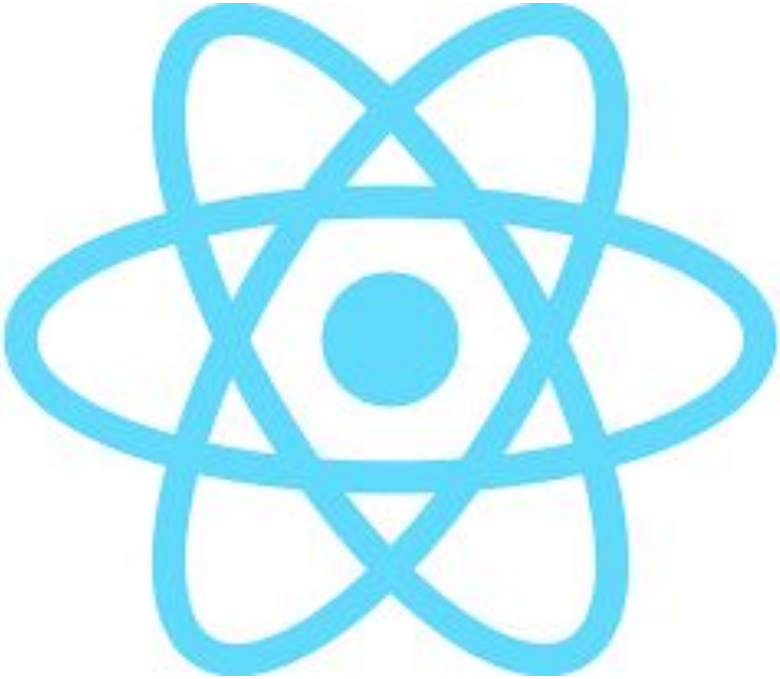
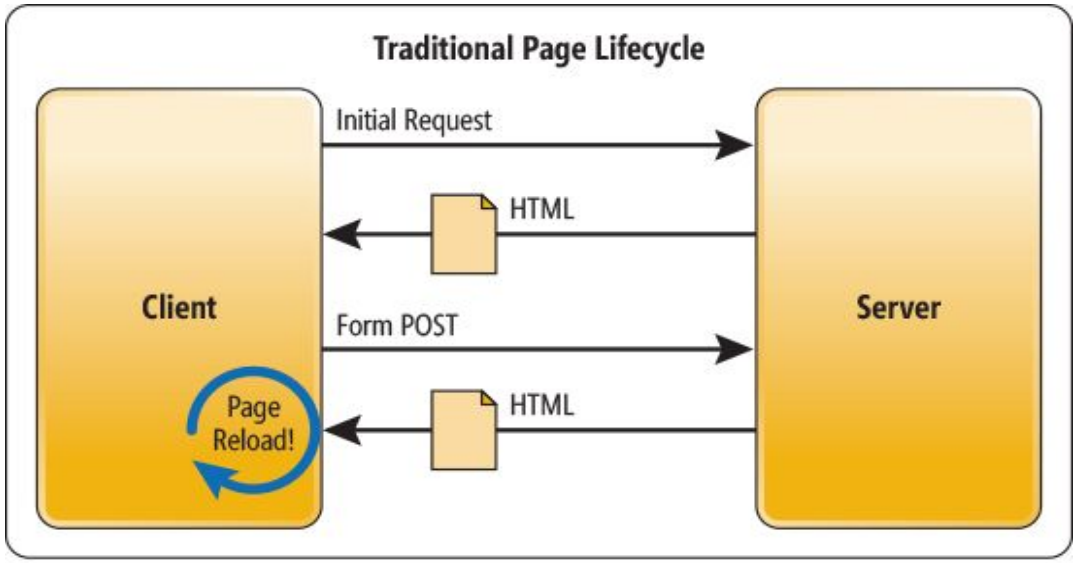


React - Part 1
(of 4)

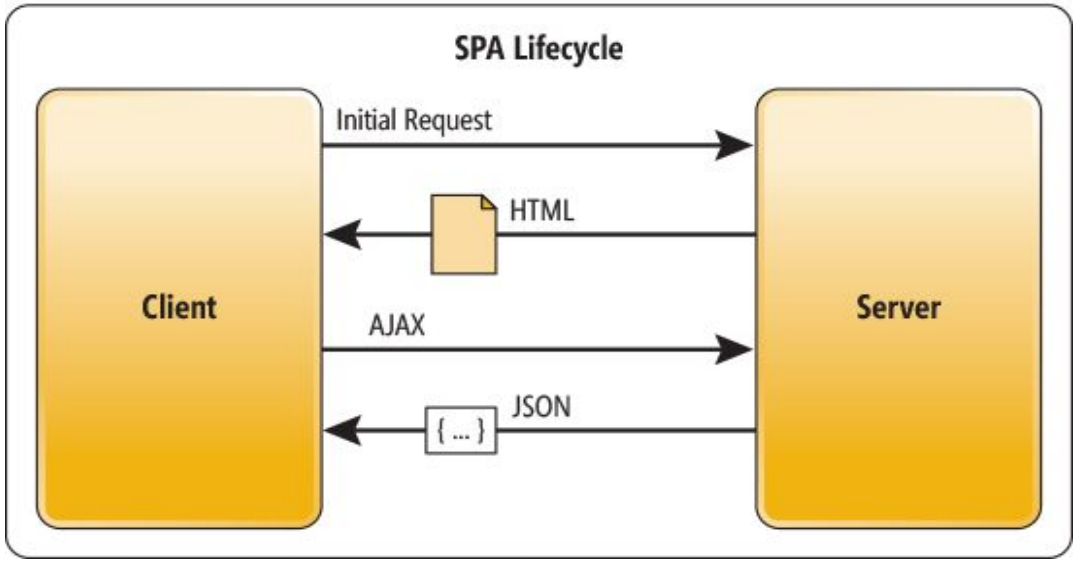
Contents

1. Javascript frameworks
2. Resources
3. Typescript
4. Components
5. JSX
6. State
7. How react works

Where is the UI constructed?



Server side rendering
e.g. Spring Thymeleaf

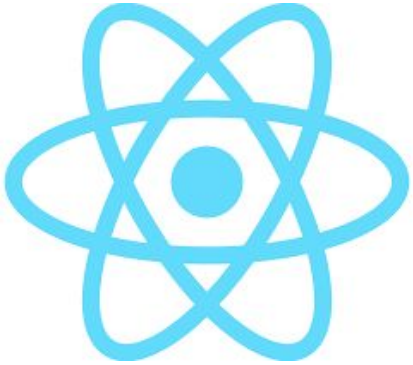


Client side rendering
‘Single **P**age **A**pplication
e.g. React

1. Javascript frameworks

There are

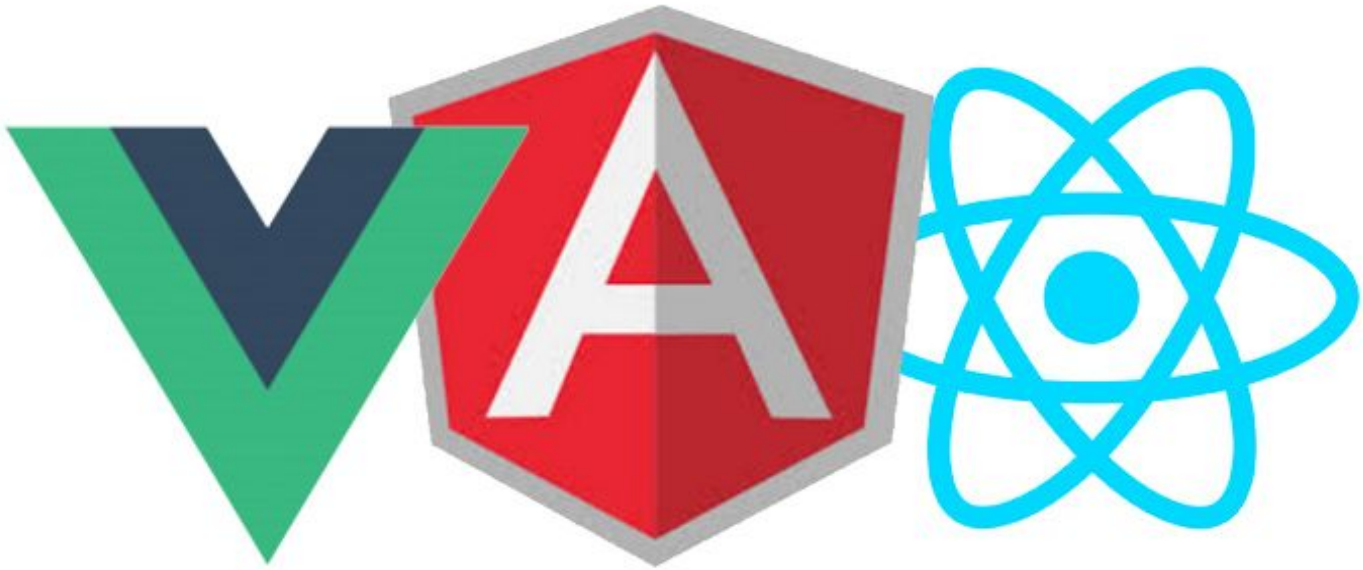
A lot of frameworks..



SVELTE

...

The big 3...



Vue **Angular** **React**

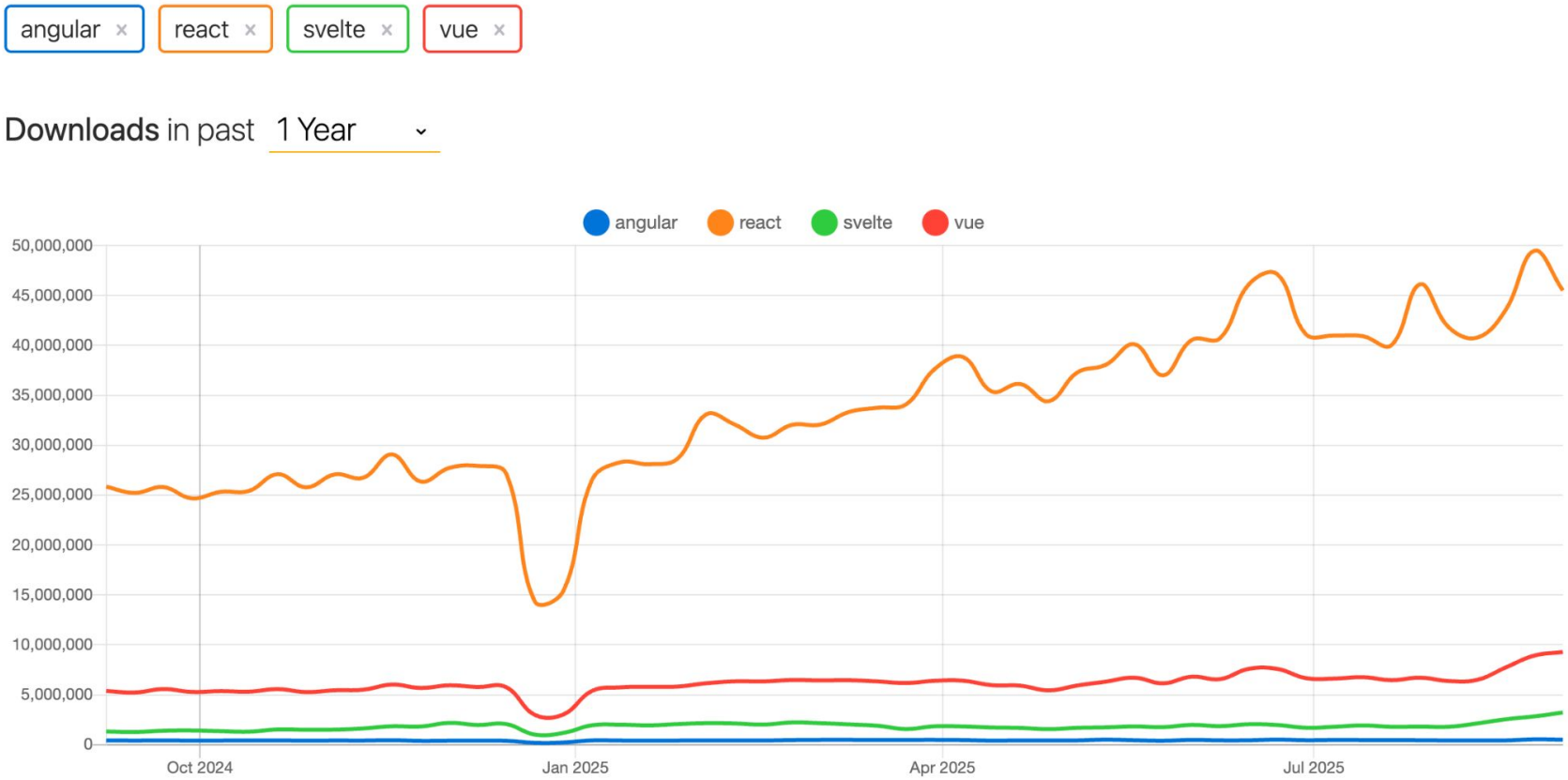
\+ **Or 'The big 4'...** SVELTE

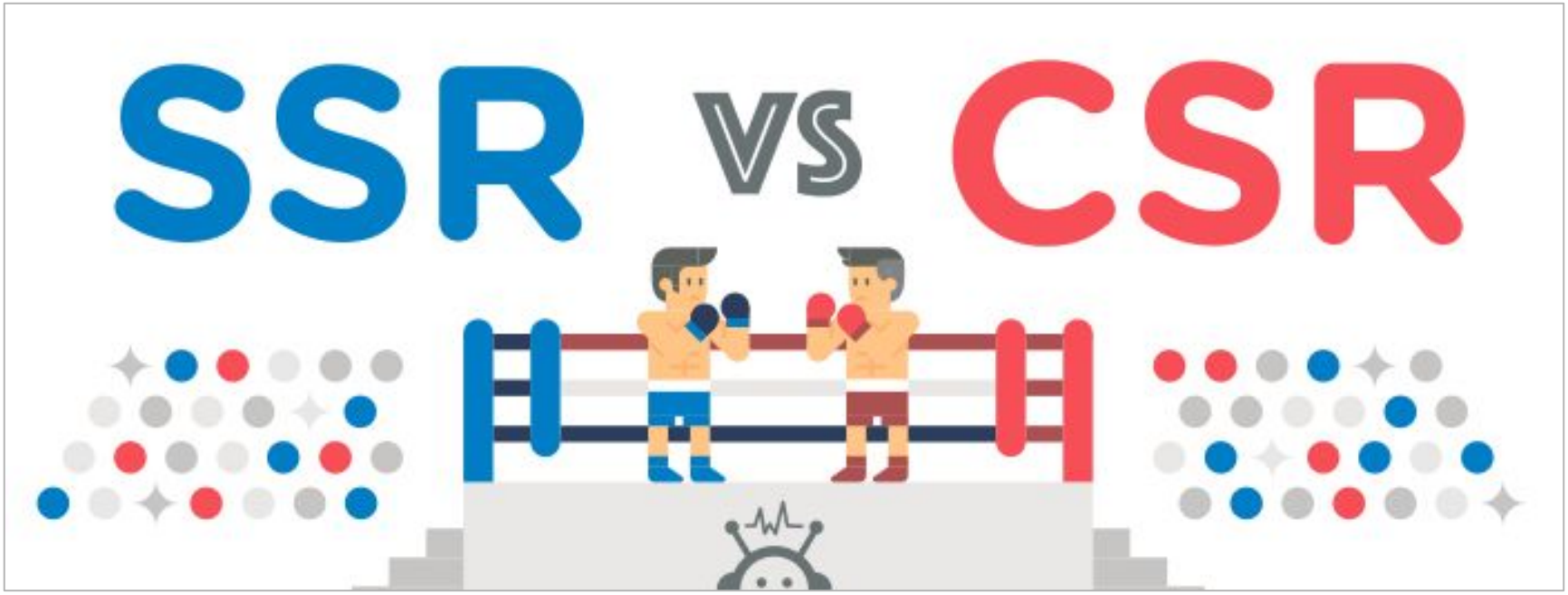
Concepts (data binding, components, props, state, routing,...) are **VERY similar**. **If you know one** of these frameworks, you can very **easily learn another**

Why we use React?

It improves your Typescript & ‘pure’ development skills
(few ‘special constructs’ ….,‘euh apart from *useEffect()* maybe 😅’)

It is very **popular** (large community, lots of tooling, extra’s,…)





Does every project need React/Vue/...?

There is nothing wrong with traditional server-side generated HTML pages

Use a JS frontend framework if

- you want to separate concerns (**single responsibility on architecture level**)
- you need an optimal UX

2. Resources

https://react.dev/learn

Search ⌘ K

Learn Refe

GET STARTED

Quick Start

Tutorial: Tic-Tac-Toe

Thinking in React

Installation

LEARN REACT

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

Is this page useful?

LEARN REACT >

Quick Start

Welcome to the React documentation! This page will give you an introduc
80% of React concepts that you will use on a daily basis.

You will learn

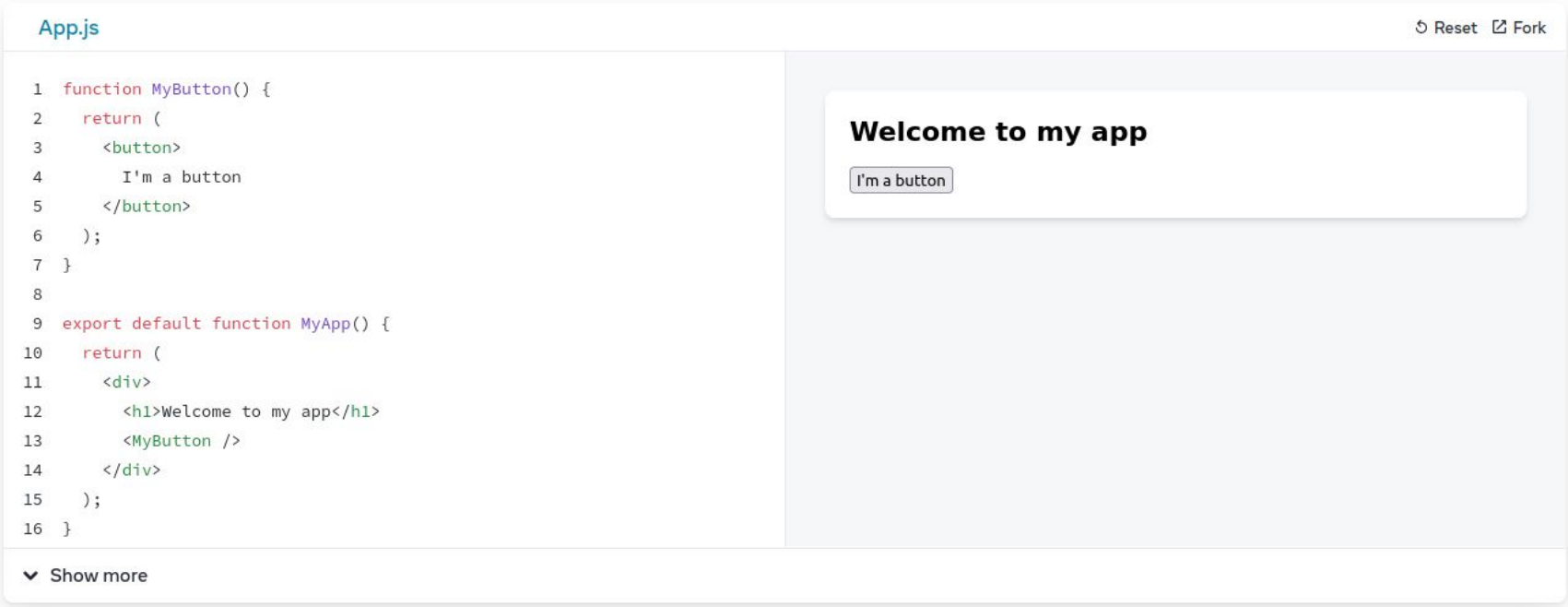
- How to create and nest components
- How to add markup and styles
- How to display data
- How to render conditions and lists
- How to respond to events and update the screen
- How to share data between components

Creating and nesting components

React apps are made out of *components*. A component is a piece of the UI (user inter

https://react.dev/learn

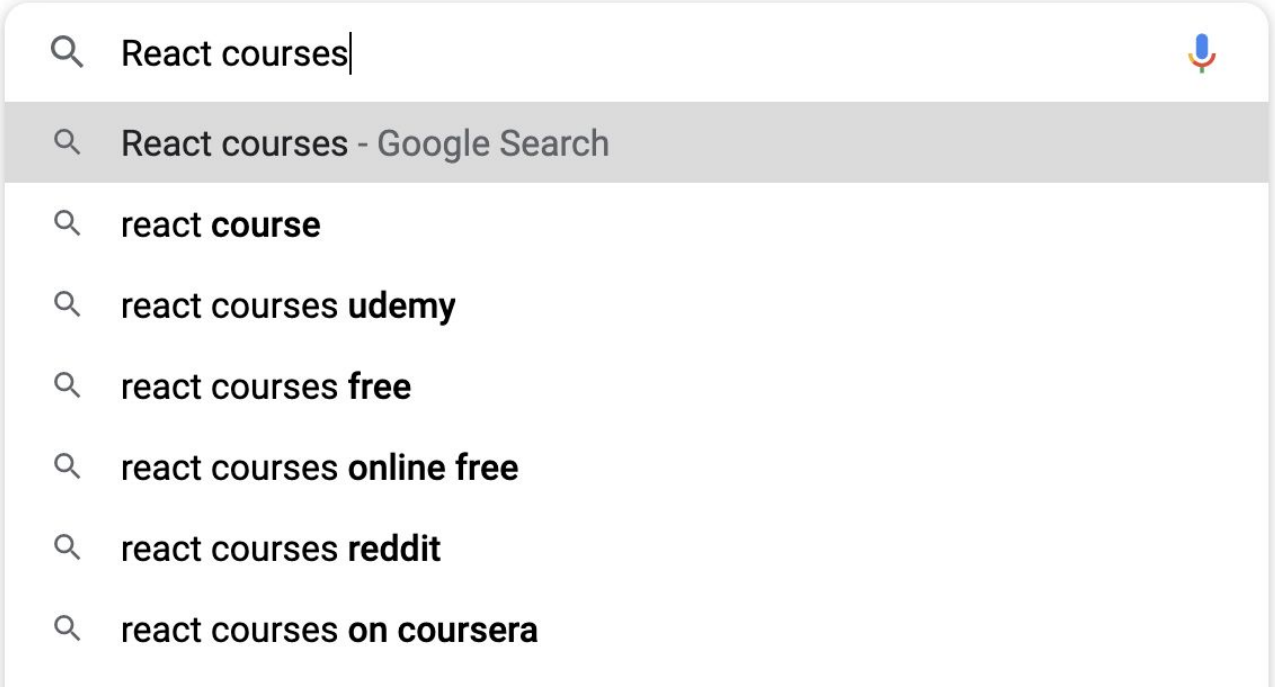
In the docs, live code editors allow you to practice what you have just read in the text!



At the end of most chapters, there are challenges you can try out.

Running through the Tic-Tac-Toe tutorial is highly recommended

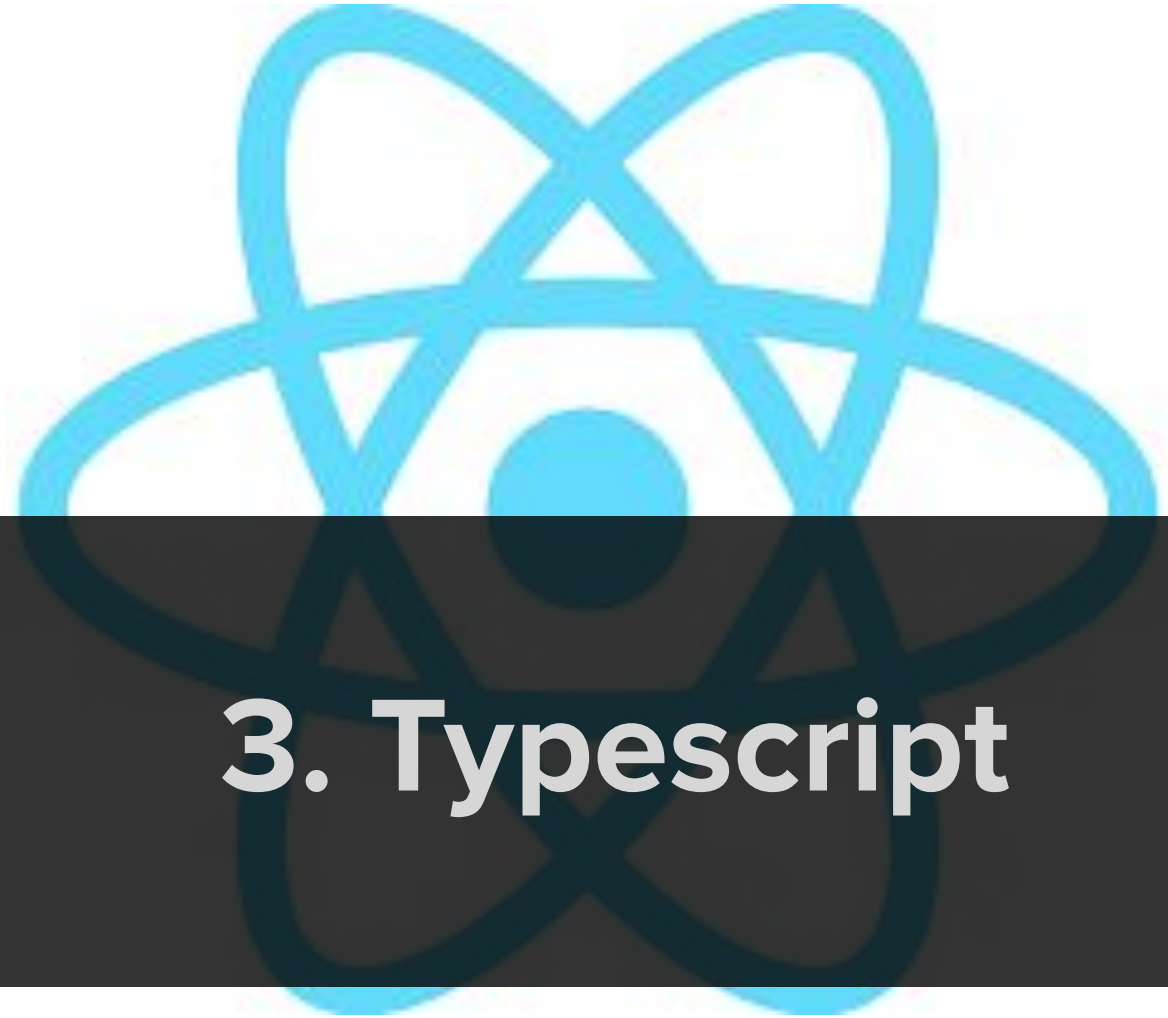
There's a wide range of (free) courses, e-books and tutorials available on the web...
pay attention only choose tutorials that work with **functional components and hooks**



+ numerous blogs, articles etc... like https://thisweekinreact.com/, https://overreacted.io/ ,...

