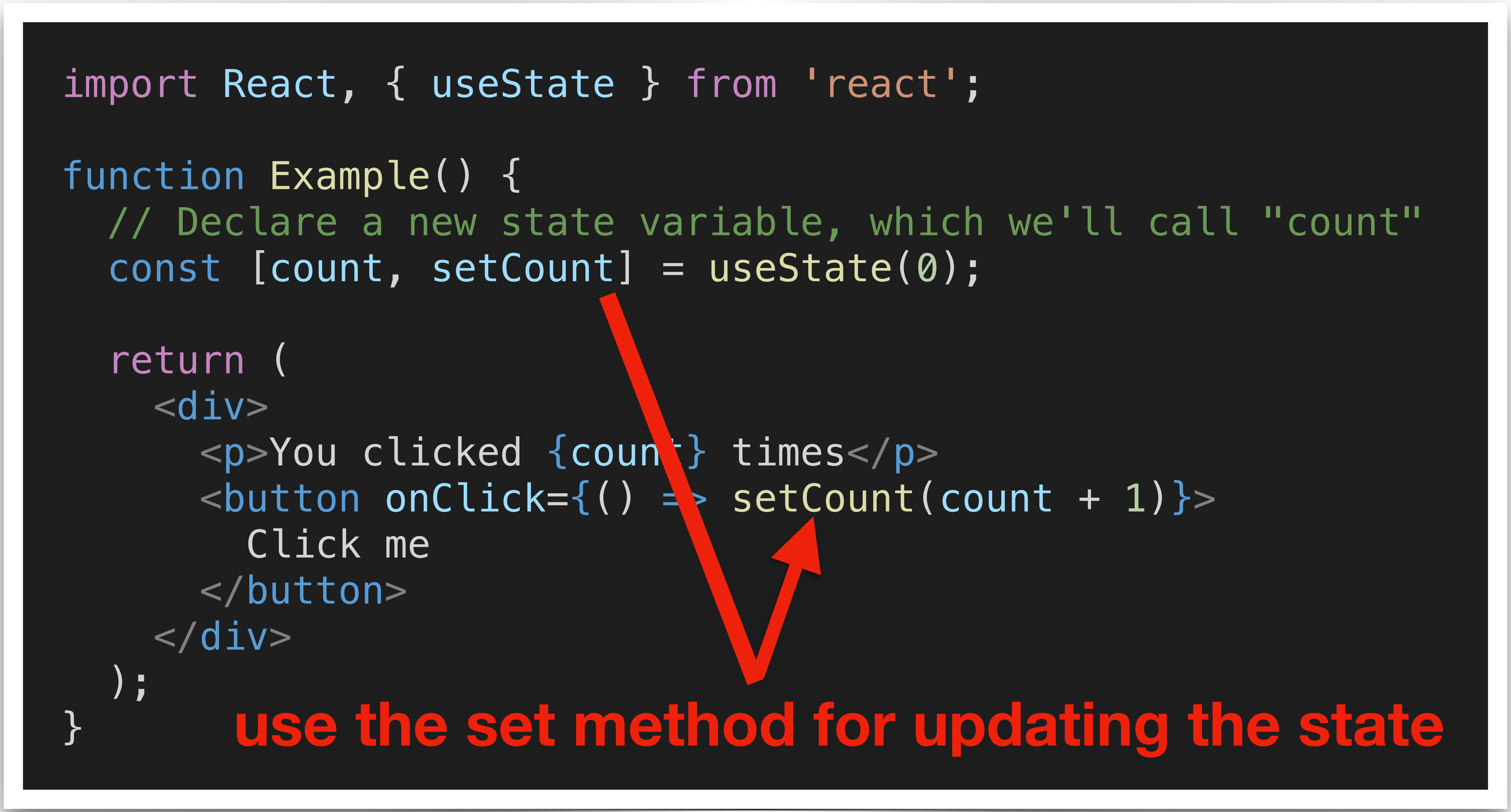


```
import React, { useState } from 'react';
```



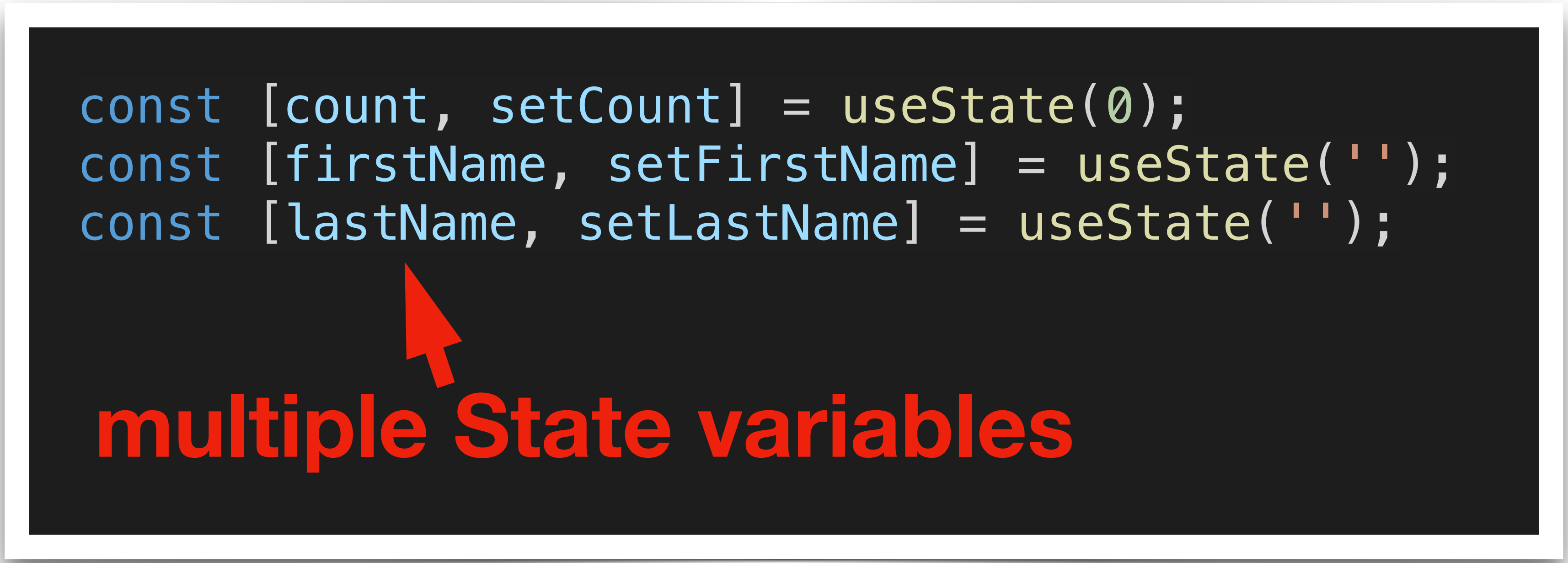
```
const [count, setCount] = useState(0);
```

Hooks use ***Destructering on arrays***, so the name of the variable and method doesn't matter => whatever makes sense.



```
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>
  );
}
```

Modifying the state variable directly doesn’t have any effect on the lifecycle of the component, doesn’t trigger a rerendering and should be avoided!



```
const [count, setCount] = useState(0);
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
```

- You can use the State Hook multiple times within a Function Component, for each state variable once.

- Make sure to apply the ***Rules of Hooks***!

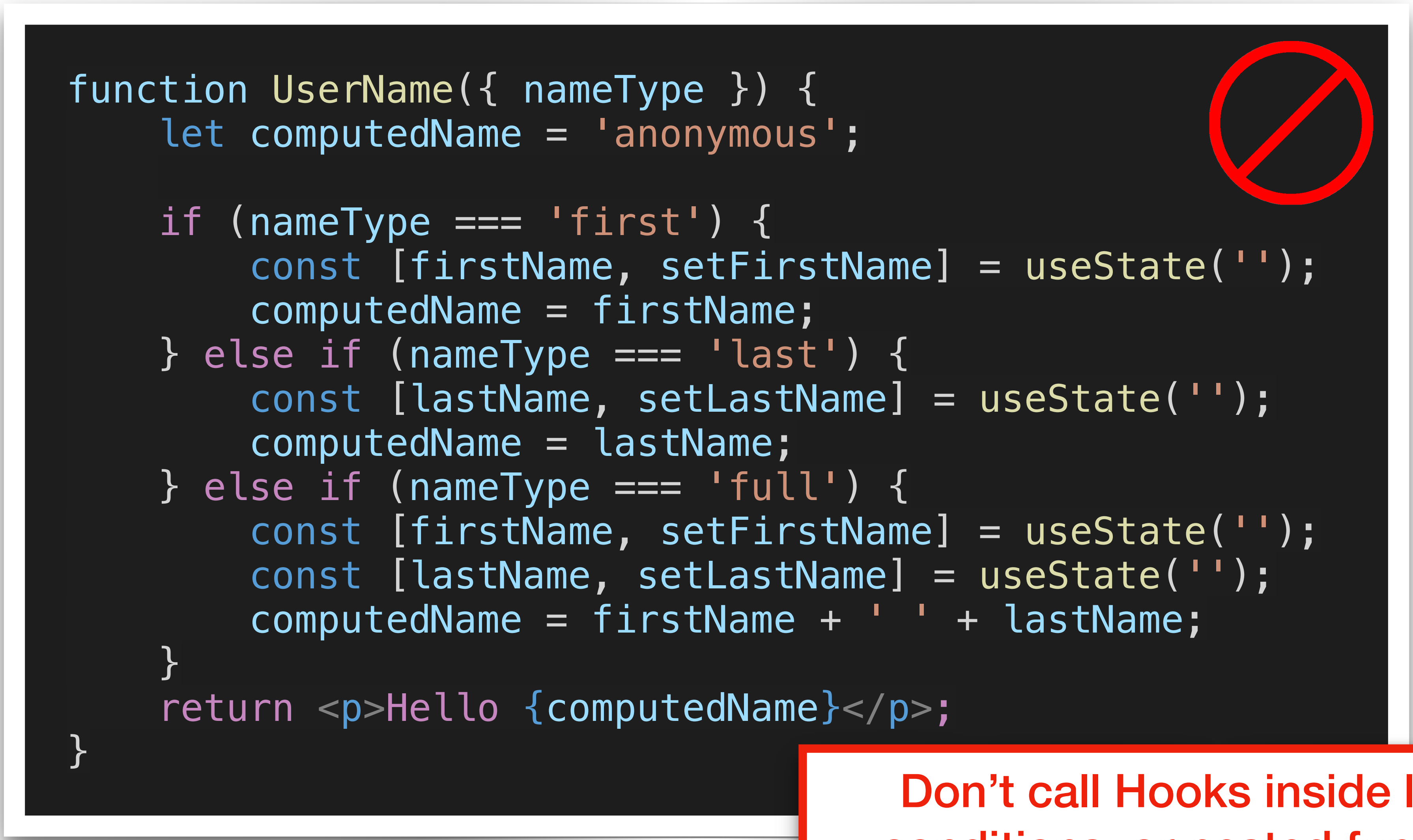
Rules of Hooks

„Hooks are JavaScript functions, but you need to follow two rules when using them.“

– *reactjs.org*

Rules of Hooks (1)

Only Call Hooks at the Top Level!



```
function UserName({ nameType }) {
    let computedName = 'anonymous';

    if (nameType === 'first') {
        const [firstName, setFirstName] = useState('');
        computedName = firstName;
    } else if (nameType === 'last') {
        const [lastName, setLastName] = useState('');
        computedName = lastName;
    } else if (nameType === 'full') {
        const [firstName, setFirstName] = useState('');
        const [lastName, setLastName] = useState('');
        computedName = firstName + ' ' + lastName;
    }
    return <p>Hello {computedName}</p>;
}
```

Don’t call Hooks inside loops, conditions, or nested functions.

Rules of Hooks (2)

Only Call Hooks from React Functions

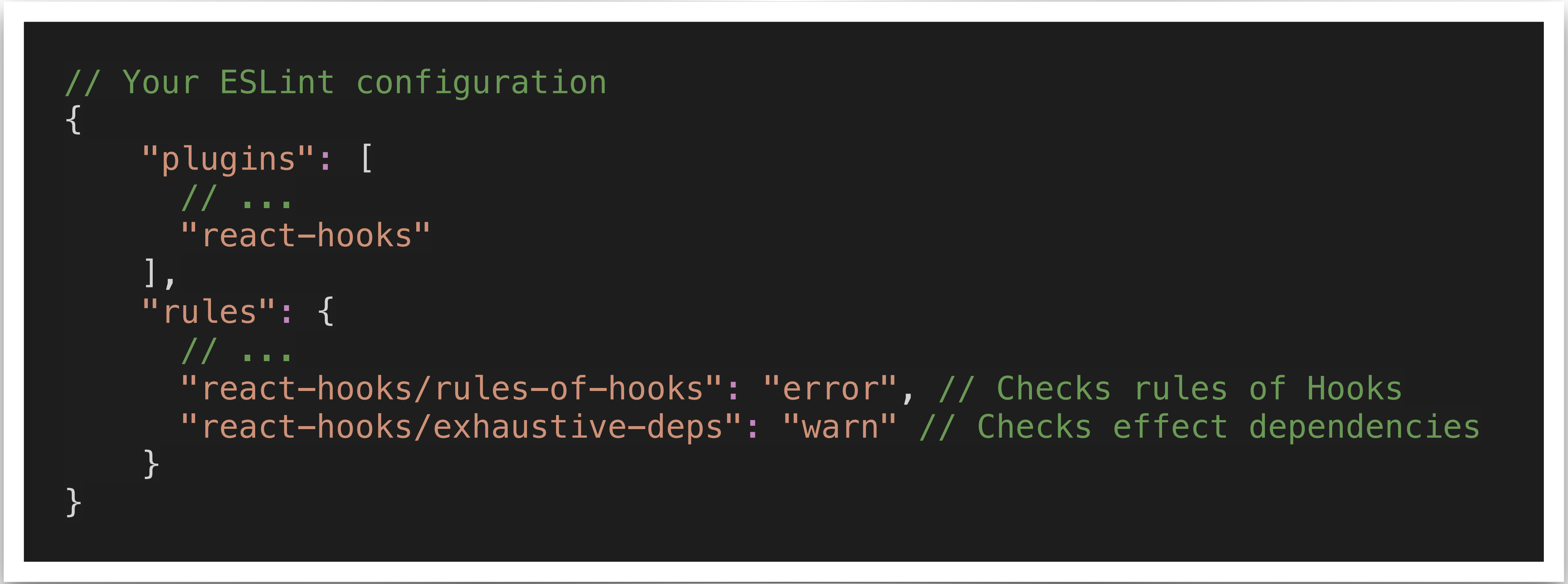
```
import { useState } from 'react';

function computeHelloString() {
    const [username] = useState('');
    return `Hello ${username}`;
}
```

Don't call Hooks from regular JavaScript functions!

Call Hooks from React function components.
Call Hooks from custom Hooks.

Rules of Hooks - ESLint Plugin

There is a nice ESLint Plugin for enforcing this two rules.



```
// Your ESLint configuration
{
    "plugins": [
      // ...
      "react-hooks"
    ],
    "rules": {
      // ...
      "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
      "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
    }
}
```

```
> npm install eslint-plugin-react-hooks —save-dev
```

Function Components (1)

Before Hooks we referred to Components implemented as a class to be ***Stateful Components*** and Components implemented as a function to be ***Stateless Components***.

This is no longer valid!

With Hooks Components implemented as a function can have state, therefor we should call them ***Function Components.***

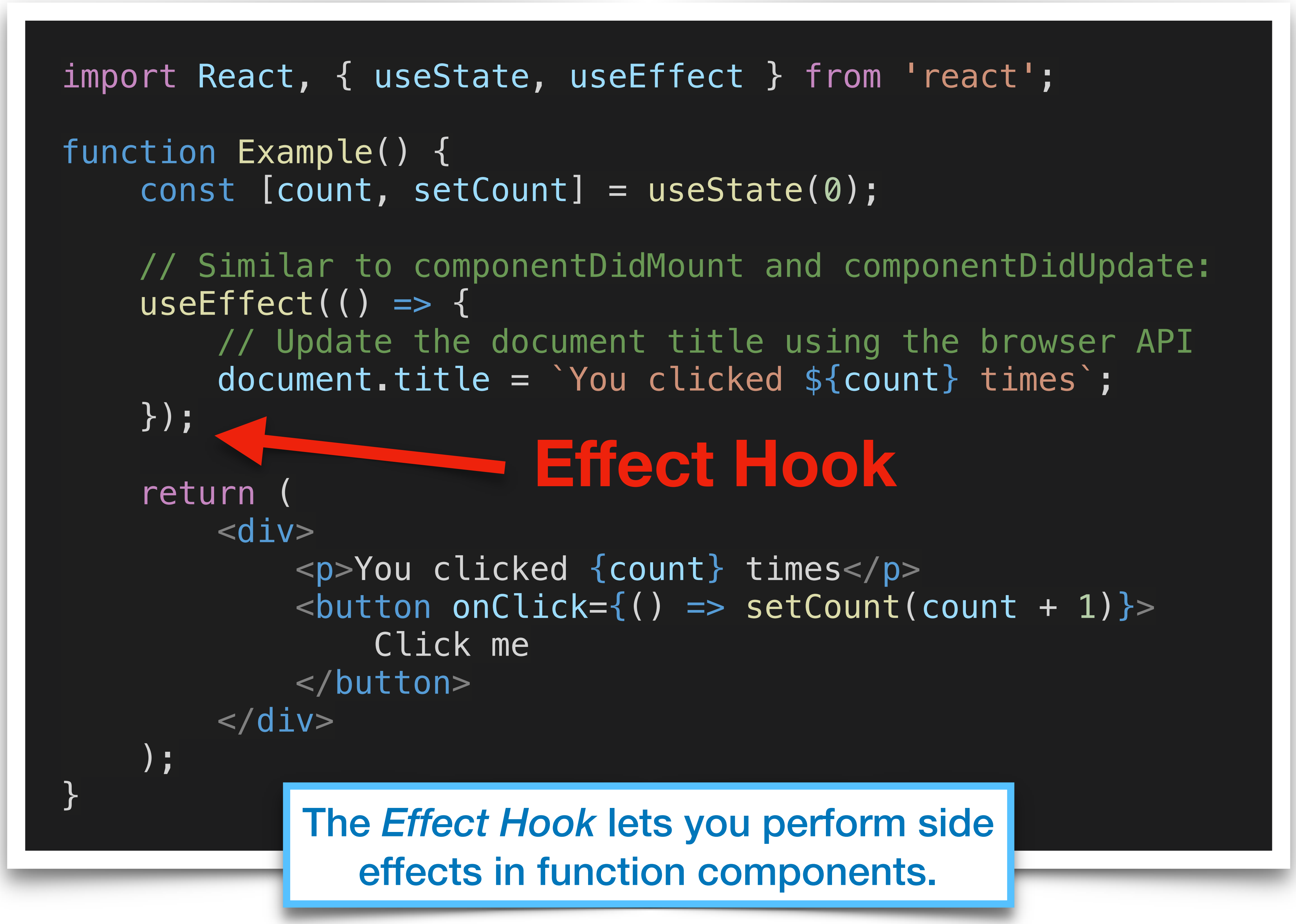
Function Components (2)

„Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's *render phase*).

Doing so will lead to confusing bugs and inconsistencies in the UI.“

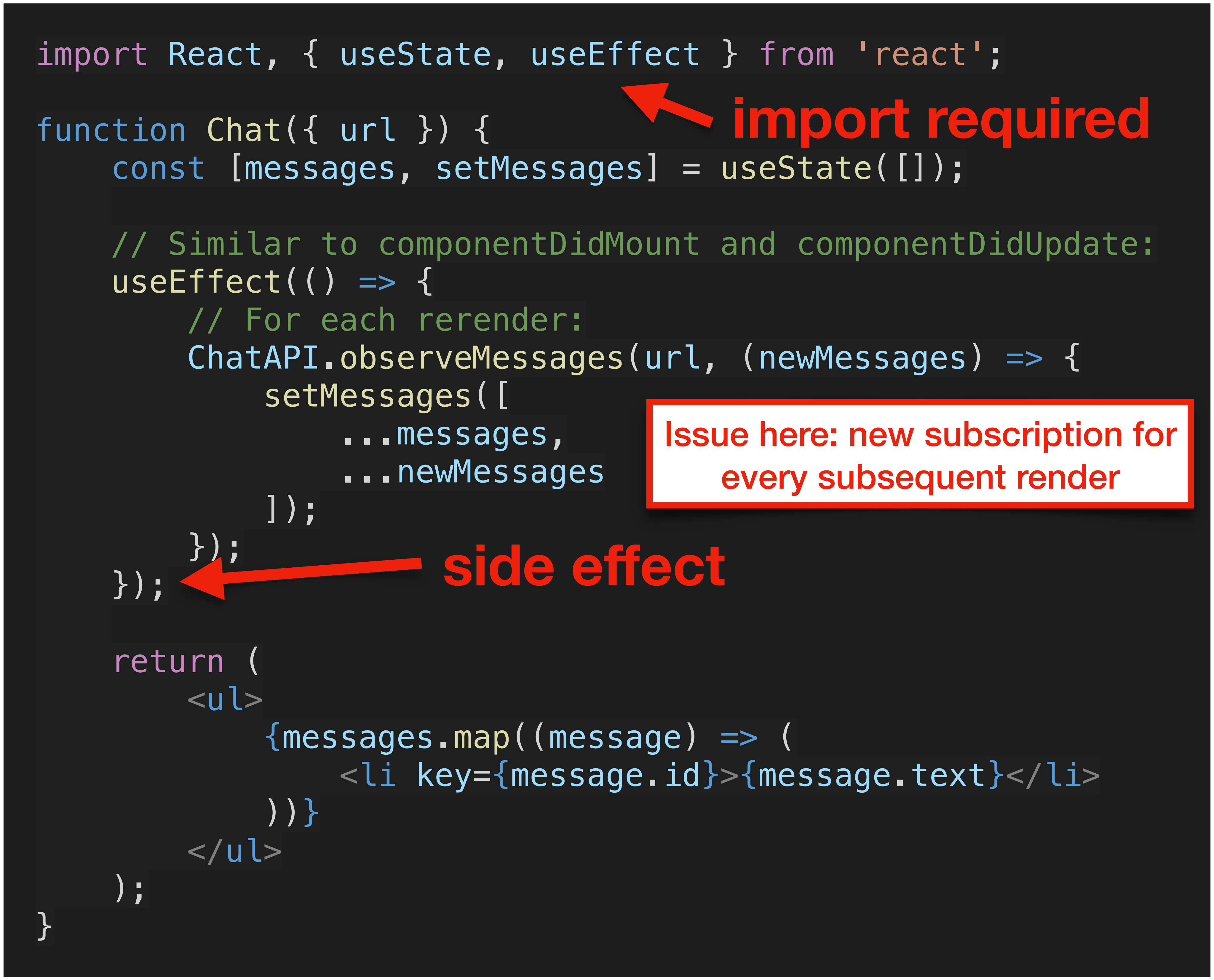
– *reactjs.org*

Effect Hook



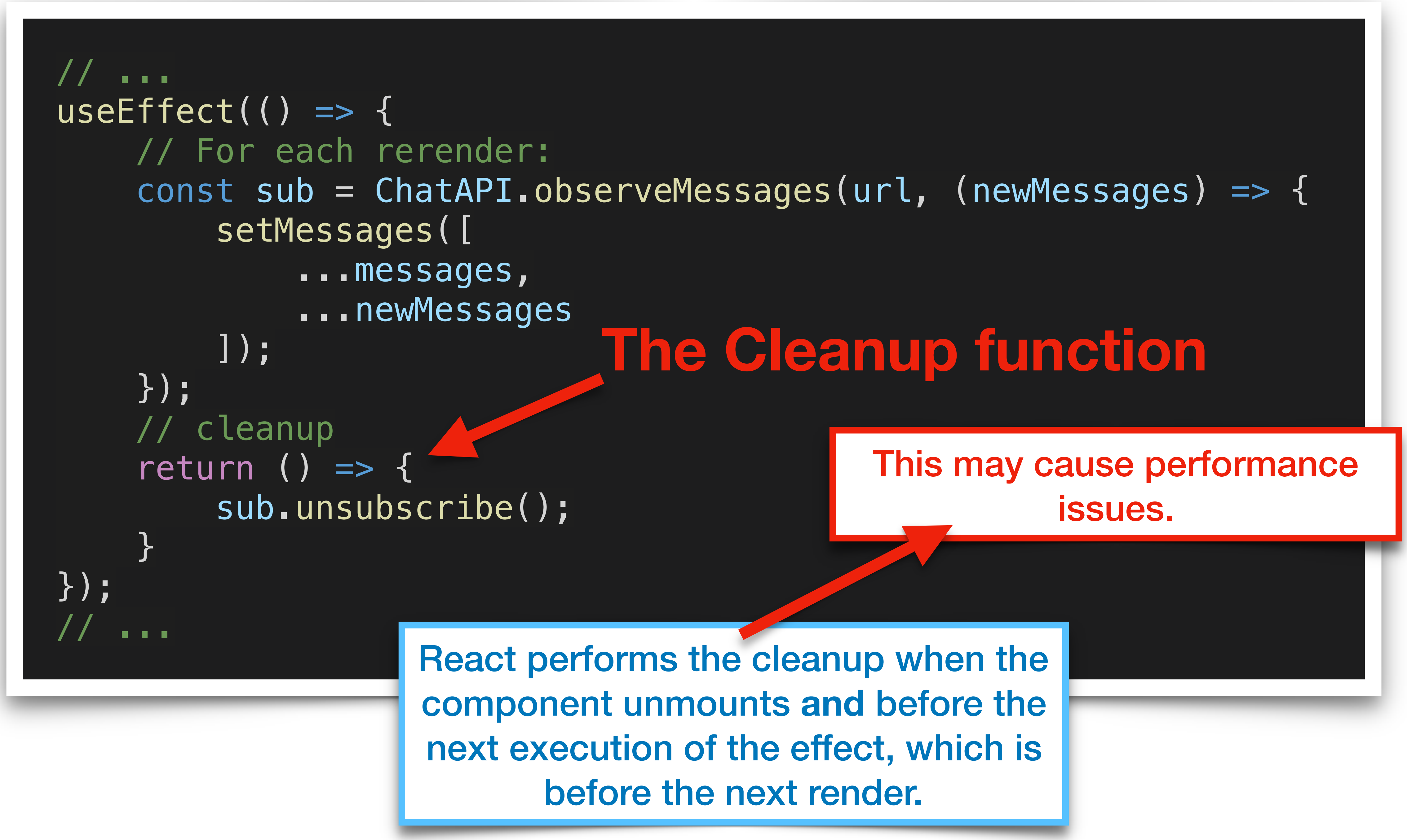
```
import React, { useState, useEffect } from 'react';

function Example() {
    const [count, setCount] = useState(0);

    // Similar to componentDidMount and componentDidUpdate:
    useEffect(() => {
        // Update the document title using the browser API
        document.title = `You clicked ${count} times`;
    });

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    );
}
```

The *Effect Hook* lets you perform side effects in function components.



```
import React, { useState, useEffect } from 'react';

function Chat({ url }) {
    const [messages, setMessages] = useState([]);

    // Similar to componentDidMount and componentDidUpdate:
    useEffect(() => {
        // For each rerender:
        ChatAPI.observeMessages(url, (newMessages) => {
            setMessages([
                ...messages,
                ...newMessages
            ]);
        });
    });

    return (
        <ul>
            {messages.map((message) => (
                <li key={message.id}>{message.text}</li>
            ))}
        </ul>
    );
}
```