Use **with care**
Use AI tools mainly to **explain code**
Code and refactor mostly yourself
Understand all the **code** you deliver
Be able to **make adjustments to code without AI tools**

During the exam, **live coding** may be asked on your project, this means you have to add a small feature to it in a limited time.



3. Typescript

Typescript

- https://www.typescriptlang.org/
- Compiled to JS
- Strongly typed
- Interfaces
- Type inference

De-facto standard for (serious) web development

Compiled

Browsers do not understand TS, it needs to be compiled to JS

- “tsc” ➜ typescript compiler

Configuration: tsconfig.json

- Used by the compiler
- Default is usually provided by tooling (vite,...)
- **Most of the time no need to change it**

Strongly typed

test.js

```
function addFive(num) {
    return num + 5
}

const result = addFive( num: 'Hello')
console.log(result) // Hello5
```

runtime error (Hello5)

test.ts

```
function addFive(num: number) {
    return num + 5
}

const result = addFive( num: 'Hello')
console.log(result)
```

TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

Change addFive() signature ⌥⇧↩ More actions... ⌥↩

compile error

Type inference

Typescript tries to be as smart as possible to **detect the type**.

You typically only need to specify explicit types in these cases

- When you define interfaces or types
- For function arguments
- When you call a function that returns a generic

```
let x = [0, 1, null];
```

```
let x: (number | null)[]
```

https://www.typescriptlang.org/docs/handbook/type-inference.html

Type inference

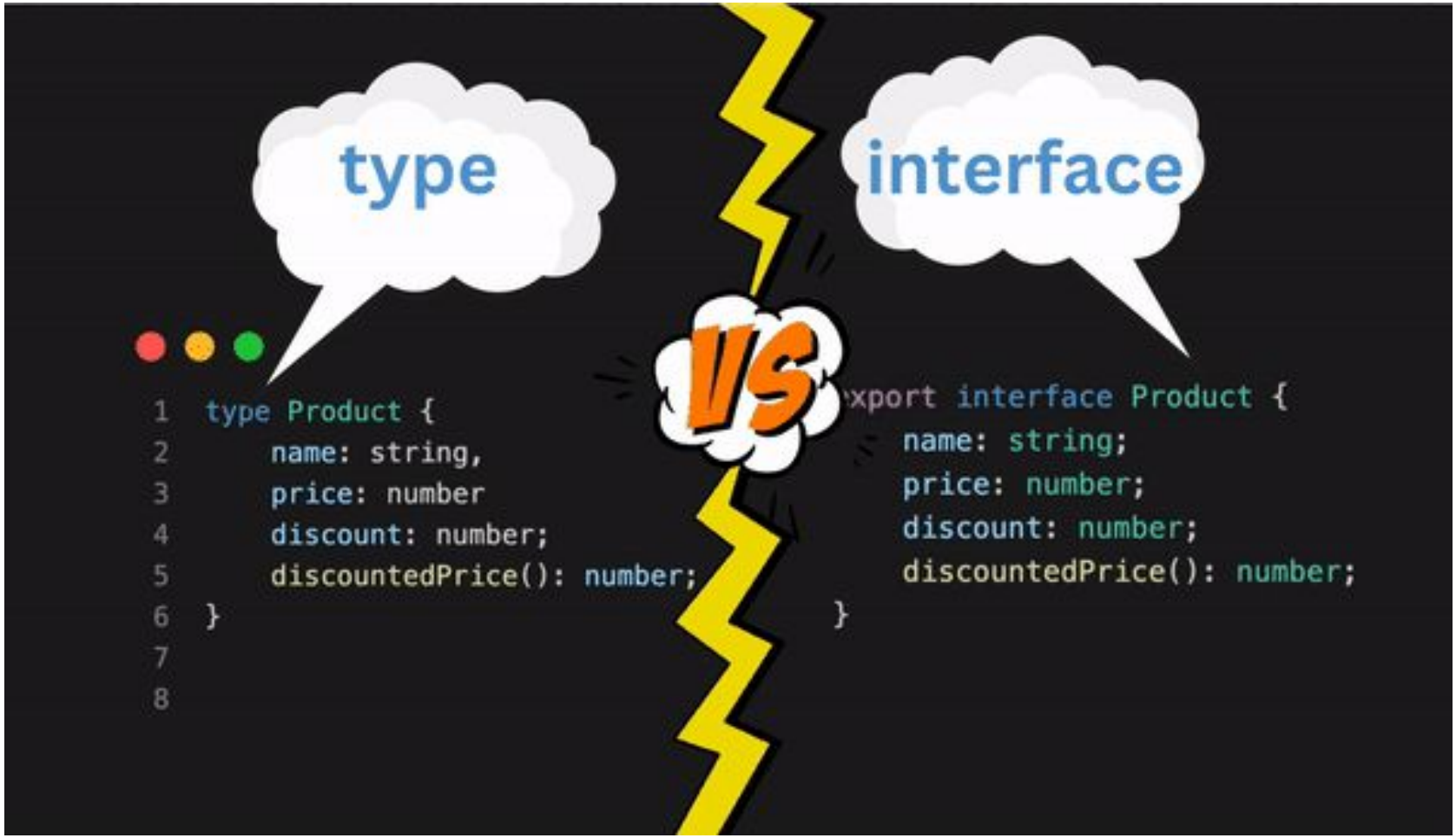
Typescript **analyses your assignments & control statements** (if, return,...) to determine what a type can be at a certain point in your code

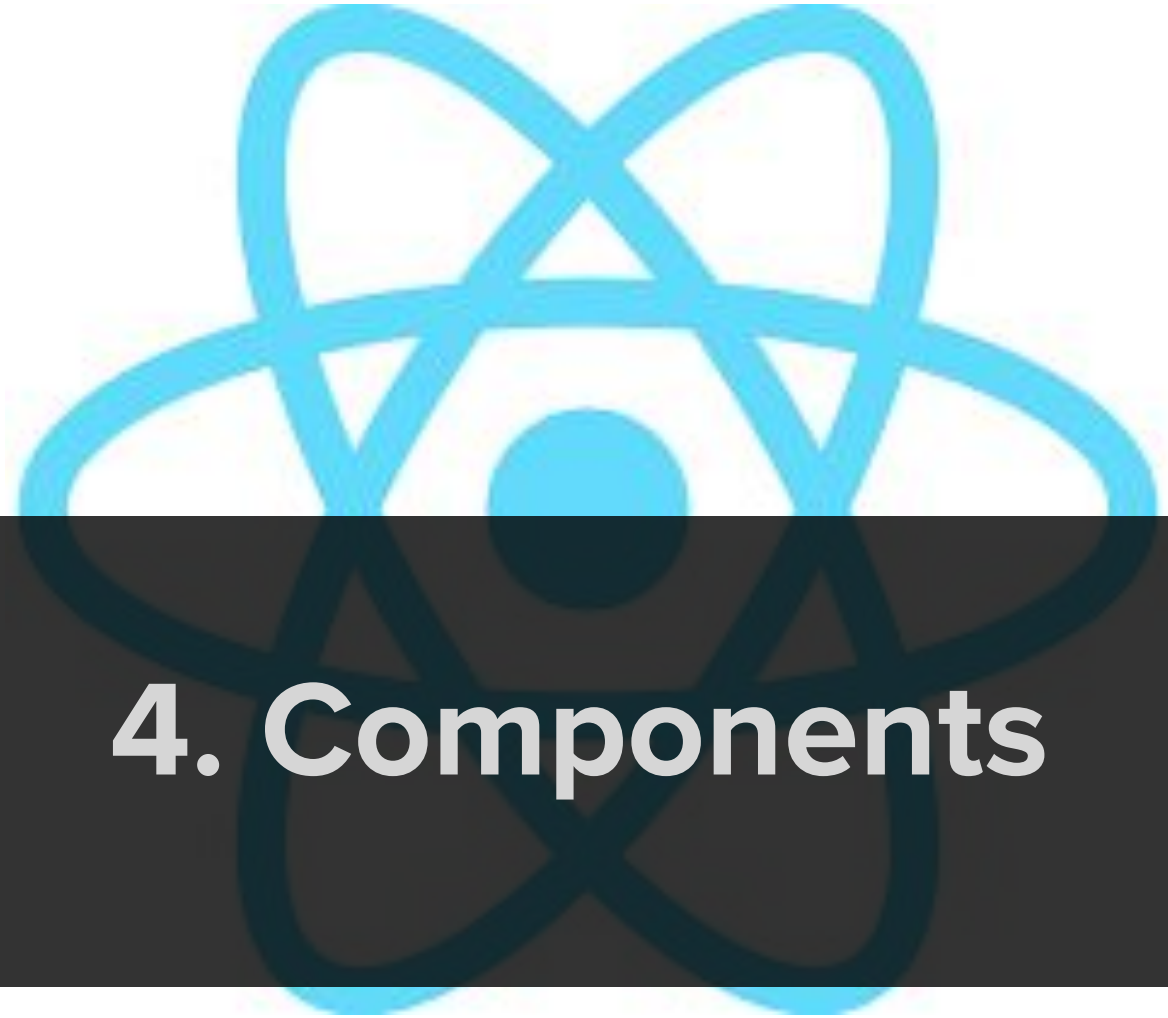
```
function example() {
  let x: string | number | boolean;

  x = Math.random() < 0.5;

  console.log(x);
              let x: boolean

  if (Math.random() < 0.5) {
    x = "hello";
    console.log(x);
                let x: string
  } else {
    x = 100;
    console.log(x);
                let x: number
  }

  return x;
         let x: string | number
}
```

https://www.typescriptlang.org/docs/handbook/2/narrowing.html#control-flow-analysis

Interfaces vs Types

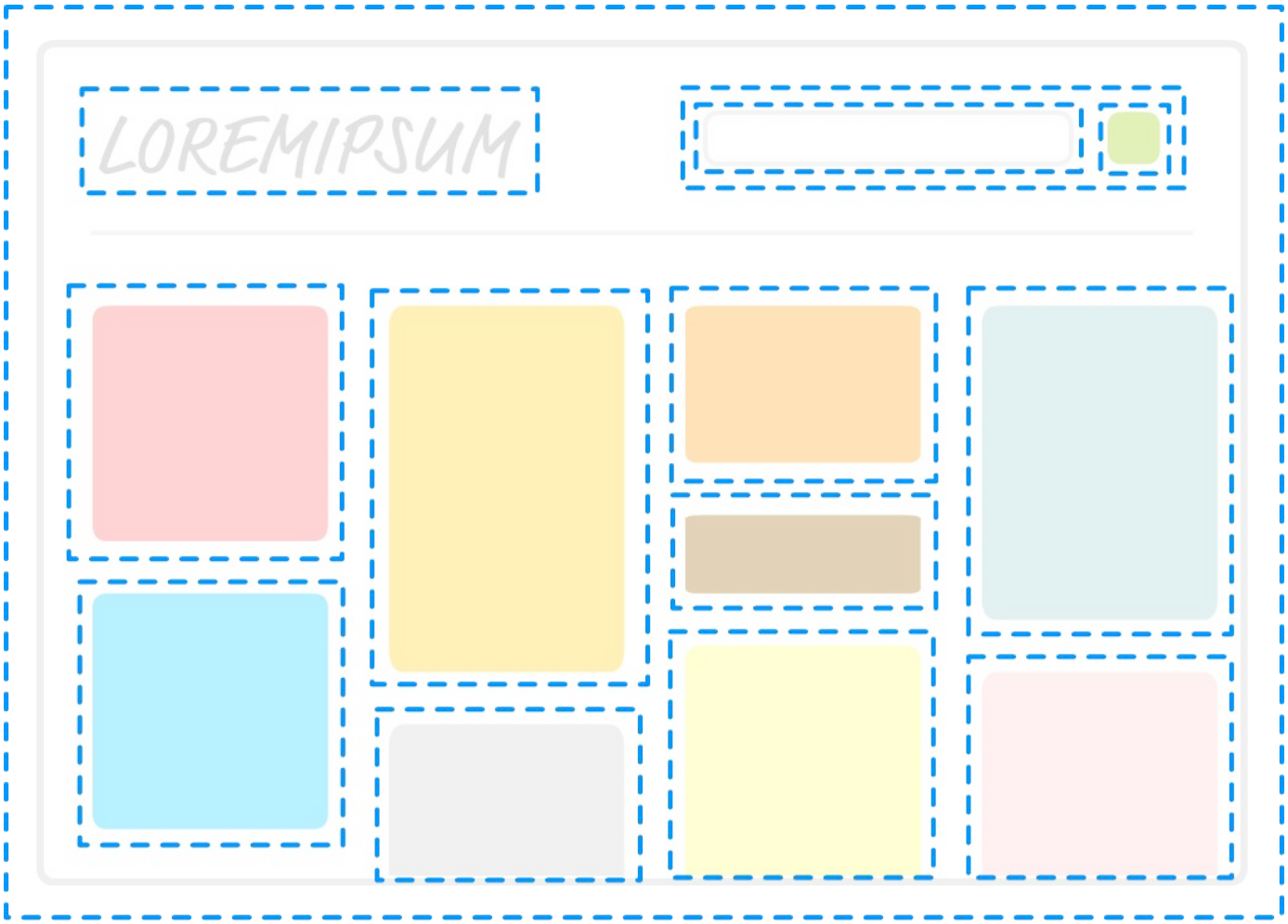
There are two syntaxes to ‘define’ things. Both have (almost) the same functionality…we will use them both in our examples





4. Components

Divide the UI into 'pieces' (= components) with their own responsibility



The UI is built from a collection of (reusable) components

A (react) component is a **function** with 0 or 1 parameter ('props') that **returns a 'JSX' element**

```
function MyComponent() {
   return <div>Hello world</div>
}
```

A component can be used in another component like an HTML (actually JSX...) element

```
<MyComponent />
```

https://react.dev/learn#components

You can 'pass' **data** (Object, Array, String, ...) to a component (from 'parent' to 'child') as an attribute

```
<MyComponent textToShow=”Hello world” />
```

This data enters the component using the **‘props’** parameter

```
function MyComponent(props) {
    <div>{props.textToShow}</div>
}
```

Or with **destructuring** (very common):

```
function MyComponent({ textToShow }) {
   <div>{textToShow}</div>
}
```

https://react.dev/learn/passing-props-to-a-component

Destructuring

this is the same as: `const name = user.name;`

Typescript

Always use **Typescript** for your React code!

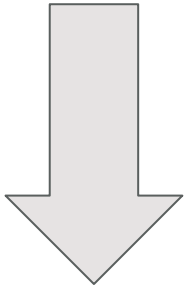
When defining a Component that has props: define an interface (or type) to specify the prop types.

```
interface MyComponentProps {
   textToShow: string
}

function MyComponent({textToShow}: MyComponentProps) {
   return <div>{textToShow}</div>
}
```

Pure functions

*“A React component must act like a **pure function** with respect to it’s props”*

same props IN



same JSX OUT

https://react.dev/reference/react/StrictMode

Props are read-only

```
function MyComponentProps(props: MyComponentProps) {
    props.textToShow = "Hahaha I'm changing you!"
    return <props.textToShow}</div>
}
```

```
▶ Uncaught TypeError: Cannot assign to read only property 'textToShow' of object '#<Object>'
    at MyComponentProps (App.tsx?t=1726491289311:21:20)
    at renderWithHooks (react-dom_client.js?v=dcfd9f2b:11546:26)
    at updateFunctionComponent (react-dom_client.js?v=dcfd9f2b:14580:28)
    at beginWork (react-dom_client.js?v=dcfd9f2b:15922:22)
    at HTMLUnknownElement.callCallback2 (react-dom_client.js?v=dcfd9f2b:3672:22)
    at Object.invokeGuardedCallbackDev (react-dom_client.js?v=dcfd9f2b:3697:24)
    at invokeGuardedCallback (react-dom_client.js?v=dcfd9f2b:3731:39)
    at beginWork$1 (react-dom_client.js?v=dcfd9f2b:19763:15)
    at performUnitOfWork (react-dom_client.js?v=dcfd9f2b:19196:20)
    at workLoopSync (react-dom_client.js?v=dcfd9f2b:19135:13)
```

The ‘children’ prop

https://react.dev/learn/passing-props-to-a-component#passing-jsx-as-children

is a specific auto-passed prop that contains the elements **within a component's start and closing tag**

The component will render these elements as its children

```
<MyList>
    <li>item 1</li>
    <li>item 2</li>
</MyList>

function MyList({children}) {
    <ul>{children}</ul>
}
```

will result in

```
<ul>
    <li>item 1</li>
    <li>item 2</li>
</ul>
```

Commonly used in component frameworks... **less common in your own code**

Check the **children-demo** on Canvas

Events

React exposes all events available in HTML. Pass a callback function that should be executed when the event occurs

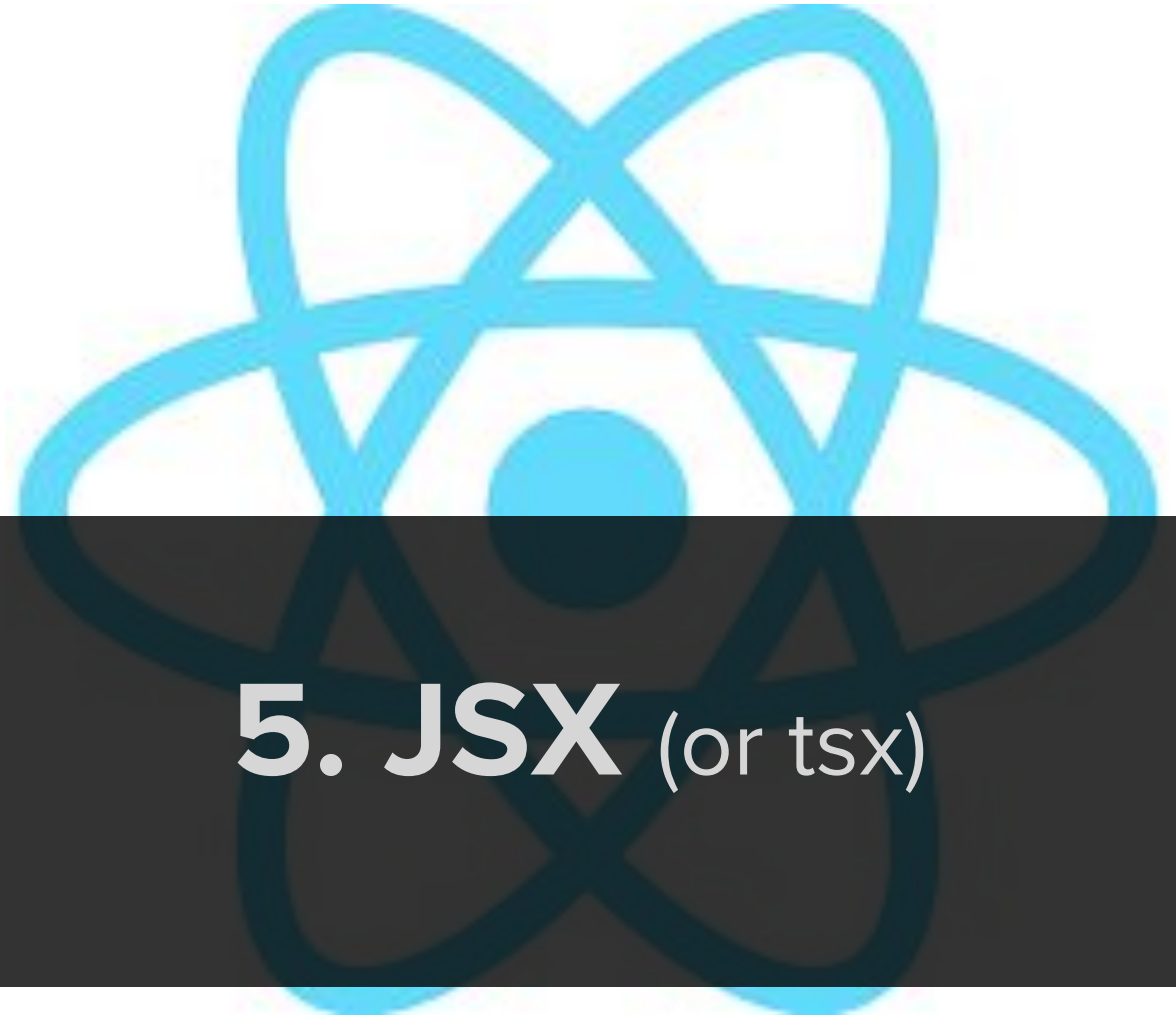
```
function MyComponent({textToShow}: MyComponentProps) {

    function handleClick() {
        console.log("I'm clicked!")
    }

    return (
        <div onClick={handleClick}>{textToShow}</div>
    )
}
```

https://react.dev/learn/responding-to-events#adding-event-handlers

Events and props

Pass a callback function as a prop to receive events in the parent of a component

```
interface MyComponentProps {
   textToShow: string
   showAlert: () => void
}

function MyComponent({textToShow, showAlert}: MyComponentProps){
   return <div onClick={showAlert}>{textToShow}</div>
}


function App() {
   return (
       <MyComponent
               textToShow="Hello"
               showAlert={() => alert("Hello")}/>
   )
}
```



5. JSX (or tsx)

const element = <h1>Hello, world!</h1>

Not HTML, Not a String

It's **JSX**

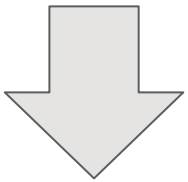
- "JavaScript XML"
- JSX is a Javascript syntax extension

Used to describe what the UI will look like (in each state)

- JSX creates HTML elements in the background using the _jsx function
- You can check this using for instance Babel: https://babeljs.io/repl/

https://react.dev/learn/writing-markup-with-jsx

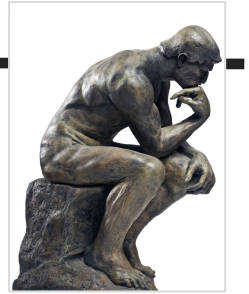
```jsx
<h1 className="title">Hello, world!</h1>
```



Transpile (esbuild, swc, babel,...)

```jsx
import { jsx as _jsx } from "react/jsx-runtime";

const el = _jsx("h1", {
  className: "title",
  children: "Hello, world!"
});
```

JSX = HTML in Javascript



So…'gone are the best practices (like MVC)?'
No! Gone is artificial separation between Javascript and HTML (and layouting CSS…)

High Cohesion and Single Responsibility

"Keep together what **changes together"**

Each **component** (part of the screen) has a specific **task** *('show a list of users')* When it **needs to change**, you will almost always touch **HTML and TS and** (layouting) **CSS**

Application-wide **CSS styling (= theming) IS** kept **separate**!