Effect Cleanup



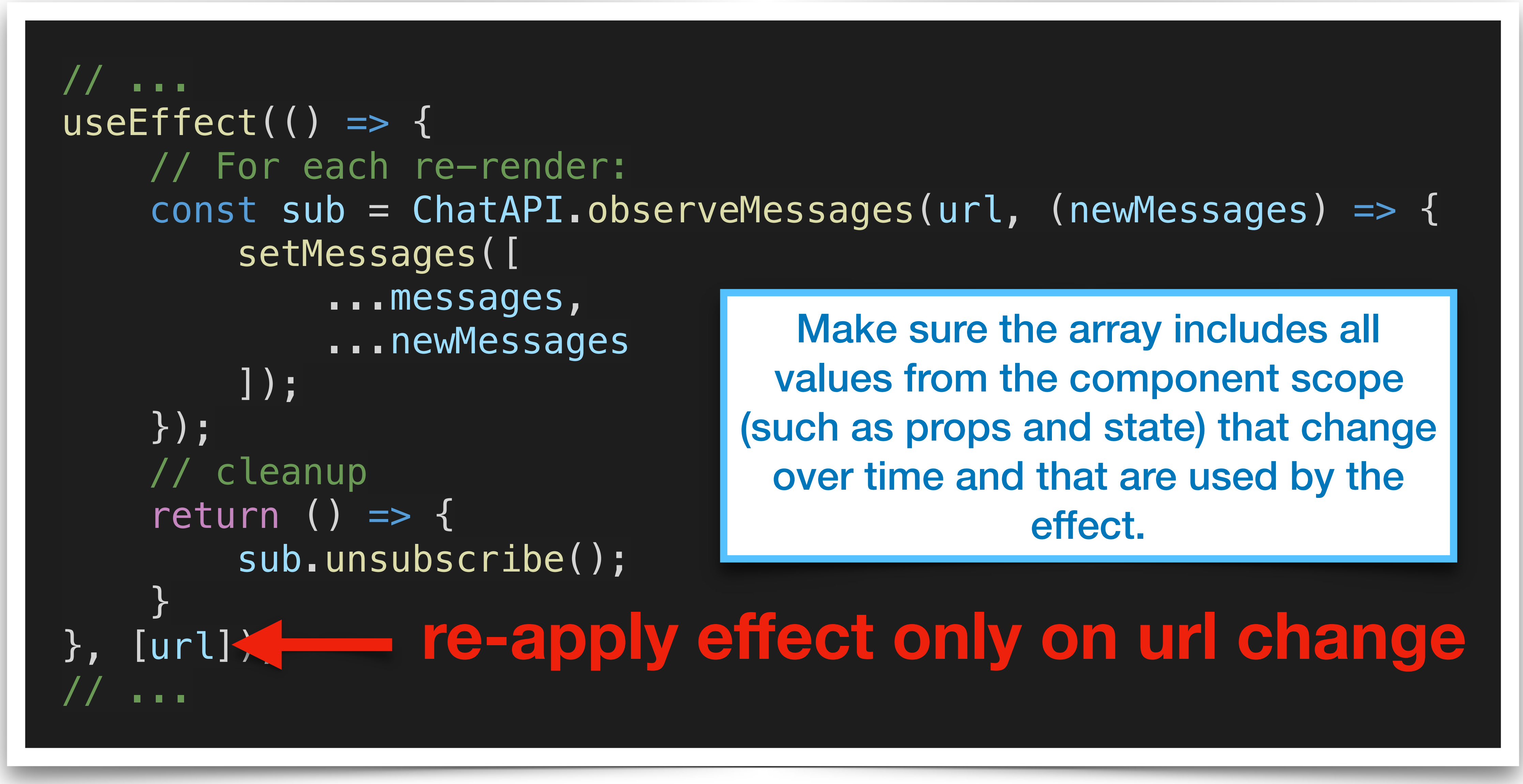
```
// ...
useEffect(() => {
    // For each rerender:
    const sub = ChatAPI.observeMessages(url, (newMessages) => {
        setMessages([
            ...messages,
            ...newMessages
        ]);
    });
    // cleanup
    return () => {
        sub.unsubscribe();
    }
});
// ...
```

The Cleanup function
This may cause performance issues.
React performs the cleanup when the component unmounts and before the next execution of the effect, which is before the next render.

Skipping Effects (1)

- In some cases, cleaning up or applying the effect after every render might create a performance problem.
- You can tell React to *skip* applying an effect if certain values haven't changed between re-renders.
- To do so, pass an array as an optional second argument to *useEffect*
- Before re-applying the effect, the passed array is checked for any change. If no members of the array changed, the effect is skipped.
- If you like to have the effect to be applied only once (while mounting, cleaned up while unmounting): pass an empty array.

Skipping Effects (2)

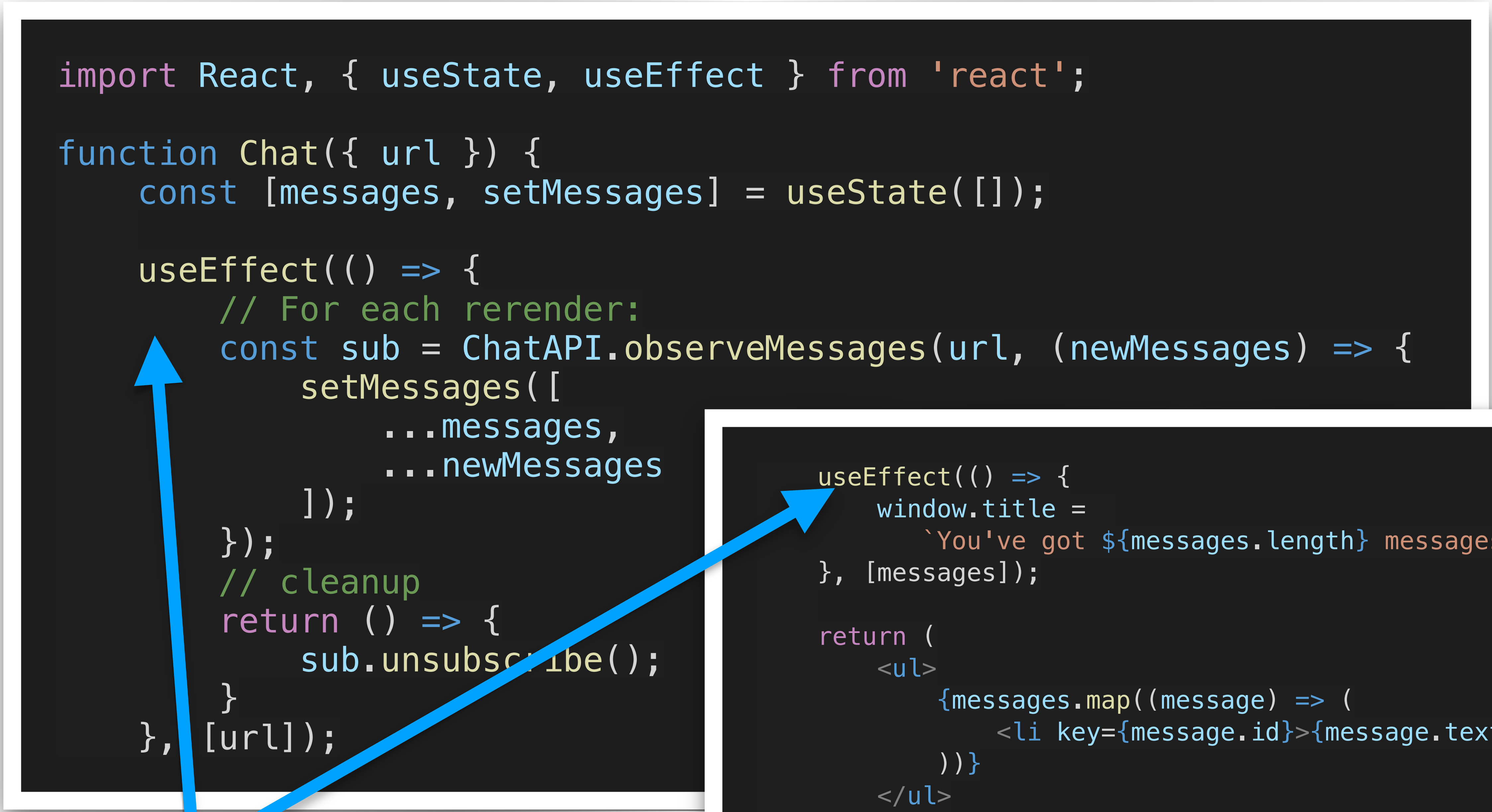


```
// ...
useEffect(() => {
    // For each re-render:
    const sub = ChatAPI.observeMessages(url, (newMessages) => {
        setMessages([
            ...messages,
            ...newMessages
        ]);
    });
    // cleanup
    return () => {
        sub.unsubscribe();
    }
}, [url]);
// ...
```

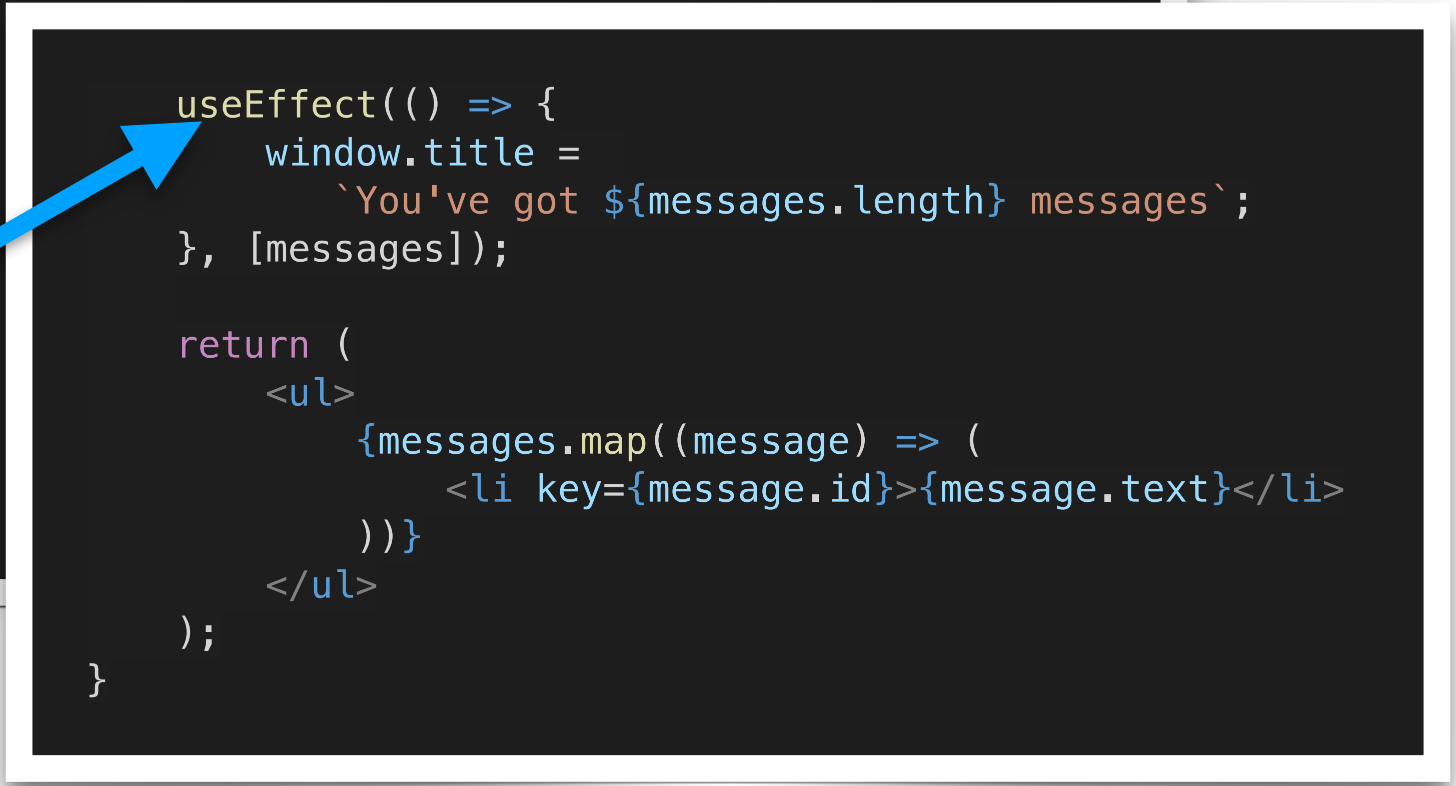
Make sure the array includes all values from the component scope (such as props and state) that change over time and that are used by the effect.

re-apply effect only on url change

Complete Example: Effects



```
import React, { useState, useEffect } from 'react';

function Chat({ url }) {
    const [messages, setMessages] = useState([]);

    useEffect(() => {
        // For each rerender:
        const sub = ChatAPI.observeMessages(url, (newMessages) => {
            setMessages([
                ...messages,
                ...newMessages
            ]);
        });
        // cleanup
        return () => {
            sub.unsubscribe();
        }
    }, [url]);
```



```
    useEffect(() => {
        window.title =
            `You've got ${messages.length} messages`;
    }, [messages]);

    return (
        <ul>
            {messages.map((message) => (
                <li key={message.id}>{message.text}</li>
            ))}
        </ul>
    );
}
```

Great separation of concerns.

HTML5 Forms with Controlled Components

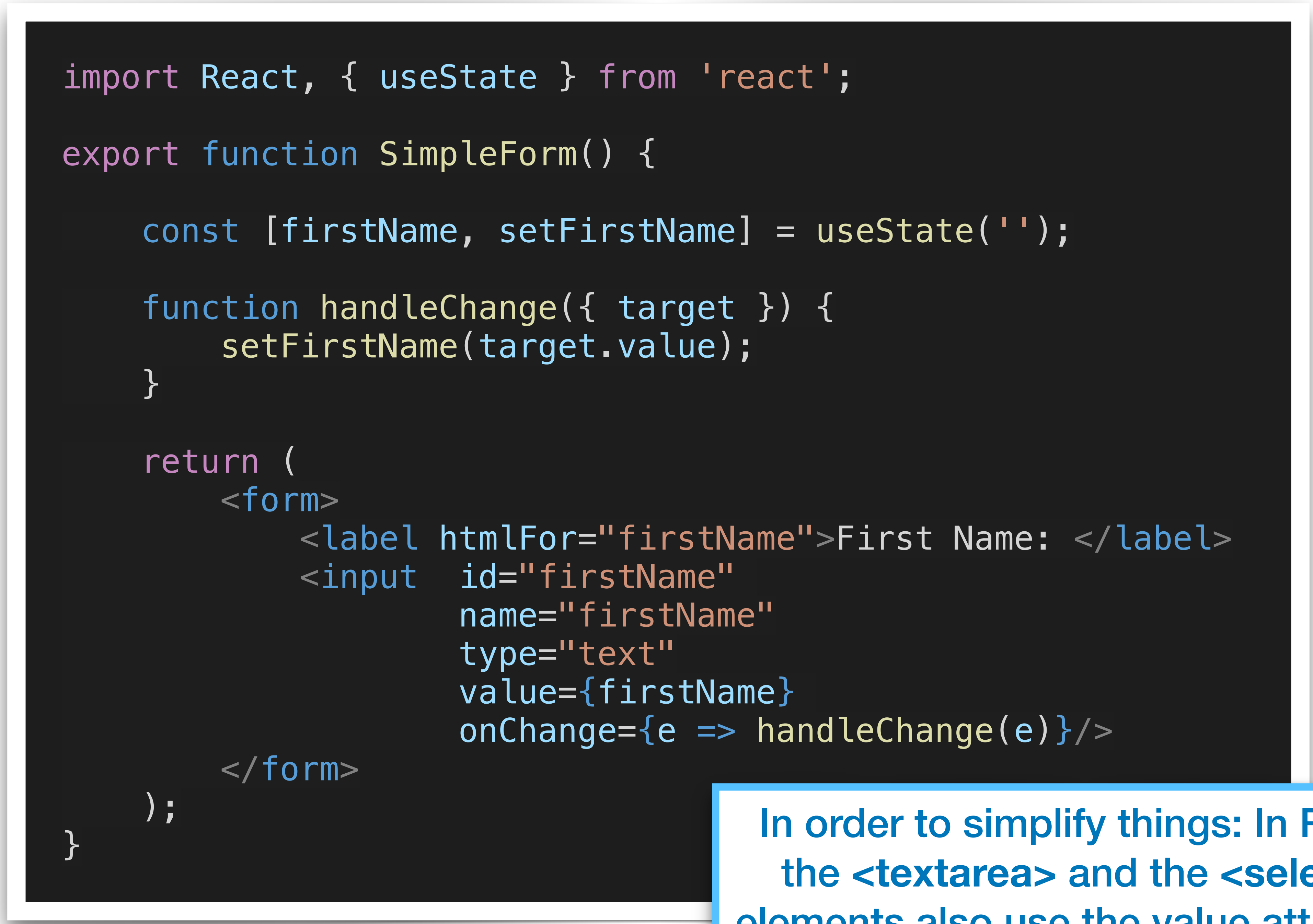
Controlled Components

- In HTML, form elements such as **<input>**, **<textarea>**, and **<select>** typically maintain their own state and update it based on user input.

- In React, mutable state is typically kept in the *state* of components, and only updated with *setState()*.

- In Controlled Components we make the components state the *„single source of truth“.*

Controlled Components



```
import React, { useState } from 'react';

export function SimpleForm() {

    const [firstName, setFirstName] = useState('');

    function handleChange({ target }) {
        setFirstName(target.value);
    }

    return (
        <form>
            <label htmlFor="firstName">First Name: </label>
            <input  id="firstName"
                    name="firstName"
                    type="text"
                    value={firstName}
                    onChange={e => handleChange(e)}/>
        </form>
    );
}
```

In order to simplify things: In React the **<textarea>** and the **<select>** elements also use the value attribute.

Form Validation

Let the user guide through your application.

Show them what they are doing wrong and how to fix it as soon as possible!

- State of the Art

Form Validation - Strategies

There are multiple ways to achieve Form Validation (Client-side):

- **Built-in form validation**

  Uses HTML5 form validation features.

- **JavaScript - The *constraint validation API***

  More and more browsers now support the constraint validation API, and it's becoming reliable.

- **JavaScript - Custom Implementation**

  Sometimes the constraint validation API is not enough.

HTML5 Built-in Validators

In HTML5 there are built-in *Validators* that can be used with the *Built-in form validation* and *constraint validation API:*

- **type**

  The ***type*** attribute of an input is also a validator. Always use the correct one,

  E.x: ***email***, ***number***, ***color***, ***date***, ***datetime-local***, ***month***, ***number***, ***range***, ***password***, ***text***, …

- **required**

  A value is required

- **minlength, maxlength**

  The minimal or maximal length of the input value

- **pattern**

  The input value has to match the given regular expression.

Built-in form validation

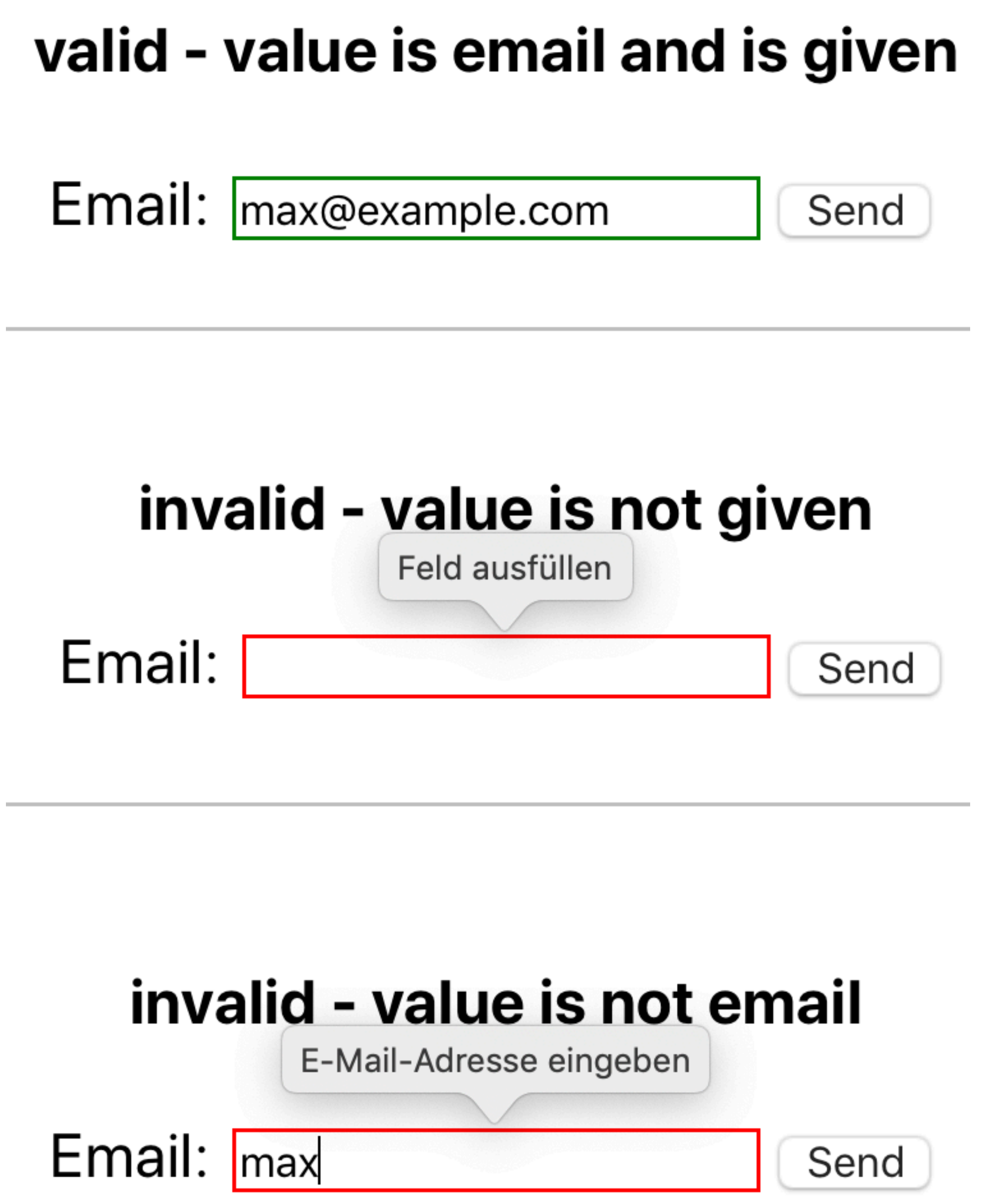
```jsx
import React, { useState } from 'react';
import './SimpleForm.css';

export function SimpleForm() {

    const [email, setEmail] = useState('');

    function handleChange({ target }) {
        setEmail(target.value);
    }

    function sendForm(e) {
        // only on valid form!!!
    }

    return (
        <form onSubmit={sendForm}>
            <label htmlFor="userEmail">Email: </label>
            <input  id="userEmail"
                    name="userEmail"
                    type="email"
                    required
                    value={email}
                    onChange={e => handleChange(e)}/>
            <button>Send</button>
        </form>
    );
}
```

```css
input {
    outline: none;
}

input:valid {
    border: 1px solid green;
}

input:invalid {
    border: 1px solid red;
}
```

type **validator**

required **validator**

Built-in form validation



valid - value is email and is given

Email: max@example.com Send

invalid - value is not given

Feld ausfüllen

Email: Send

invalid - value is not email

E-Mail-Adresse eingeben

Email: max Send

Built-in form validation

The Built-in form validation has some **disadvantages**:

- **No immediate user guidance**

  Error messages are shown on form submit.

  There is no way to show immediately an input ***hint, error*** or ***success message*** depending on the inputs validity and/ or touched state - while the user is typing.

- **The error messages are pre-styled and pre-defined**

  We'd like to show a custom message with our custom design and behavior

- **No Cross-field validation**

  Validation happens on input element basis.

Form validation - our strategy

Let us define an own validation strategy:

- **Each input is in the state: (un-)touched and (in-)valid**

  Show a ***hint message*** if the input is in state ***untouched,*** an ***error message*** if the input is in state ***invalid*** or a ***success message*** if the input is in state ***valid***.

  Show the input in a ***custom style (css)*** for the states ***valid*** and ***invalid***.

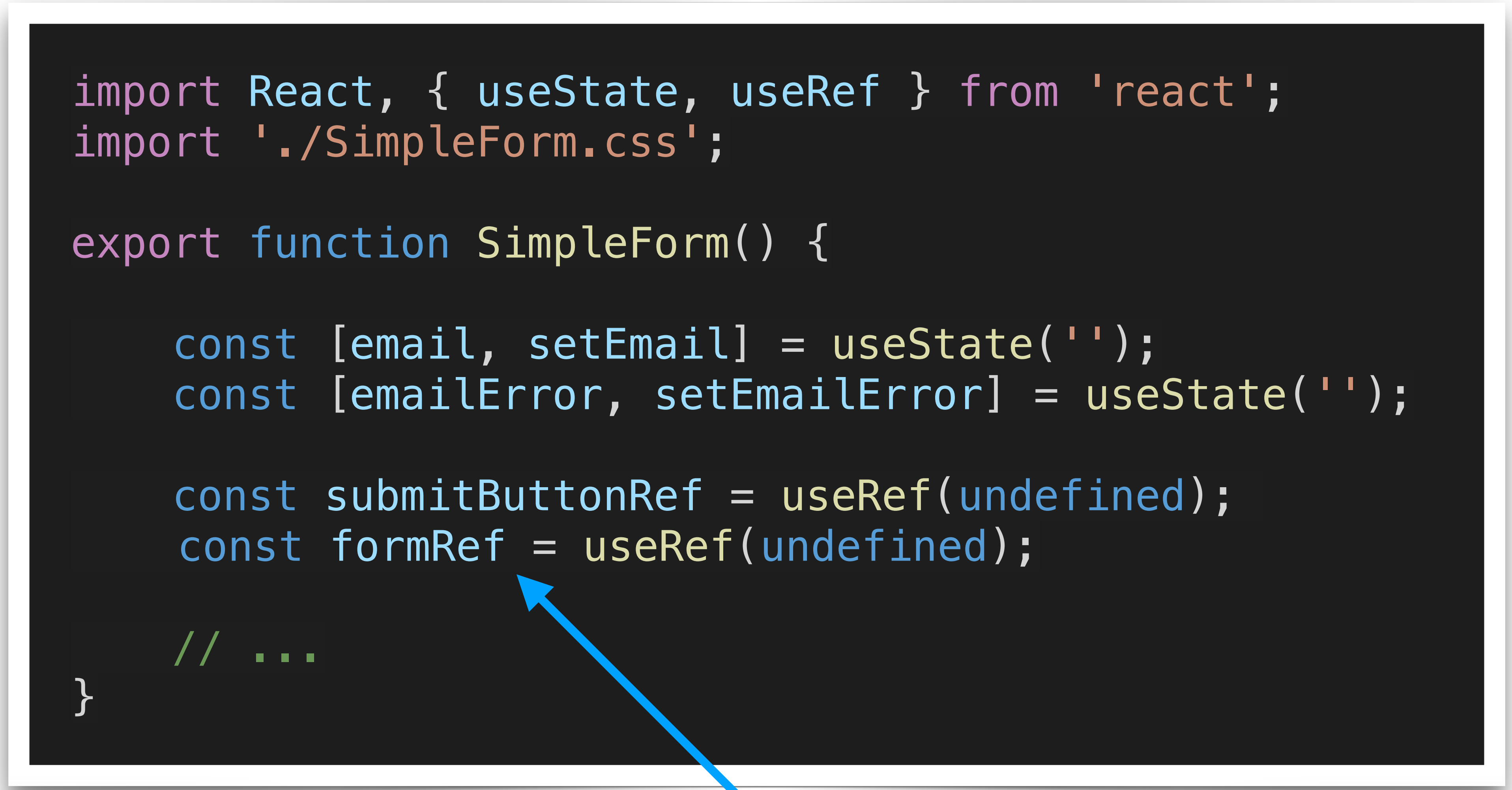
- **The error message depends on the kind of the error.**

  Depending on the actual error a custom error message is shown.