JSX

So we can use JSX (more or less) like we use HTML

```
<div>Static text</div>
```

But with a lot **more power**
Use {} to add javascript

```
export default function Avatar() {
  const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
  const description = 'Gregorio Y. Zara';
  return (
    <img
      className="avatar"
      src={avatar}
      alt={description}
    />
  );
}
```

JSX

JSX is Javascript (after compilation) so you can pass JSX as a parameter, as a return value, put it in a variable etc...

```
function MyComponent({input}) {
    return <div>{input}</div>
}

function MyApp() {
  const url = <a href="www.kdg.be">Kdg</a>
  return <MyComponent input={url}/>
}
```

JSX and CSS

'class' is a keyword in JS, so to add a CSS class to a JSX element we have to use 'className' in React:

```
<div className="myCSSClass">Text</div>
```

To to add a class conditionally (very common)

```
<div className={condition ? "cssClass1" : "cssClass2"}>Text</div>
```

With the ‘style’ attribuut you can pass *CSS* directly
(for instance if you need to calculate a CSS props)

```
<div style={{ fontWeight: 'bold' }}>Text</div>
```

Please note the double ‘{{‘: you pass an object

6. State

Imperative programming

Tell the software to perform **a sequence of steps** (an ‘algorithm’)

```
a = b + c;
d = doSomethingWith(a);
```

Declarative programming

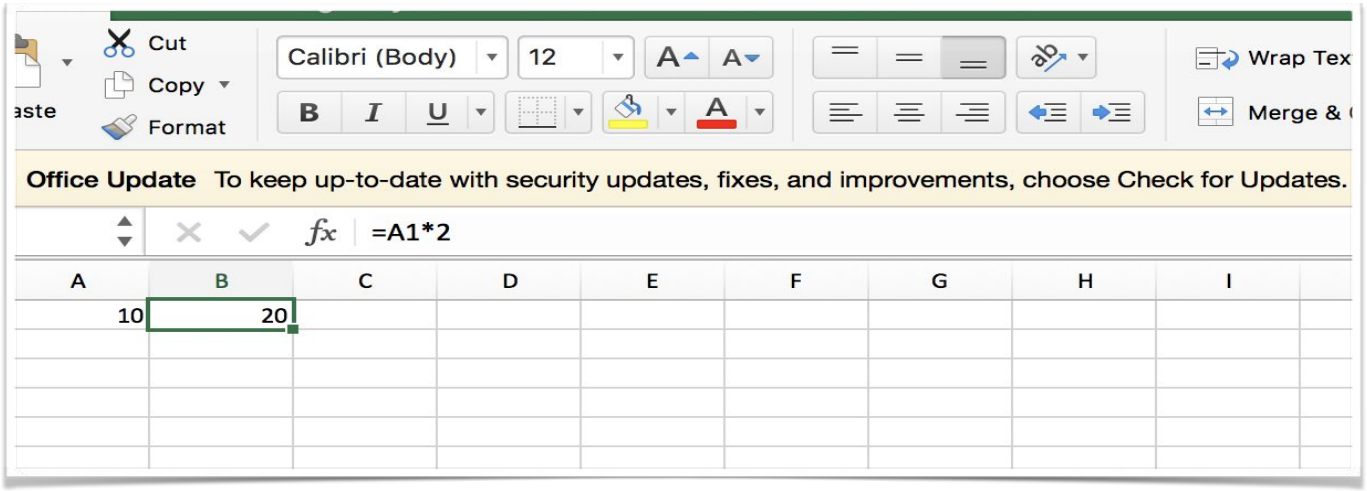
Describe what you want from the software

```
select * from t_users
where firstname = Leo
```

(SQL is declarative)

Reactive programming

Let the software react to changes



(like in a spreadsheet...one cell reacts to the changes in another cell)

React is a declarative + reactive framework

"Declarative": 'describe' the design for each **'state'** of your app

"Reactive": React will update and render the UI when **state** changes

Handling **state** is one of the most **complex tasks** in building IT systems

"A list of users", "A shopping basket"

Who owns it? (backend, database,...)
Where lives it? (database, backend, frontend, caches,...)
When does it **change**?
How to handle **replication**, **caching**, **concurrent access,...**

The UI on screen is a **function of the application's state**: *the data to show, the settings,...*

If the **state changes** (caused by user interaction, an API call that returns data...), the **UI adapts**

Recap

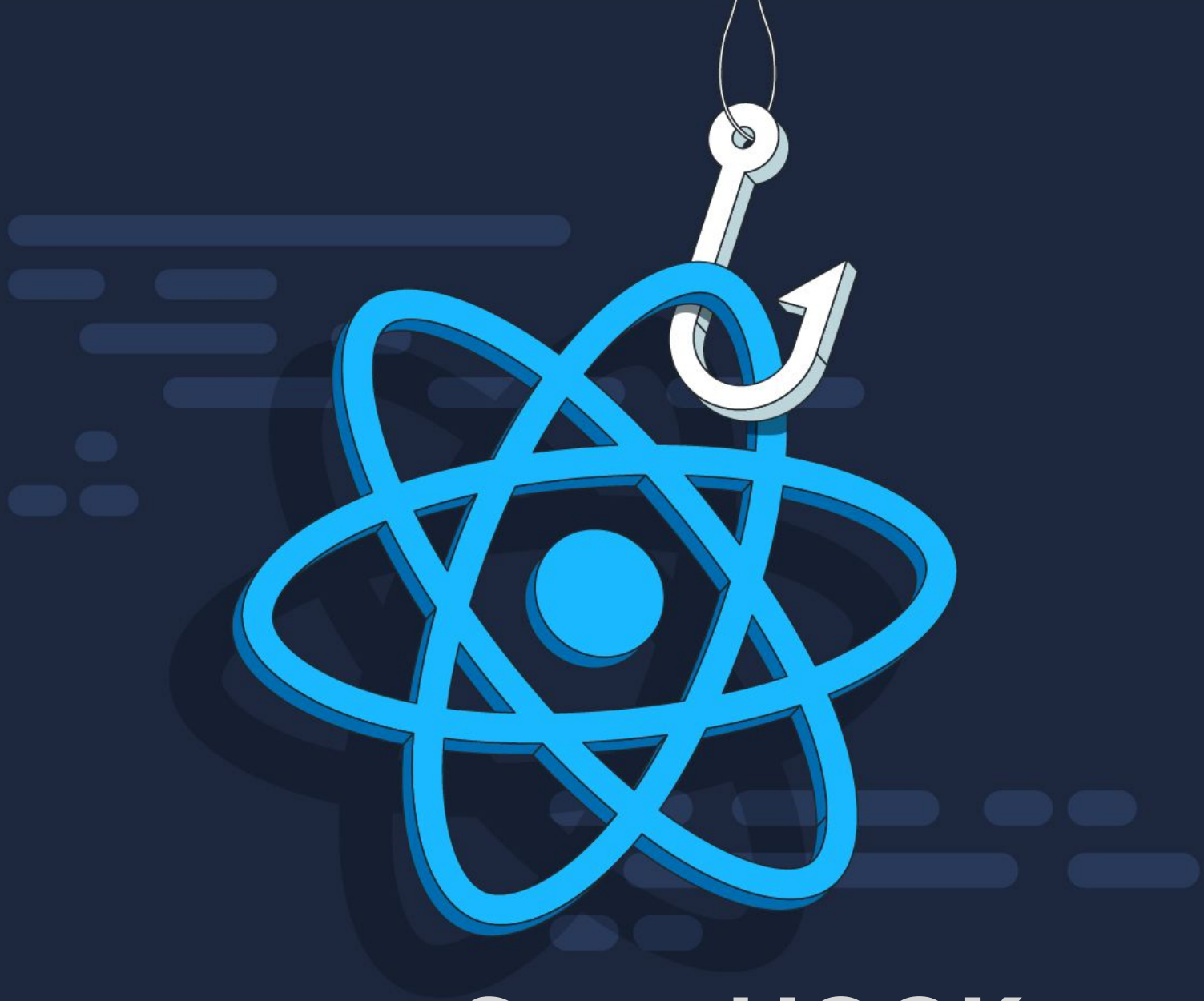
UI = f(state)

PROPS IN

JSX OUT

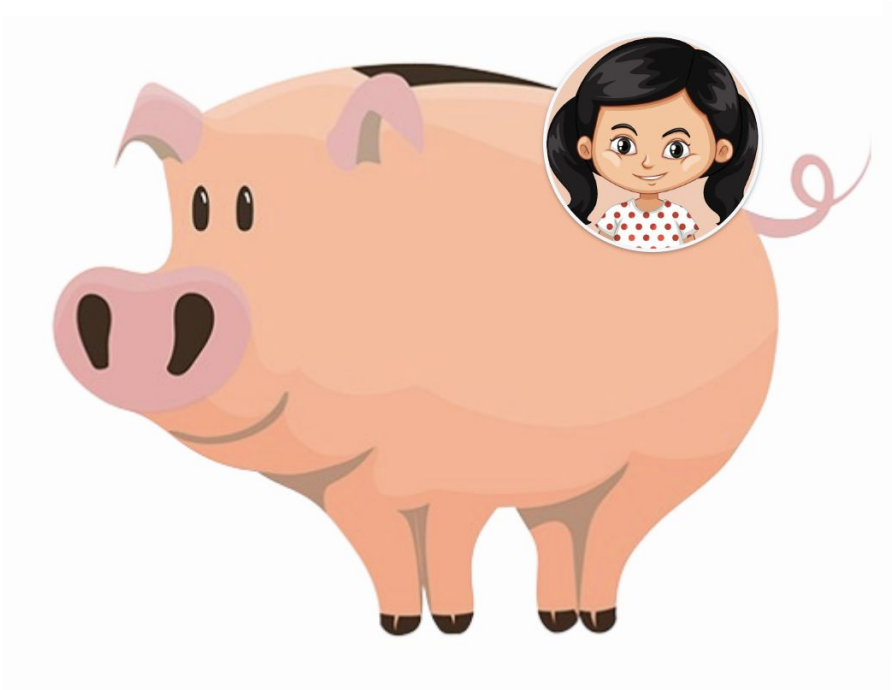
INTERNAL STATE

useState hooks



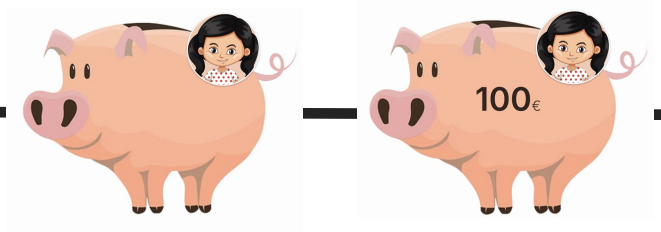
useState HOOK

This Piggybank component can be in two internal states



amount is **not shown**

amount is **shown**



The component can be in two boolean **'states'**
This is represented by following line in the component

```
const [showAmount, setShowAmount] = useState(false)
```

Initially **showAmount** will be **false**. When you call **setShowAmount** in the code, for instance in an onClick handler:

```
onClick={() => setShowAmount(!showAmount)}/>
```

the component function ***PiggyBank*** will be called again with **showAmount** having the value **true**
This way you can return the correct view for this state:

```
{showAmount &&
    <div className="fields">
        <div className="balance">{account.balance}</div>
    </div>
}
```

Conditional rendering using the && operator

Hooks are pieces of functionality that can be “attached” to a functional component (“Hooked into”).

When you want to **attach private state to a component** you can ‘use’ the useState hook.

```
const [value, setValue] =
  useState(startingState)
```

value: value of the state
setValue: function used to mutate the state.
startingState: value of the state at initialization (default = undefined)

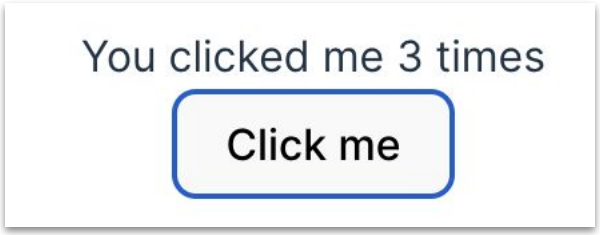
This is array destructuring, useState returns an array
- index 0: value
- index 1: setter function

You can choose the names of both the value and the function
(this not the case when destructuring an object!)

A change in the state (caused by a ‘setValue’ call) triggers a ‘re-render’ of the component that contains the useState hook

RERENDER = your component function is called again, with the new state value

By the way, this is one of our favourite exam questions:
’What happens when you call a state setter?’



re-render = **recall** the component function

```
export function UseStateDemo() {
    const [count, setCount] = useState(0)

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

Calling `setCount` when `count = 1` will schedule a recall of `UseStateDemo()` with `count` set to `2`

Aha!

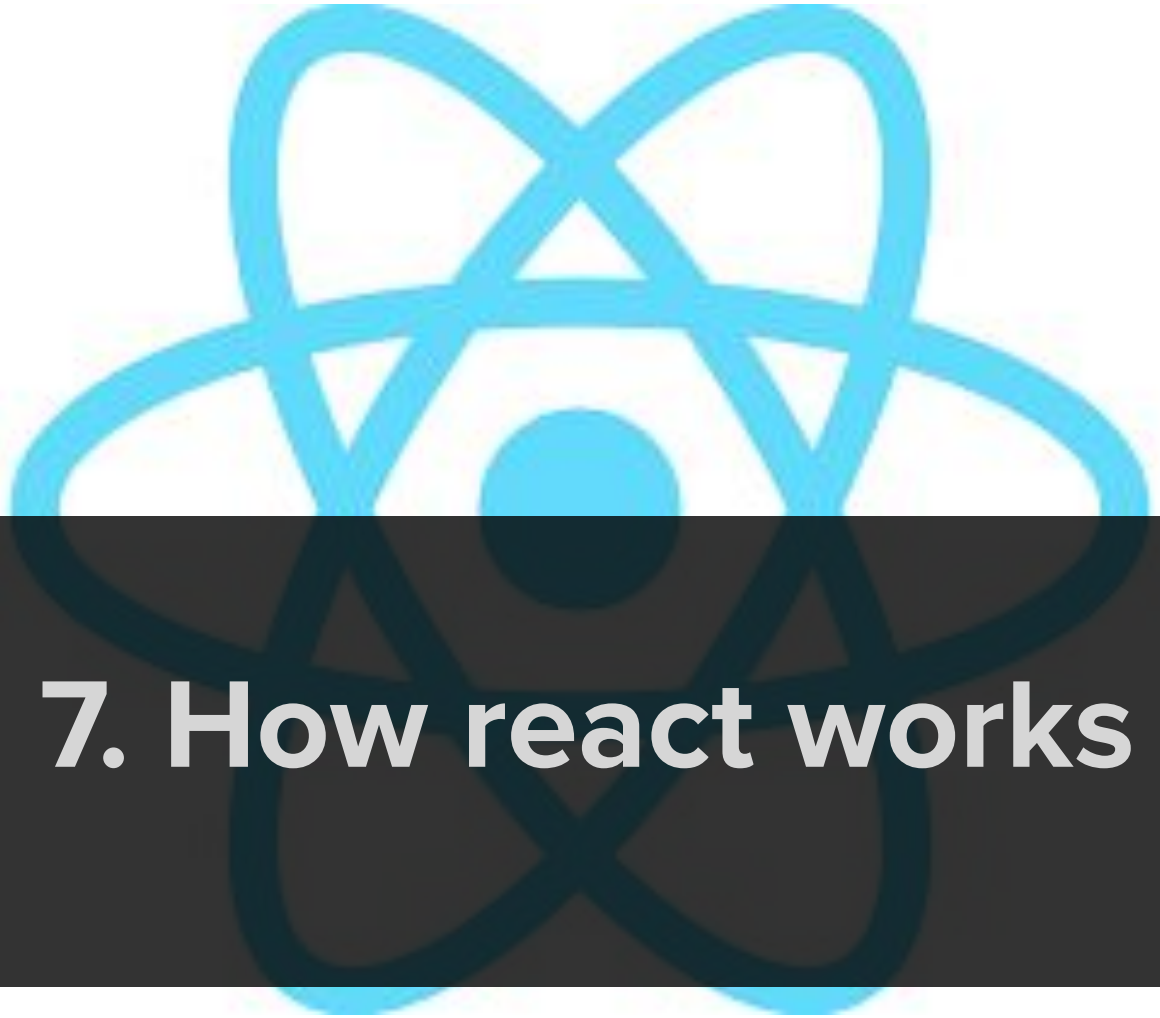
De setter (setCount) is executed **async** by React and you **can not ‘wait’ for it**. Only on the *next rerender* (= next call to the component function), count will have the *new value*.

https://react.dev/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time

https://react.dev/learn/queueing-a-series-of-state-updates

Lifting state up

- Start off using state closest to where it is used.
- If the state is needed in another component then move the state up to a shared parent
- This is called: “Lifting up state”



7. How react works

Change detection

How does React know that your state had changed?

1. An event happens (button click, API call returns,...). In the event handler, a setter (returned from the useState hook) is called:

```
<button onClick={() => setShowAmont(!showAmount)}/>
```

2. React compares the passed value with the previous value.
   For objects this is done by reference. Always provide a new object (shallow or deep copy) or the change will not be detected!

3. If they differ, react calls the component function of the component that invoked the setter

   function PiggyBank() is called

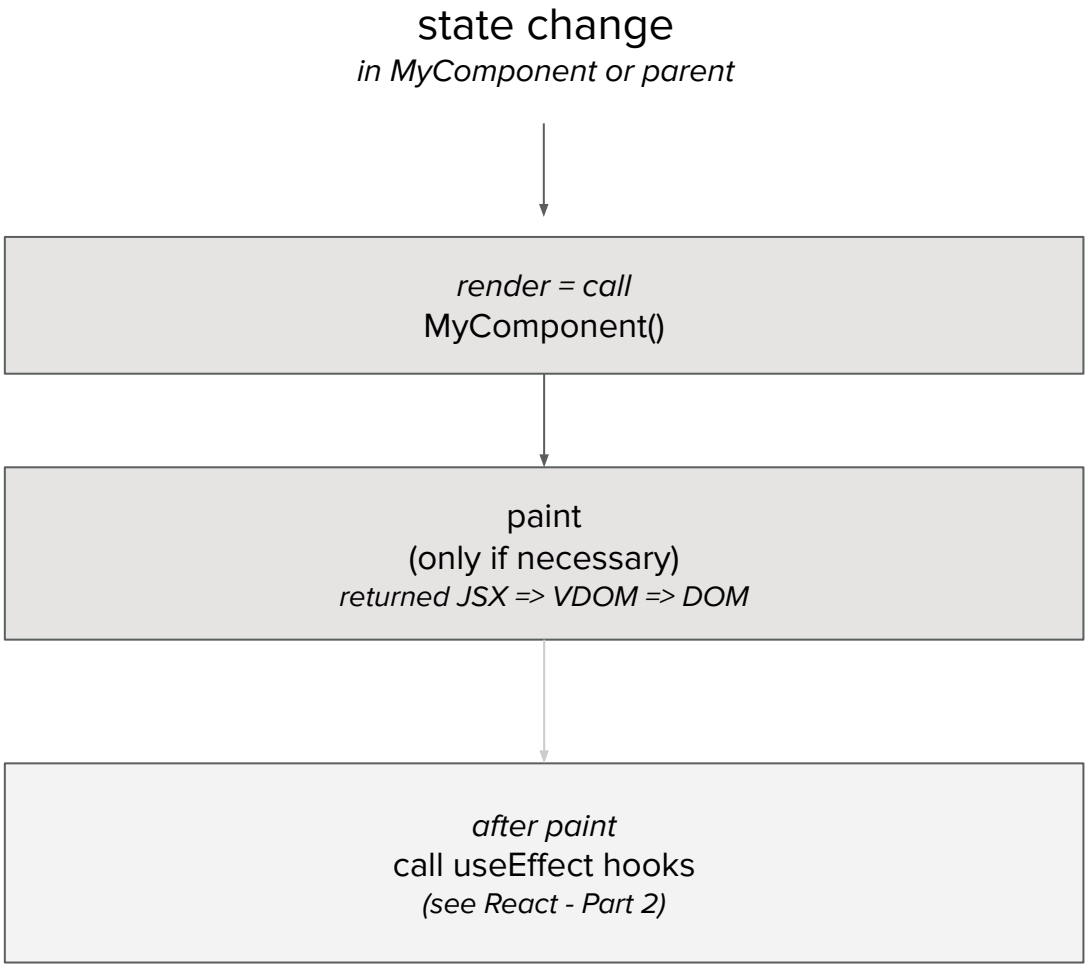
4. As a consequence the component functions of all the children (= tags in the JSX) of this component are called (‘rendering’)

5. The returned JSX is converted to ‘virtual DOM’ elements (‘painting’)

Rendering & Painting

By ‘rendering’ we mean ‘React calls your component function’

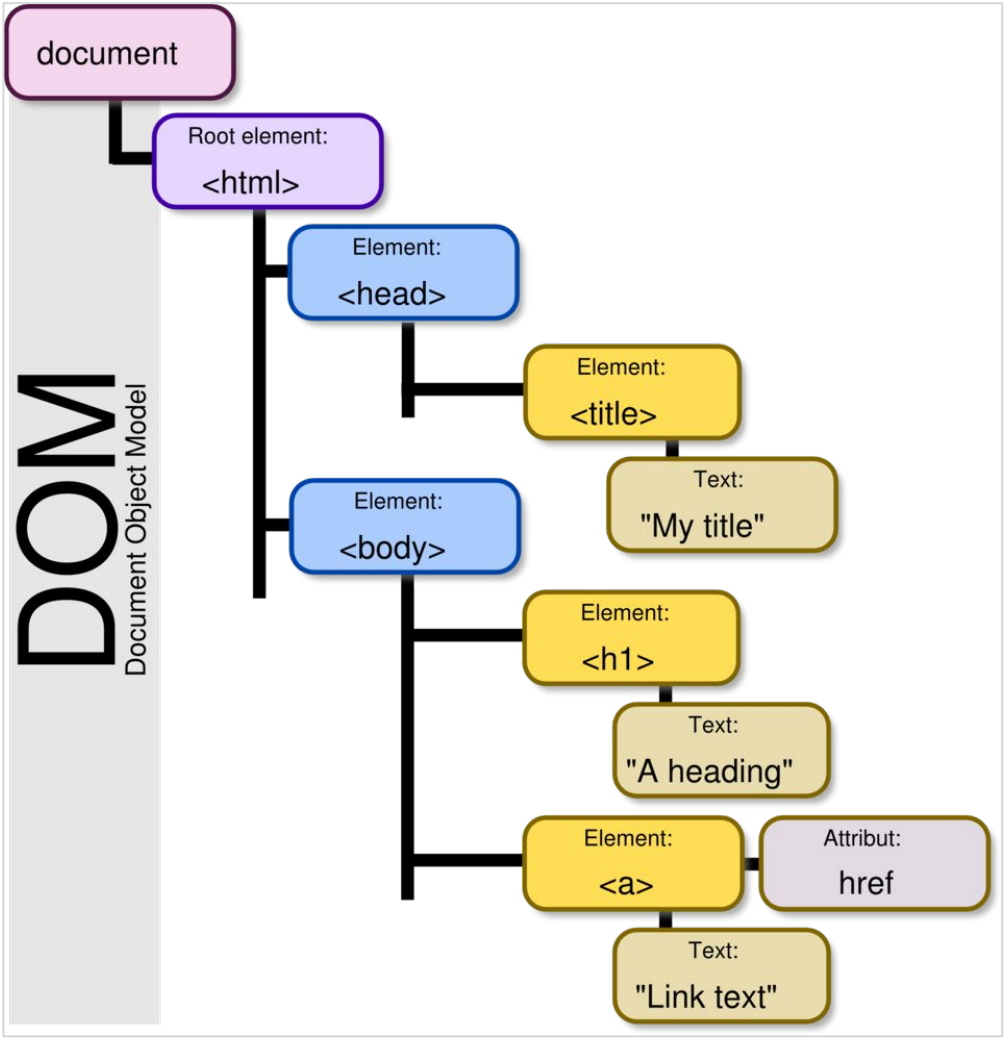
By ‘painting’ we mean ‘React uses the returned JSX to update a virtual DOM and (eventually) update the real (browser) DOM‘



Virtual DOM

What is the real DOM?

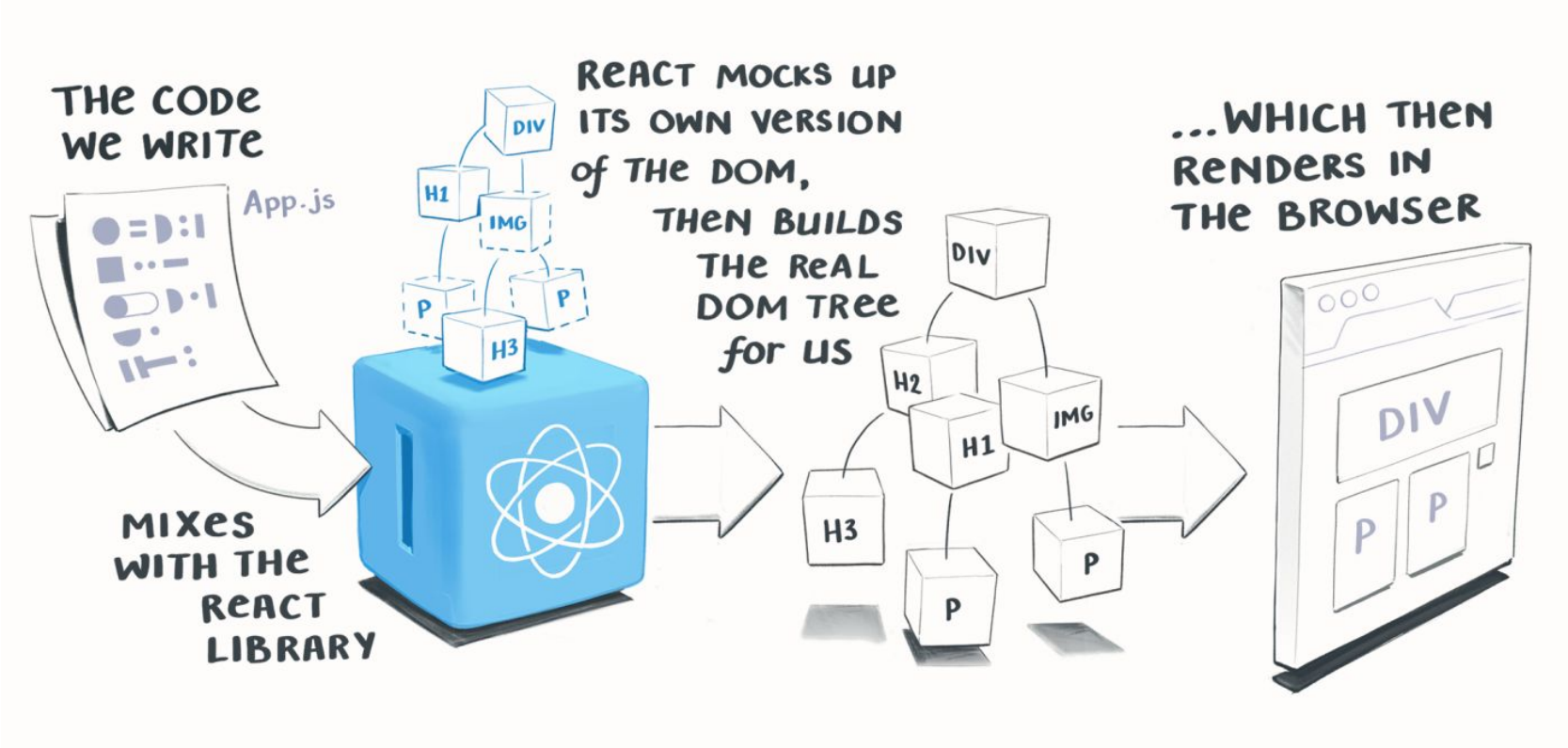
Tree of all html elements built by the browser to represent the page in memory



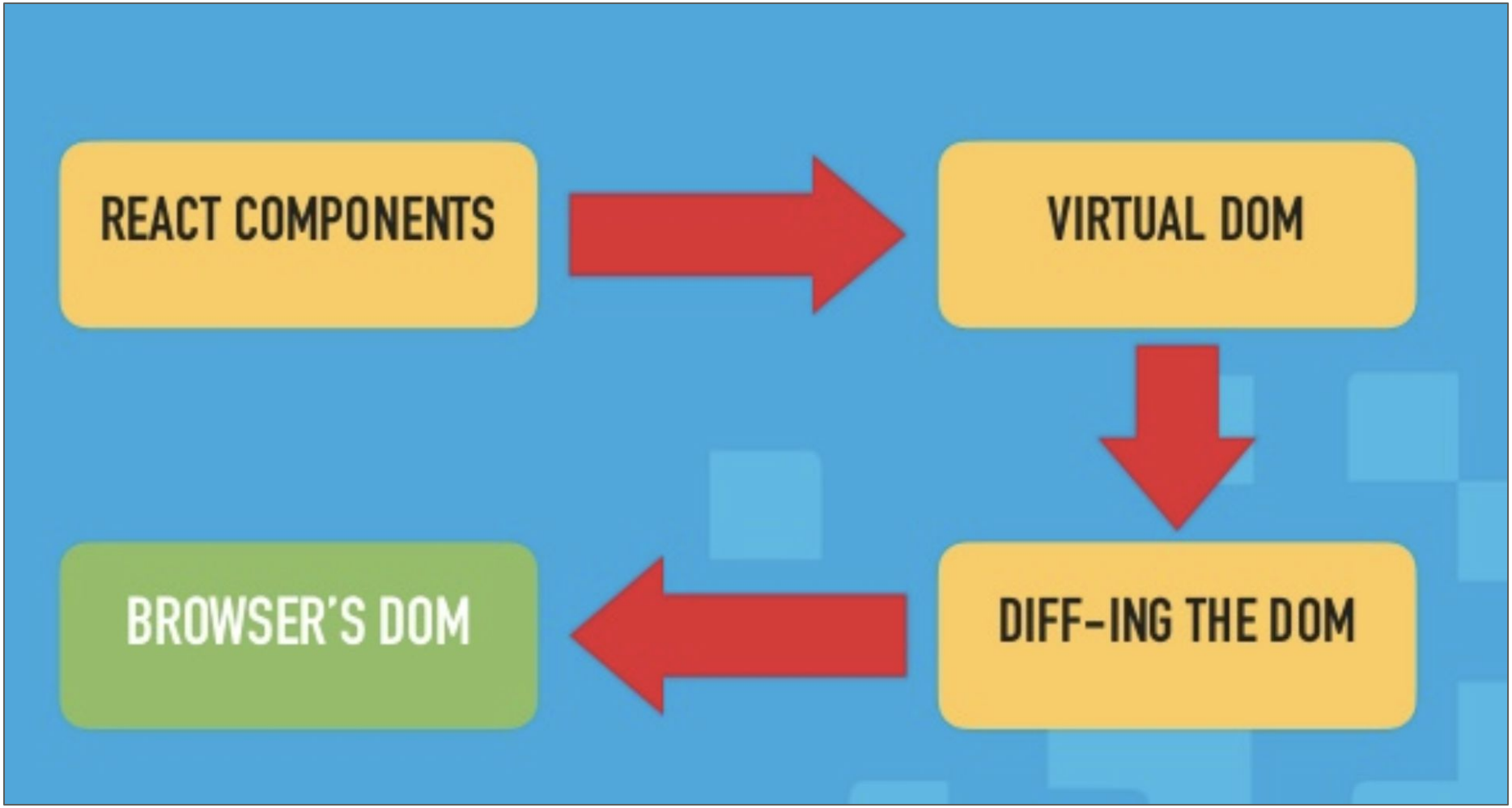
Virtual DOM

The VDOM is a virtual representation of the UI that is managed by React and synchronized with the 'real' browser DOM. **Only elements in the VDOM that differ from the real DOM on the screen are updated (DIFFING THE DOM)**

That process is called “Reconciliation” and makes the declarative approach work: *You tell React what state you want the app in and React makes sure the DOM matches that state*



Virtual DOM

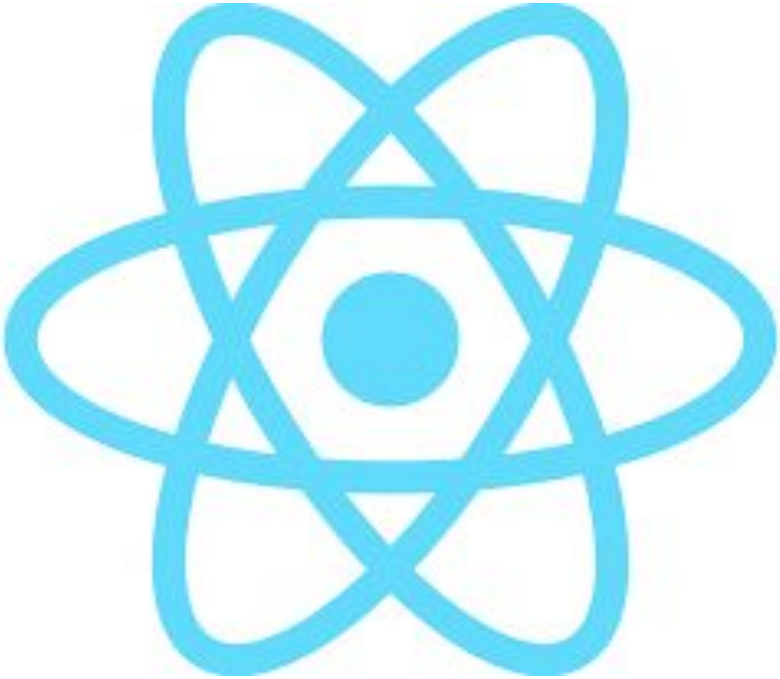


"Reconciliation"

What to expect in Part 2?

- useEffect and the rules of hooks
- Apps with multiple pages using React Router
- Backend communication with React Query

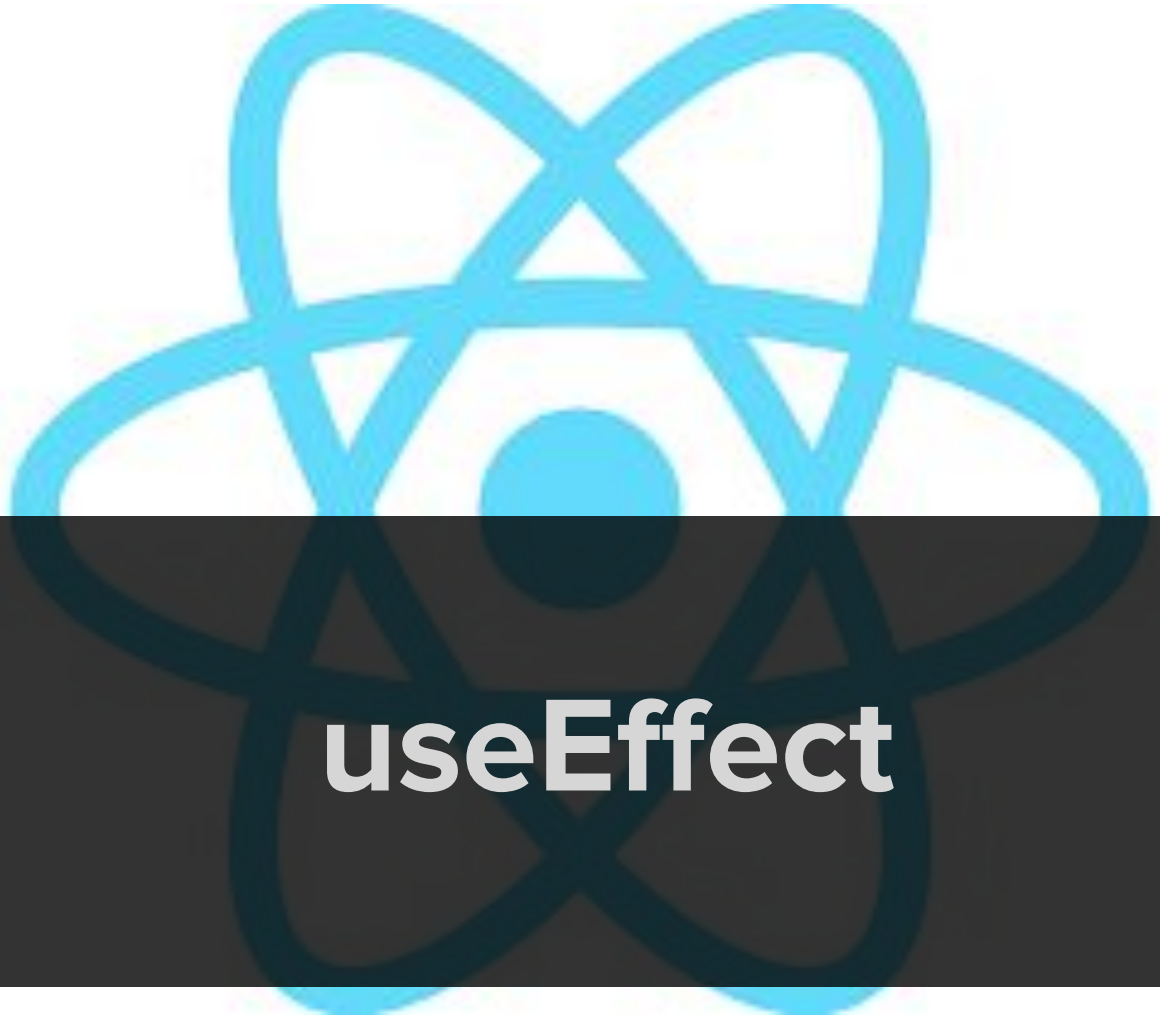

That's all Folks!



React - Part 2

Contents

1. useEffect
2. Routing
3. Backend communication
4. Component frameworks



useEffect

Recap

UI = f(state)

PROPS IN

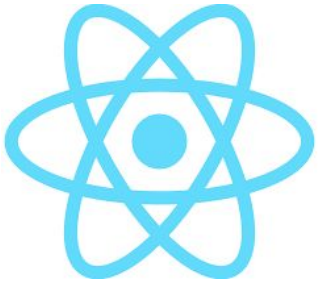
JSX OUT

INTERNAL STATE

useState hooks

PiggyBank

React



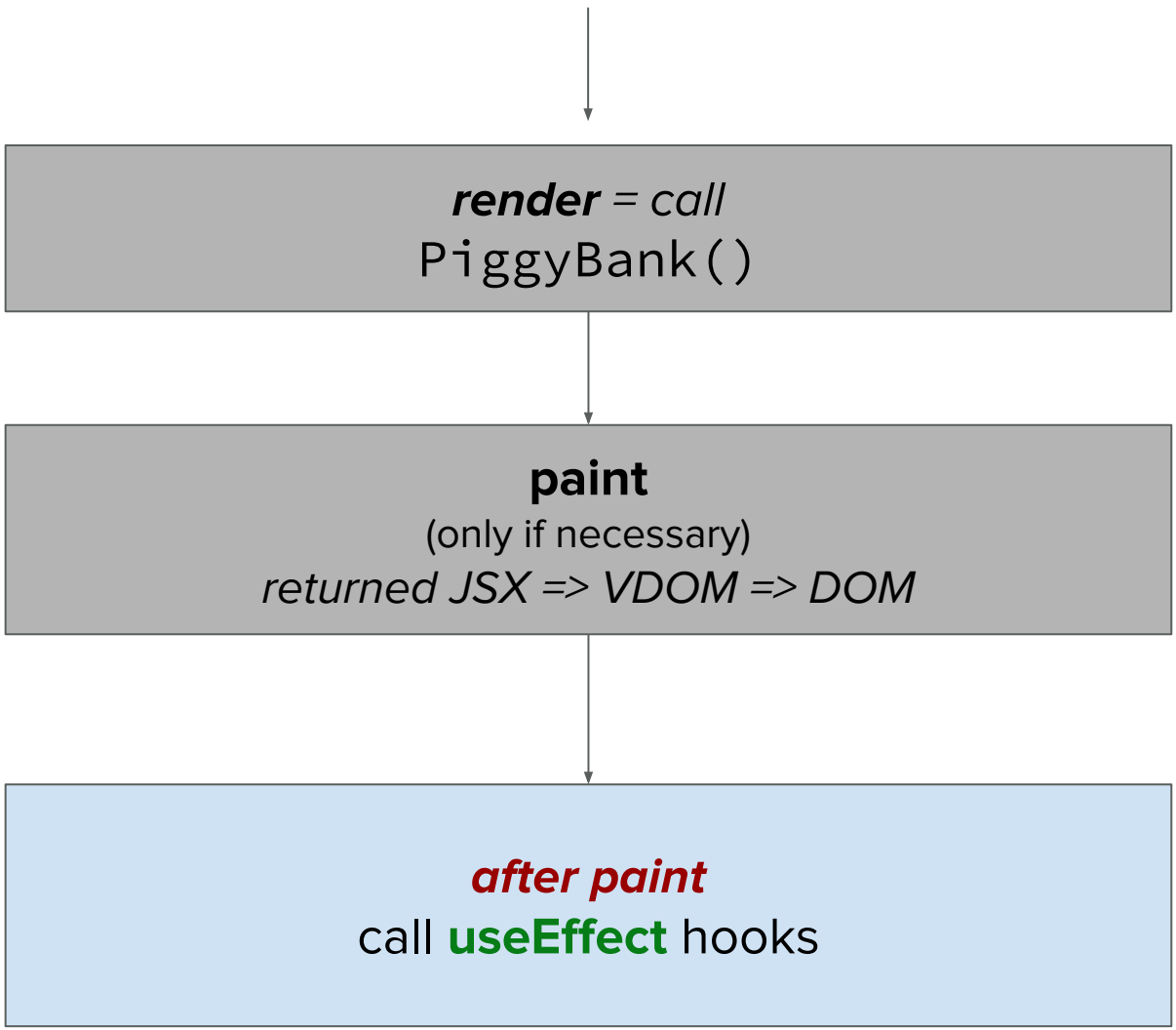
"I have a state change!":
`setShowAmount(true)`

"Roger that, I will **re-call** you and `showAmount` will be true when I do so":
`PiggyBank()`

"Thanks for calling me bro, here is the JSX for my new state"

"Thx! I will check if it differs from what is on the screen and repaint if necessary

Android? => Re-composition


state change
in PiggyBank (or parent of PiggyBank)
***render** = call*
`PiggyBank()`
paint
(only if necessary)
returned JSX => VDOM => DOM
after paint
call **useEffect** hooks

Side effects

Functional component = **PURE function**
Called again with the same props & internal state it should return the same JSX to be rendered

But “sometimes” an application has to **do something else** then render data = **apply side effects**

If caused by an **interaction** (click, swipe,...) this is done in **event handlers**

```
<OwnerBadge owner={account.owner} onClick={() => setShowAmount(true)}/>
```

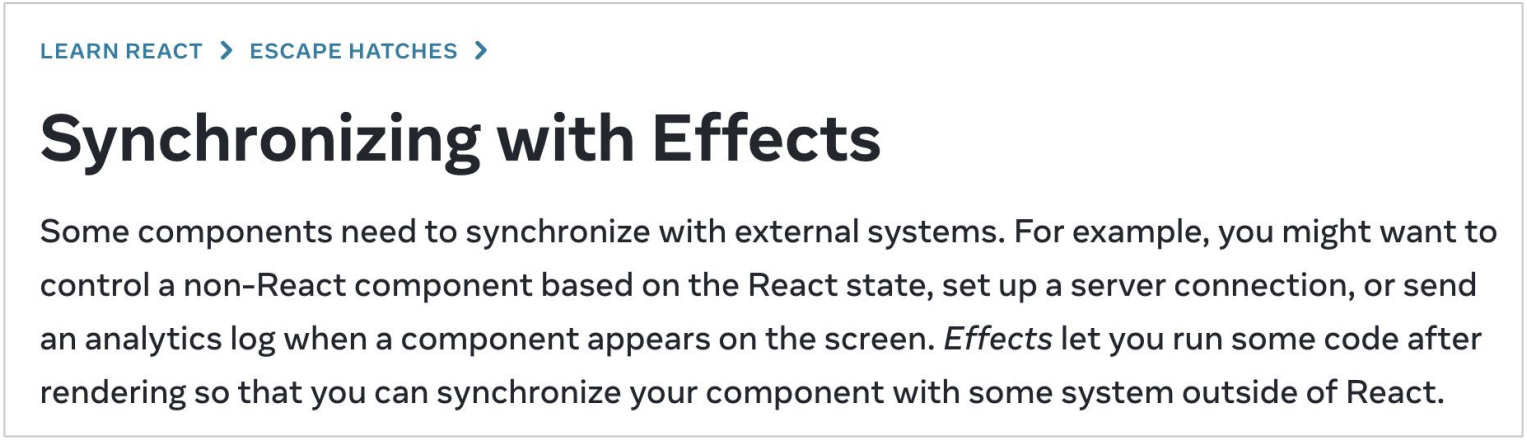
But what if you want to do something that can not be placed in an event handler?

useEffect

`useEffect` allows you to **synchronize an external API** with Reacts declarative/reactive way of doing things

Examples

- **get data from a REST-endpoint**
- (un)subscribe to/from a chat room web socket
- write/read a setting to/from local storage
- use a native browser API (`setInterval`,...)
- ...



```
useEffect(
  () => {
    // effect code (runs after render)
    return () => {
      // cleanup code (runs before unmount or re-run)
    };
  },
  [dependencies] // array of values the effect depends on
);
```

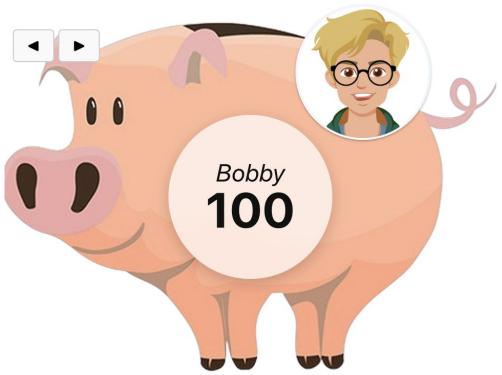
```
useEffect(
  () => {
    // effect code (runs after render)
    return () => {
      // cleanup code (runs before unmount or re-run)
    };
  },
  [dependencies] // array of values the effect depends on
);
```

- Used to run “side effects” **after** a render.

- Function with two arguments:
  - a function that is executed when the useEffect hook runs
    - Can return a function that runs when the component ‘unmounts’ (is removed from the screen) to clean up.
  - a “dependency list” (optional): an array of variables that trigger a re-run of the function when they change.

- Dependency list determines when to run the effect (again).
- **Always runs at least once** (after the first render = ‘mount’)

Example



ESC key pressed

```
useEffect( effect: () : (() => void) | undefined => {
    if (!showFields) return
    const onKeyDown : (e: KeyboardEvent) => void = (e: KeyboardEvent) : void => {  Show usages
        if (e.key === "Escape") setShowFields( value: false)
    }
    window.addEventListener( type: "keydown", listener: onKeyDown)
    return () : void => window.removeEventListener( type: "keydown", listener: onKeyDown)
}, deps: [showFields])
```

Rules of hooks

`useState` and `useEffect` are the most famous React hooks, but there are others like `useRef`, `useCallback`,... (we will discuss these later on), and can also write your own custom hooks.

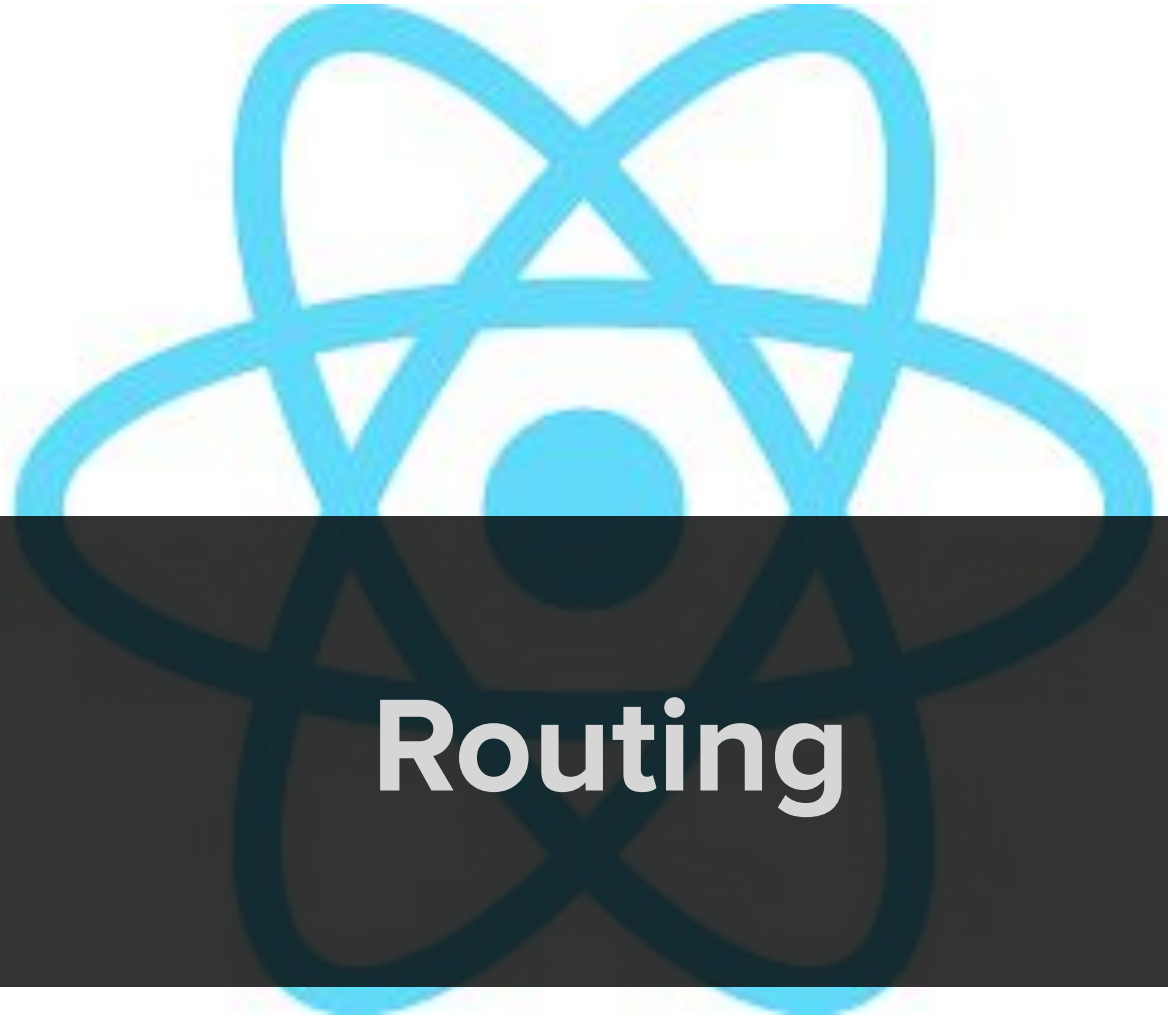
But wait! There are some rules ...

Hooks can only be used if we adhere to "the rules of hooks"

1. Name always start with 'use'
2. Only call hooks on the top-level (not conditionally, not in a loop,...)!
3. Only call hooks from React functions
   a. React components (are functions!)
   b. Custom hooks

Aha!

Why?
Because React relies on the order in which hooks are called to return the right value



Routing

React is a **library** rather than a **framework**... it doesn't give you a turnkey for things like *routing, backend communication, UI components, forms,...*

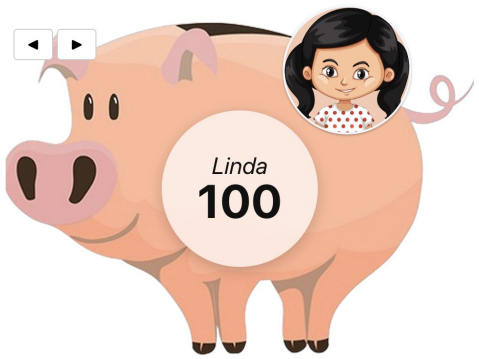
flexibility

- making choices
extra configuration

Routing

= Attaching a component to a URL

localhost:5173/piggybanks/1



localhost:5173/piggybanks/2

Navigating from one route (URL) to another does not trigger a full page reload:
= **client side routing**

A router library is not included in React, you can choose your own routing solution. There are a lot of options: Next.js router, Tanstack router, React Router,...

The fall into two types: **page-based** or **declarative** (or a mix of both)

Page-based routing

The routing is **file-system driven**: every file inside a `pages/` folder automatically becomes a route. You don't write `<Route>` components; the file name defines the route.

```
/pages
   index.js          →  "/"
   about.js          →  "/about"
   blog/[id].js      →  "/blog/:id"
```

Next.js example

Declarative routing

Routes are explicitly declared in JSX with `<Route>` components.

```
export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

React Router example

We use React Router (declarative) in this exercise.

Configuration is usually done at the top level (App.tsx)

```
function App() {
   return (
       <BrowserRouter>
           <Routes>
               <Route path="/piggybanks/:id" element={<PiggyBankDetail/>}/>
               <Route path="/piggybanks" element={<PiggyBankList/>}/>
               <Route path="/" element={<Navigate to="/piggybanks"/>}/>
           </Routes>
       </BrowserRouter>
   )
}
```

`<BrowserRouter>` activates routing using the browser history API.
There are also routers for React native, testing,...

`<Routes>` will select the **`<Route>`** that best matches the URL.

('/' matches any URL, `<Navigate>` redirects to another route)

Check the **router-demo** on Canvas

```
<Route path="/piggybanks/:id">
```

:<param-name> can be used to pass a parameter
The React framework recognizes this pattern and passes the id from the url to the component using the **useParams hook**.

```
function Piggybank() {
  const { id } = useParams()
  …
}
```

User navigates to /pigybanks/3 => useParams returns 3

There are two ways to navigate to a route.