- **Hide each submit trigger (submit button) if the form is in an invalid state.**

  If ***any*** of the inputs of a form ***is invalid*** than the complete ***form is invalid***.

Form validation - our strategy

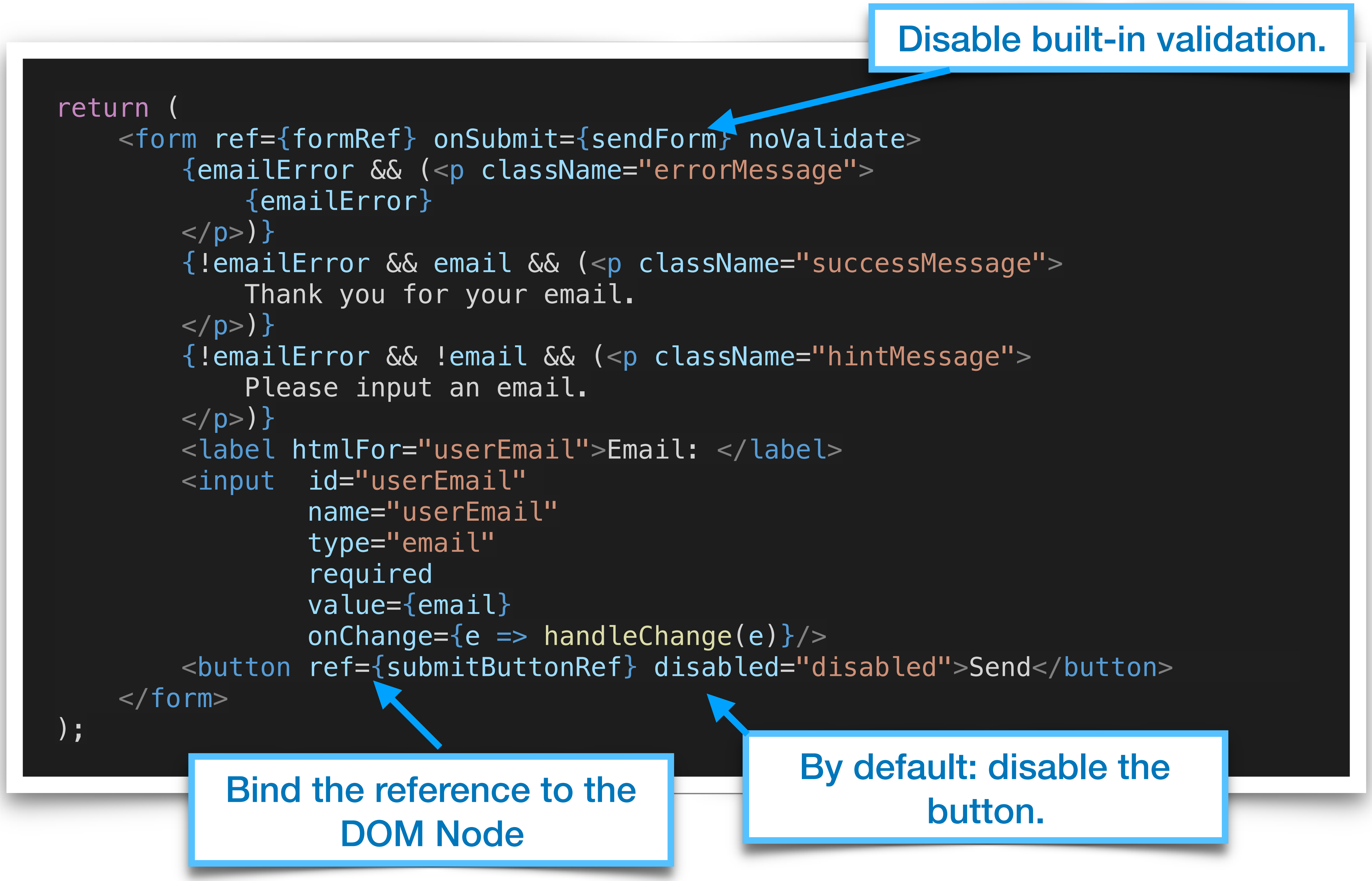


```
import React, { useState, useRef } from 'react';
import './SimpleForm.css';

export function SimpleForm() {

    const [email, setEmail] = useState('');
    const [emailError, setEmailError] = useState('');

    const submitButtonRef = useRef(undefined);
    const formRef = useRef(undefined);

    // ...
}
```

A ***reference*** can be bound directly to a DOM Node.

We use this (later in code) to set the buttons ***disabled*** attribute directly and to check the forms ***validity*** state.

Form validation - our strategy

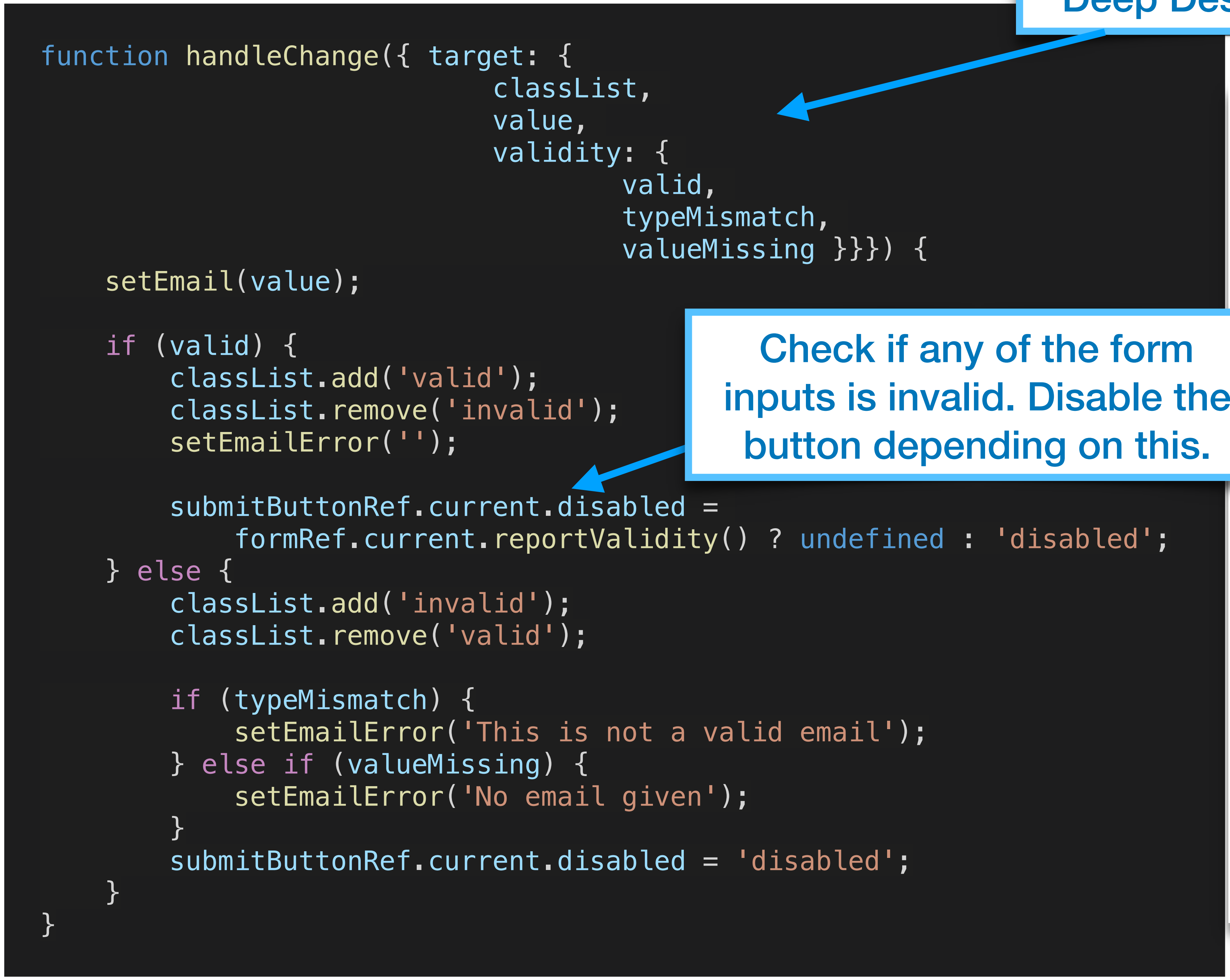


```
return (
    <form ref={formRef} onSubmit={sendForm} noValidate>
        {emailError && (<p className="errorMessage">
            {emailError}
        </p>)}
        {!emailError && email && (<p className="successMessage">
            Thank you for your email.
        </p>)}
        {!emailError && !email && (<p className="hintMessage">
            Please input an email.
        </p>)}
        <label htmlFor="userEmail">Email: </label>
        <input  id="userEmail"
                name="userEmail"
                type="email"
                required
                value={email}
                onChange={e => handleChange(e)}/>
        <button ref={submitButtonRef} disabled="disabled">Send</button>
    </form>
);
```

Disable built-in validation.
Bind the reference to the DOM Node
By default: disable the button.

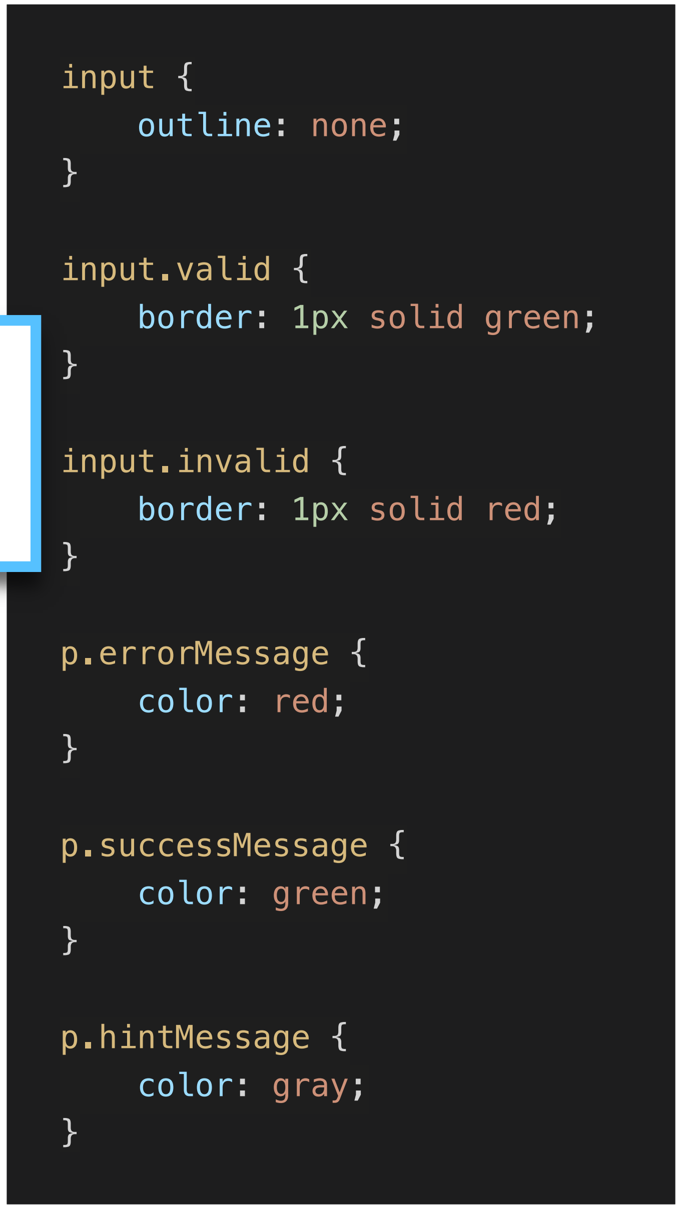
Form validation - our strategy

Deep Destructering



```
function handleChange({ target: {
                            classList,
                            value,
                            validity: {
                                valid,
                                typeMismatch,
                                valueMissing }}}) {
    setEmail(value);

    if (valid) {
        classList.add('valid');
        classList.remove('invalid');
        setEmailError('');

        submitButtonRef.current.disabled =
            formRef.current.reportValidity() ? undefined : 'disabled';
    } else {
        classList.add('invalid');
        classList.remove('valid');

        if (typeMismatch) {
            setEmailError('This is not a valid email');
        } else if (valueMissing) {
            setEmailError('No email given');
        }
        submitButtonRef.current.disabled = 'disabled';
    }
}
```

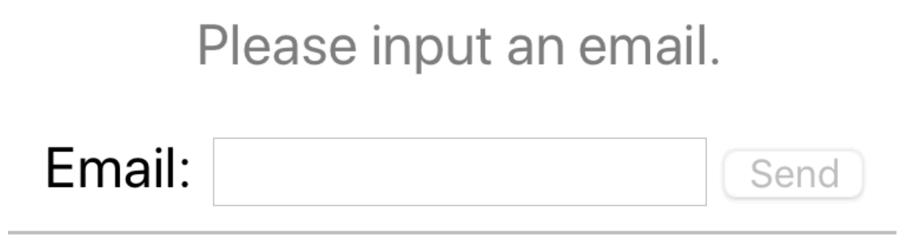
Check if any of the form inputs is invalid. Disable the button depending on this.



```
input {
    outline: none;
}

input.valid {
    border: 1px solid green;
}

input.invalid {
    border: 1px solid red;
}

p.errorMessage {
    color: red;
}

p.successMessage {
    color: green;
}

p.hintMessage {
    color: gray;
}
```

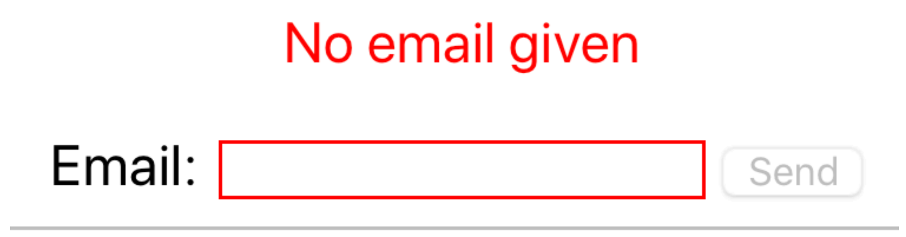
Form validation - our strategy

valid - field is untouched


Please input an email.

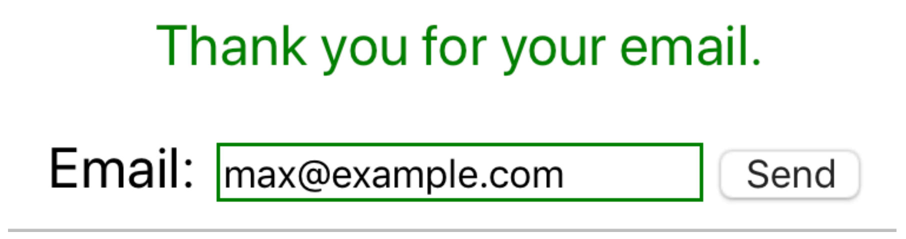
Email:

invalid - value is not given


No email given

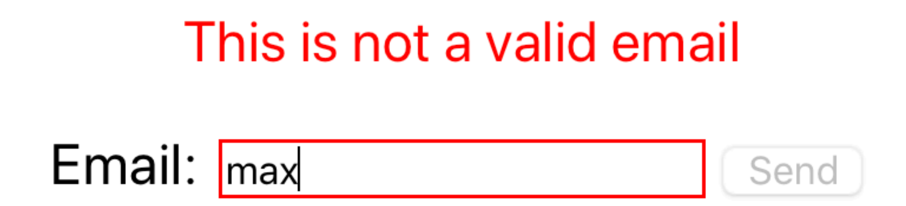
Email:

valid - value is email and is given


Thank you for your email.

Email: max@example.com

invalid - value is not email


This is not a valid email

Email: max

Testing

Why Testing? (1)

- **Does a Module behave as expected?**

  Think in ***Contracts***:

  What is the ***Invariant*** of the Module?

  What are the ***pre-*** and ***post-conditions***?

- **Does the (Sub-) System behave as expected?**

  Same as above.

- **During the lifespan of the App: Any *Regressions*?**

  Things might change over time…

Why Testing? (2)

- **Is the App what the User expects?**

  Does the App meet the expectations of your Apps Users?

- **Is the App what we (or our client) expect?**

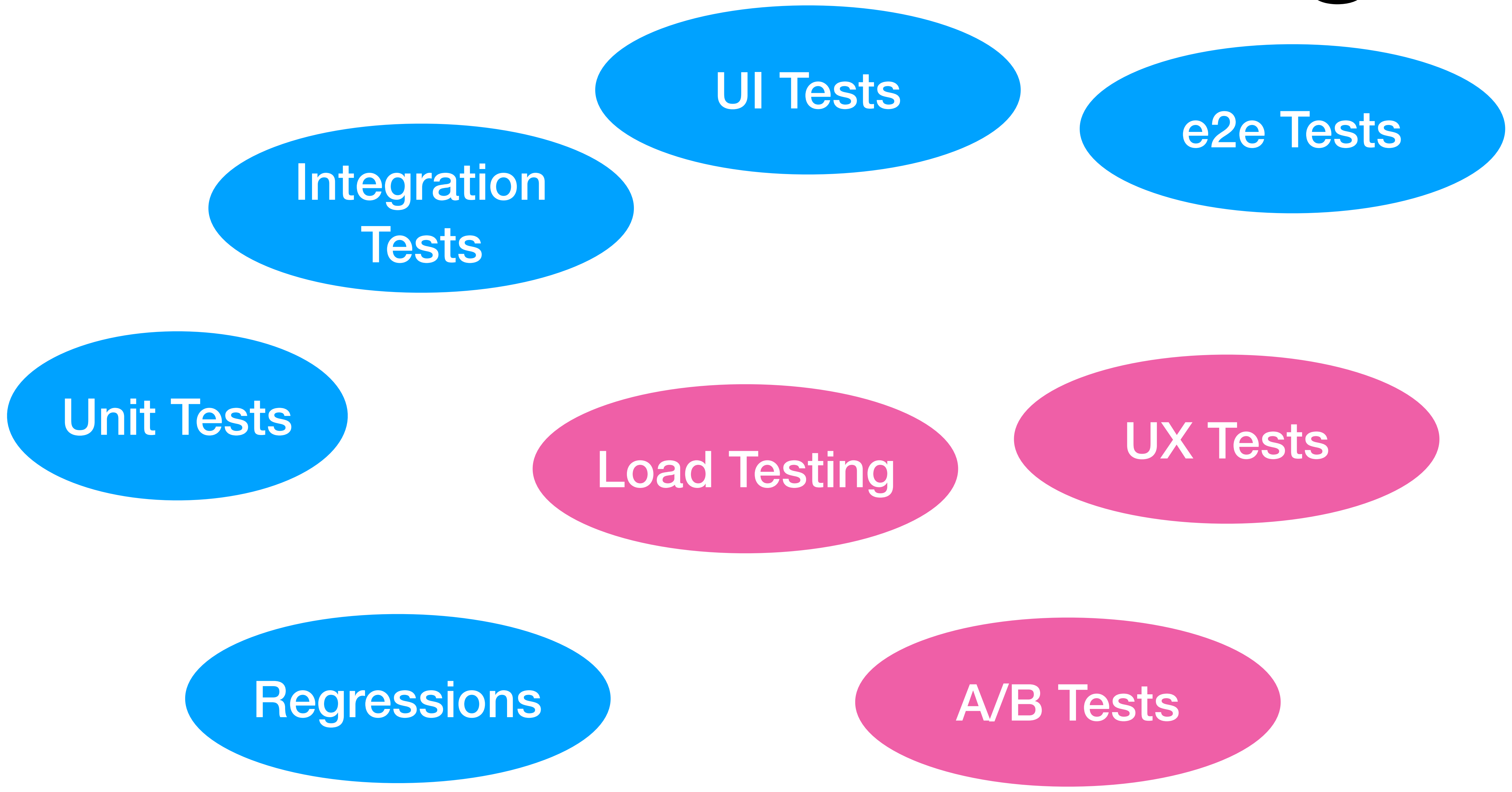
  Have we meet the ***requirements?***

  Are there better ways to do things?

- **Does our App perform well in a real world scenario?**

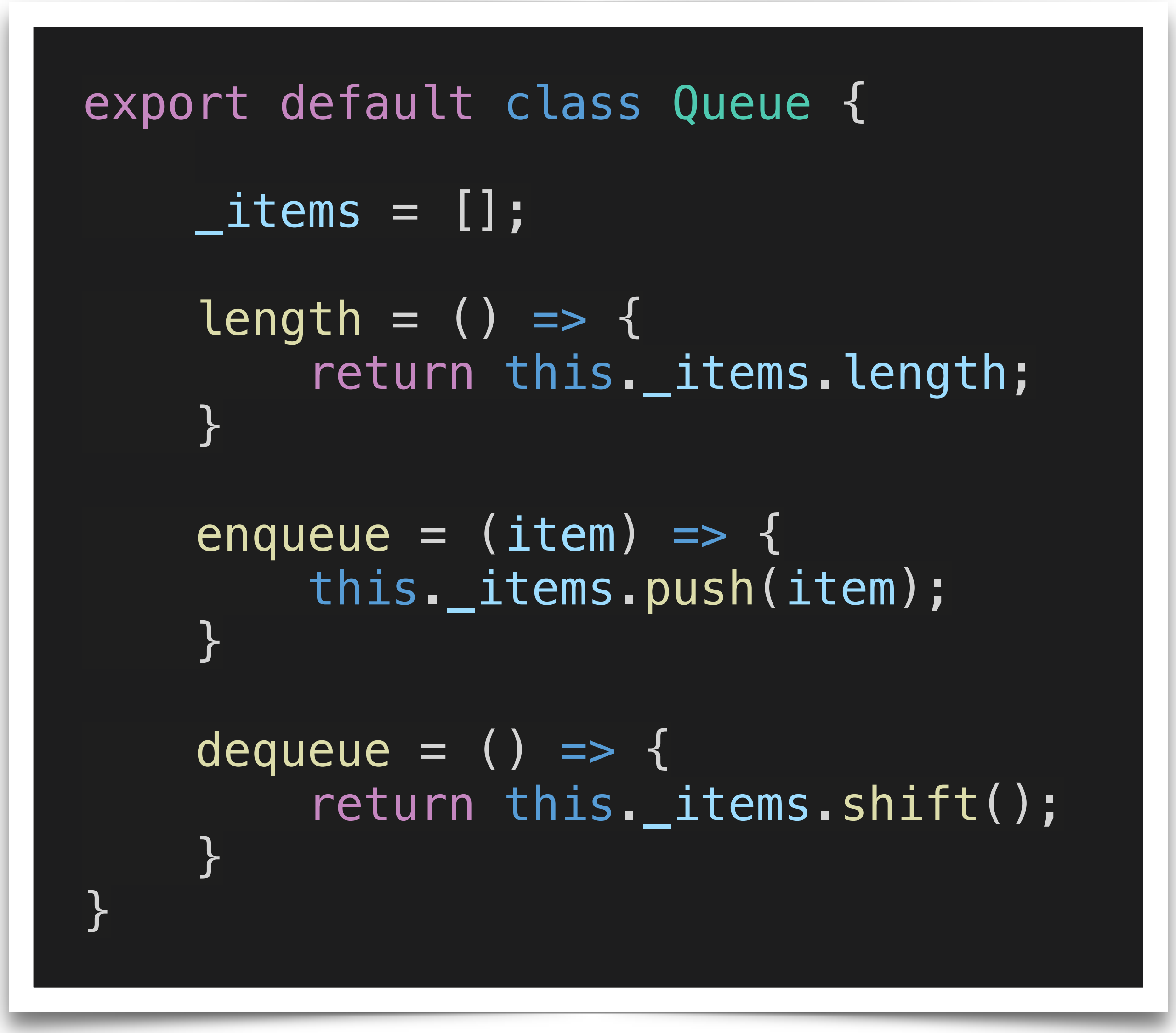
  Quite often the answer is: ***NO***.

The World of Testing



🟦 **We focus on these.** 🟪 **Not covered here.**

Think in Contracts



```
export default class Queue {

    _items = [];

    length = () => {
        return this._items.length;
    }

    enqueue = (item) => {
        this._items.push(item);
    }

    dequeue = () => {
        return this._items.shift();
    }
}
```

Invariant:
q.length() >= 0

Pre: l1 := q.length()
q.enqueue(a)
Post: l1 + 1 = q.length()

Pre: l1 := q.length() and l1 > 0
r := q.dequeue()
Post: r != undefined and l1 = q.length() + 1

Pre: l1 := q.length() and l1 = 0
r := q.dequeue()
Post: r = undefined and l1 = q.length()

That's all? ***NO!***

Is the App Correct?

If all tests pass, your App still might be incorrect.

You've probably forgotten some test cases.

- State of the Art

Black-Box Testing

While defining the Contract for your Unit,
consider only its public api.

Design a solid api for your Unit.