`<Link to="url">`

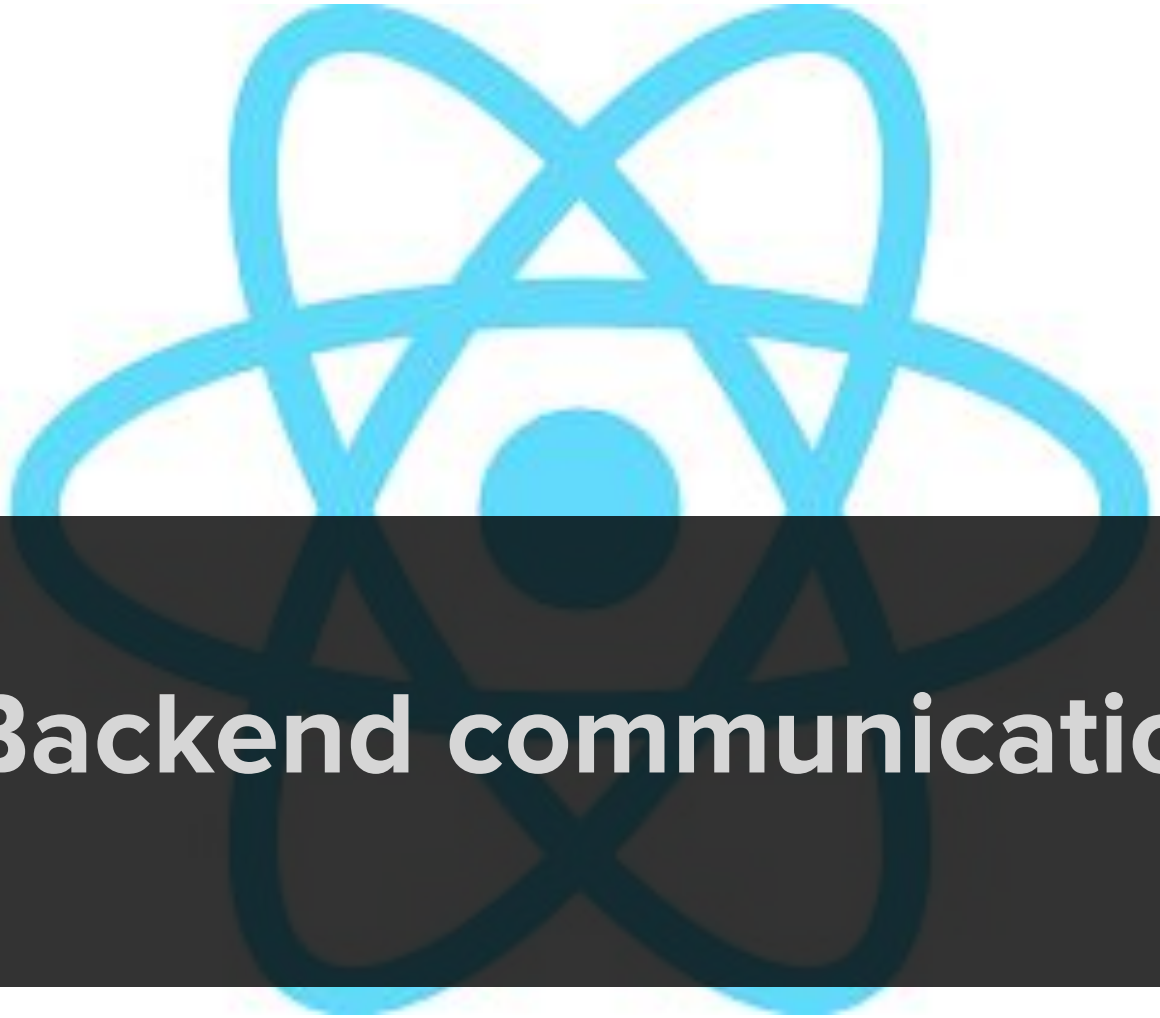
Used to put navigation in JSX. Generates an <a> tag behind the scenes

```
<Link to="/about">About</Link>
```

`useNavigate hook`

Used to handle dynamic (imperative) navigation in Javascript

```
import {useNavigate, Link} from "react-router-dom";


function SomeComponent() {
  const navigate = useNavigate();
  const createRecipe = async (data) => {
        const newRecipe = await storeRecipeInDB(data);
        navigate(`/recipes/${newRecipe.id}`);
  }

  return (
    <Link to="/home"}><button>Go back</button></Link>
  );
}
```



Backend communication

Option 1 - useEffect with boilerplate

To handle HTTP calls, you can use the standard **fetch** API (or pick a more advanced solution like **axios**, **xior**, **ky**...)

On top of that you will need **useEffect** to initiate the call. Typically you want to do this after 'first render' of the component (= after the first time `Todos()` is called).
Pass an empty dependency array to accomplish this.

```
export function Todos() {
   const [todos, setTodos] = useState<Todo[]>([]);

   useEffect(() => {
     async function fetchTodos() {
          const {data} = await axios.get<Todo[]>(URL);
          setTodos(data);
     }
     fetchTodos();
   },[]);

   return (
    <div>
      <ul>{todos.map((todo) => <li key={todo.id}>{to
    </div>
   )
}
```

useEffect does not accept an async function, we have to work with a separate async function that we call immediately

Empty dependency array: the effect runs only once, on first render

Asynchronous communication **takes time** and **can go wrong**...so we need to provide **loading** and **error** handling...this leads to a lot of boilerplate code

```
export function Todos() {
   const [todos, setTodos] = useState<Todo[]>([]);
   const [isLoading, setIsLoading] = useState(true);
   const [isError, setIsError] = useState(false);

   useEffect(() => {
       async function fetchTodos() {
           try {
               const response = await axios.get<Todo[]>(URL);
               setTodos(response.data);
           } catch (e) {
               setIsError(true);
           }
           setIsLoading(false);
       }
       fetchTodos();
   }, []);


   if (isLoading) {
       return <div>Loading todos</div>
   }

   if (isError) {
       return <div>Todos could not be loaded</div>
   }

   return (
    <div>
      <ul>{todos.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>
    </div>
   )
}
```

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> *-- Phil Karlton*

Let's make this quote a bit broader: **naming things and managing distributed state**

Example

The value shown by a piggybank has multiple copies in the running app:

- data.json ⟶ "the ultimate truth"
- React Query cache (see further)
- your in memory model
  (mostly a ref to rquery cached object)
- element in the virtual DOM (React 'cache')
- element in the real DOM

Syncing **asynchronous state** - frontend-backend - is extra challenging:

- *When to get new data from the backend?*
- *Save one element in a collection or all at once?*
- *Embed data or make extra calls?*
- *What if a call fails?*

Option 2 - React Query

React Query is a library that helps you to handle async state (and his problems). It does this by using **useEffect** and **useState** under the hood**,** with several features (like caching) on top of that.

There are (of course) alternatives...

Comparison | React Query vs SWR vs Apollo vs RTK Query vs React Router

Use *`QueryClientProvider`* to supply a QueryClient instance to your code.
(This is a "`context`", more on that later!)

```
const queryClient = new QueryClient()

function App() {
  return (
    // Provide the client to your App
    <QueryClientProvider client={queryClient}>
      <Todos />
    </QueryClientProvider>
  )
}
```

Define functions that do the actual data manipulation.
This can be *anything that returns a promise*.

```
export async function getTodos(){
    const {data: todos} = await axios.get<Todo[]>('/todos');
    return todos;
};

export function postTodo(todo: Todo): Promise<Todo> {
    return axios.post('/todos, todo)
};
```

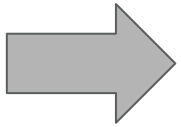
Uses these functions in custom hooks to get (***useQuery***) and update (***useMutation***) data

```
function Todos() {
  const queryClient = useQueryClient()
  const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

  const mutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['todos'] })
    },
  })

  return (
    <div>
      <ul>{query.data?.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>

      <button
        onClick={() => {
          mutation.mutate({
            id: Date.now(),
            title: 'Do Laundry',
          })
        }}
      >
        Add Todo
      </button>
    </div>
  )
}
```

(error and loading flags omitted here for clarity)

React Query (and similar libraries) reduce the amount of 'boilerplate' code significantly by handling **error and loading states** for you

```
const URL = 'https://jsonplaceholder.typicode.com/albums';

export function Albums () {
    const [albums, setAlbums] = useState( initialState: []);
    const [isLoading, setIsLoading] = useState( initialState: true);
    const [isError, setIsError] = useState( initialState: false);

    useEffect( effect: () => {
        async function fetchAlbums() {
            try {
                const response = await axios.get(URL);
                setAlbums(response.data);
                setIsLoading( value: false);
            } catch (e) {
                setIsError( value: true);
                setIsLoading( value: false);
            }
        }

        fetchAlbums();
    }, deps: []);

    if (isLoading) {
        return <div>Error loading albums</div>
    }

    if (isError) {
        return <div>Albums could not be loaded</div>
    }

    return (
        <div className="albums">
            {albums.map(({id, title}) => (
                <h1 key={id}>{title}</h1>))
            }
        </div>
    )
}
```



```
const URL = 'https://jsonplaceholder.typicode.com/albums';

async function getAlbums() : Promise<Album[]> { Show usages new *
    const {data: albums} = await axios.get<Album[]>(URL);
    return albums;
}

export function Albums() : Element { Show usages new *
    const {isLoading, isError, data: albums} =
        useQuery(
            {
                queryKey: ['albums'],
                queryFn: getAlbums
            });

    if (isLoading) {
        return <div>Loading albums</div>
    }

    if (isError || typeof albums == "undefined") {
        return <div>Albums could not be loaded</div>
    }

    return (
        <div className="albums">
            {albums.map(({id, title} : Album ) : Element => (
                <h1 key={id}>{title}</h1>))
            }
        </div>
    )
}
```

Check the **rquery-demo** on Canvas, compare with and without RQuery

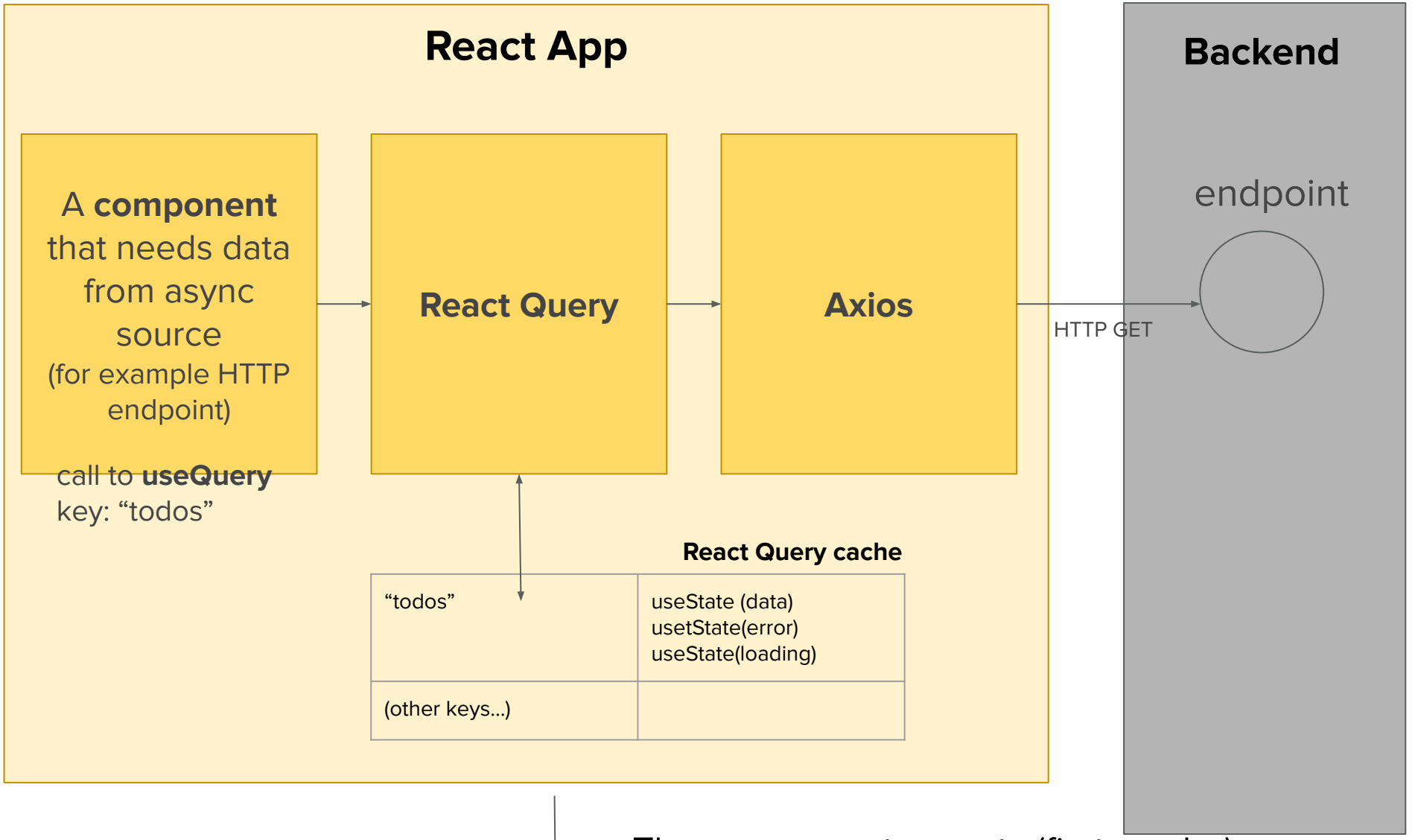
The **query 'key'** [`'todos'`] is used by React Query internally to **cache the data**

```
const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

const mutation = useMutation({
  mutationFn: postTodo,
  onSuccess: () => {
    // Invalidate and refetch
    queryClient.invalidateQueries({ queryKey: ['todos'] })
  },
})
```

React Query will **refetch** the data when the cache becomes *'stale'* (= no longer valid). This will happen after a configurable time or when you call `queryClient.invalidateQueries`

Additional debugging needed?
Check the [React Query devtools]

Sidenote: if you optionally want to try-out the new [React Suspense] feature: React query [supports it]



In these cases React Query will check the cache and refetch if stale

- The component mounts (first render),
- The browser tab regains focus (if configured)
- The network reconnects after being offline.

Lots of additional features...

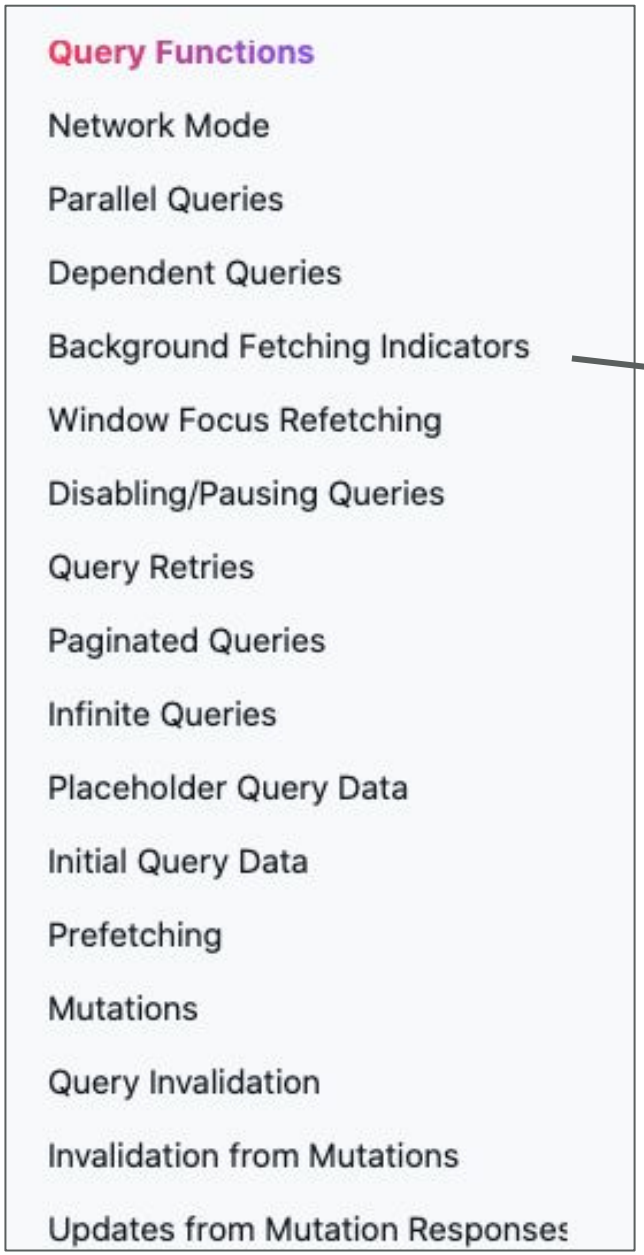
- Query Functions
- Network Mode
- Parallel Queries
- Dependent Queries
- Background Fetching Indicators
- Window Focus Refetching
- Disabling/Pausing Queries
- Query Retries
- Paginated Queries
- Infinite Queries
- Placeholder Query Data
- Initial Query Data
- Prefetching
- Mutations
- Query Invalidation
- Invalidation from Mutations
- Updates from Mutation Responses

React query will (out of the box) retry 4 times (can be configured) to load data when a 404 is encountered

- Query Functions
- Network Mode
- Parallel Queries
- Dependent Queries
- Background Fetching Indicators
- Window Focus Refetching
- Disabling/Pausing Queries
- Query Retries
- Paginated Queries
- Infinite Queries
- Placeholder Query Data
- Initial Query Data
- Prefetching
- Mutations
- Query Invalidation
- Invalidation from Mutations
- Updates from Mutation Responses

You can specify that a query only runs when a specific condition is met (eg. that another query is finished)

```
// Get the user
const { data: user } = useQuery({
  queryKey: ['user', email],
  queryFn: getUserByEmail,
})

const userId = user?.id

// Then get the user's projects
const {
  status,
  fetchStatus,
  data: projects,
} = useQuery({
  queryKey: ['projects', userId],
  queryFn: getProjectsByUser,
  // The query will not execute until the userId exists
  enabled: !!userId,
})
```



Refetching at a specific interval (**polling**) is also supported

Filtering

To filter data coming from a backend (or any external source), there are 2 options:

1. Client-side filtering

Get all the data from the backend, filter it in memory on the client

Avoid this if you expect to retrieve large amounts of data

Snappy (for instance: user is typing in ‘filterByName’ textfield => immediate feedback

2. Server-side filtering

Retrieve only the data you need

Only the data you need is transferred over the wire

If you have a slow backend, the user may experience delays
(consider using something like useDebounce)

If you use React Query, make sure to **attach the filter to the query key**, or it will not refetch when the filter changes

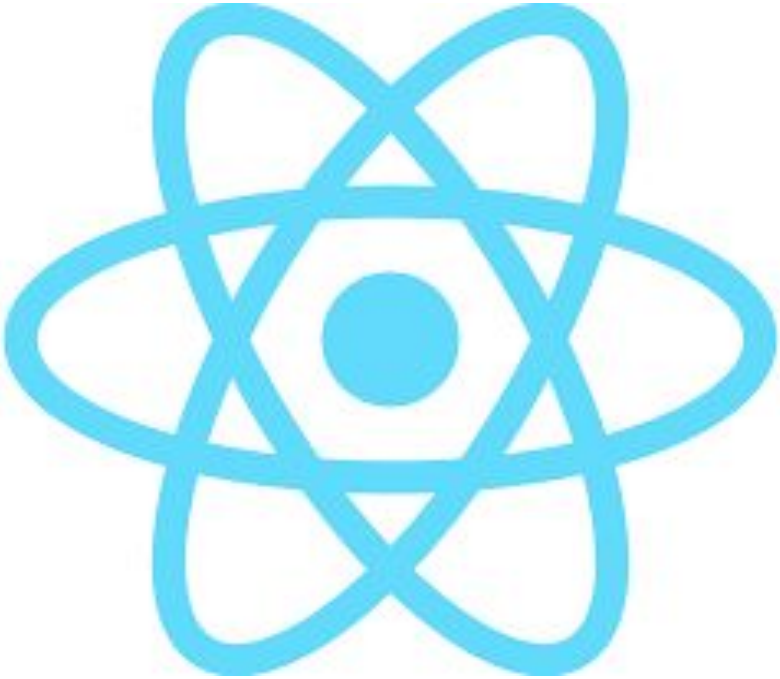
Check the **filtering** demo on Canvas

What to expect in Part 3?

- Design systems and component frameworks
- Creating custom hooks and using hook libraries
- Handling forms with React Hook Form
- Managing application wide state with React Context

KdG Karel de Grote Hogeschool


That's all Folks!

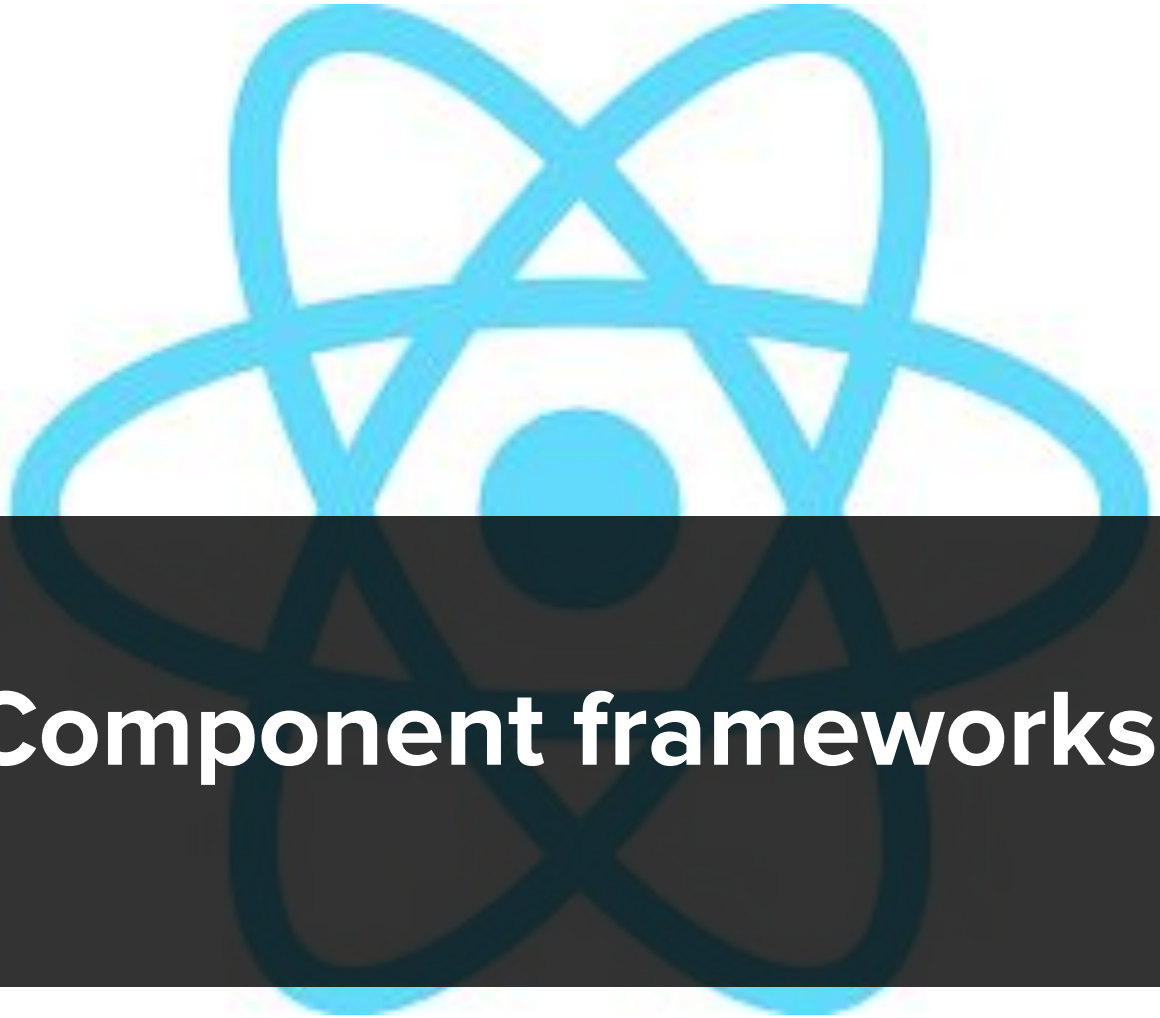


React - Part 3
(of 4)

Contents

Component frameworks

Form handling



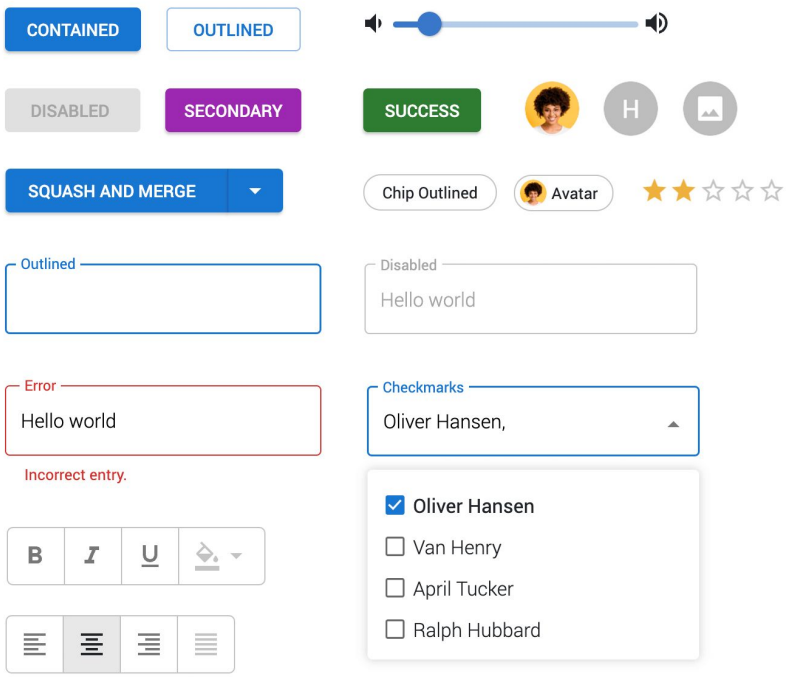
Component frameworks

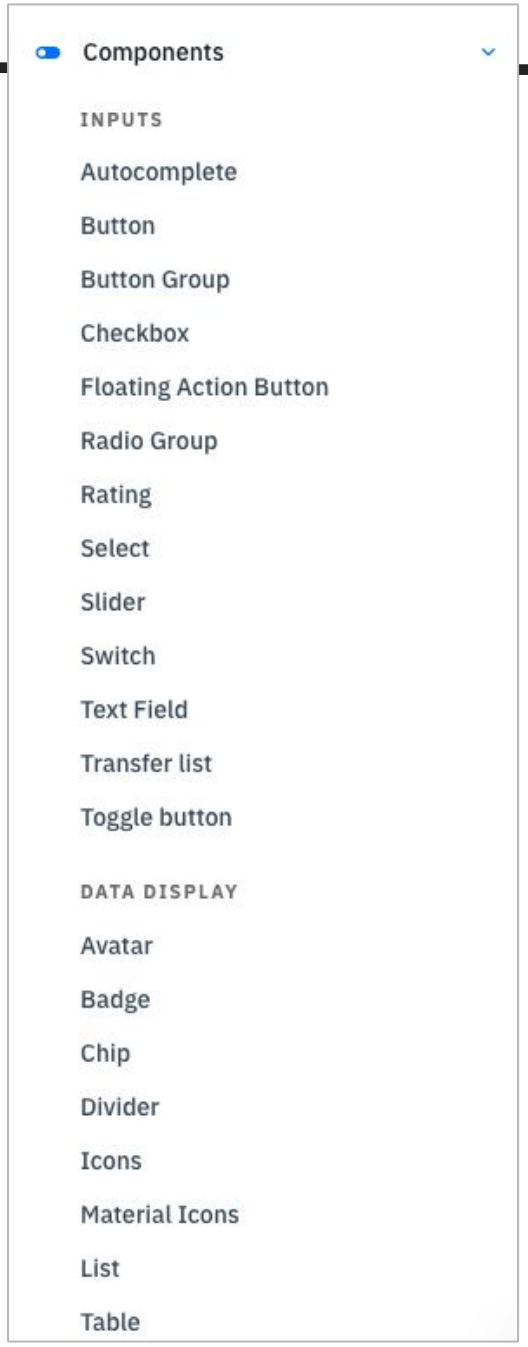
Almost every (large) application needs general functionality like lists, menus, cards, forms, app-bars, drawers,...

Do not reinvent the wheel

Component framework

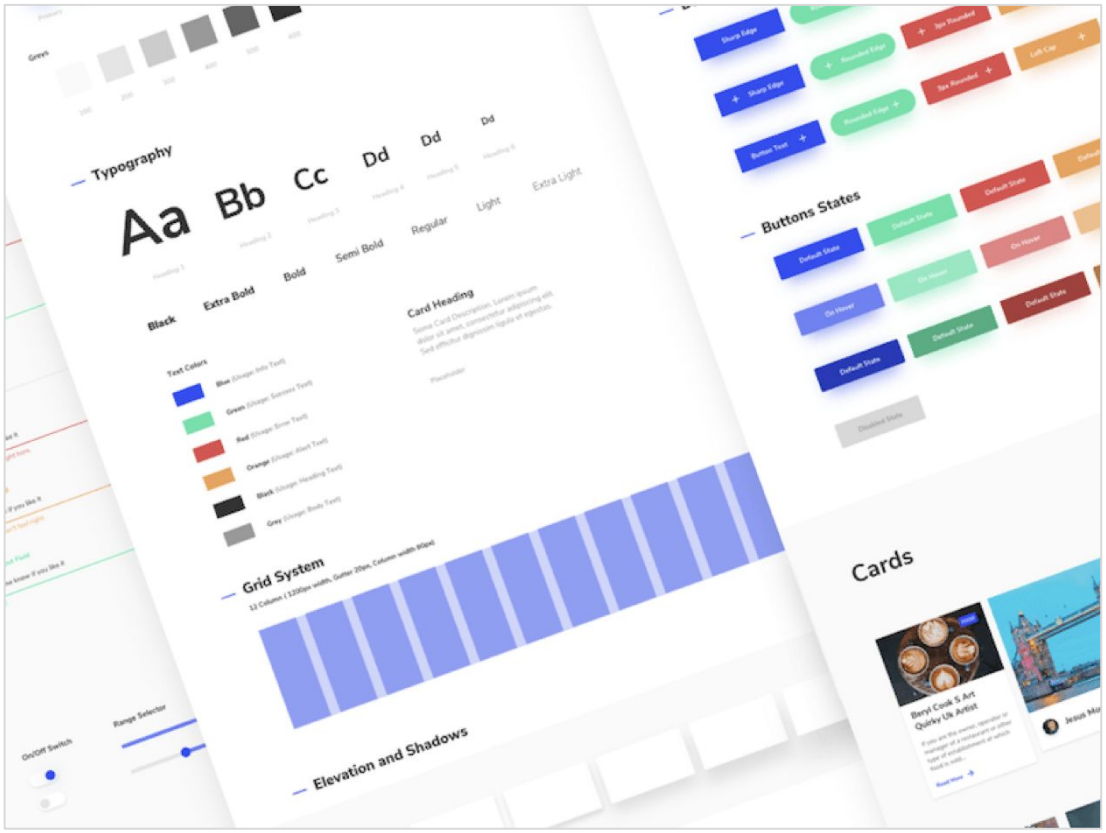
A **collection of components** that follow the rules of a certain **design system**. Can be adapted to your needs using CSS and theming.





Design system

A collection of **guidelines**, **design tokens** and **resources** (icons, css files, components, code snippets, etc.) that together form a **consistent set of rules** and a **toolbox for designing** a user interface.

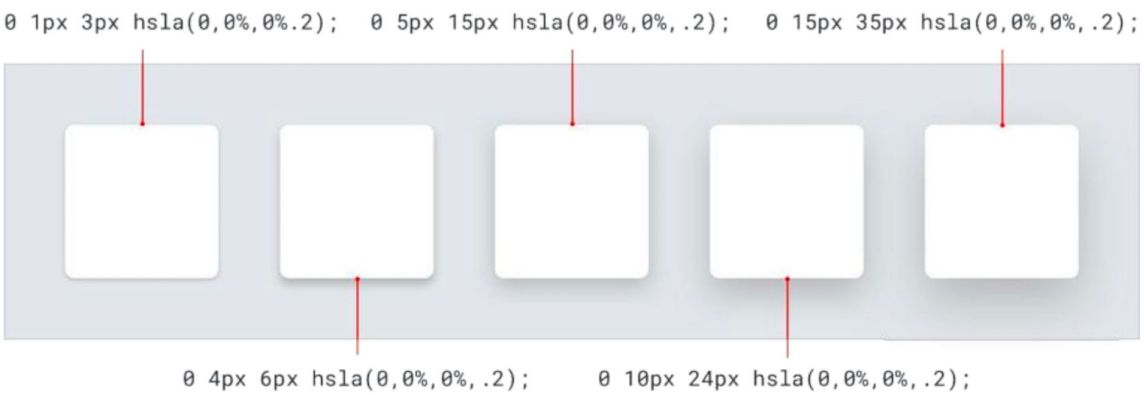


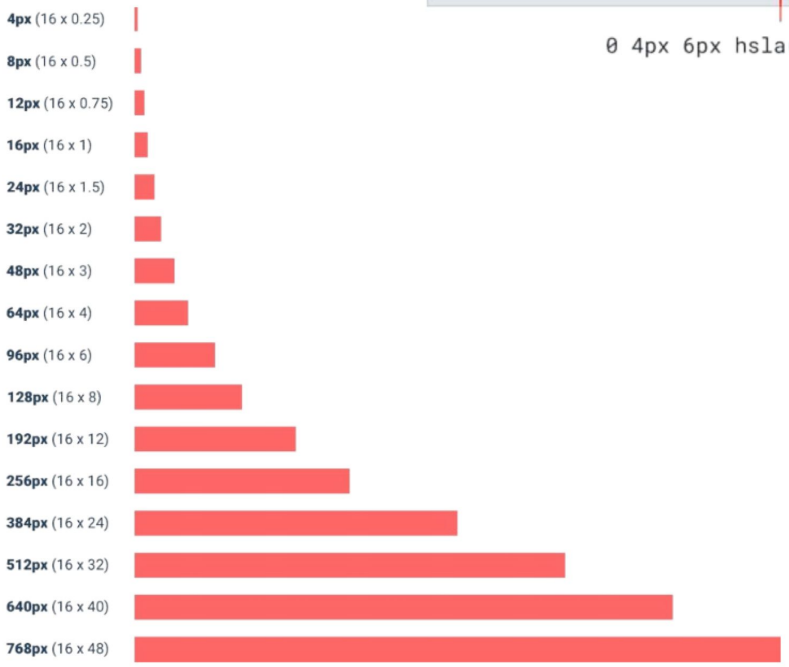
Design tokens

The atomic elements that make up a design system
(colors, shadows, fonts, spacing/grid, animations etc...)


SO MANY OPTIONS, SO LITTLE TIME

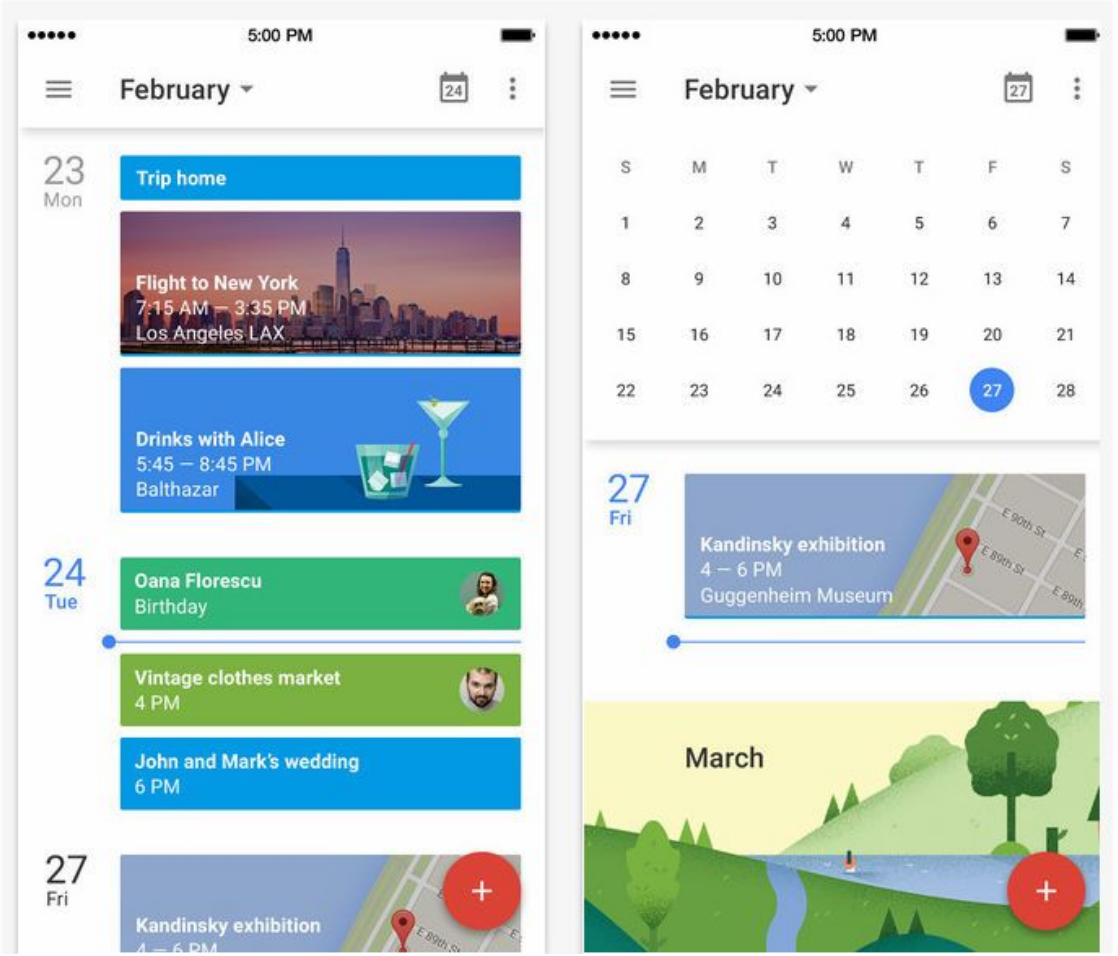
In a design system, choices are made for you to reduce the number of options and create **consistent interfaces**.





Material design

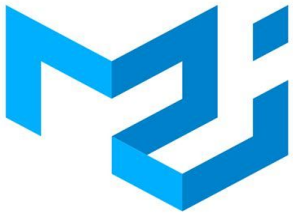
Design system by Google, used in all its products.

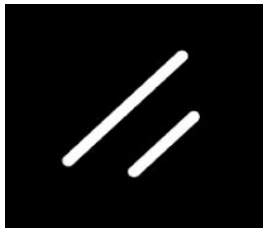


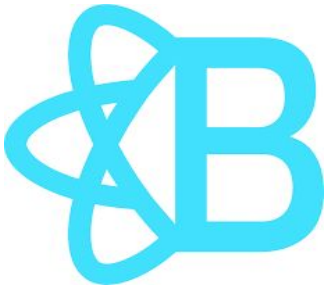
https://material.io/design/

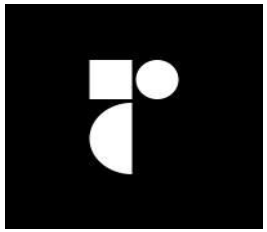
Component frameworks

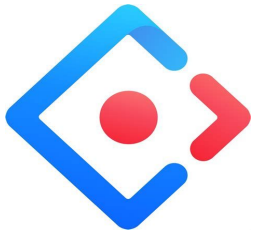
The are **a lot** of component frameworks. Each framework follows a specific design system (unless it is headless).

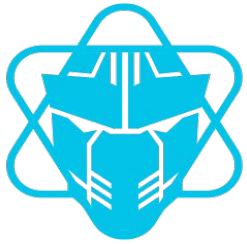


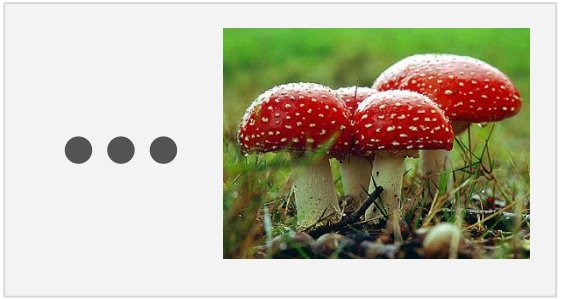








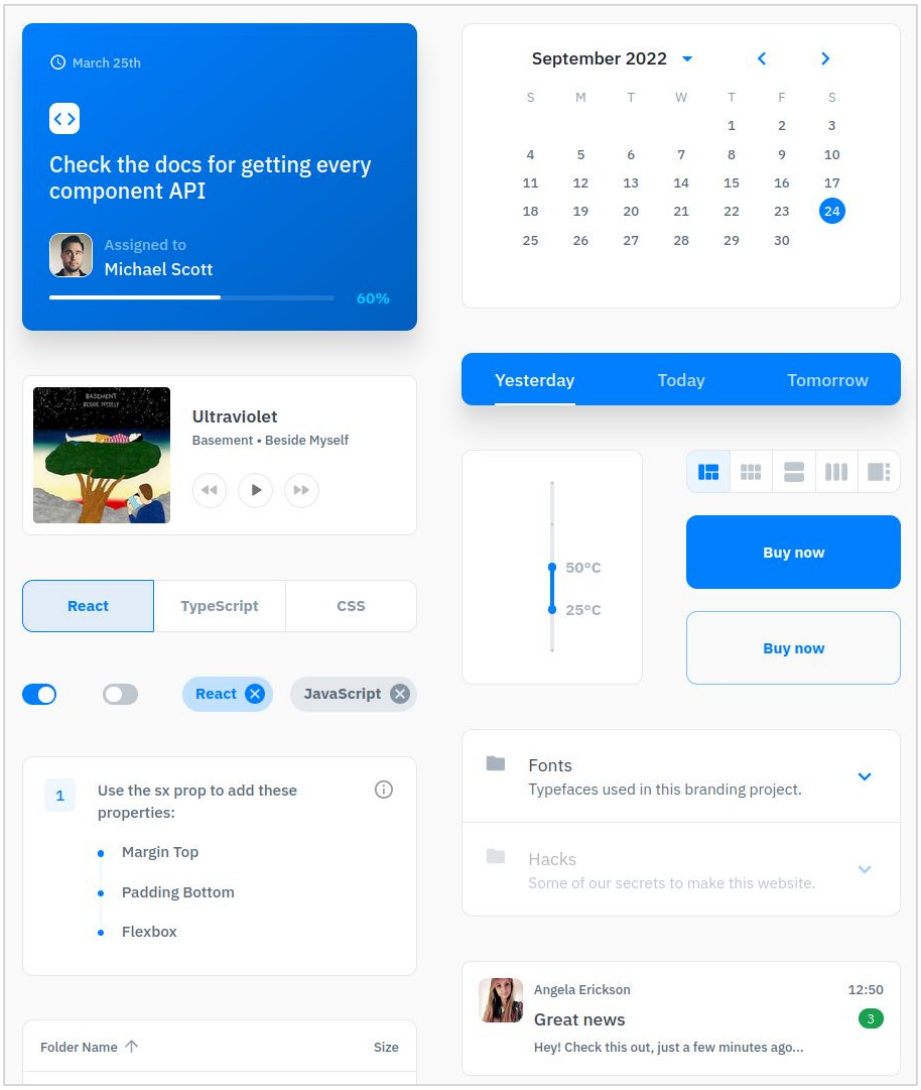




MUI

One of the **most used and extensive** componente frameworks. We will apply it in the exercise and example code, but **you are free to pick another**!

MUI adheres to the **material design system**.



Check the **mui-demo** on Canvas

Do you really need a component framework?

Option - Do not use a component framework, style your div's with a CSS framework like bootstrap, tailwind,...
Create your own components using these code 'blocks'.

MyButton.tsx

```
<div className="text-white bg-blue-700 hover:bg-blue-800 focus:ring-4 focus:ring-blue-300 font-medium rounded-lg text-sm px-5 py-2.5 mr-2 mb-2
dark:bg-blue-600 dark:hover:bg-blue-700 focus:outline-none dark:focus:ring-blue-800">
        Save
</div>
```

Option - Use an 'unstyled' component framework, style it with your preferred styling solution

Base UI, *Radix UI*, *Headless UI*,...,

Option - Use a styled component framework like *MUI*, *Shadcn*, *Mantine*, *Ant Design*,..

```
<Button>Save</Button>
```

Powerful components with clearly defined props that adhere to a design system
Less wrapping in own components needed

A stronger vendor lock-in
Less flexible (adaptable, reusable) than e.g. tailwind code blocks of headless

Most of the component frameworks promote a CSS-in-JS approach which means you can **embed styles in Javascript**.

This is not necessary a bad thing. Remember we talked about JSX being HTML-in-JS and '**keep together what changes together**'.
Moreover, it gives you a lot of extra power like dynamic styles, dynamic theming and dynamic responsiveness *(with dynamic we mean it is just JS so you can use if tests, ternary expressions etc to calculate styles directly from your state)*

For example in MUI, most component have an sx property that accepts a JS object with CSS styles.

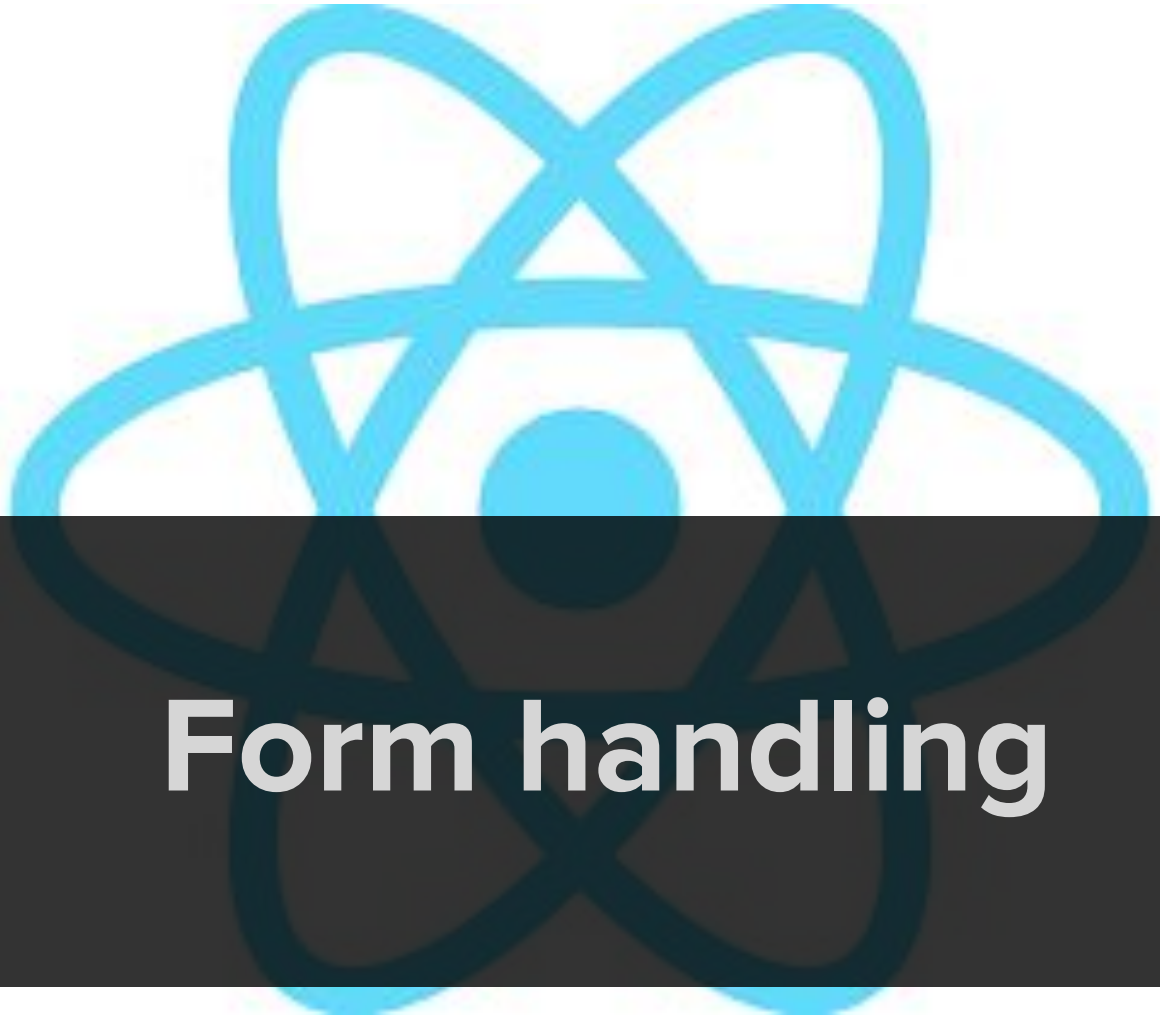
```
<Avatar
  alt="Remy Sharp"
  src="/static/images/avatar/1.jpg"
  sx={{ width: 24, height: 24 }}
/>
```

This does not mean you cannot change global styles for all your components (like colors, fonts,...)! This is done through a theming mechanism. Utilities like Box and Stack are also very handy for (theme-aware styling.

Besides the sx prop, you will also see the use of styled components in MUI example code. Styled components often lead to code bloat, but can be useful for a reusable component with more complex theme-aware styling.

```
import { Box } from "@mui/material";

export default function Example() {
   return (
       <Box
           sx={{
               p: 2,                    // theme.spacing(2)
               bgcolor: "primary.main",
               "&:hover": { bgcolor: "primary.dark" },
           }}
       >
           Hello World
       </Box>
   );
}
```

```
import { styled } from "@mui/material/styles";
import Button from "@mui/material/Button";

const MyButton = styled(Button)(({ theme }) => ({
   padding: theme.spacing(2),
   backgroundColor: theme.palette.secondary.main,
   "&:hover": {
       backgroundColor: theme.palette.secondary.dark,
   },
}));

export default function Example() {
   return <MyButton>Click Me</MyButton>;
}
```



Form handling

Controlled components

Vanilla way of working with inputs: use "**controlled components**"

- The value and the `onChange` function of the component are provided by the programmer.
- The component is now "controlled"

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

```
<input
   type="text"
   placeholder="First Name"
   value={firstName}
   onChange={({ target }) => setFirstName(target.value)}
/>
```

This has to be done for every form field!

Check the **controlled-form-demo** on Canvas

Controlled components

This quickly becomes hard to manage

- you need a `useState` for each form field
- it can be non-performant
  (rerender of the complete form with each keystroke...)

What if we add validation, error states, dynamic fields, ... on top of that?

Use a library for this!