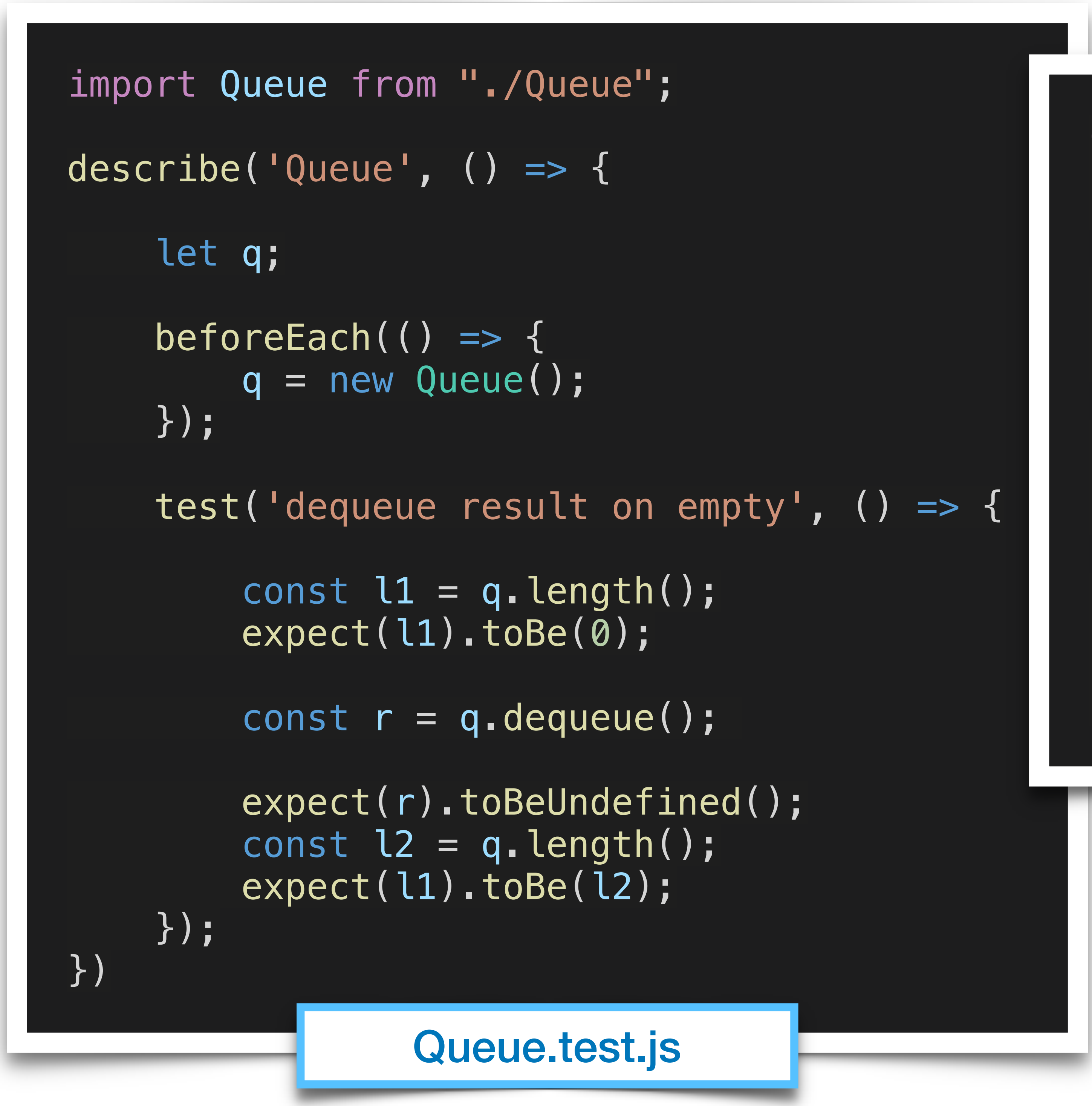
- State of the Art

Jest

„Jest is a delightful JavaScript Testing Framework with a focus on simplicity."

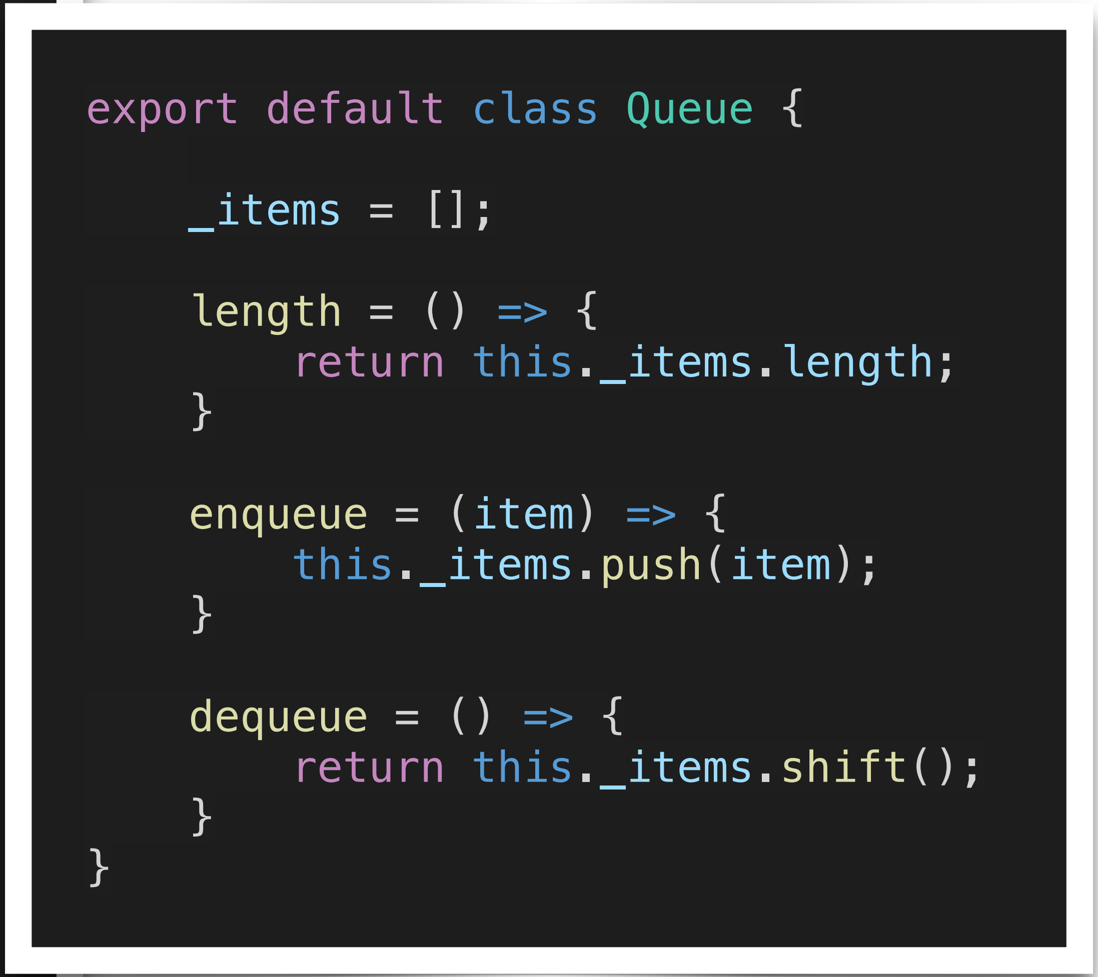
– *jestjs.io*

Jest - Example



```
import Queue from "./Queue";

describe('Queue', () => {

    let q;

    beforeEach(() => {
        q = new Queue();
    });

    test('dequeue result on empty', () => {

        const l1 = q.length();
        expect(l1).toBe(0);

        const r = q.dequeue();

        expect(r).toBeUndefined();
        const l2 = q.length();
        expect(l1).toBe(l2);
    });
})
```

Queue.test.js



```
export default class Queue {

    _items = [];

    length = () => {
        return this._items.length;
    }

    enqueue = (item) => {
        this._items.push(item);
    }

    dequeue = () => {
        return this._items.shift();
    }
}
```

Pre: **l1 := q.length() and l1 = 0**

r := q.dequeue()

Post: **r = undefined and l1 = q.length()**

Common Matchers

- **.toBe(value)**

  Test exact equality, usually on primitives or on object references.

- **.toEqual(object)**

  Recursively checks every field of an object or array.

- **.not.toBe(value) / .not.toEqual(object) / …**

  You can also test for the opposite of a matcher.

Truthiness Matchers

- **.toBeNull(value)**

  Matches only *null*.

- **.toBeUndefined(v)/ .toBeDefined(v)**

  Matches only *undefined* / *defined*.

- **.toBeTruthy(v) / .toBeFalsy(v)**

  Matches anything that an *if* statement treats as *true* / *false.*

Number Matchers

- **.toBeGreaterThan(n)**
- **.toBeGreaterThanOrEqual(n)**
- **.toBeLessThan(n)**
- **.toBeLessThanOrEqual(n)**
- **.toBeCloseTo(0.1)**

  For floating point equality, because you don't want a test to depend on a tiny rounding error.

String Matchers

- **.toMatch(/regex/)**

  Check strings against regular expressions.

Arrays and iterables Matchers

- **.toContain(item)**

  Check if an array or iterable contains a particular item.

Exception Matchers

- **expect(fn).toThrow(Exception)**

  Test that a particular function throws an error when it's called.

Testing Asynchronous Code

When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test.

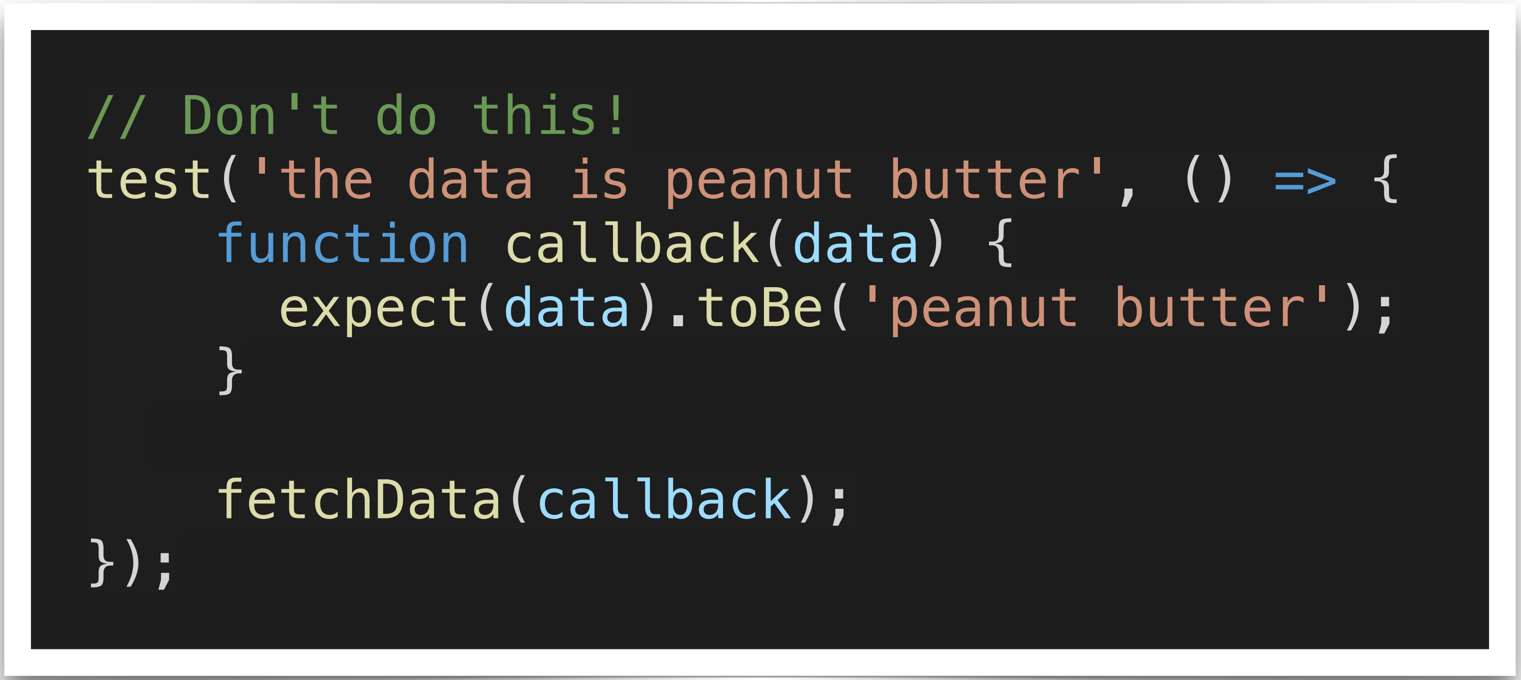
Jest has several ways to handle this.

– *jestjs.io*

Testing Callbacks

By default, Jest tests complete once they reach the end of their execution.

That means this test will *not* work as intended.

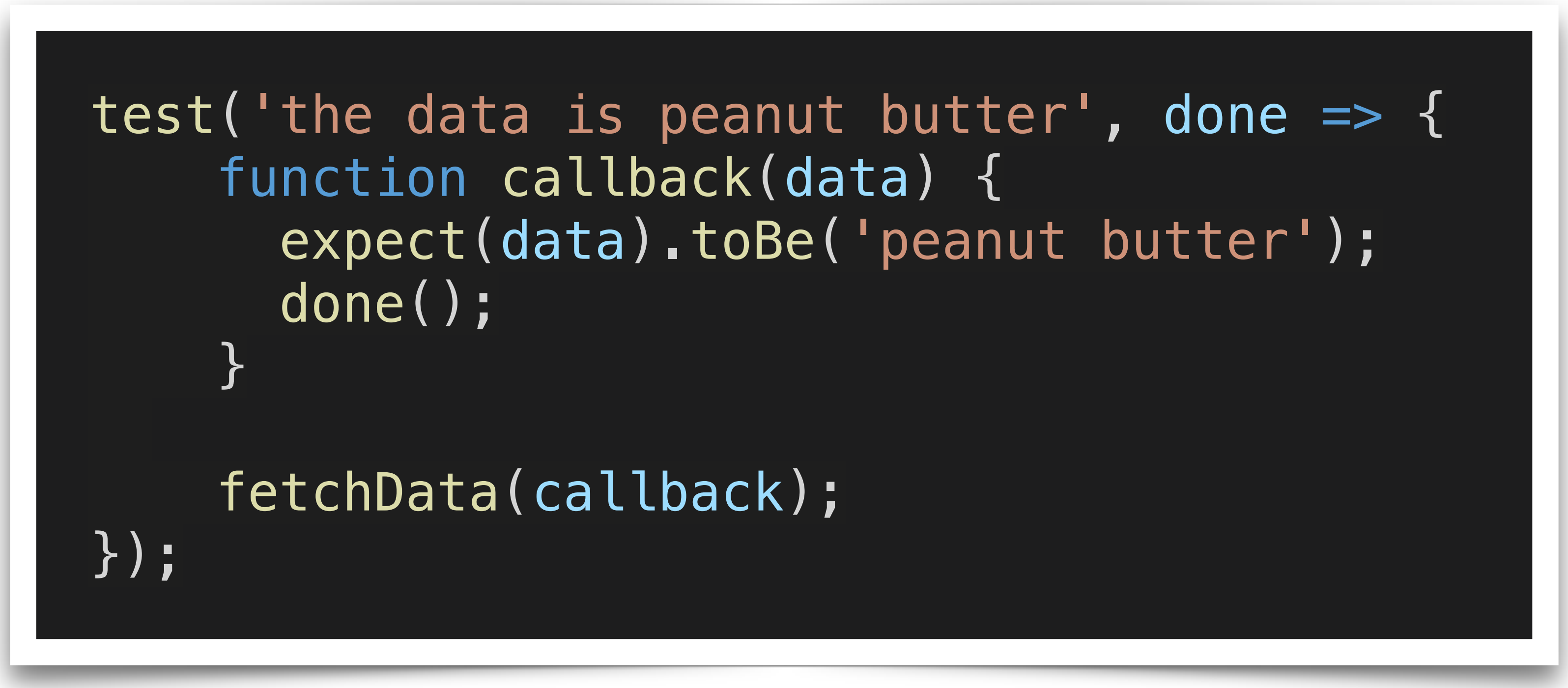


```
// Don't do this!
test('the data is peanut butter', () => {
    function callback(data) {
      expect(data).toBe('peanut butter');
    }

    fetchData(callback);
});
```

Testing Callbacks

Instead of putting the test in a function with an empty argument, use a single argument called *done*.

Jest will wait until the *done* callback is called before finishing the test.

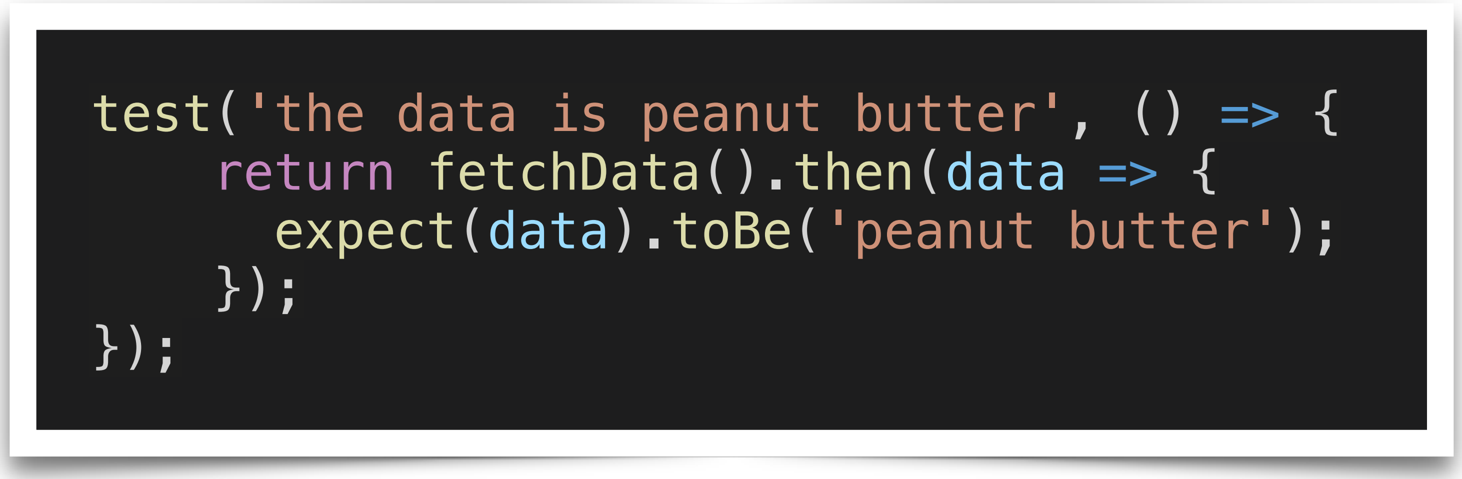


```
test('the data is peanut butter', done => {
    function callback(data) {
      expect(data).toBe('peanut butter');
      done();
    }

    fetchData(callback);
});
```

Testing Promises

Return a promise from your test, and Jest will wait for that promise to resolve. If the promise is rejected, the test will automatically fail.

Be sure to return the promise - if you omit this *return* statement, your test will complete before the promise returned

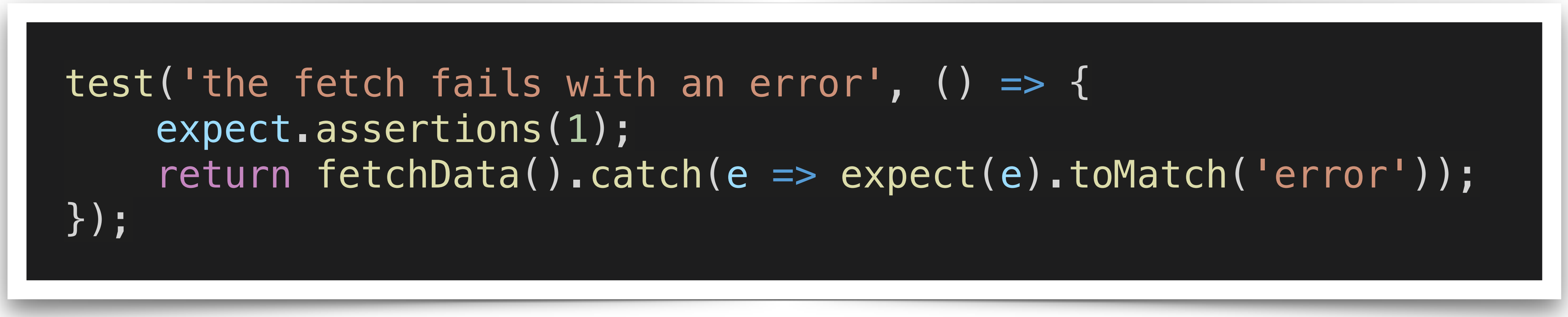


```
test('the data is peanut butter', () => {
    return fetchData().then(data => {
      expect(data).toBe('peanut butter');
    });
});
```

Testing Promises

If you expect a promise to be rejected use the *catch* method. Make sure to add *expect.assertions* to verify that a certain number of assertions are called.

Otherwise a fulfilled promise would not fail the test.



```
test('the fetch fails with an error', () => {
    expect.assertions(1);
    return fetchData().catch(e => expect(e).toMatch('error'));
});
```

Testing React Components

Testing React Components

On top of Jest, we have multiple options to Test React Components:

- **react-test-renderer**

  react-test-renderer is a library for rendering React components to pure JavaScript objects. We can use it for asserting the behavior of our components and for *snapshot testing*.

- **react-testing-library**

  The ***react-testing-library*** is a very lightweight solution for testing React components.

  Its primary guiding principle is: The more your tests resemble the way your software is used, the more confidence they can give you.

- **Enzyme**

  Airbnbs heavy ***alternative*** to react-testing-library - We won't focus on this.

react-test-renderer - Example

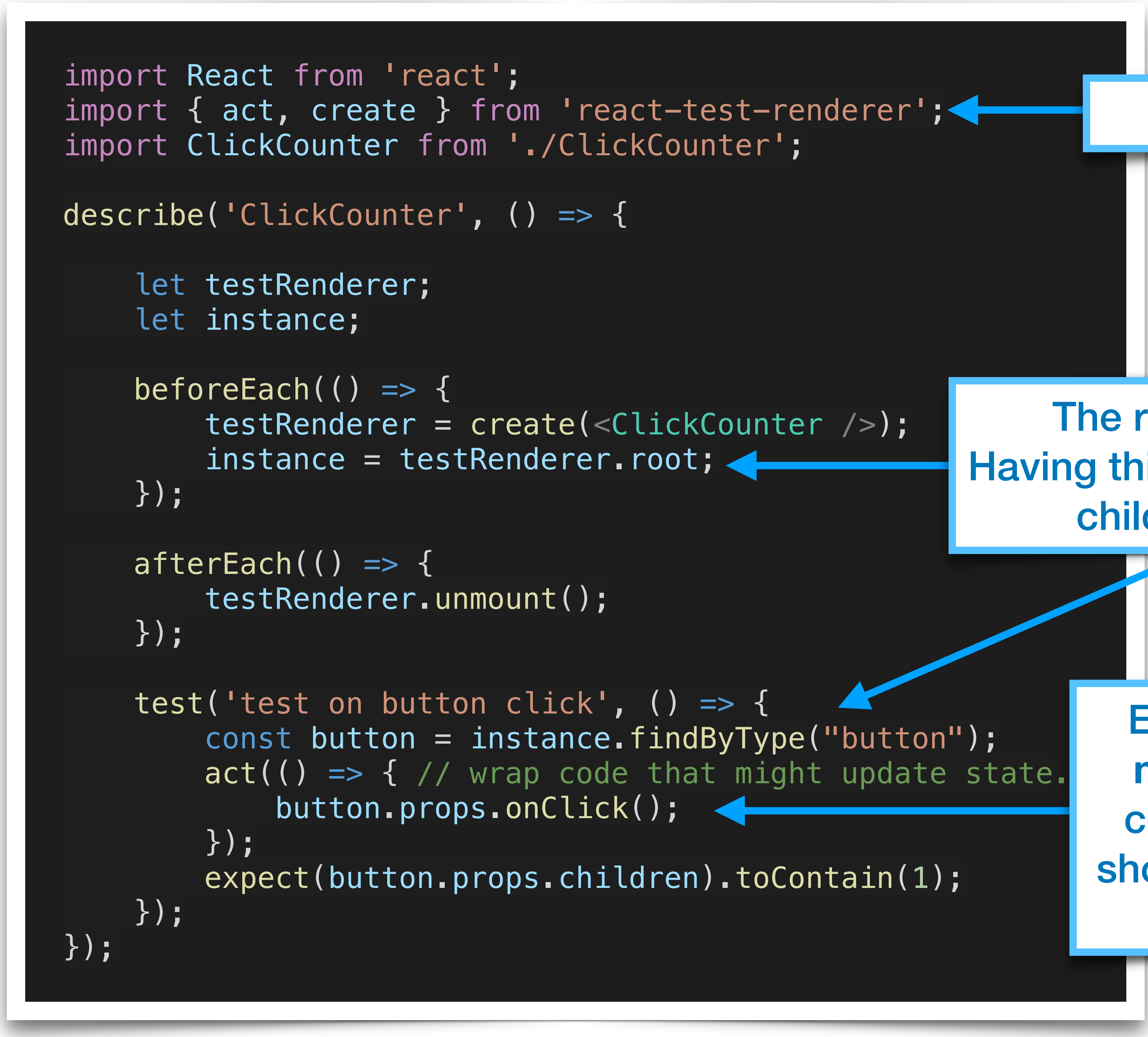
```
import React, { useState } from 'react';

export default function ClickCounter() {
    const [numClicks, setNumClicks] = useState(0);

    function increment() {
        setNumClicks(numClicks + 1);
    }

    return (<button onClick={increment}>You clicked {numClicks} times.</button>)
}
```

We'd like to test, if the text of the button changes correctly after a click on this button.

Three Children here.

```
> npm i --save-dev react-test-renderer
```

react-test-renderer - Example



```
import React from 'react';
import { act, create } from 'react-test-renderer';
import ClickCounter from './ClickCounter';

describe('ClickCounter', () => {

    let testRenderer;
    let instance;

    beforeEach(() => {
        testRenderer = create(<ClickCounter />);
        instance = testRenderer.root;
    });

    afterEach(() => {
        testRenderer.unmount();
    });

    test('test on button click', () => {
        const button = instance.findByType("button");
        act(() => { // wrap code that might update state.
            button.props.onClick();
        });
        expect(button.props.children).toContain(1);
    });
});
```

Important imports.

The root component. Having this, we can access all child components.

Each action which **might change the** components state, should be wrapped in ***act*.**

react-test-renderer - instance

If you have an ***instance*** (either the root instance or as a result of on selector function) one can usually do:

- **access the children**

  .children returns the children of an element instance (E.x. the text node of a button).