React Hook Form (RHF) helps with creating forms and managing their state

- State management is done using refs to the DOM element iso useState (‘uncontrolled’ components RHF tracks state internally)
- Support for validation, often used in combination with a schema validator like Zod or Yup
- Can integrate with UI libraries that do not expose a ref by using the “controller” component. This “isolates” rerenders

```tsx
interface ISignUpFormValues {
    name: string;
    email: string;
}

function App() {
    const { register, handleSubmit} = useForm<ISignUpFormValues>();

    const [user, setUser] = useState<ISignUpFormValues>();

    const onSubmit: SubmitHandler<ISignUpFormValues> = (data) => {
        setUser(data);
        console.log(data)
    };

    return (
        <>
            <form onSubmit={handleSubmit(onSubmit)} noValidate>
                <input placeholder="Name"{...register("name")}/>
                <input placeholder="Email" {...register("email")}/>
                <button type="submit">Submit</button>
            </form>

            <pre>{JSON.stringify(user, null, 4)}</pre>
        </>
    );
}
```

Check the **react-hook-form-demo** on Canvas

Controlled form

React state management
[firstName, setFirstName] = useState('')

Coupled via value and onChange

Firstname

rerender** of complete form **on every keystroke

RHF register

RHF internal state management
"firstName" field

{...register} attaches event handlers and a ref to the native HTMLinput field

Firstname

***no rerender** on keystroke, fast, simple but...*
...not usable for (complex) UI fields that do not expose a ref to underlying DOM element

RHF <Controller>

RHF internal state management
"firstName" field

Coupled via value and onChange to RHF's internal state management no ref to DOM element needed

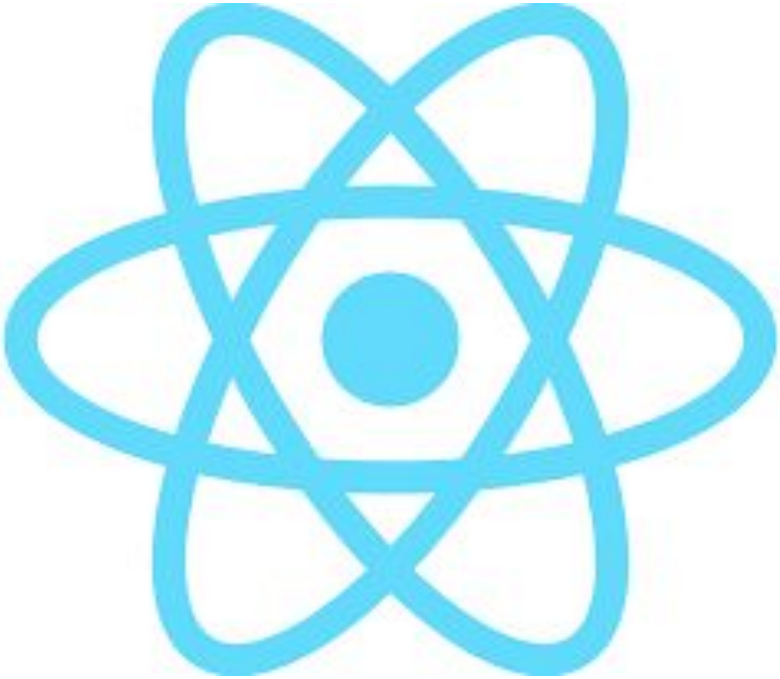
10€ 20€ 50€ 100€ 200€

***render isolation:** only the field itself (wrapped in a <Controller> using a **render prop)** is rerendered*

```
<Controller
    name="amount"
    control={control}
    render={({field: {value, onChange}} : { field: ControllerRenderProps<{ amount: ... } ) : Element => (
        <DonateInput presets={[10, 20, 50, 100, 200]} amount={value} onAmount={onChange}/>
    )}
/>
```

Render prop
Passing JSX or a function that returns JSX as a prop


That's all Folks!



React - Part 4
(of 4)

Contents

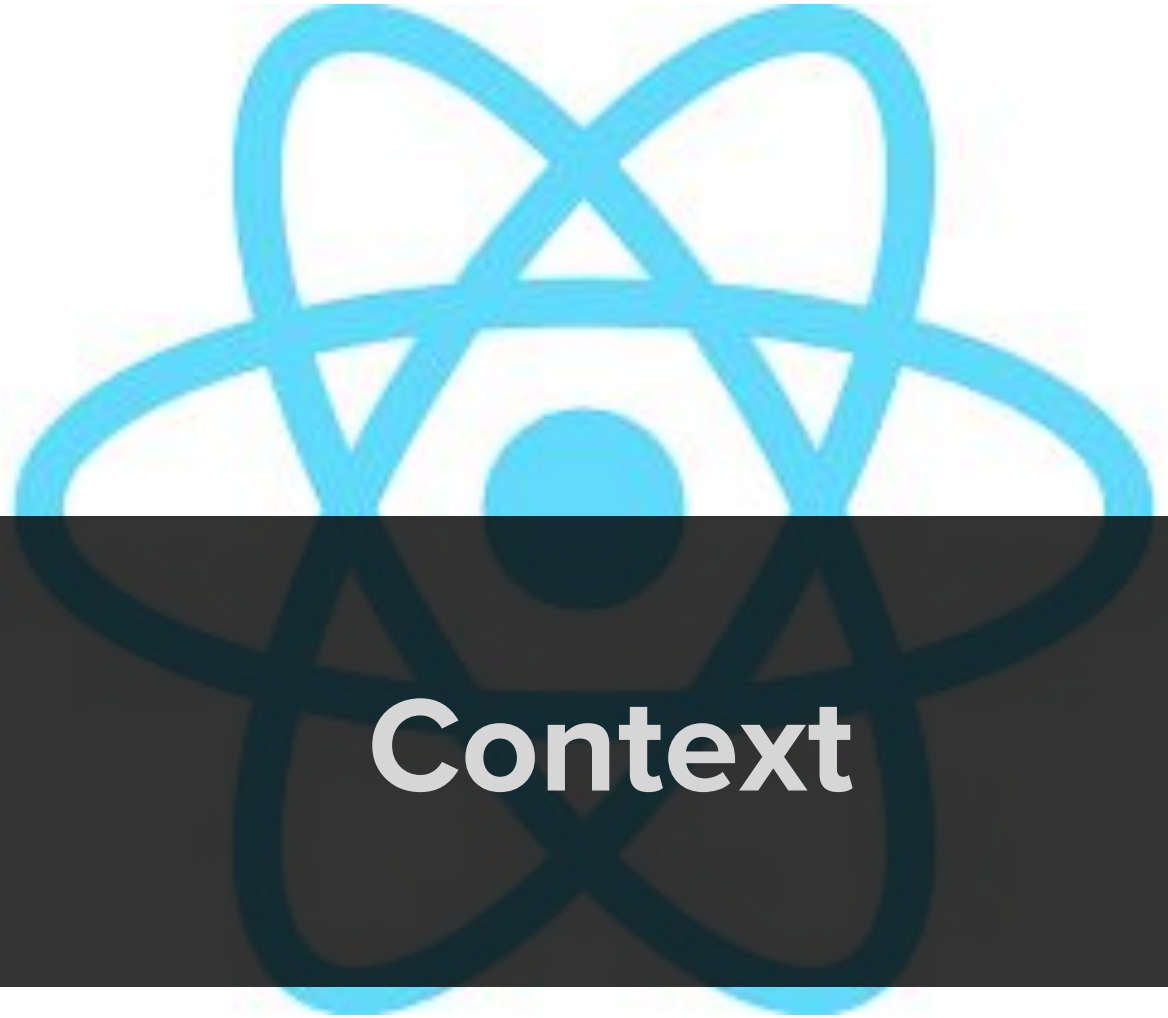
Context

Hook libs

useRef

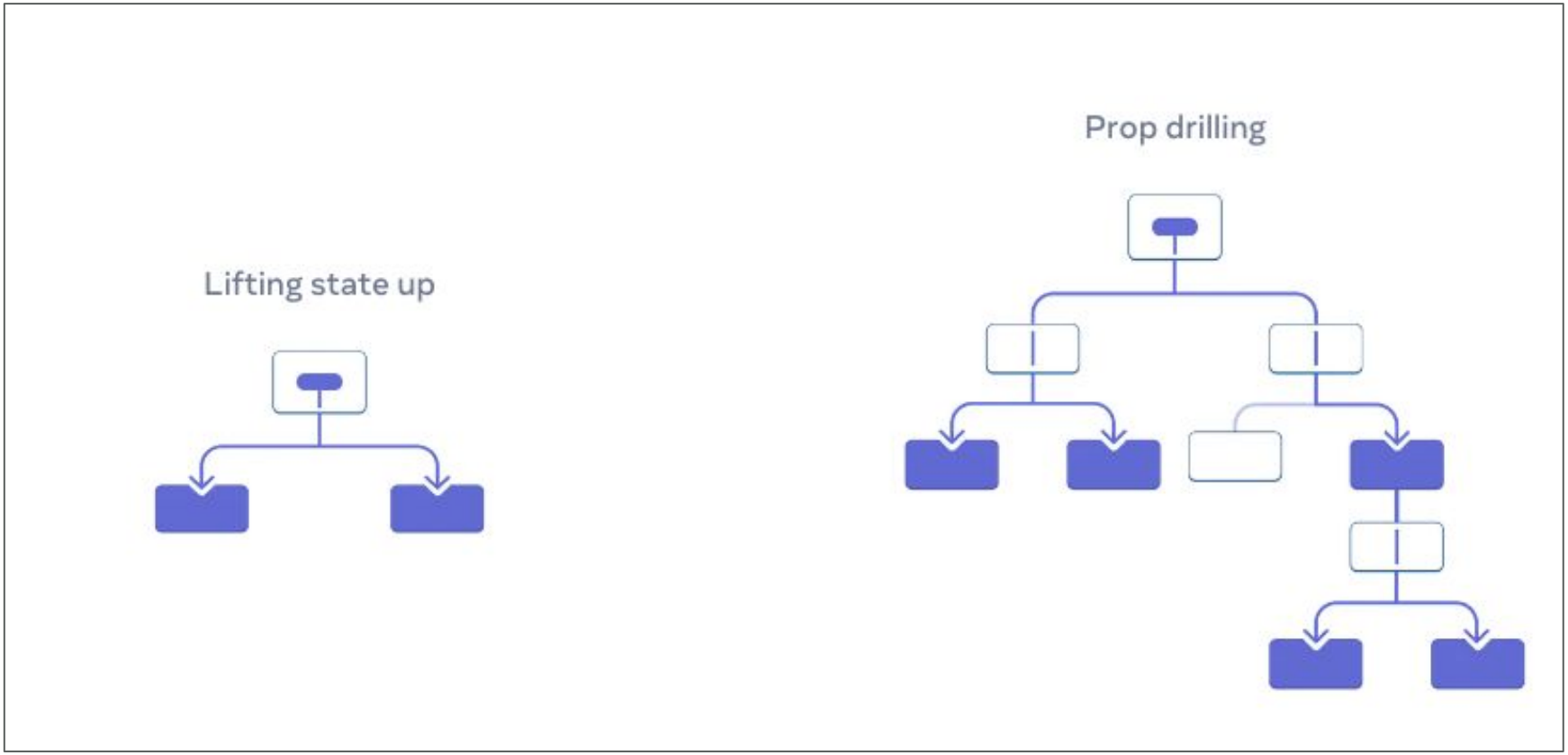
State management libs

Optimisations



Context

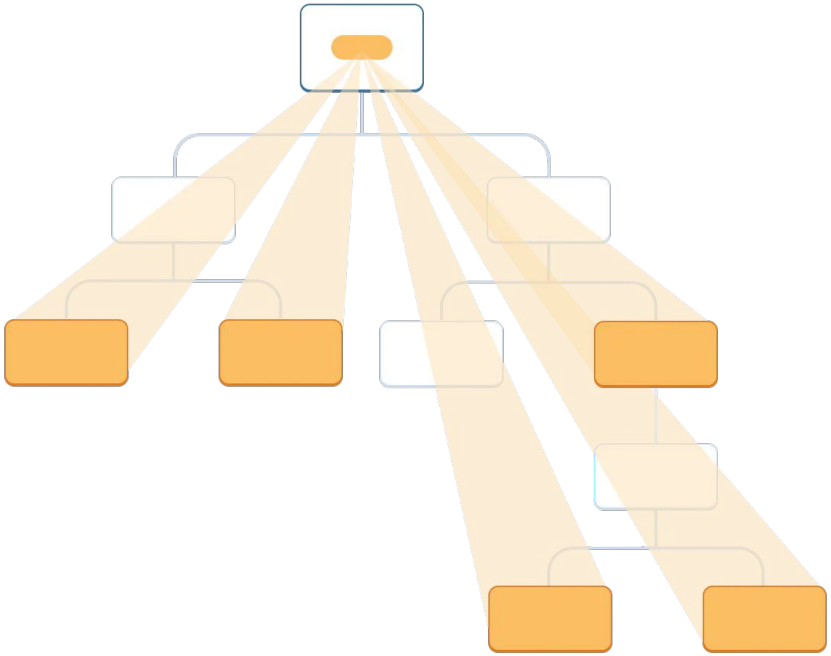
Prop drilling

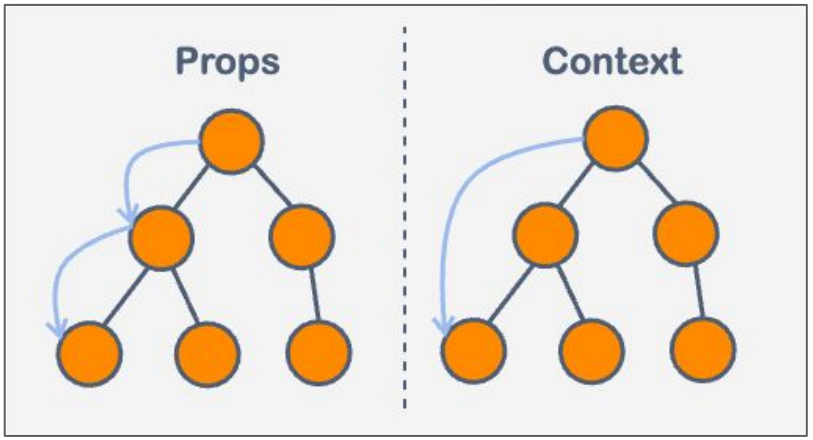


If you have **deep component trees** with components that **need the same state**, *'lifting state up'* might lead to prop drilling = the shared state sit's on top of the tree and needs to be passed down through a lot of levels of components

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.





I'm a context, I provide values (and functions) to everyone I surround

<ContextProvider>

<ComponentA>

I do not care about this context

<ComponentB/>

I like what you provide, thanks!

</ComponentA>

</ContextProvider>

≃ **dependency injection**, the React way...

- Complex apps can have many levels of nested components
  - Passing state through all children can get very complicated and confusing.
  - Code gets hard to read and maintain.

- What if we could keep all related state in one component and access it from all components that are ‘below’ it in the tree?
  - **Context to the rescue!**

- Context is designed to share data that can be considered “global” for a tree of React components, e.g.:
  - Authenticated user
  - Theme
  - Language
  - Shopping basket
  - ...

- Context docs

A context is a component that ‘wraps’ a (part of) the component tree and **provides** a **value** object to the child components (**consumers**) in that tree .

All consumers of a context will **re-render** whenever the **provider’s value prop changes**.

Advantages and disadvantages of context:

- **Cleaner code** that is easier to read. Less “noise” in components because less properties need to be explicitly passed down
- Related state is kept together

- Components are now tightly coupled to the context(s) they use:
  - Components get **harder to reuse**
  - Components get **harder to test**

Context are very useful, but not the best solution in all situations! See section “Before you use context” and use cases for context

How to “use context”

1. Create a context with `React.createContext()` (e.g. ‘MyContext’)

2. Add the component ‘`MyContext.Provider`’ to the tree and provide a value. The provided value is made accessible to all components that use this context.

3. use the `‘useContext’` hook in a component that is a child to the provider component.

Check the **context-demo** on Canvas

Component types

Props only component

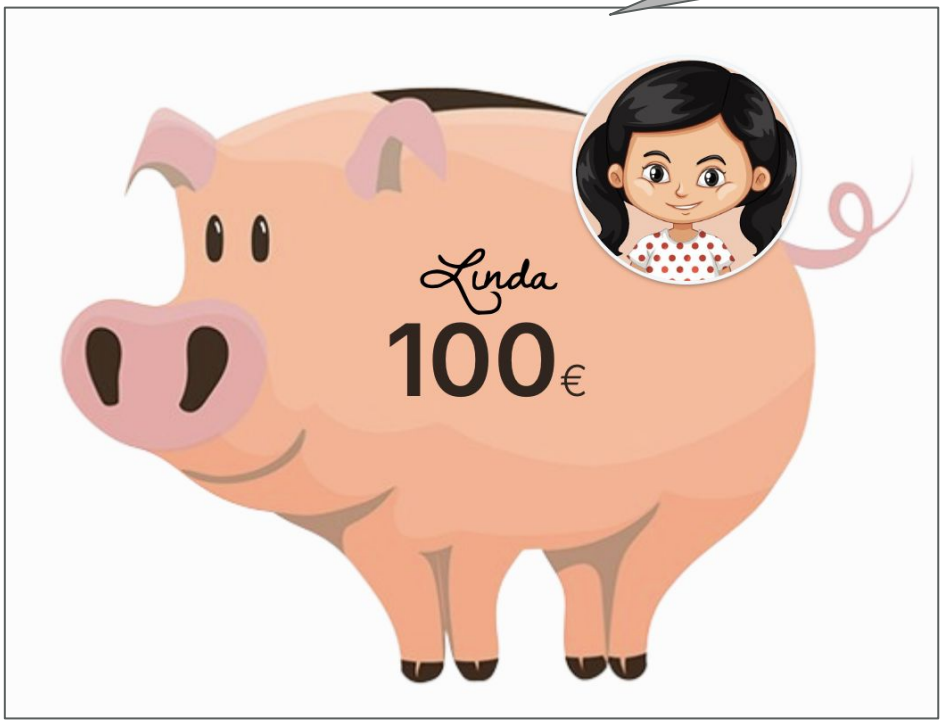
`input props,`

`callback props,`

`children`

`local state`

Just give me data and I will do my job



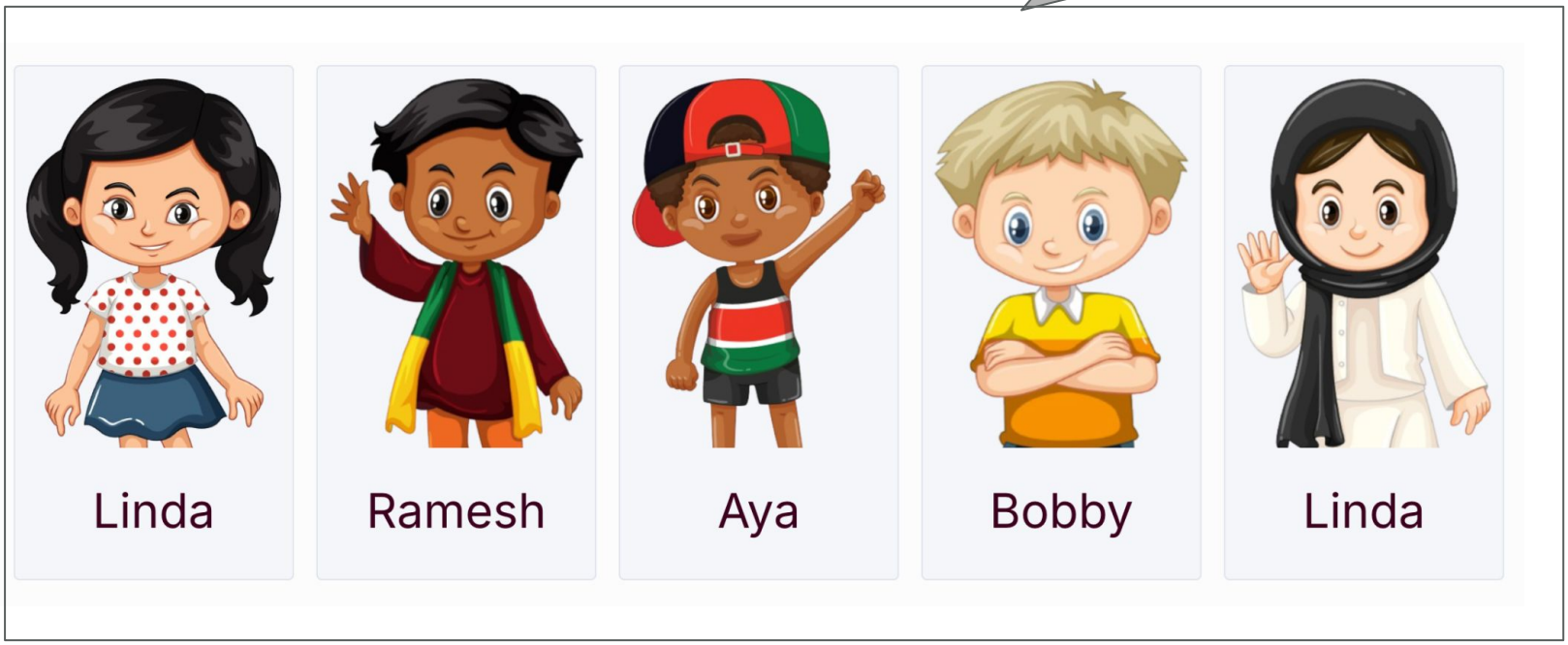
PiggyBankView

Component with side-effects

`useEffect`,

`useQuery`,

`...`



PiggyBankListPage

Routing aware component

`useParams`

I assume that I am coupled to a URL that contains an ID

PiggyBankDetailPage

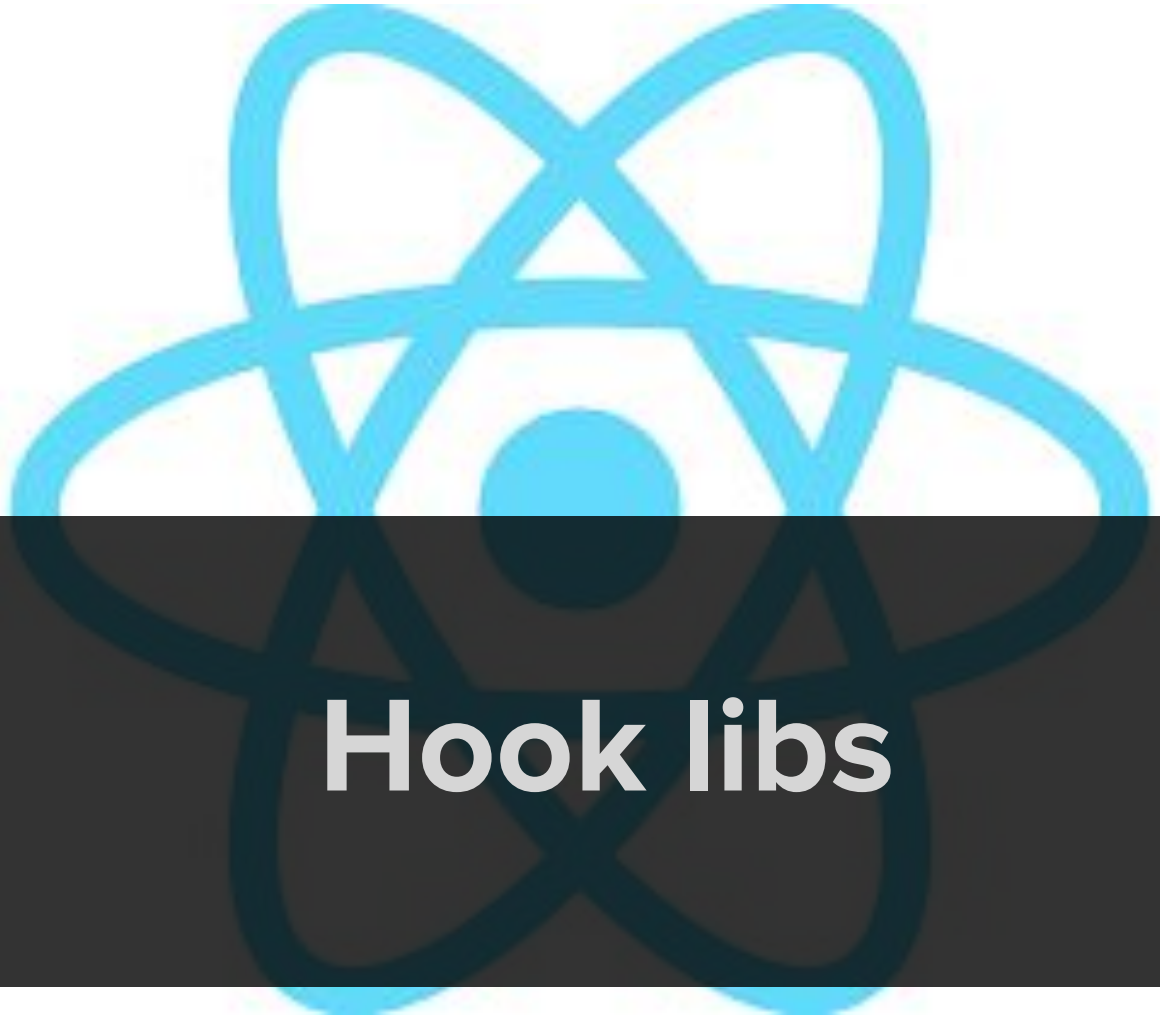
(this component also has side-effects: it loads his data based on the route id)

Context aware component

`useContext`

I assume that I'm wrapped in a context that supplies me a LoggedInUser

PiggyBankDetailPage



Hook libs

Recap - Rules of hooks

useState and *useEffect* are the most famous React hooks, but there are others like *useRef*,,... (we will discuss these later on), and can also write your own **custom hooks**.

But wait! There are some rules ...

Hooks can only be used if we adhere to "the rules of hooks"

1. Name always start with 'use'
2. Only call hooks on the top-level (not conditionally, not in a loop,...)!
3. Only call hooks from React functions
   a. React components (are functions!)
   b. Custom hooks

Why?
Because React relies on the order in which hooks are called to return the right value

Build your own hooks

You can call other hooks (useState, useEffect, custom hooks,...) from your own hook. Return an object containing your data, relevant states, ...