- **access the props**

  .props might be attributes or bounded callbacks. Use this to read/ write attributes or to executed bounded callbacks (like ***onClick()*** or ***onFocus()***)

  If you execute a callback, make sure to wrap it into ***act(() => {*** */* callback call */* ***})***

- **locate child instances**

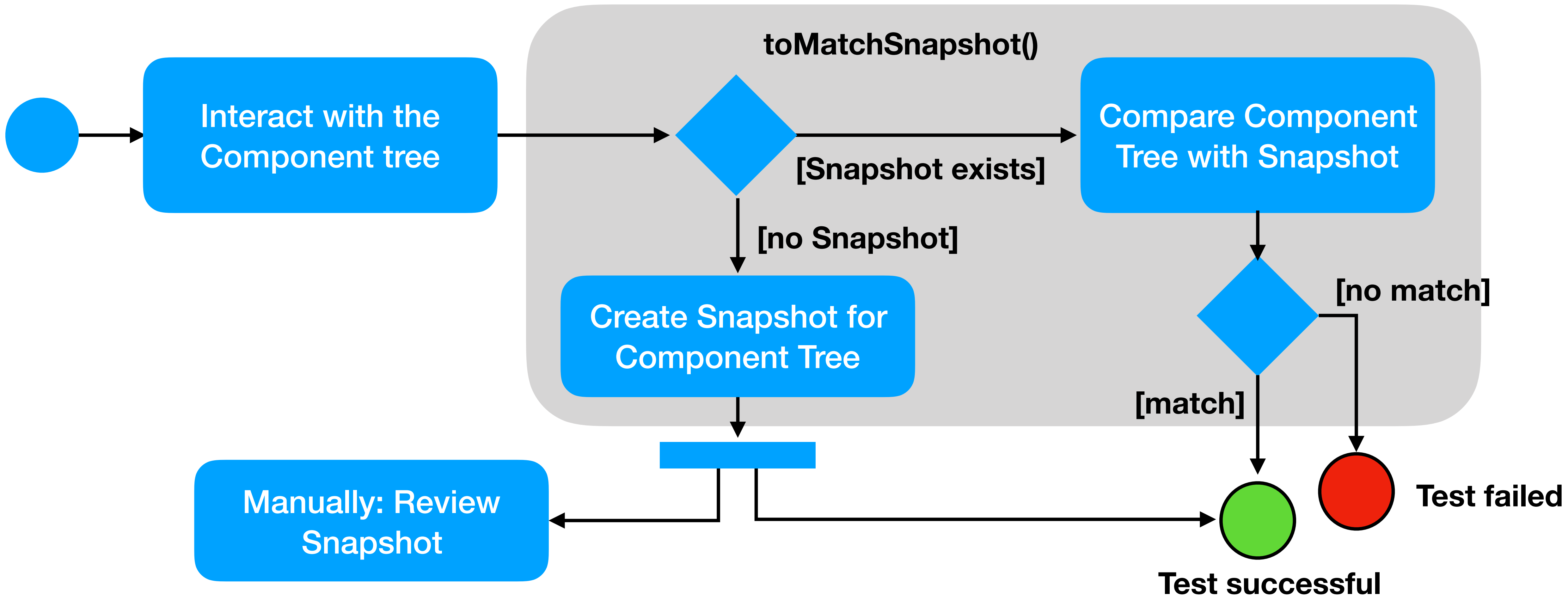
  Use: ***find(), findByType(), findByProps(), findAll(), findAllByType(), findAllByProps()***

https://reactjs.org/docs/test-renderer.html#testinstancefind

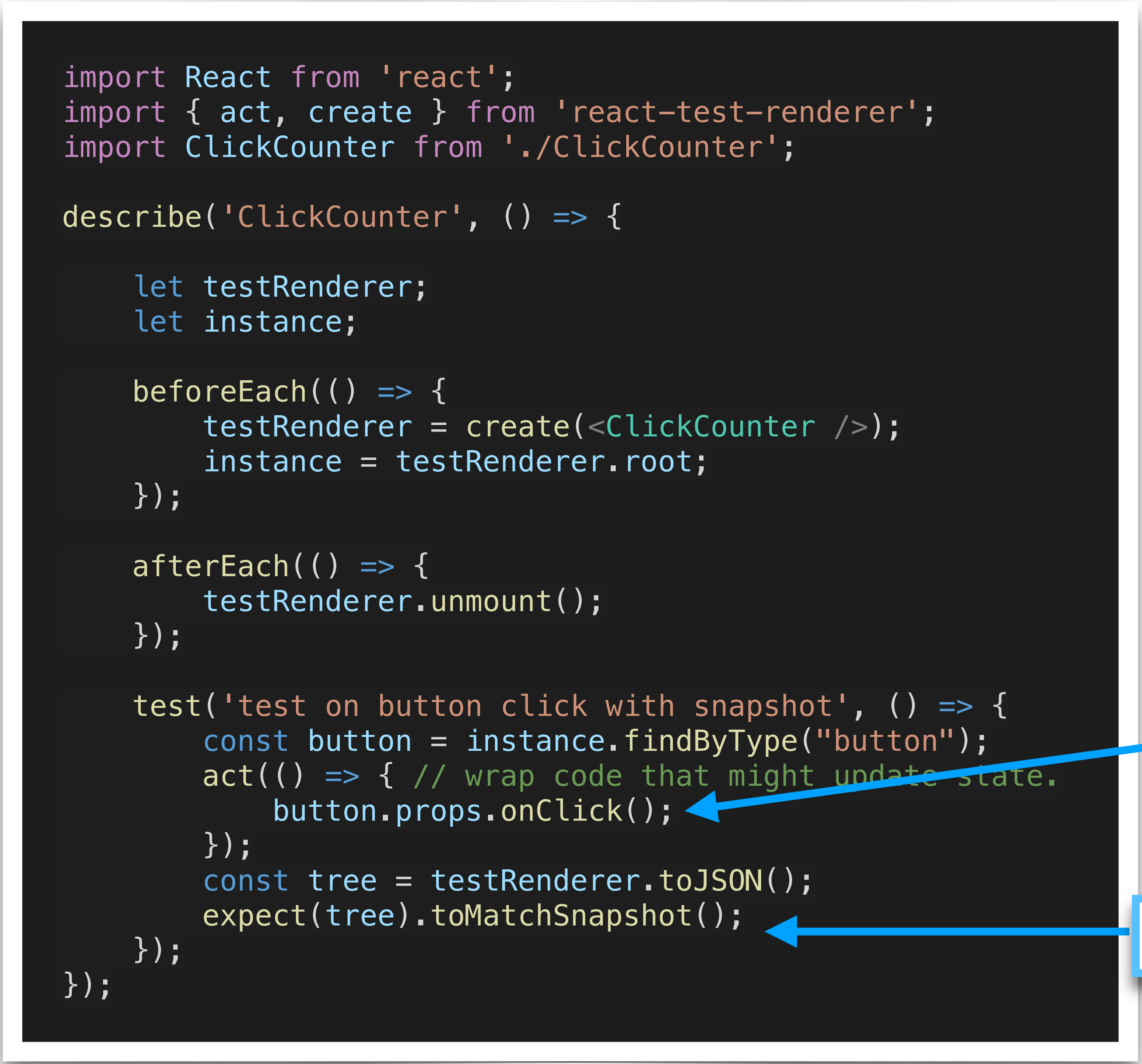
Snapshot Testing

We can use ***react-test-renderer*** and ***Jests toMatchSnapshot() matcher*** to perform snapshot testing.

Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.


toMatchSnapshot()
Interact with the Component tree
[Snapshot exists]
Compare Component Tree with Snapshot
[no Snapshot]
Create Snapshot for Component Tree
[no match]
[match]
Manually: Review Snapshot
Test failed
Test successful

Snapshot Testing - Example

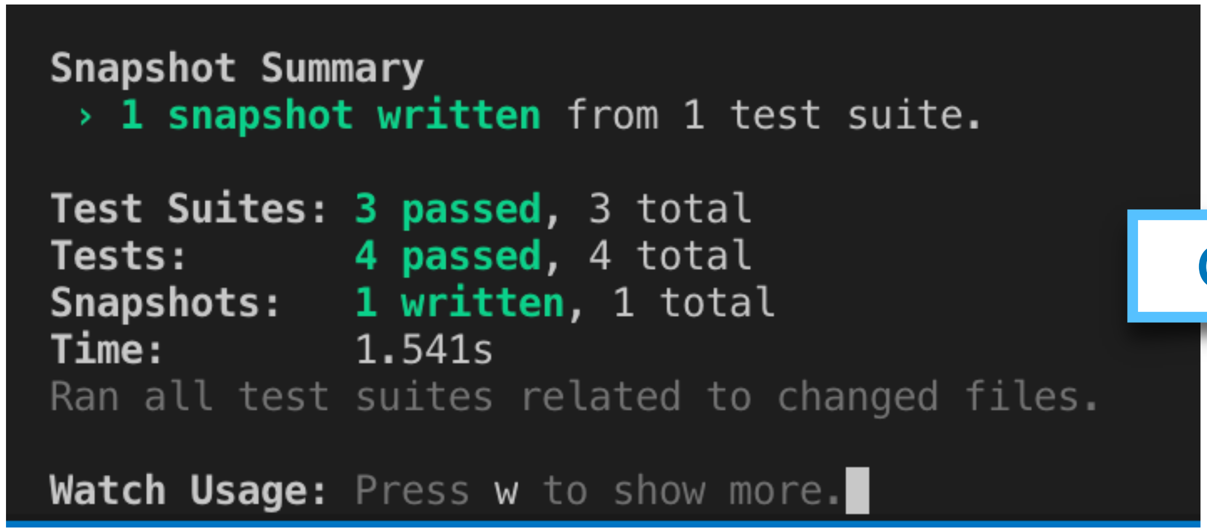


```
import React from 'react';
import { act, create } from 'react-test-renderer';
import ClickCounter from './ClickCounter';

describe('ClickCounter', () => {

    let testRenderer;
    let instance;

    beforeEach(() => {
        testRenderer = create(<ClickCounter />);
        instance = testRenderer.root;
    });

    afterEach(() => {
        testRenderer.unmount();
    });

    test('test on button click with snapshot', () => {
        const button = instance.findByType("button");
        act(() => { // wrap code that might update state.
            button.props.onClick();
        });
        const tree = testRenderer.toJSON();
        expect(tree).toMatchSnapshot();
    });
});
```

Interaction with the component tree.

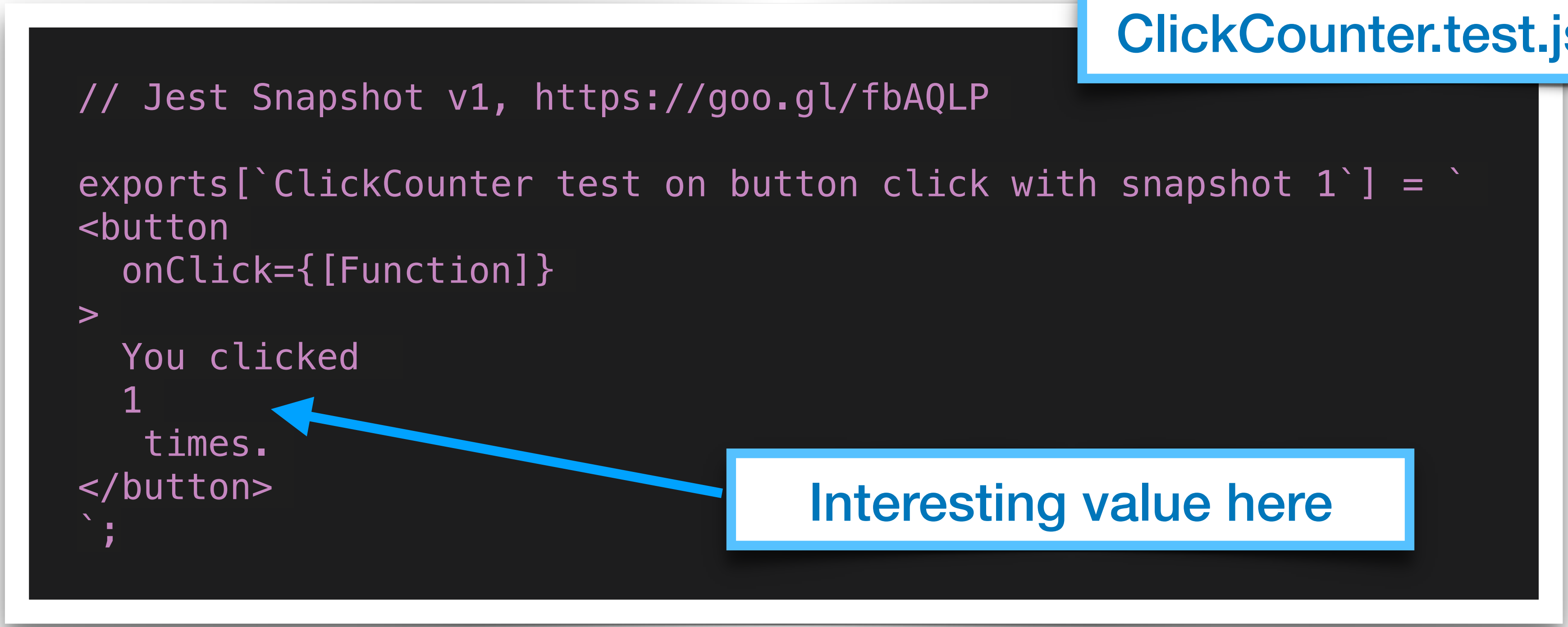
Create or match snapshot

Snapshot Testing - Create Snapshot



```
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 3 passed, 3 total
Tests:       4 passed, 4 total
Snapshots:   1 written, 1 total
Time:        1.541s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Console Output from ***npm test***

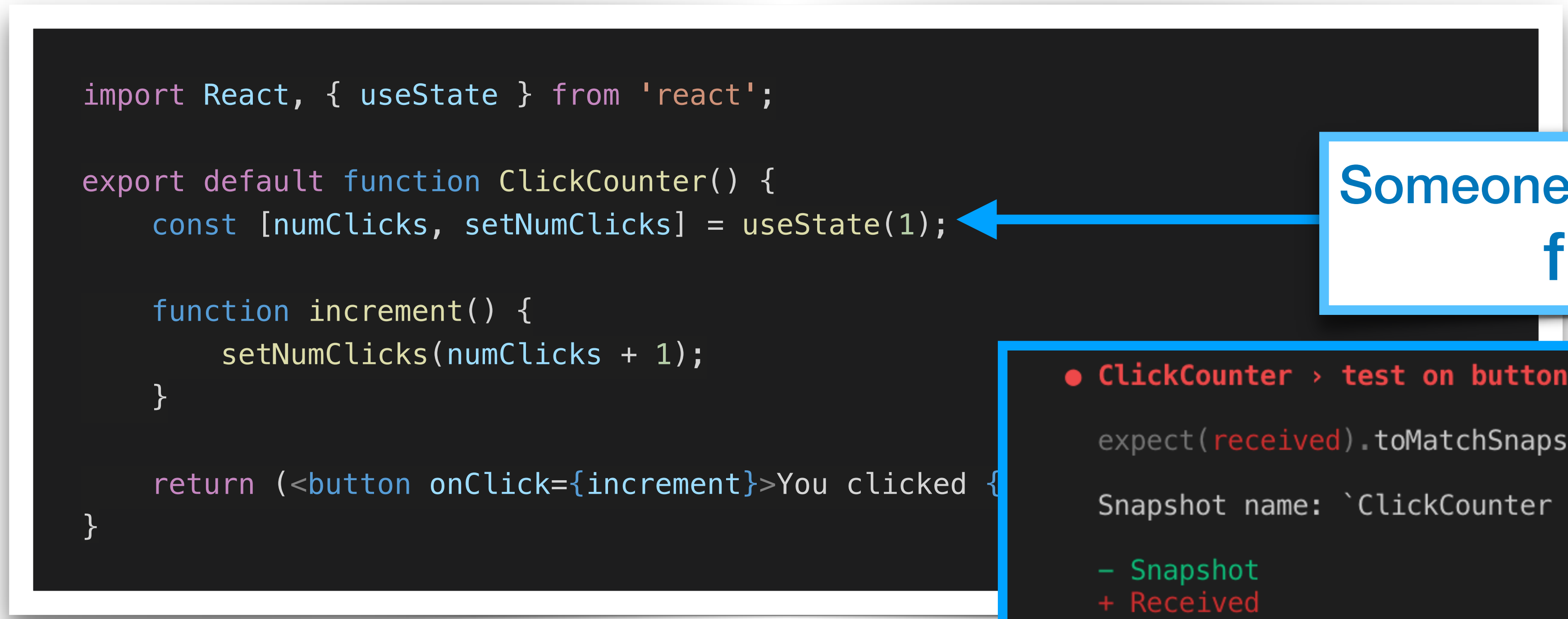


ClickCounter.test.js.snap

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`ClickCounter test on button click with snapshot 1`] = `
<button
  onClick={[Function]}
>
  You clicked
  1
   times.
</button>
`;
```

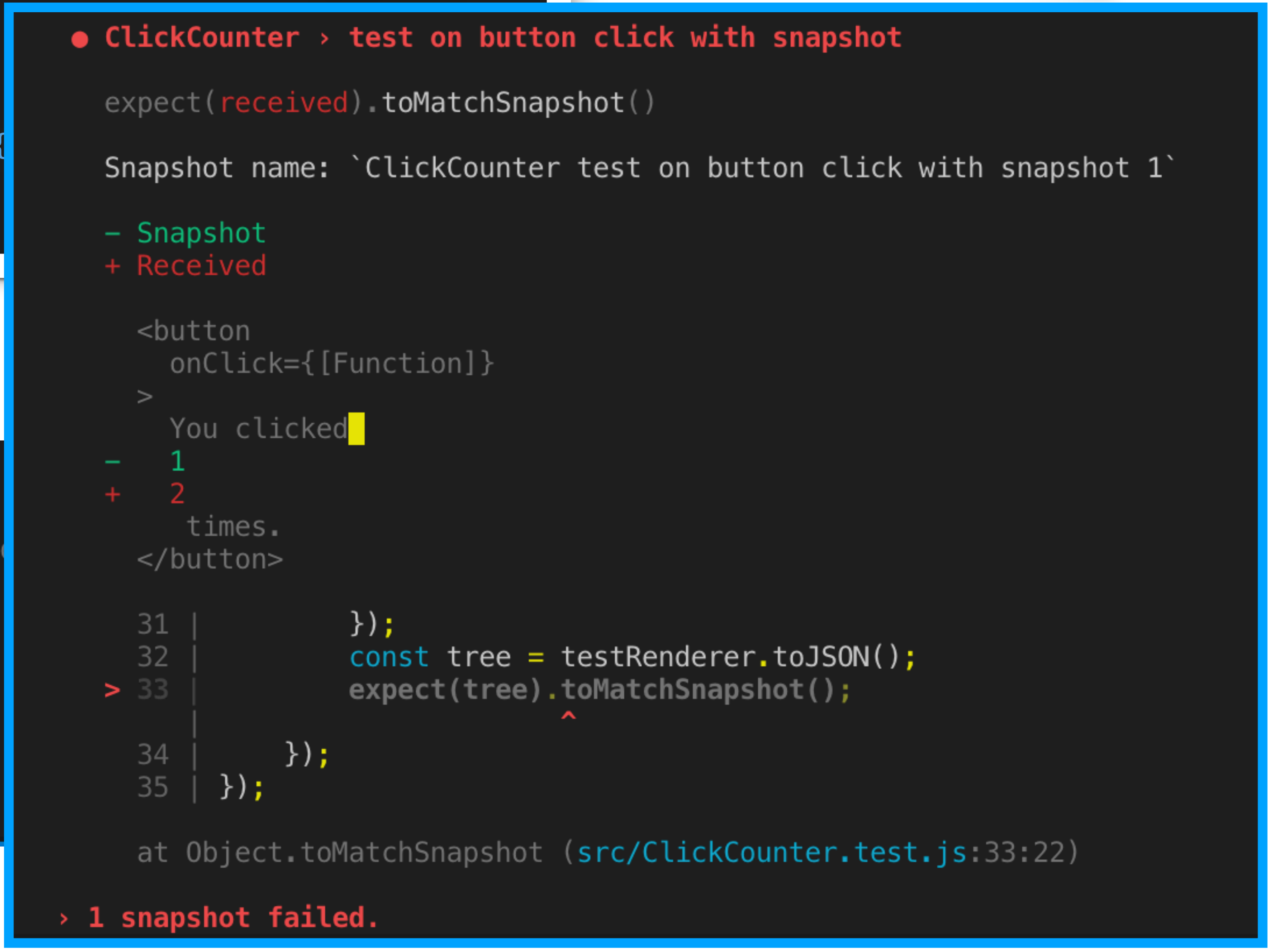
Interesting value here

Snapshot Testing - regression

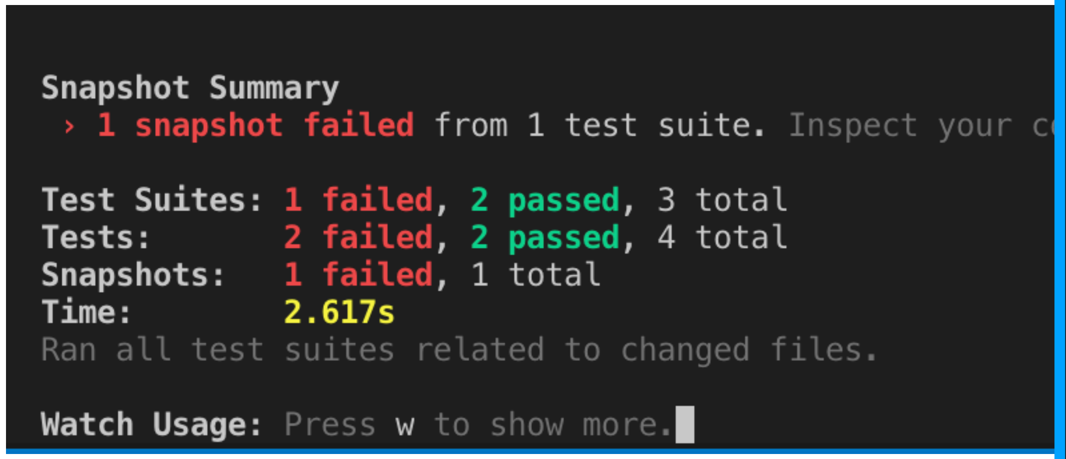


```
import React, { useState } from 'react';

export default function ClickCounter() {
    const [numClicks, setNumClicks] = useState(1);

    function increment() {
        setNumClicks(numClicks + 1);
    }

    return (<button onClick={increment}>You clicked {
}
```

Someone changed initial value
from 0 to 1



```
● ClickCounter › test on button click with snapshot

  expect(received).toMatchSnapshot()

  Snapshot name: `ClickCounter test on button click with snapshot 1`

  - Snapshot
  + Received

    <button
      onClick={[Function]}
    >
      You clicked
  -   1
  +   2
       times.
    </button>

    31 |         });
    32 |         const tree = testRenderer.toJSON();
  > 33 |         expect(tree).toMatchSnapshot();
       |                      ^
    34 |     });
    35 | });

    at Object.toMatchSnapshot (src/ClickCounter.test.js:33:22)

› 1 snapshot failed.
```



```
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect your c

Test Suites: 1 failed, 2 passed, 3 total
Tests:       2 failed, 2 passed, 4 total
Snapshots:   1 failed, 1 total
Time:        2.617s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Redux

„Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test."

– *redux.js.org*

Redux - Three principles

Redux can be described in three fundamental principles:

- **Single source of truth**

  The *state* of your whole application is stored in an object tree within a ***single store***.

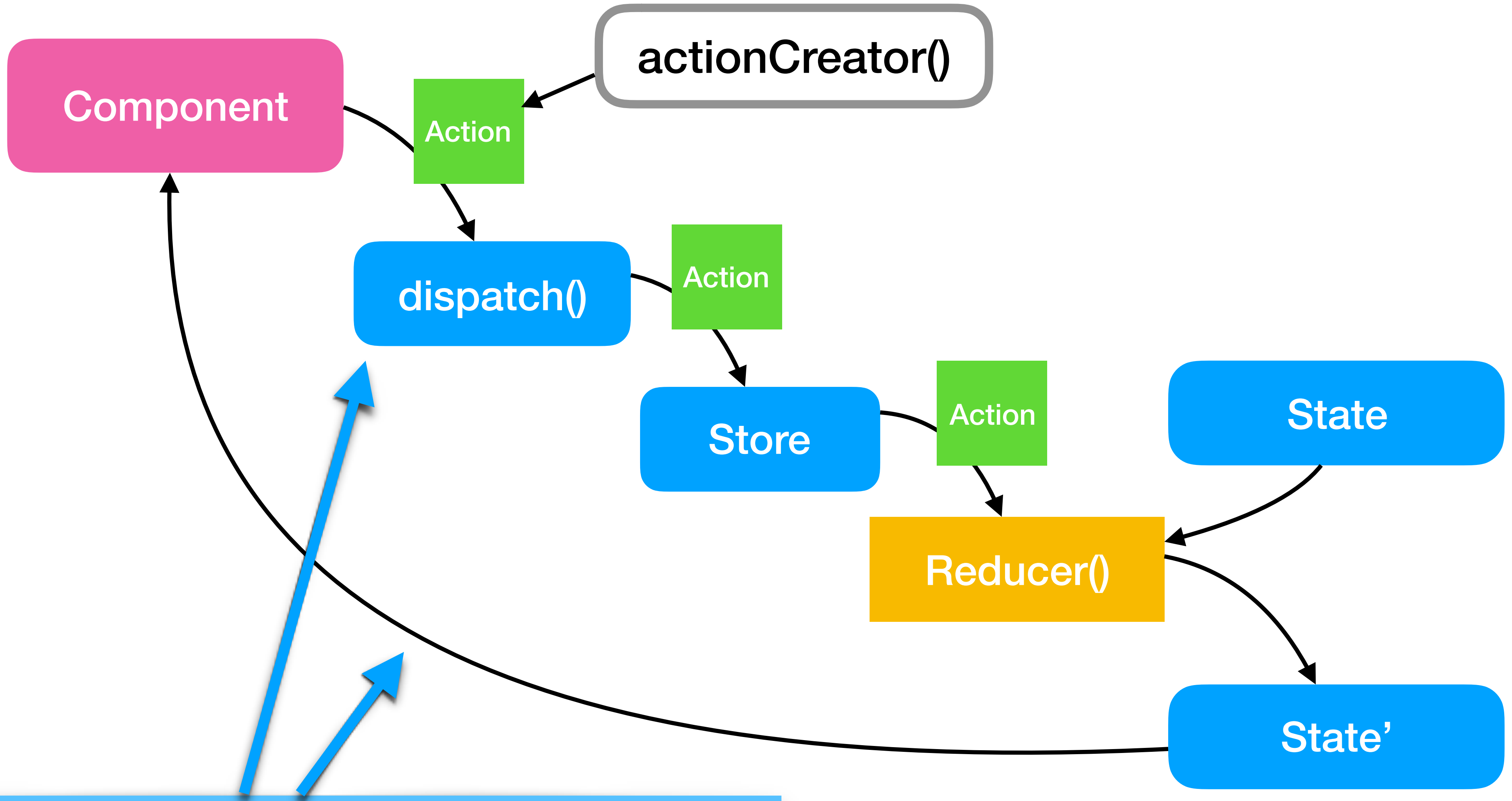
- **State is read-only**

  The only way to change the state is to emit an ***action***, an object describing what happened.

- **Changes are made with pure functions**

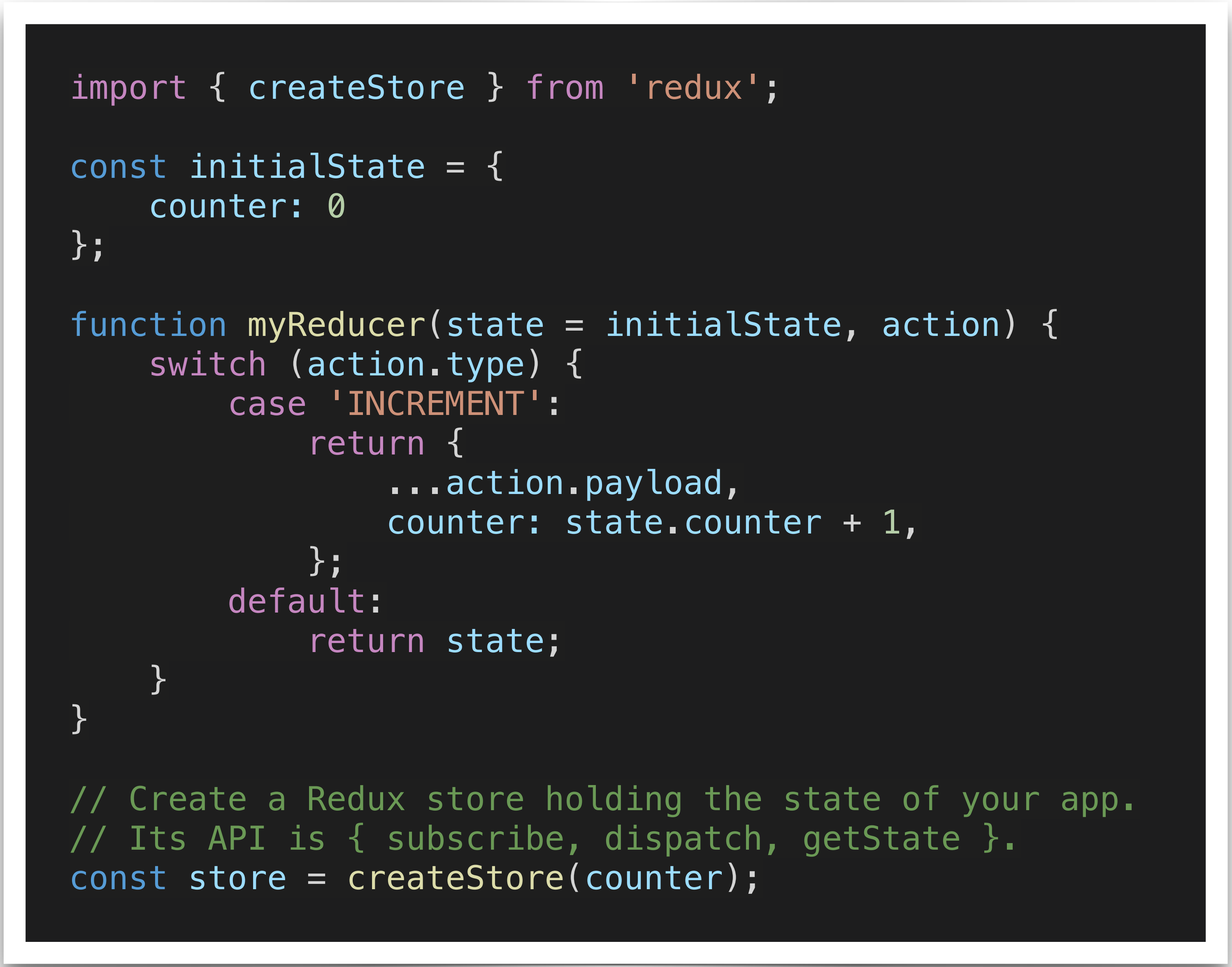
  To specify how the state tree is transformed by actions, you write pure ***reducers***.

Redux building parts

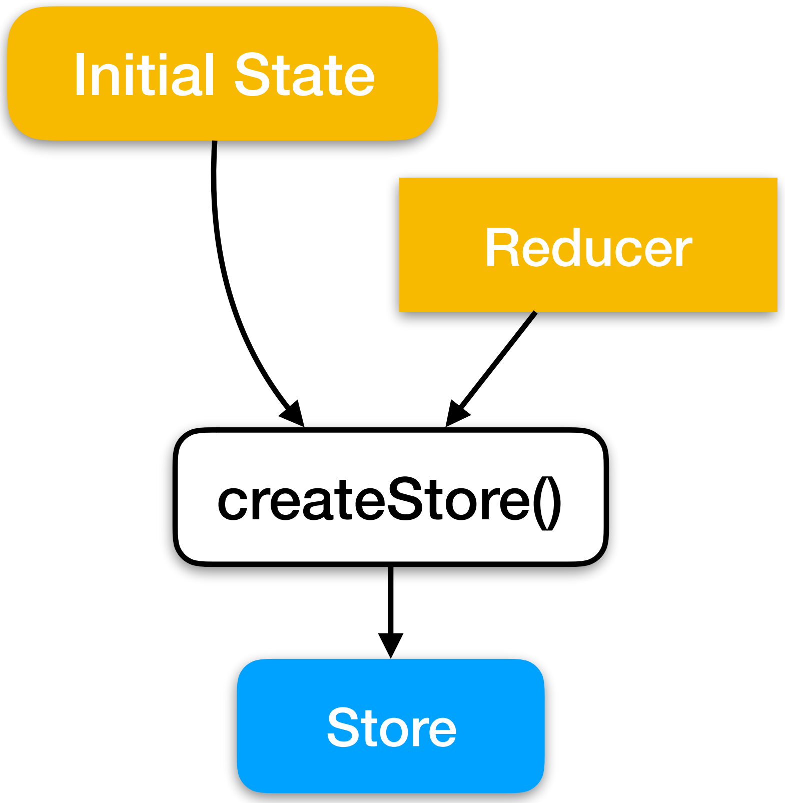


Usually this is handled/ simplified using ***react-redux***

Setting up Redux

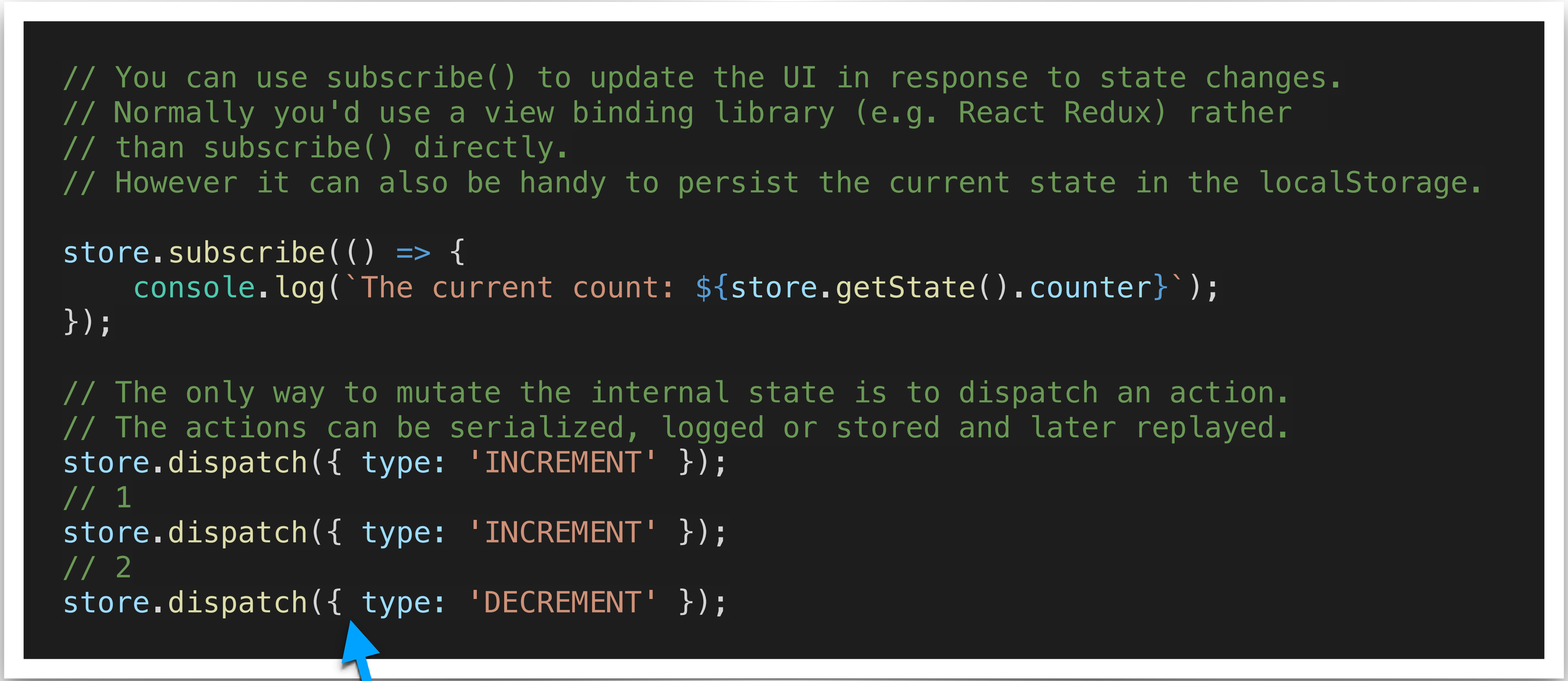


```
import { createStore } from 'redux';

const initialState = {
    counter: 0
};

function myReducer(state = initialState, action) {
    switch (action.type) {
        case 'INCREMENT':
            return {
                ...action.payload,
                counter: state.counter + 1,
            };
        default:
            return state;
    }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
const store = createStore(counter);
```



```
> npm install redux
```

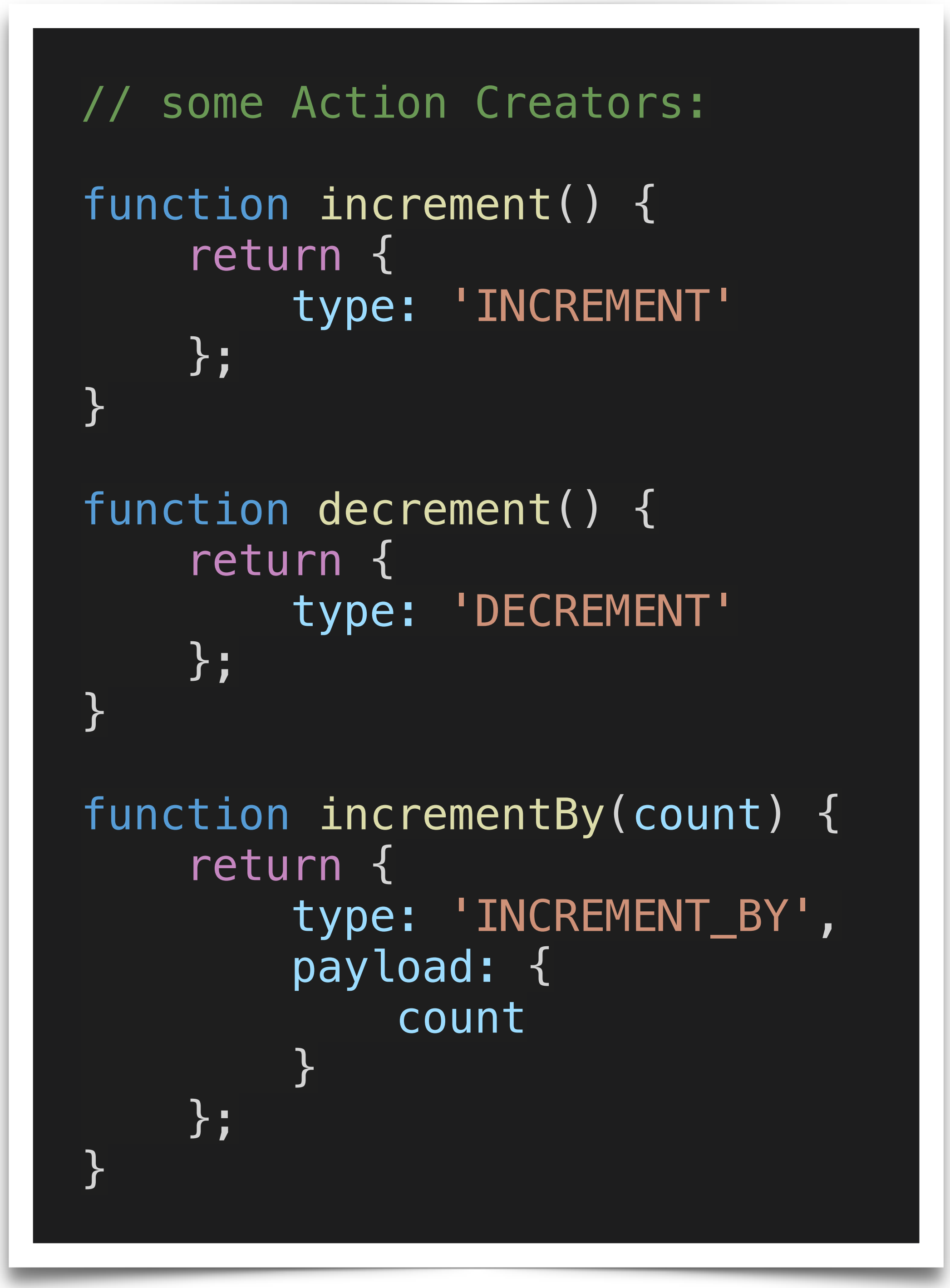
Using Redux



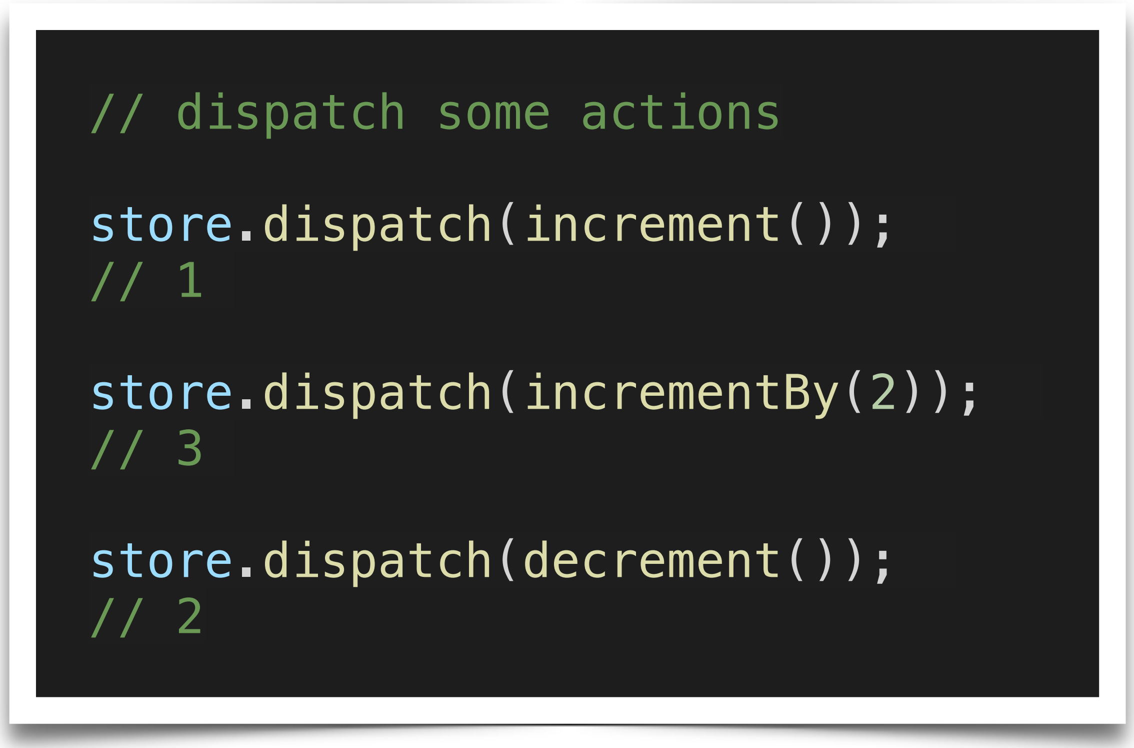
```
// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather
// than subscribe() directly.
// However it can also be handy to persist the current state in the localStorage.

store.subscribe(() => {
    console.log(`The current count: ${store.getState().counter}`);
});

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'INCREMENT' });
// 1
store.dispatch({ type: 'INCREMENT' });
// 2
store.dispatch({ type: 'DECREMENT' });
```

In simple cases the ***Action*** object can be created „on the fly“ but it's recommended to use *Action* ***Creators*** to be make your code more readable, especially while creating complex Action objects.

Action Creators



```
// some Action Creators:

function increment() {
    return {
        type: 'INCREMENT'
    };
}

function decrement() {
    return {
        type: 'DECREMENT'
    };
}

function incrementBy(count) {
    return {
        type: 'INCREMENT_BY',
        payload: {
            count
        }
    };
}
```



```
// dispatch some actions

store.dispatch(increment());
// 1

store.dispatch(incrementBy(2));
// 3

store.dispatch(decrement());
// 2
```

Pure Functions

A pure function is a function that has the following properties:

- Its return value is the same for the same arguments

  (no variation with local static variables, non-local variables, mutable reference arguments or input streams from I/O devices)

- Its evaluation has no side effects

  (no variation with local static variables, non-local variables, mutable reference arguments or input streams from I/O devices)

– *en.wikipedia.org*

Pure Functions - Examples

```
function nextFiveDays() {
    return currentTimestamp() + daysInMillis(5);
}
```

Returns a new value every call

```
function nextFiveDays(currentTimestamp) {
    return currentTimestamp + daysInMillis(5);
}
```

Pure Functions - Examples

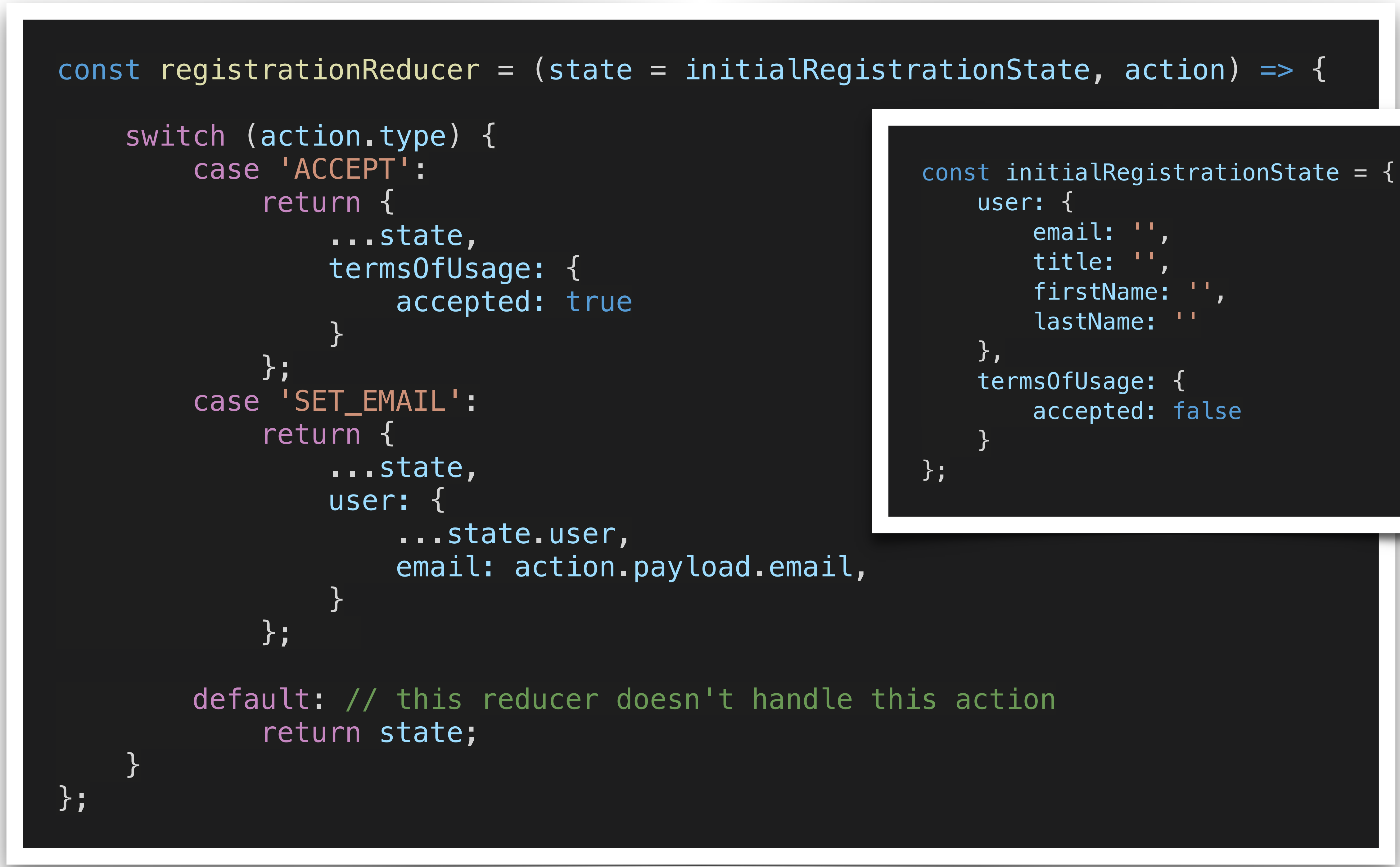
```
function incrementAndReturn() {
    return counter++;
}
```

🚫

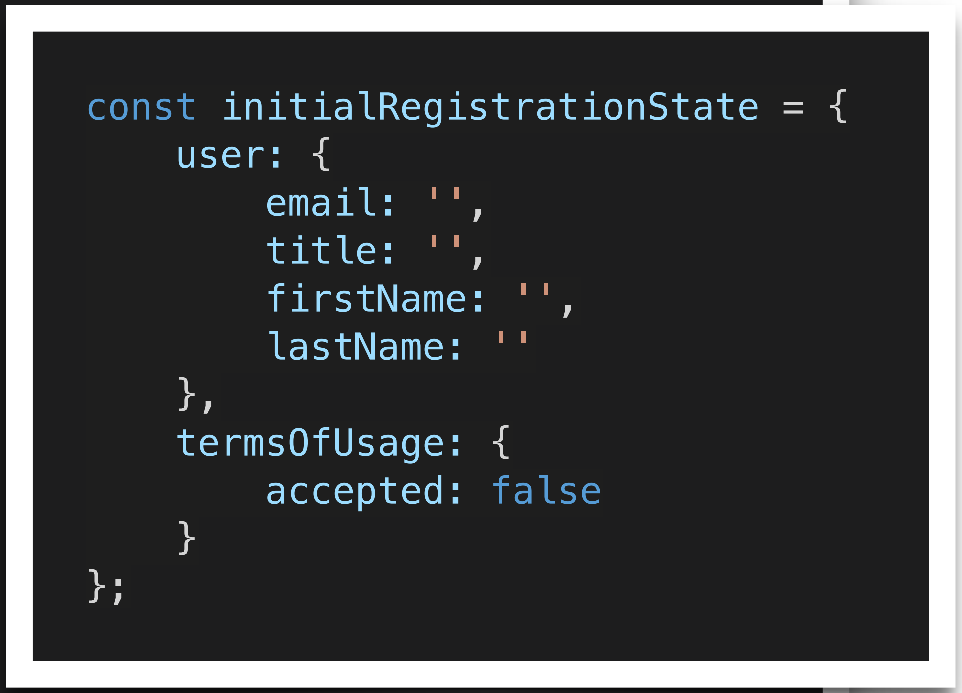
Returns a new value every call
Side Effect: Outer Variable modified.

```
function incrementAndReturn(counter) {
    return counter + 1;
}
```

Reducers as Pure Functions



```
const registrationReducer = (state = initialRegistrationState, action) => {

    switch (action.type) {
        case 'ACCEPT':
            return {
                ...state,
                termsOfUsage: {
                    accepted: true
                }
            };
        case 'SET_EMAIL':
            return {
                ...state,
                user: {
                    ...state.user,
                    email: action.payload.email,
                }
            };

        default: // this reducer doesn't handle this action
            return state;
    }
};
```



```
const initialRegistrationState = {
    user: {
        email: '',
        title: '',
        firstName: '',
        lastName: ''
    },
    termsOfUsage: {
        accepted: false
    }
};
```

Redux Middleware

„Middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer“

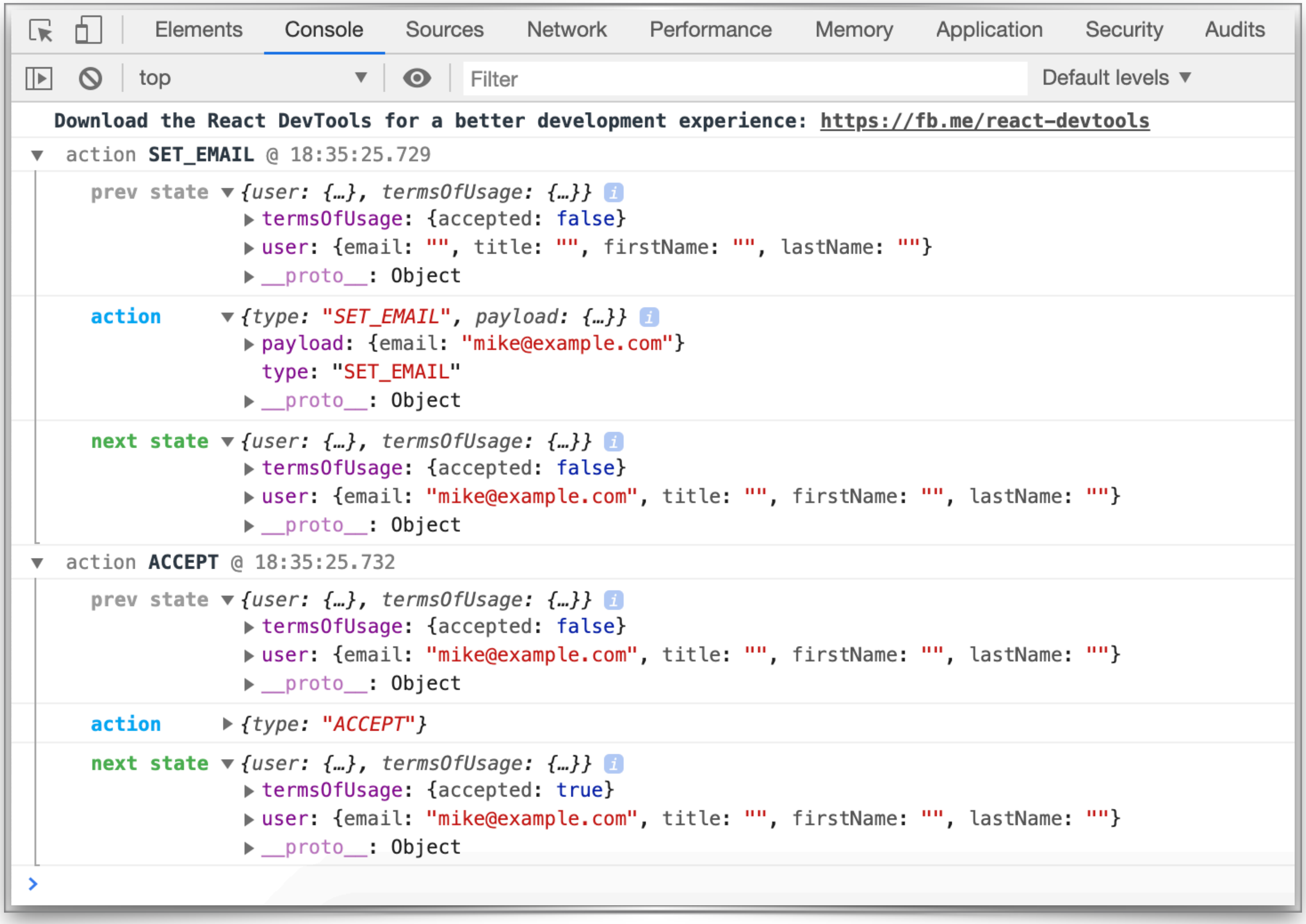
– *redux.js.org*

Redux Middleware - logger

```
import { createStore, applyMiddleware } from 'redux';
import logger from 'redux-logger'; // the logger middleware
import {
    registrationReducer,
    acceptTermsOfUsage,
    setEmail
} from './store/registration';


// apply the logger middleware
const store = createStore(registrationReducer, applyMiddleware(logger));

// dispatch some actions
store.dispatch(setEmail('mike@example.com'));
store.dispatch(acceptTermsOfUsage());
```

```
> npm install redux-logger
```

Redux Middleware - logger



```
Download the React DevTools for a better development experience: https://fb.me/react-devtools
▼ action SET_EMAIL @ 18:35:25.729
    prev state ▼{user: {…}, termsOfUsage: {…}}
                 ▶termsOfUsage: {accepted: false}
                 ▶user: {email: "", title: "", firstName: "", lastName: ""}
                 ▶__proto__: Object
    action     ▼{type: "SET_EMAIL", payload: {…}}
                 ▶payload: {email: "mike@example.com"}
                  type: "SET_EMAIL"
                 ▶__proto__: Object
    next state ▼{user: {…}, termsOfUsage: {…}}
                 ▶termsOfUsage: {accepted: false}
                 ▶user: {email: "mike@example.com", title: "", firstName: "", lastName: ""}
                 ▶__proto__: Object
▼ action ACCEPT @ 18:35:25.732
    prev state ▼{user: {…}, termsOfUsage: {…}}
                 ▶termsOfUsage: {accepted: false}
                 ▶user: {email: "mike@example.com", title: "", firstName: "", lastName: ""}
                 ▶__proto__: Object
    action     ▶{type: "ACCEPT"}
    next state ▼{user: {…}, termsOfUsage: {…}}
                 ▶termsOfUsage: {accepted: true}
                 ▶user: {email: "mike@example.com", title: "", firstName: "", lastName: ""}
                 ▶__proto__: Object
>
```

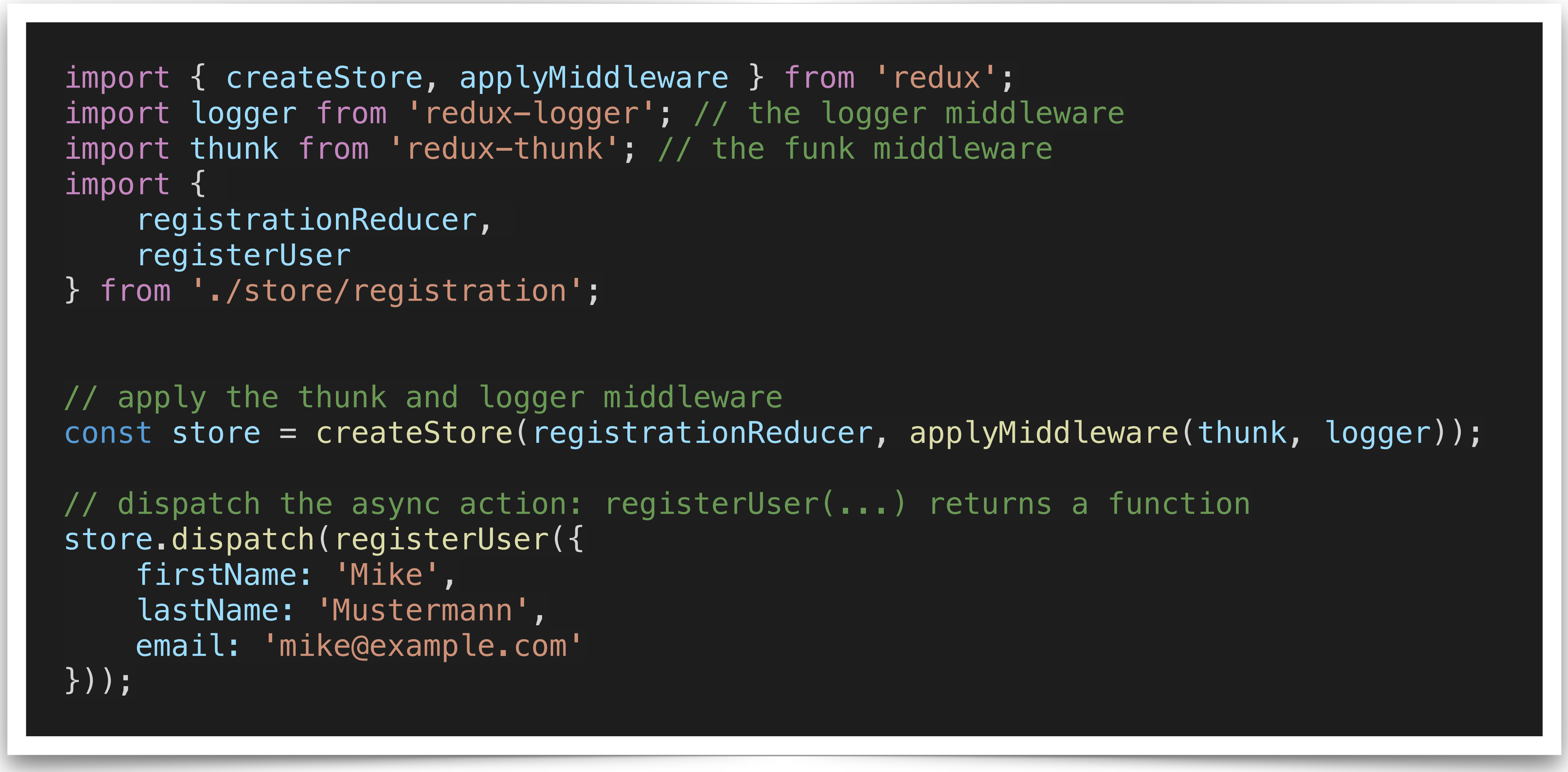
Redux Async Workflow

- Without middleware, Redux store only supports synchronous data flow.

- Asynchronous middleware like ***redux-thunk*** wraps the store's *dispatch()* method and allows you to dispatch something other than actions, for example, functions or Promises.

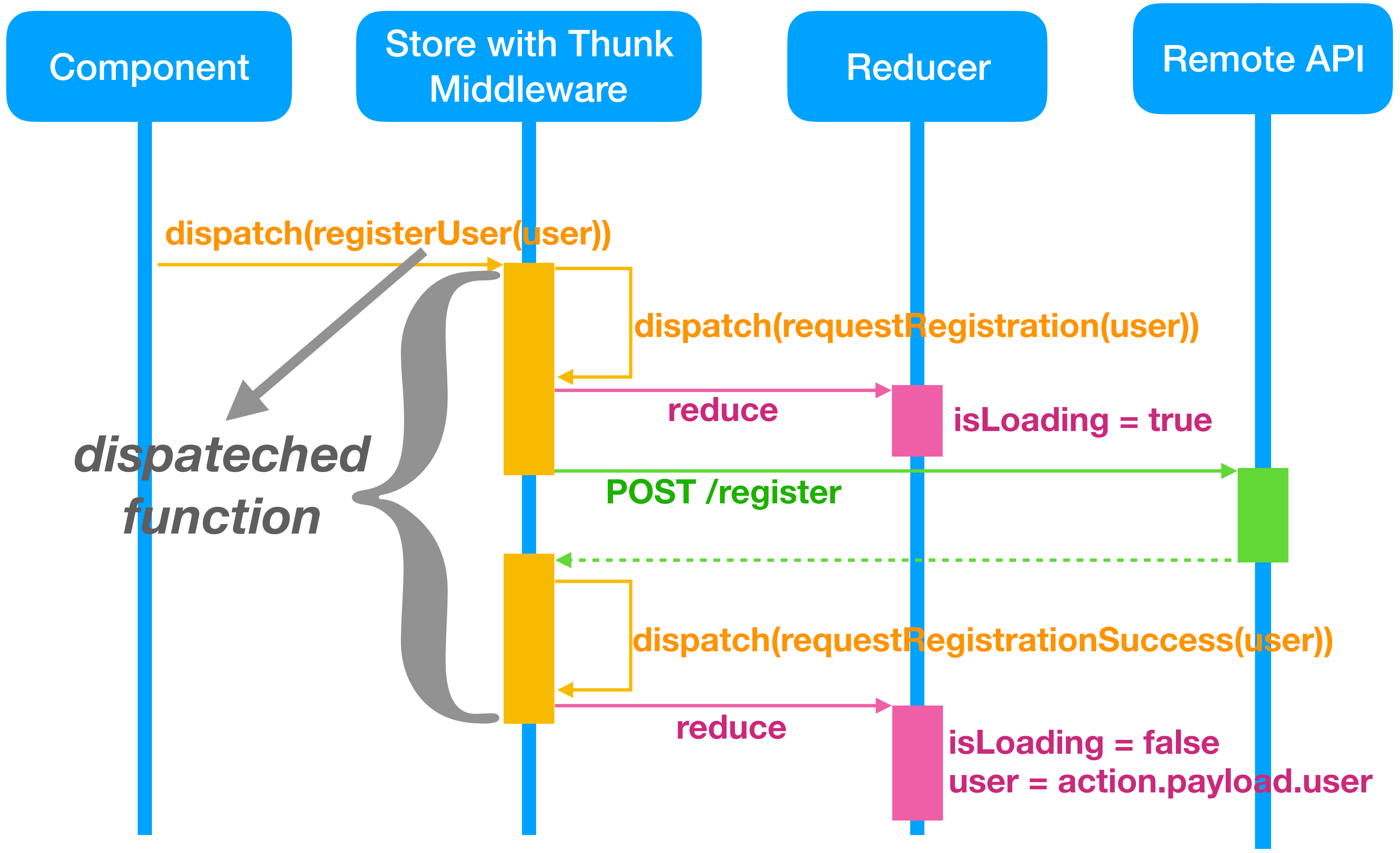
```
> npm install redux-thunk
```

Redux Async Workflow

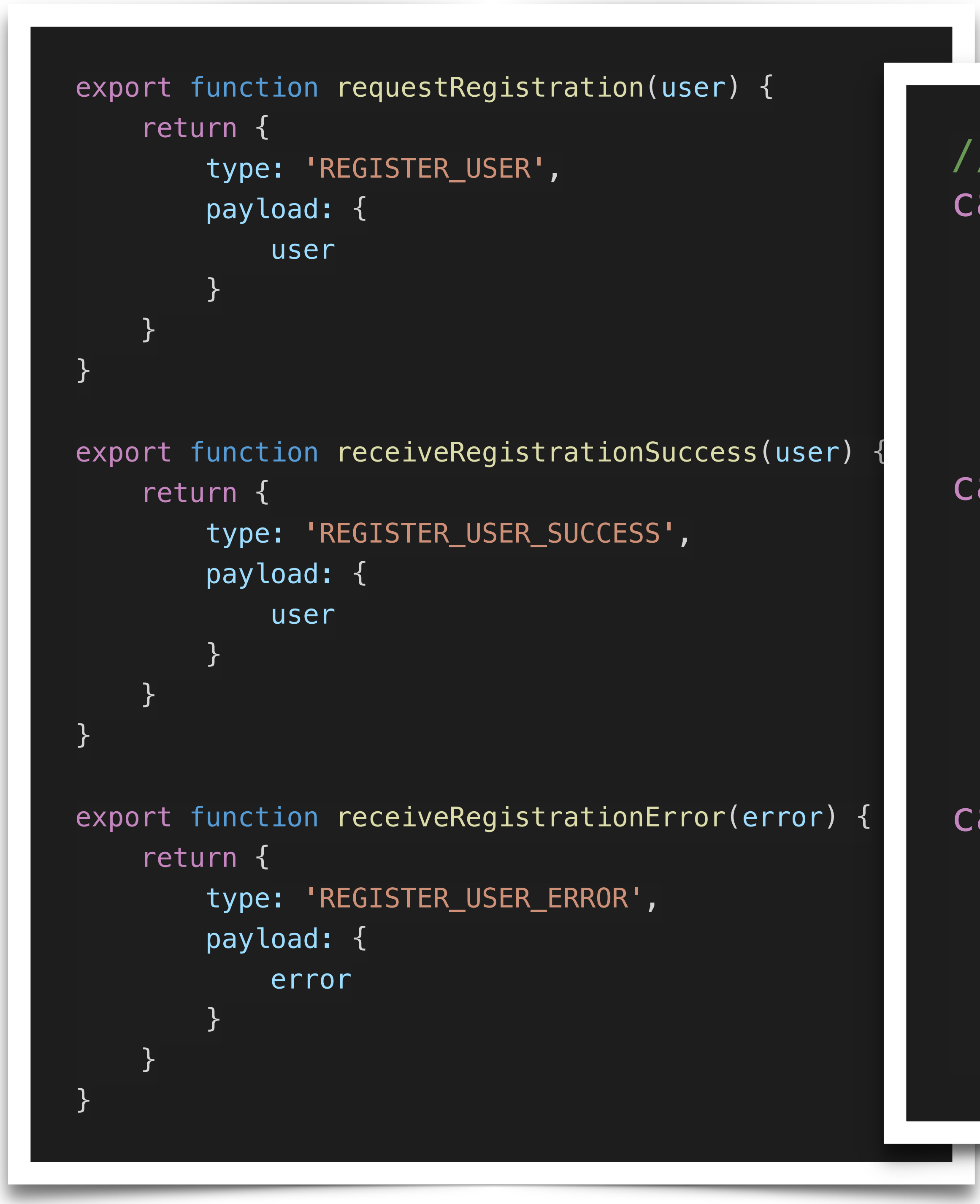


```
import { createStore, applyMiddleware } from 'redux';
import logger from 'redux-logger'; // the logger middleware
import thunk from 'redux-thunk'; // the funk middleware
import {
    registrationReducer,
    registerUser
} from './store/registration';


// apply the thunk and logger middleware
const store = createStore(registrationReducer, applyMiddleware(thunk, logger));

// dispatch the async action: registerUser(...) returns a function
store.dispatch(registerUser({
    firstName: 'Mike',
    lastName: 'Mustermann',
    email: 'mike@example.com'
}));
```

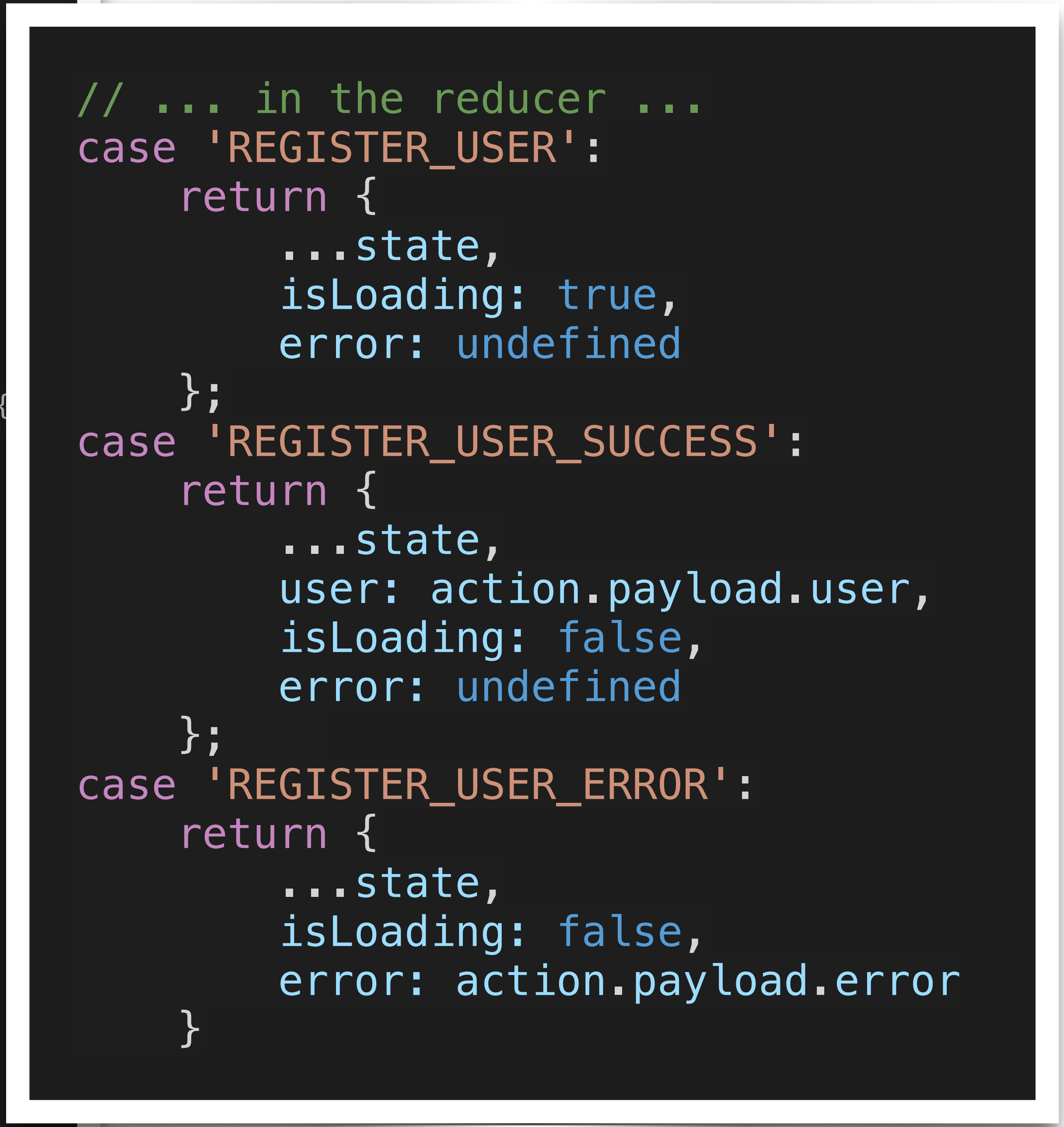
Redux Async Workflow


Component
Store with Thunk Middleware
Reducer
Remote API
dispatch(registerUser(user))
dispatch(requestRegistration(user))
reduce
isLoading = true
POST /register
dispatched function
dispatch(requestRegistrationSuccess(user))
reduce
isLoading = false
user = action.payload.user

Redux Async Workflow

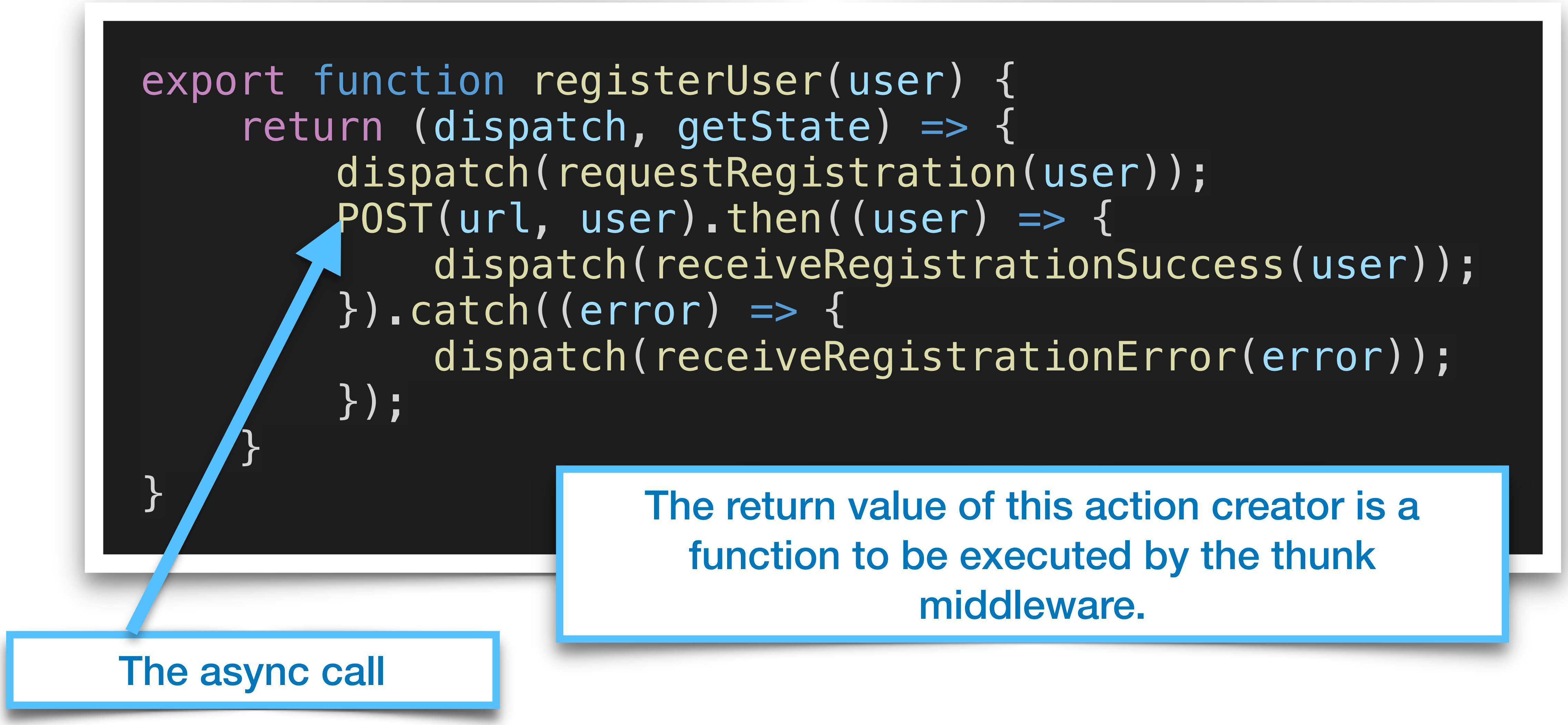


```
export function requestRegistration(user) {
    return {
        type: 'REGISTER_USER',
        payload: {
            user
        }
    }
}

export function receiveRegistrationSuccess(user) {
    return {
        type: 'REGISTER_USER_SUCCESS',
        payload: {
            user
        }
    }
}

export function receiveRegistrationError(error) {
    return {
        type: 'REGISTER_USER_ERROR',
        payload: {
            error
        }
    }
}
```



```
// ... in the reducer ...
case 'REGISTER_USER':
    return {
        ...state,
        isLoading: true,
        error: undefined
    };
case 'REGISTER_USER_SUCCESS':
    return {
        ...state,
        user: action.payload.user,
        isLoading: false,
        error: undefined
    };
case 'REGISTER_USER_ERROR':
    return {
        ...state,
        isLoading: false,
        error: action.payload.error
    }
```

Redux Async Workflow



```
export function registerUser(user) {
    return (dispatch, getState) => {
        dispatch(requestRegistration(user));
        POST(url, user).then((user) => {
            dispatch(receiveRegistrationSuccess(user));
        }).catch((error) => {
            dispatch(receiveRegistrationError(error));
        });
    }
}
```

The return value of this action creator is a function to be executed by the thunk middleware.

The async call

React Redux

„React Redux is the official React binding for Redux.

It lets your React components read data from a Redux store, and dispatch actions to the store to update data."

– *react-redux.js.org*

Setting up React Redux

```
import { Provider } from 'react-redux';
// ...

const store = createStore(myReducer);

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

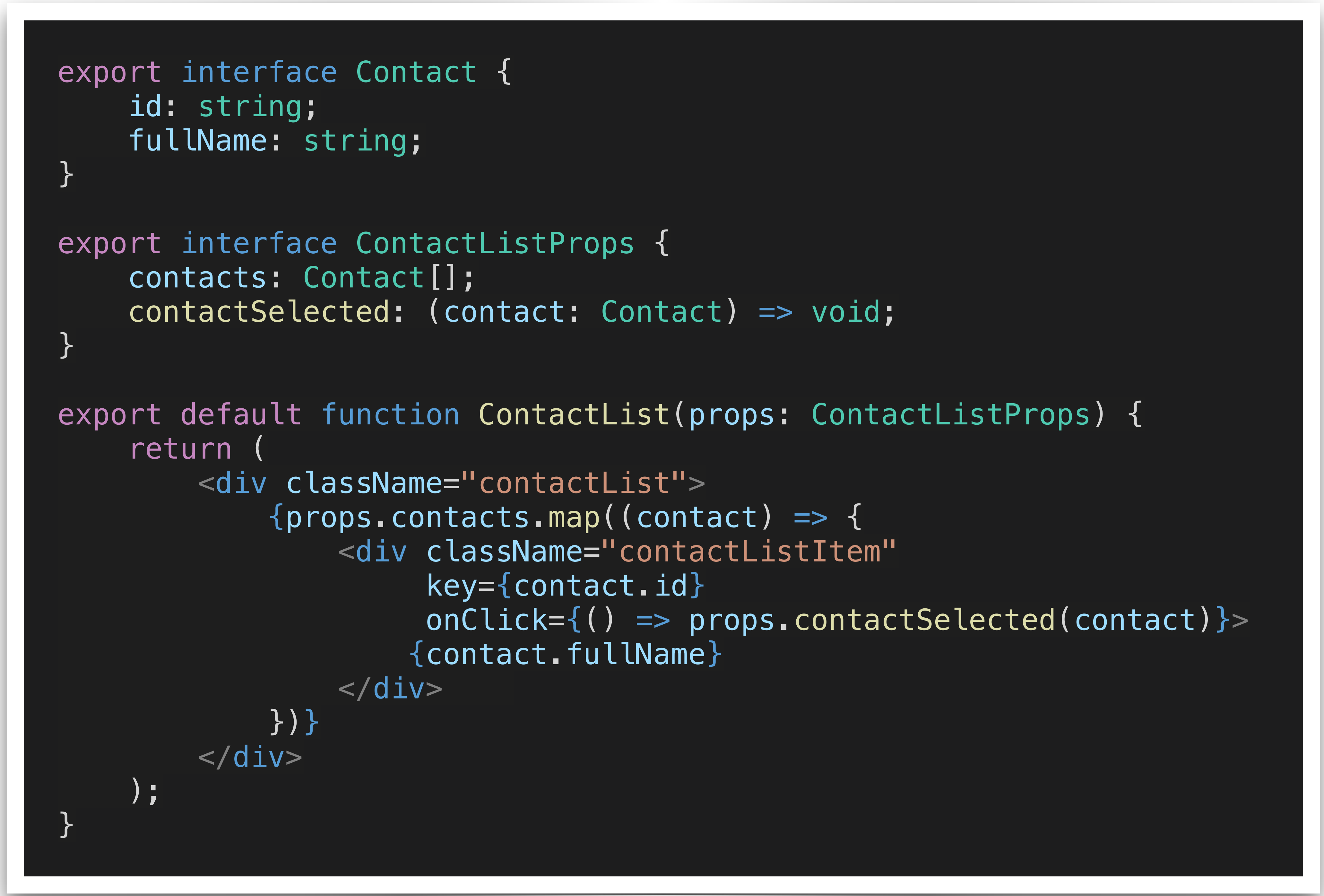
```
> npm install react-redux
```

Presentational and Container Components

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Described by Dan Abramov:
https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Presentational Component



```
export interface Contact {
    id: string;
    fullName: string;
}

export interface ContactListProps {
    contacts: Contact[];
    contactSelected: (contact: Contact) => void;
}

export default function ContactList(props: ContactListProps) {
    return (
        <div className="contactList">
            {props.contacts.map((contact) => {
                <div className="contactListItem"
                     key={contact.id}
                     onClick={() => props.contactSelected(contact)}>
                    {contact.fullName}
                </div>
            })}
        </div>
    );
}
```

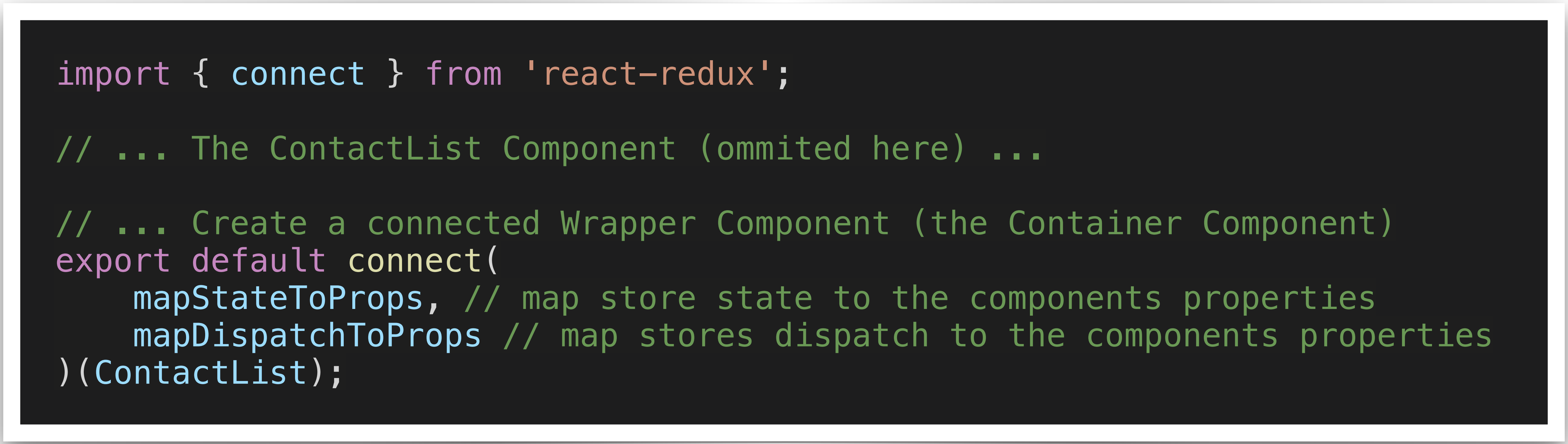
Container Component with connect()

- The ***connect()*** function connects a React component to a Redux store.

- It provides its connected component with the pieces of the data it needs from the store, and the functions it can use to dispatch actions to the store.

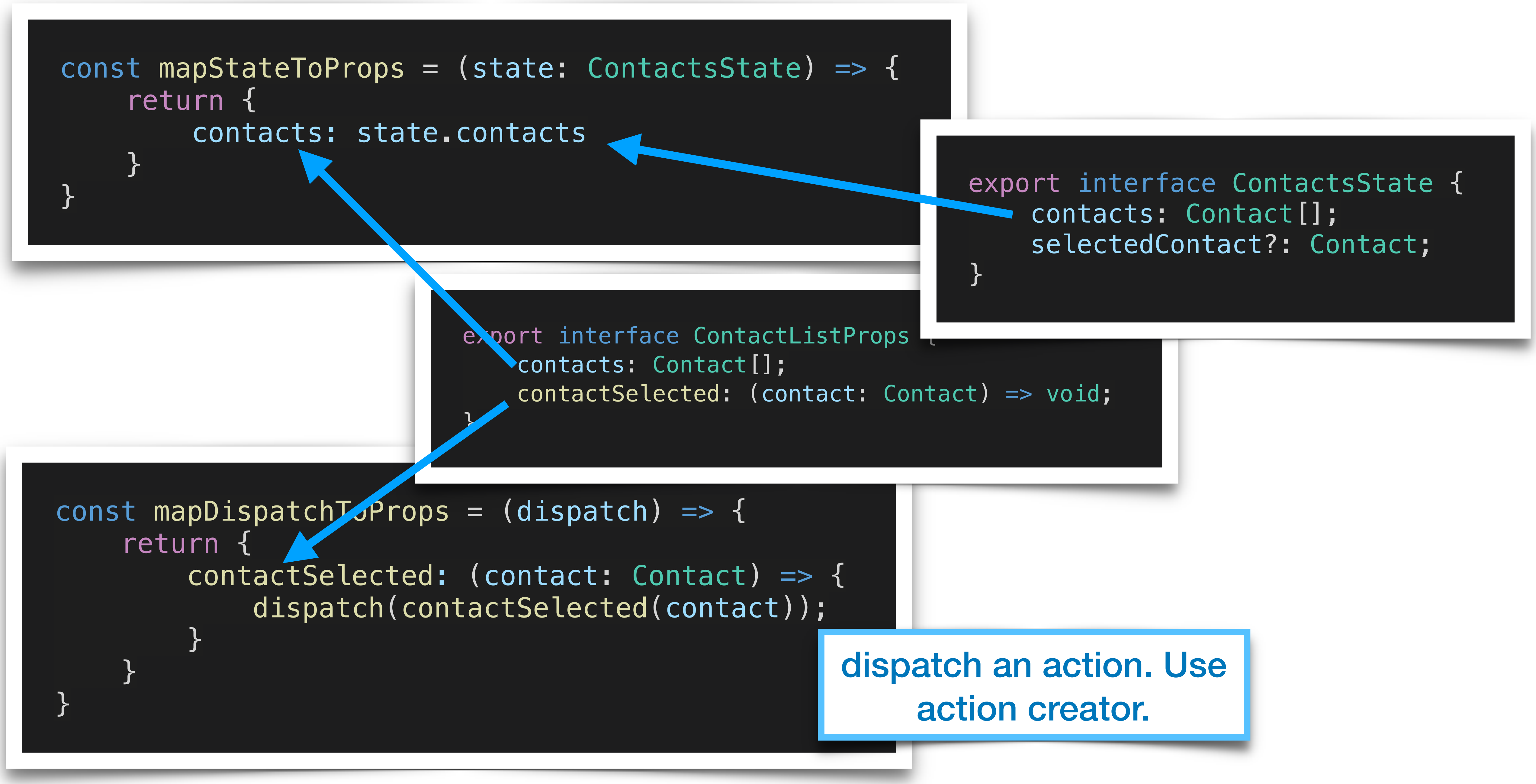
- It does not modify the component class passed to it; instead, it returns a new, connected component class that wraps the component you passed in.

Container Component with connect()



```
import { connect } from 'react-redux';

// ... The ContactList Component (ommited here) ...

// ... Create a connected Wrapper Component (the Container Component)
export default connect(
    mapStateToProps, // map store state to the components properties
    mapDispatchToProps // map stores dispatch to the components properties
)(ContactList);
```

mapStateToProps() and mapDispatchToProps()



```
const mapStateToProps = (state: ContactsState) => {
    return {
        contacts: state.contacts
    }
}
```

```
export interface ContactsState {
    contacts: Contact[];
    selectedContact?: Contact;
}
```

```
export interface ContactListProps {
    contacts: Contact[];
    contactSelected: (contact: Contact) => void;
}
```

```
const mapDispatchToProps = (dispatch) => {
    return {
        contactSelected: (contact: Contact) => {
            dispatch(contactSelected(contact));
        }
    }
}
```

dispatch an action. Use action creator.

Presentational and Container Components

„Update from 2019: I wrote this article a long time ago and my views have since evolved. In particular, I don't suggest splitting your components like this anymore. […]

Hooks let me do the same thing without an arbitrary division. […]"

Described by Dan Abramov:
https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

React Redux and Hooks

- React Redux now offers a set of hook APIs as an alternative to the existing *connect()* Higher Order Component.

- These APIs allow you to subscribe to the Redux store and dispatch actions.

- The most common used hooks are:

    - *useSelector()*

    - *useDispatch()*

Redux Form

„The best way to manage your form state in Redux.“

– *redux-form.com*

Why Redux Form?

- Creating Forms with Controlled Components leads to very verbose (boilerplate) code.
- The HTML5 built-in validation is not sufficient enough, usually a custom strategy/ implementation is required. But…
- Managing the state of each input control as well as the state of the complete form is very error-prone.
- It would be nice to keep the current state of each form in the Redux store.

Magic needed!

Redux Form - Building Parts

- To connect your React form components to your Redux store you'll need the following pieces from the ***redux-form*** package:

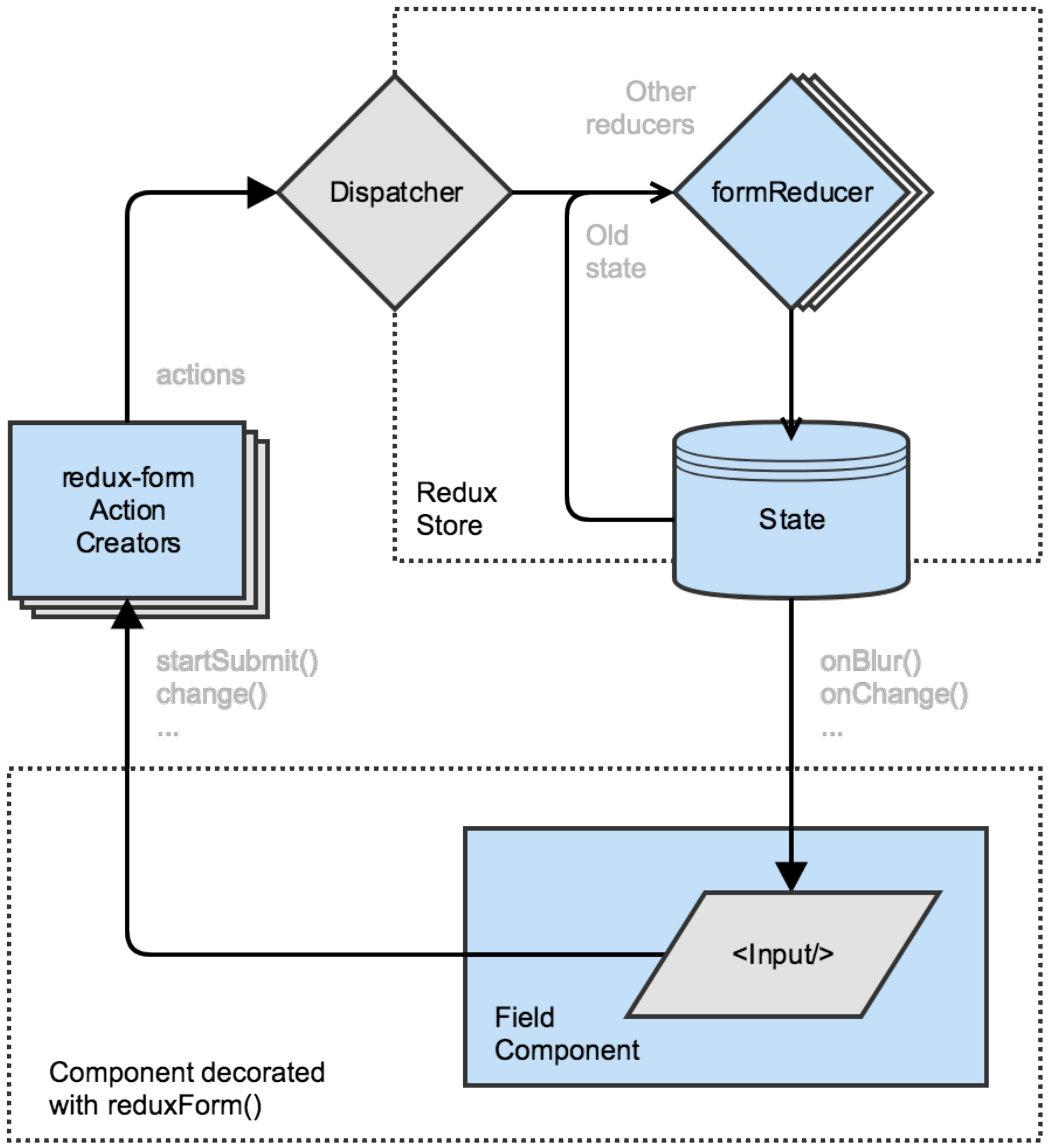
    - Redux Reducer: ***formReducer***

    - React HOC: ***reduxForm()***

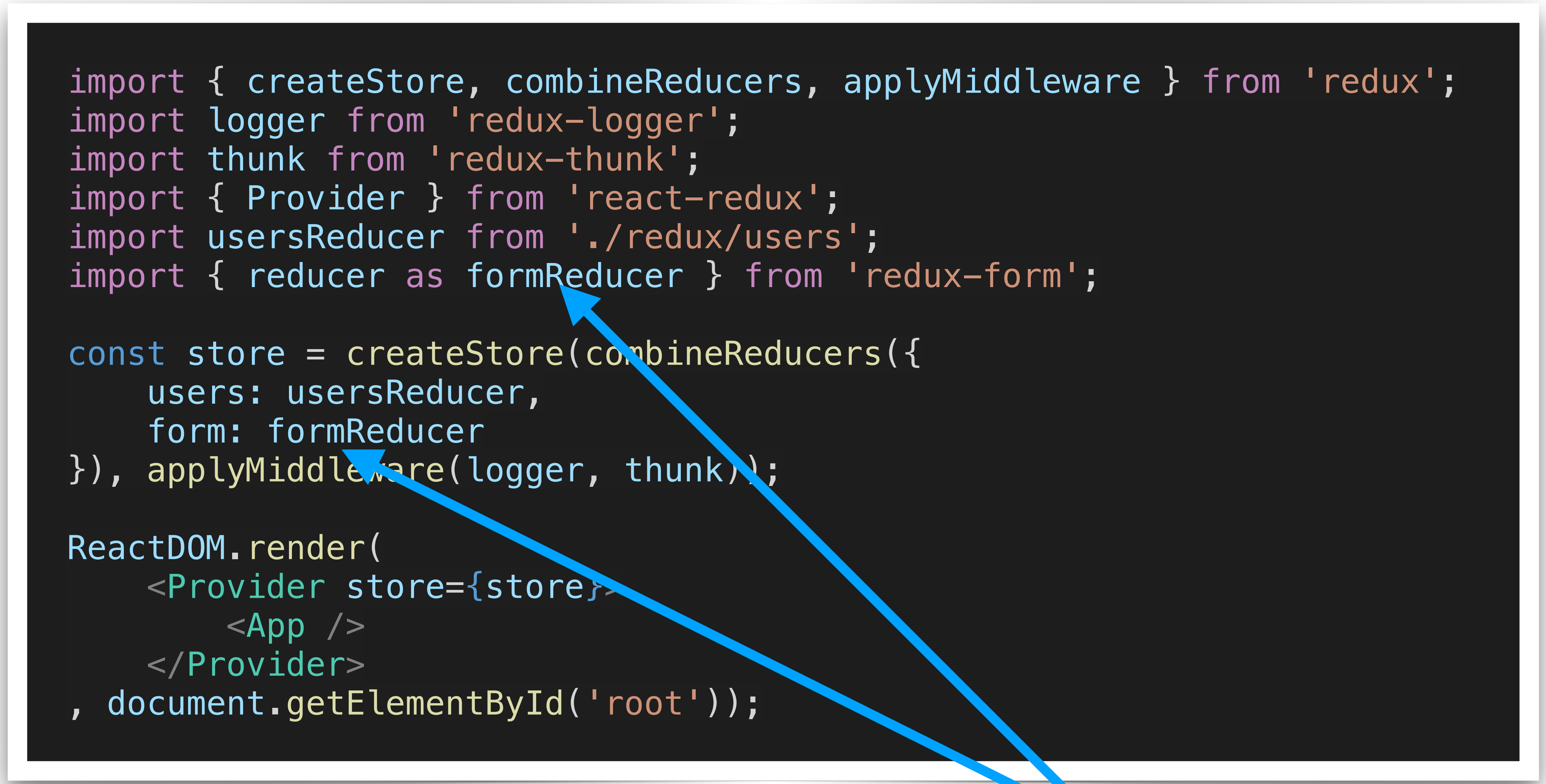
    - Component: ***<Field/>***

```
> npm install redux-form
```

Redux Form - Data Flow



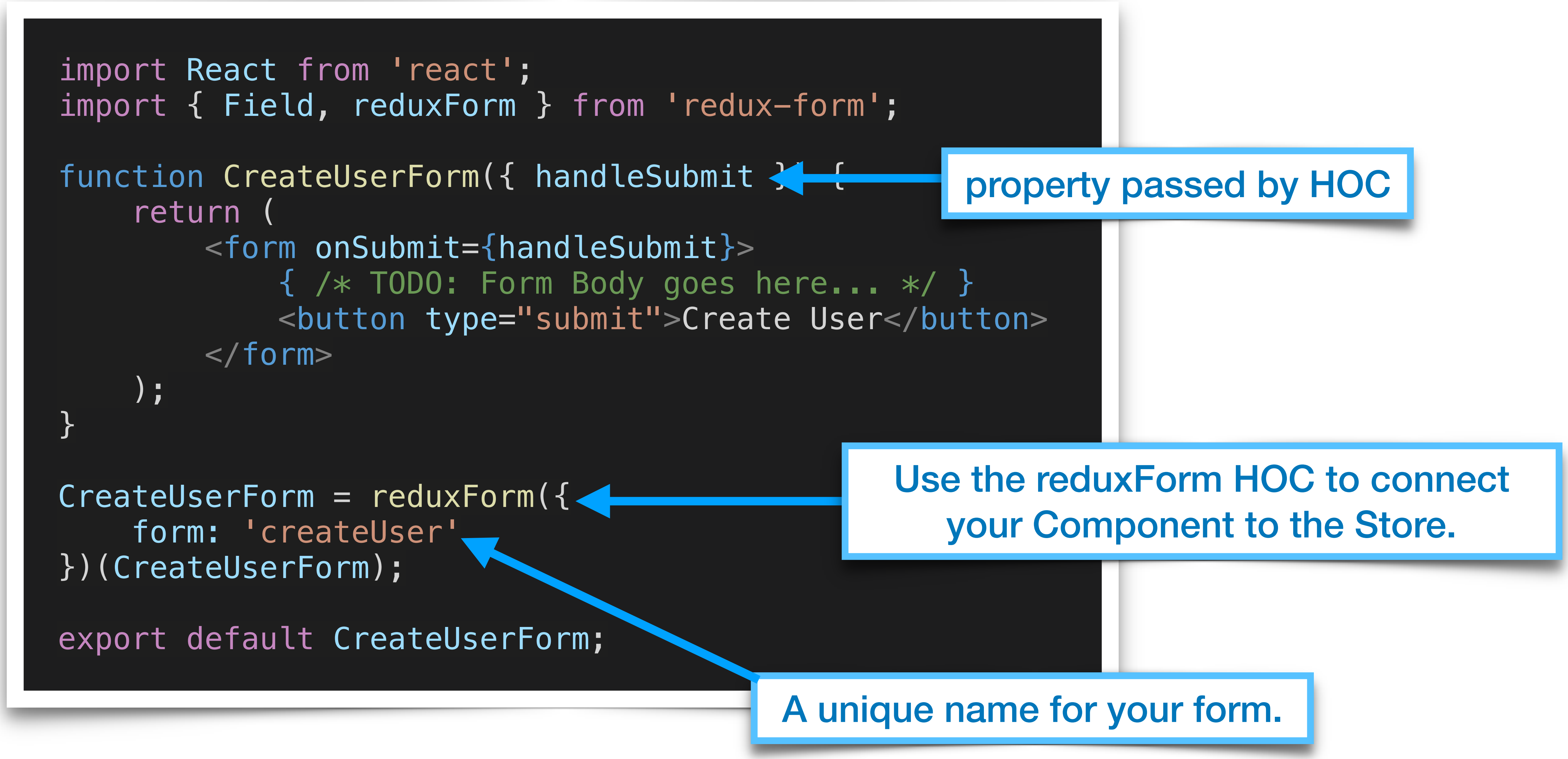
Redux Form - Form Reducer



```
import { createStore, combineReducers, applyMiddleware } from 'redux';
import logger from 'redux-logger';
import thunk from 'redux-thunk';
import { Provider } from 'react-redux';
import usersReducer from './redux/users';
import { reducer as formReducer } from 'redux-form';

const store = createStore(combineReducers({
    users: usersReducer,
    form: formReducer
}), applyMiddleware(logger, thunk));

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>
, document.getElementById('root'));
```

Add the ***formReducer*** from the ***redux-form*** module to your store. Use ***combineReducers()***.

Redux Form - reduxForm() HOC



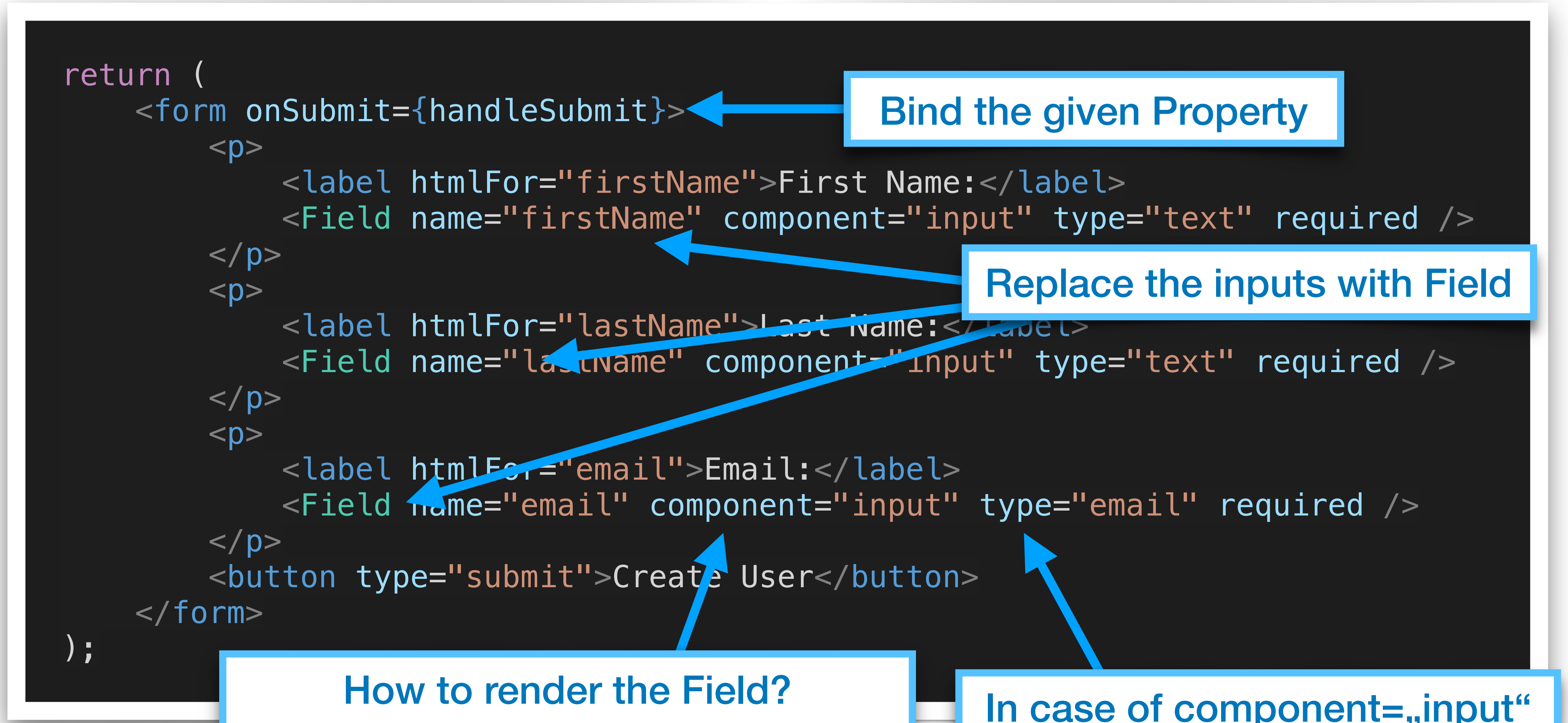
```
import React from 'react';
import { Field, reduxForm } from 'redux-form';

function CreateUserForm({ handleSubmit }) {
    return (
        <form onSubmit={handleSubmit}>
            { /* TODO: Form Body goes here... */ }
            <button type="submit">Create User</button>
        </form>
    );
}

CreateUserForm = reduxForm({
    form: 'createUser'
})(CreateUserForm);

export default CreateUserForm;
```

property passed by HOC

Use the reduxForm HOC to connect your Component to the Store.

A unique name for your form.

Redux Form - Field Component



```
return (
    <form onSubmit={handleSubmit}>
        <p>
            <label htmlFor="firstName">First Name:</label>
            <Field name="firstName" component="input" type="text" required />
        </p>
        <p>
            <label htmlFor="lastName">Last Name:</label>
            <Field name="lastName" component="input" type="text" required />
        </p>
        <p>
            <label htmlFor="email">Email:</label>
            <Field name="email" component="input" type="email" required />
        </p>
        <button type="submit">Create User</button>
    </form>
);
```

Bind the given Property

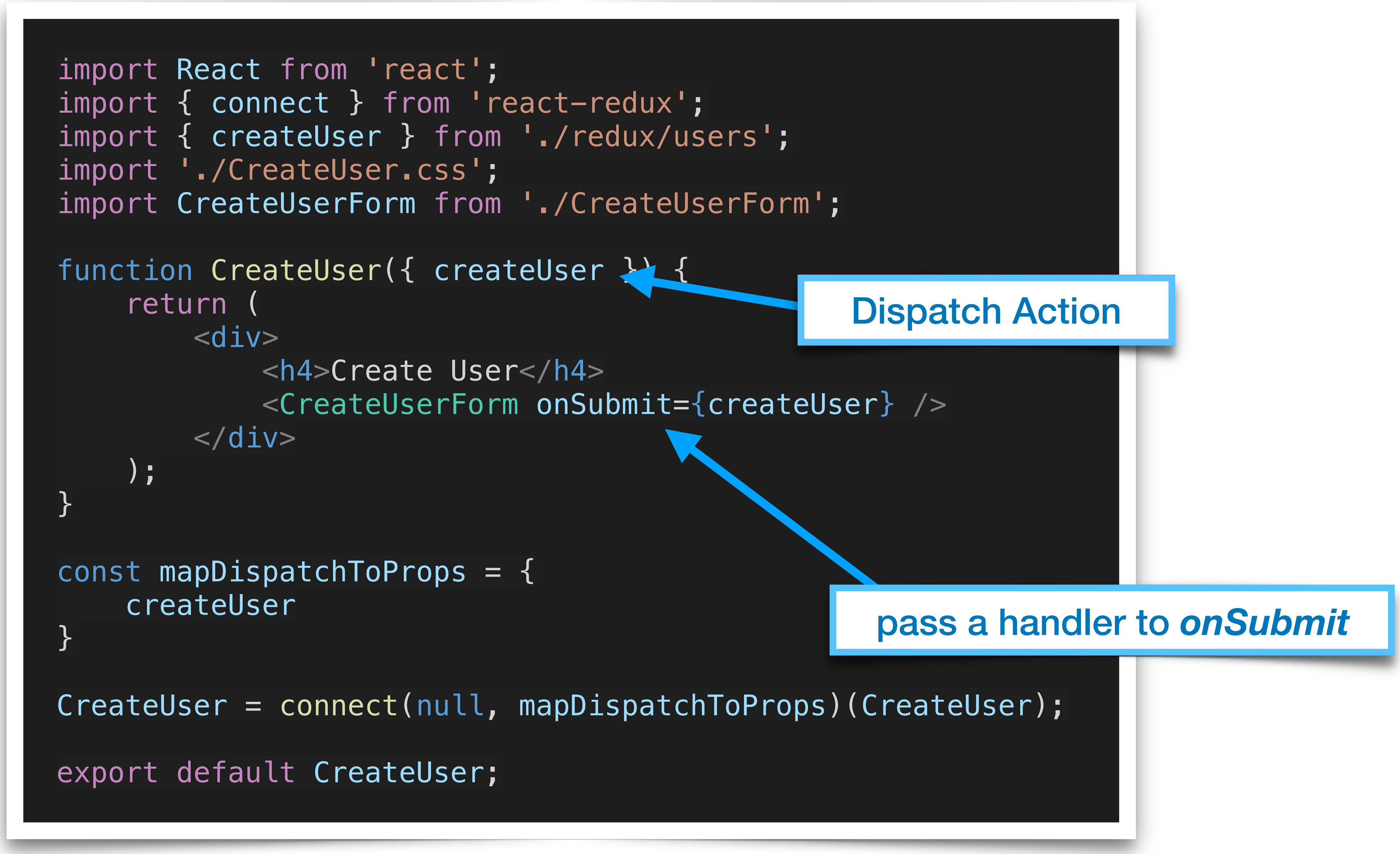
Replace the inputs with Field

How to render the Field?
input, select, textarea, …

Or Custom Component

In case of component=„input“

Redux Form - Reacting to submit



```
import React from 'react';
import { connect } from 'react-redux';
import { createUser } from './redux/users';
import './CreateUser.css';
import CreateUserForm from './CreateUserForm';

function CreateUser({ createUser }) {
    return (
        <div>
            <h4>Create User</h4>
            <CreateUserForm onSubmit={createUser} />
        </div>
    );
}

const mapDispatchToProps = {
    createUser
}

CreateUser = connect(null, mapDispatchToProps)(CreateUser);

export default CreateUser;
```