Example **`useToggle`** hook

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

export default function useToggle(
  initialValue: boolean
): [boolean, () => void] {
  const [value, setValue] = useState<boolean>(initialValue);

  const toggleFunction = () => setValue(!value);

  return [value, toggleFunction];
}
```

Can be used in the same way a useState hook is used, but it only exposes a toggle function.

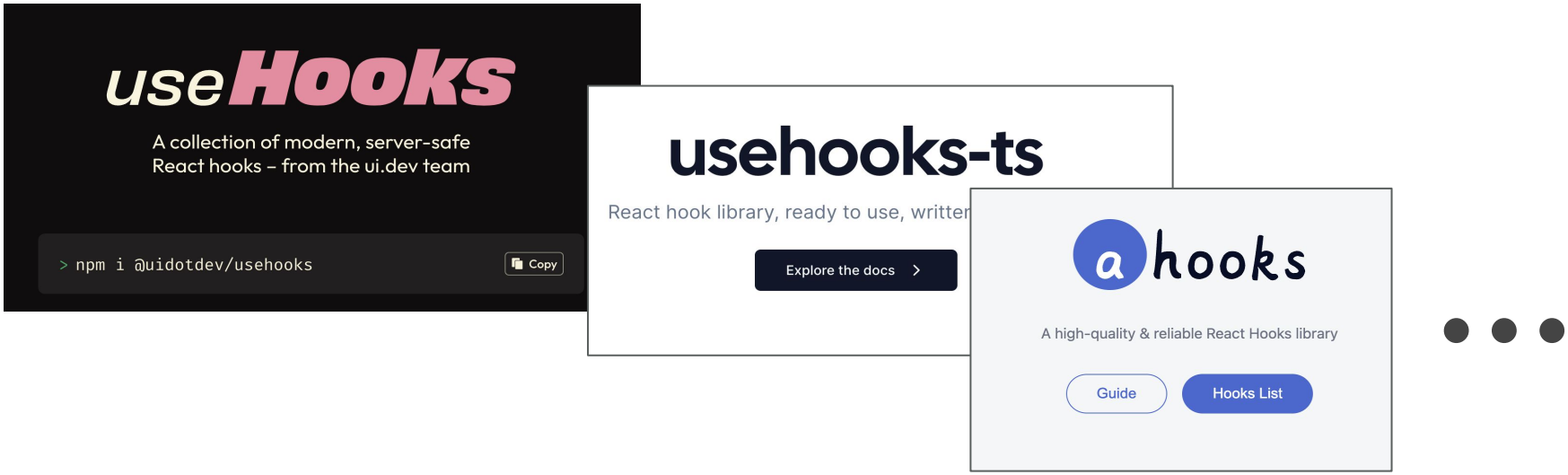
```
const [someValue, toggle] = useToggle(true);
```

Example: **usePiggyBank** hook

```
export function usePiggyBank(id: string) {
    const {isLoading, isError, data: piggybank} =
useQuery({
        queryKey: ['piggybanks', id],
        queryFn: () => getPiggyBank(id),
    })

    return {
        isLoading,
        isError,
        piggybank,
    }
}
```

This hook uses the **useQuery** hook (provided by React Query) which in turn uses **useState** and **useEffect** internally.

The are lots of **hook libraries** available to attach all kinds of functionality to your components and hooks



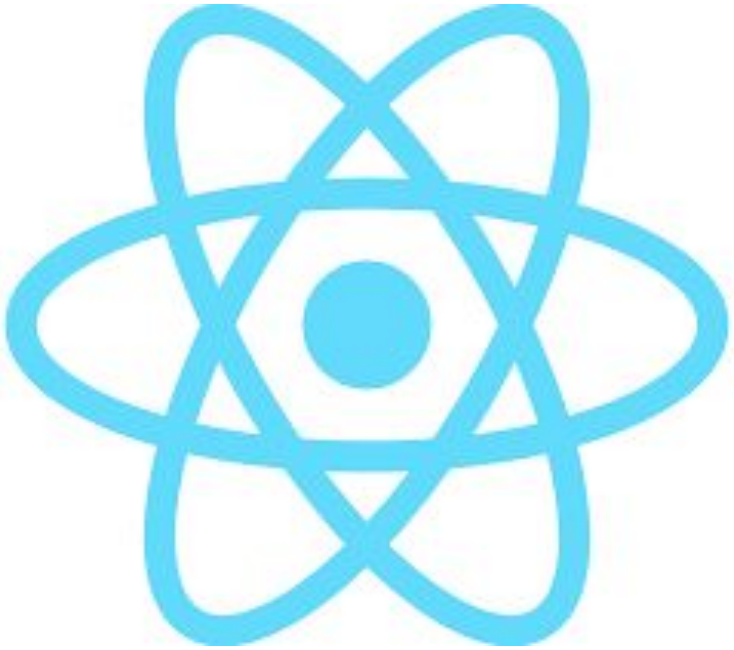
Example

https://usehooks.com/

PiggyBankView.tsx

```
const [showFields, setShowFields] = useLocalStorage("show-fields", true)
```

By replacing useState with useLocalStorage, the component now stores the value of showFields in local storage instead of in memory...



useRef

useRef

The useState hook allows you to attach ‘memory’ to a functional component (or to a custom hook)

Calling a setter returned from **useState** triggers a **rerender** (= call of the component function)

There may be situations where you want to remember and manipulate something **without triggering a rerender**:

- Storing timeout IDs
- Storing and manipulating DOM elements
- Storing other objects that aren’t necessary to calculate the JSX.

If your component needs to store some value, but it doesn’t impact the rendering logic, choose refs.

https://react.dev/learn/referencing-values-with-refs

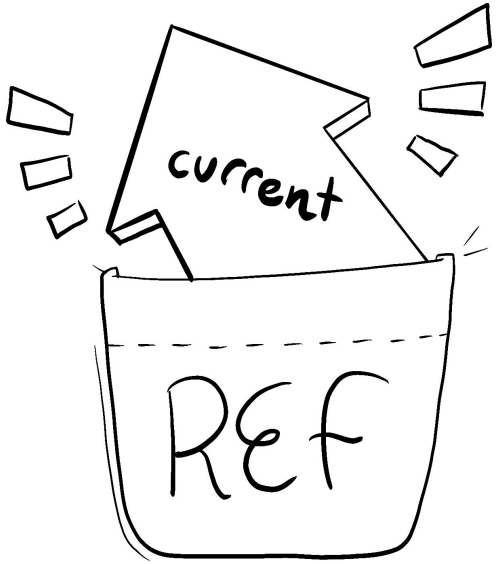
Refs are an “escape hatch” you won’t need often

useRef

```
const ref = useRef(0);
```

returns

```
{
  current: 0 // The value you passed to useRef
}
```




```
// Inside of React
function useRef(initialValue) {
  const [ref, unused] = useState({ current: initialValue });
  return ref;
}
```

useRef

refs	state
`useRef(initialValue)` returns `{ current: initialValue }`	`useState(initialValue)` returns the current value of a state variable and a state setter function (`[value, setValue]`)
Doesn't trigger re-render when you change it.	Triggers re-render when you change it.
Mutable—you can modify and update `current` 's value outside of the rendering process.	"Immutable"—you must use the state setting function to modify state variables to queue a re-render.
You shouldn't read (or write) the `current` value during rendering.	You can read state at any time. However, each render has its own snapshot of state which does not change.

Refs are an "escape hatch" you won't need often

Manipulating the DOM using refs

One of the advantages of a framework like React is that you **do not need to manipulate the DOM**. This makes code much more ‘reactive’, readable and testable.

~~`document.getElementById(), document.querySelector(), ...`~~

But sometimes it may be necessary to have access to the DOM:

- focus an input field
- scroll into view,...
- ...

React allows this. It ‘exposes’ a ref attribute on every JSX element (div, input,...). When you assign your ref (created with useRef) to this attribute, React will assign the DOM element to ref.current

Manipulating the DOM using refs

```
export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

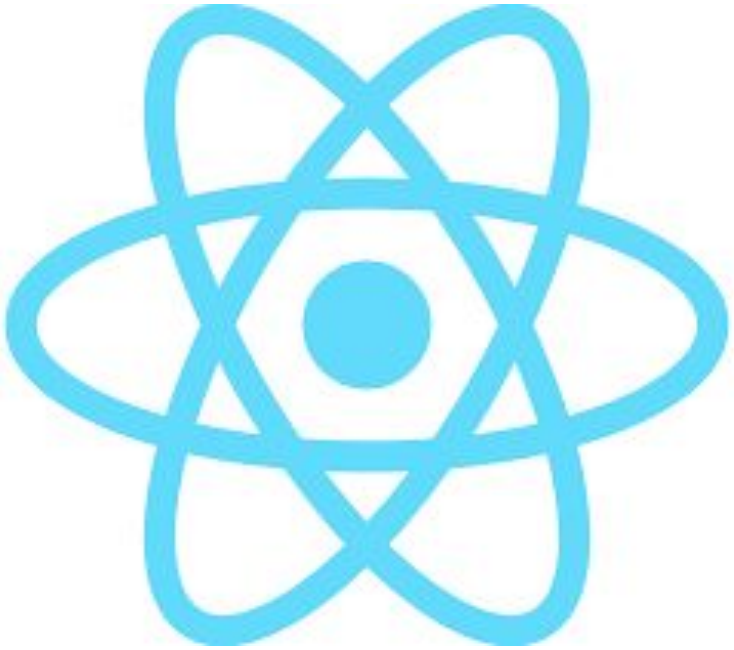
https://react.dev/learn/manipulating-the-dom-with-refs#example-focusing-a-text-input

Manipulating the DOM using refs

If you want to ‘expose a ref’ on a component that you have built yourself, you will need to foresee the ref as a prop

The ref is ‘forwarded’ to the JSX element, react will assign the actual DOM element to ref.current

```
function MyInput({ref, ...props}) {
    return (
        <>
            <div>This is a super fancy input</div>
            <input {...props} ref={ref}/>
        </>
}


function App() {
    const inputRef = React.useRef()
    return <MyInput ref={inputRef} placeholder="Hello"/>
}
```



State management libraries

State management libraries

- State shared by multiple components can get complicated, confusing and hard to read / maintain when the application grows larger
- Using the "**lifting up state**" principle will help a lot, but it can only get you so far.
  - When state needs to be passed down a few components ("Prop drilling"), this still lead to confusing and hard to maintain situations
- → **Context** can reduce the "prop drilling" and clean up your app
- → Backend state can be handled by libraries like **React Query**

- But the larger your app gets, the more state gets added. If you reach a point where state becomes too difficult to manage or too cluttered, consider using a **library** specialized in managing **state.**

State management libraries

State management libraries help you manage your state in a structured, efficient way.

- Adds a uniform, structured way of updating state
- Abstracts away some of the complexity
- Reduces the chance to introduce bugs

- Adds (a lot of) complexity to your app
- Might involve (a lot of) refactoring
- (sometimes) Steep learning curve

Use the concept of “stores” that contain data (state). React to mutations on the data

State management libraries

Most popular / used state management libraries:

1. Redux
2. Zustand
3. MobX
4. Jotai
5. Recoil

Mobx

Jōtai

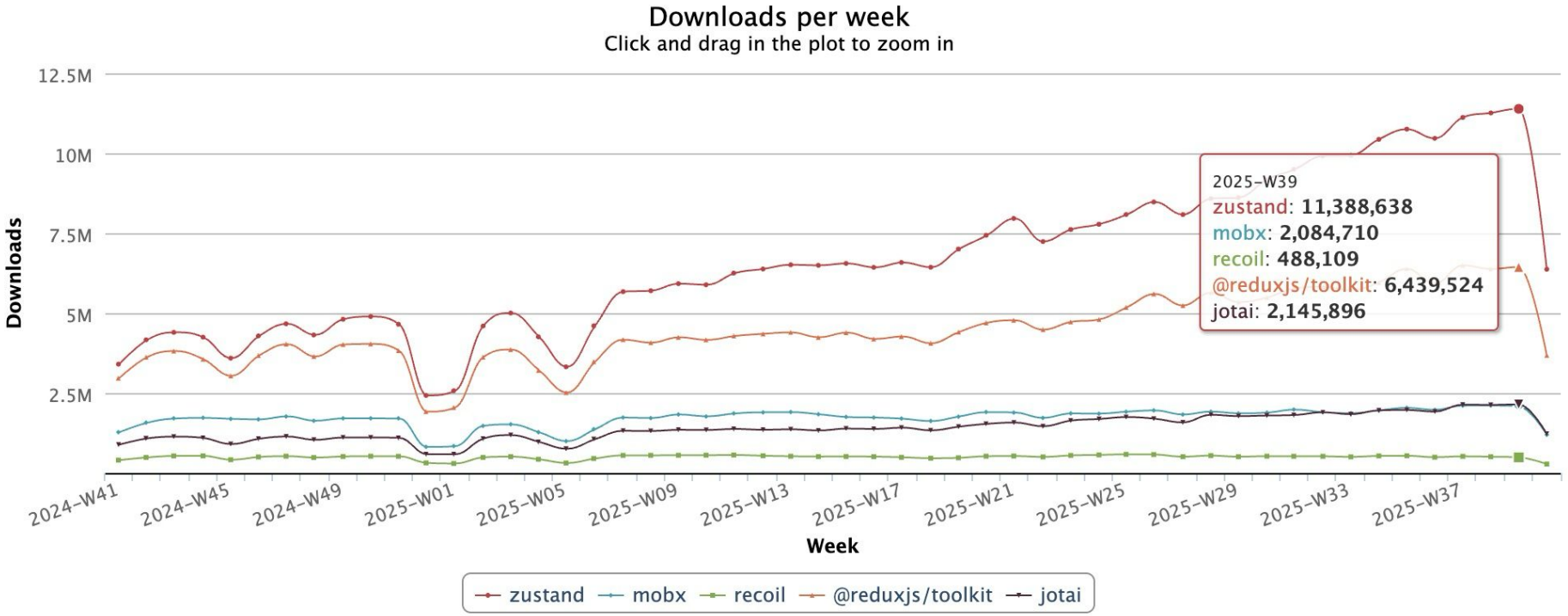
Redux

recoil

no longer maintained

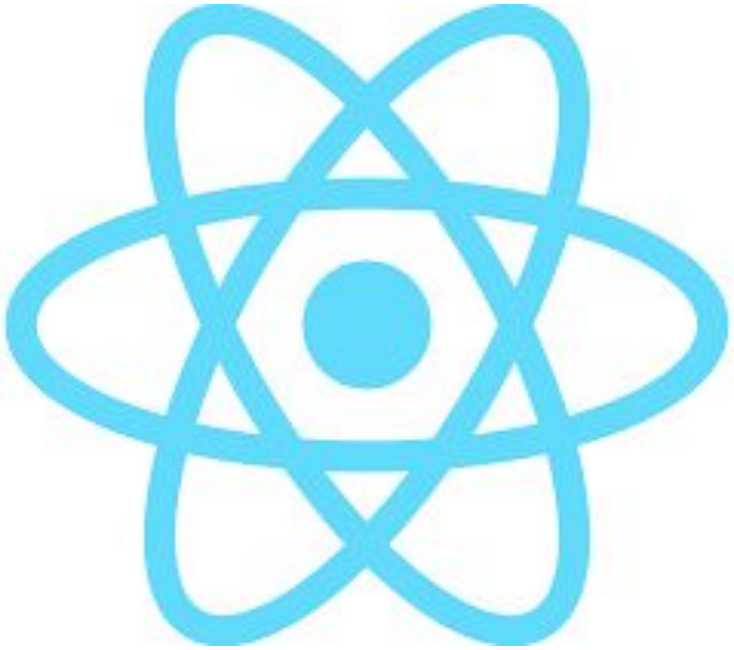
State management libraries

[oktober 2024 tot oktober 2025]



Check the **zustand-demo** on Canvas

Check the **recoil-demo** on Canvas



React Compiler

Re-redering elements

React components often re-render even when their props/state haven’t changed.

Consequences:

- Slower UI performance ⚡
- Wasted CPU cycles, especially in large apps
- React DevTools shows frequent re-renders for parent/child components

Solution:

Optimize component updates at compile time

How React Compiler Helps

babel-plugin-react-compiler (used in Babel) analyzes JSX during build to reduce unnecessary re-renders.

🧐 ***?? Memoization*** *= storing a function's result to* ***reuse it later*** *when given the same inputs, avoiding recalculation.*

Features:

- **Automatic memoization hints** for functional components
- Optimizes component updates without changing code logic
- Modes: all | annotation | infer
  - **`annotation`** → Only components marked `"use memo"` are optimized.
  - **`infer`** → Auto-detects components; `"use memo"` can override.
  - **`all`** → Aggressive, optimizes everything; `"use no memo"` to exclude.

⚡ Optional note: **SWC** is an alternative modern compiler for React (used in Vite for speed) but **does not yet perform React Compiler optimizations**.

Enable React Compiler

1. **Install Babel plugin:**

```
> npm install -D babel-plugin-react-compiler
```

2. **Update vite.config.ts:**

```ts
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react({
      babel: {
        plugins: [
         ['babel-plugin-react-compiler', {
             compilationMode: 'all' // 'all' |
         'annotation' | 'infer'
         }],
        ],
  })],
});
```

3. **Run dev/build** → React Compiler automatically reduces unnecessary re-renders.

Check the **react-compiler-demo** on Canvas


That's all Folks!

KdG Karel de Grote Hogeschool

Programming 6

Frontend security

Image Designed by vectorjuice / Freepik

Contents

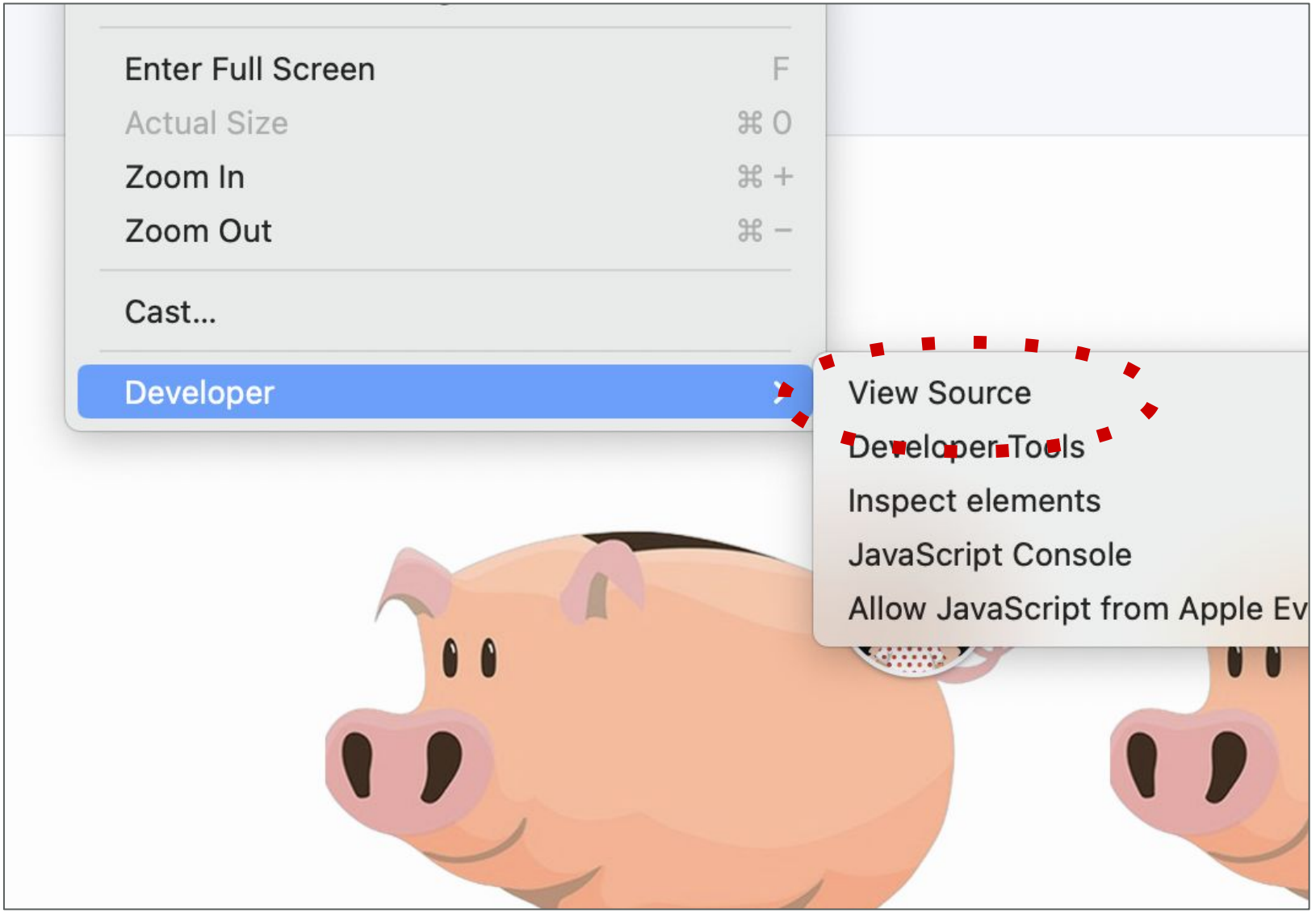
Security is a very broad topic...
We limit ourselves in these slides to aspects related to the development of **single page applications** (React...) in combination with an IDP (like Keycloak...) and a backend (like Spring...)

Authentication

CORS

XSS

Authentication


Enter Full Screen F
Actual Size ⌘ 0
Zoom In ⌘ +
Zoom Out ⌘ −
Cast...
Developer
View Source
Developer Tools
Inspect elements
JavaScript Console
Allow JavaScript from Apple Ev

React applications run in a browser. Source code is **public** by definition.

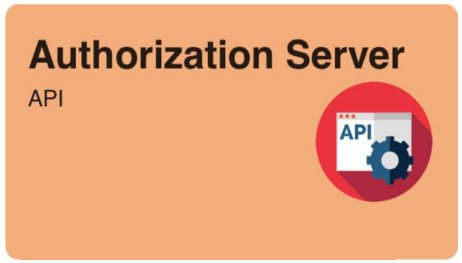
You cannot store client secrets in a browser
(for instance to prove to a token endpoint that your app is really the app and not an attacker with the secret)

→ **Use an IDP with redirects to handle login**
(or use a BFF (backend for frontend) with a session cookie….)

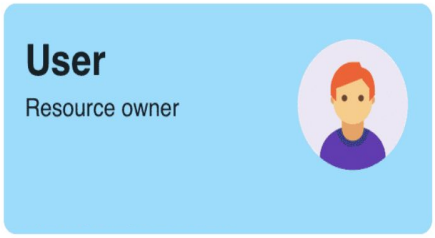
Browsers are vulnerable to XSS (see further)

→ **Do NOT store tokens in localstorage**
→ **Use short token lifespans, refresh token rotation,..**

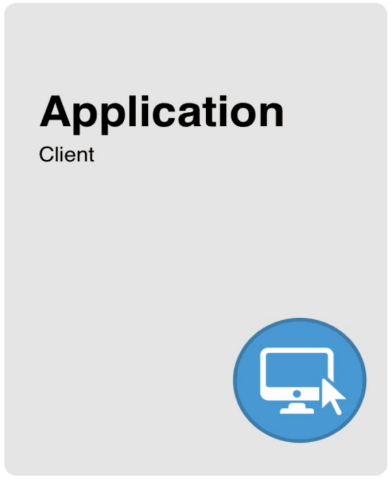
Who’s dancing?



Keycloak IDP



Human, service,..



React client

Spring backend

Using which tokens?

Identity (ID) token

Contains identity information about the resource owner (name, profile pic,...).

It proves that a user has been authenticated.

Can be used to show user info in the UI.

Access token

Provides access to protected resources. Powerful token! Short lifespan!

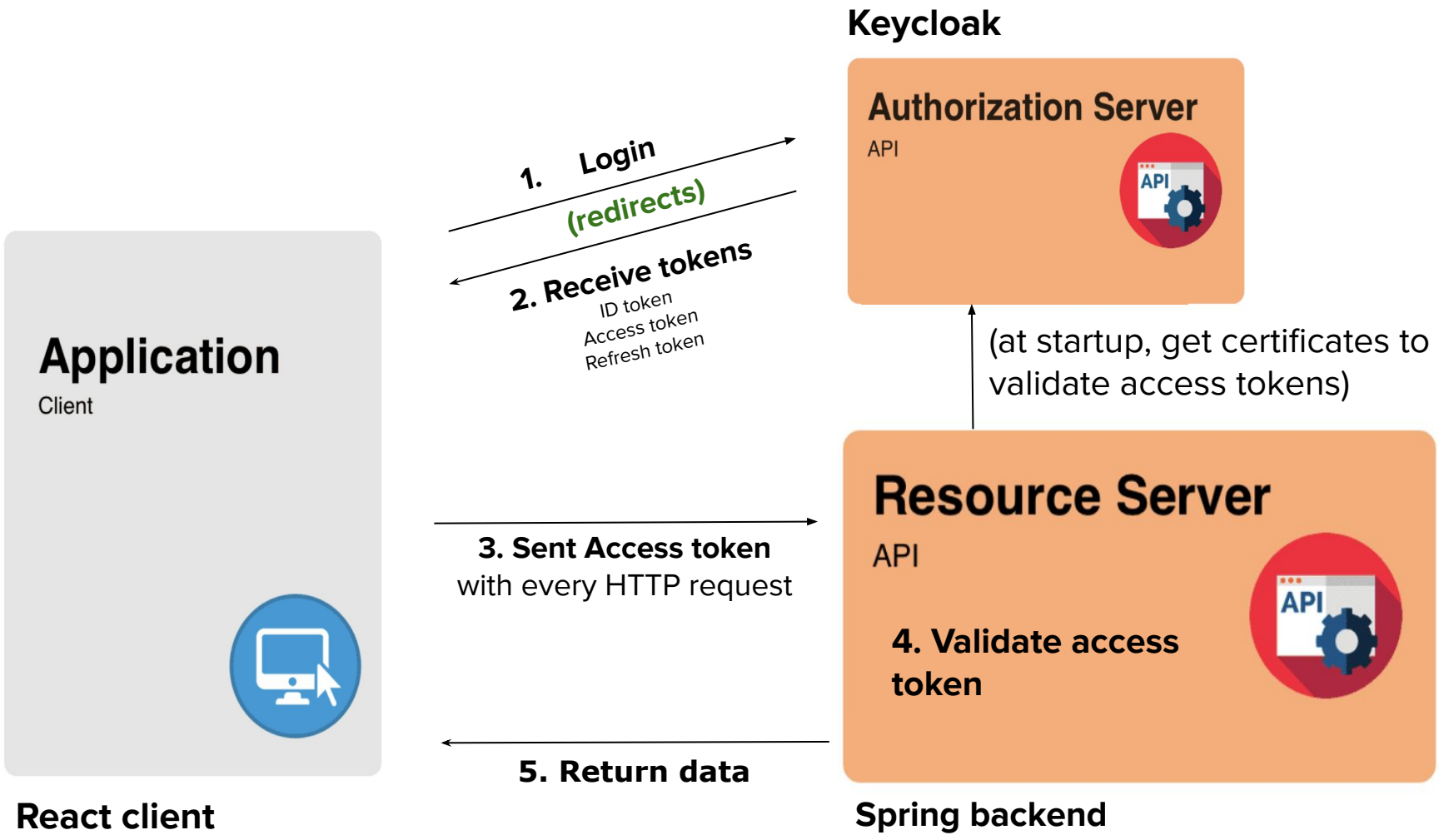
Added to the Authorization header ('bearer' token)

Can contain a role (admin,...) that can be used in the UI to show/hide certain pages/buttons

Refresh token

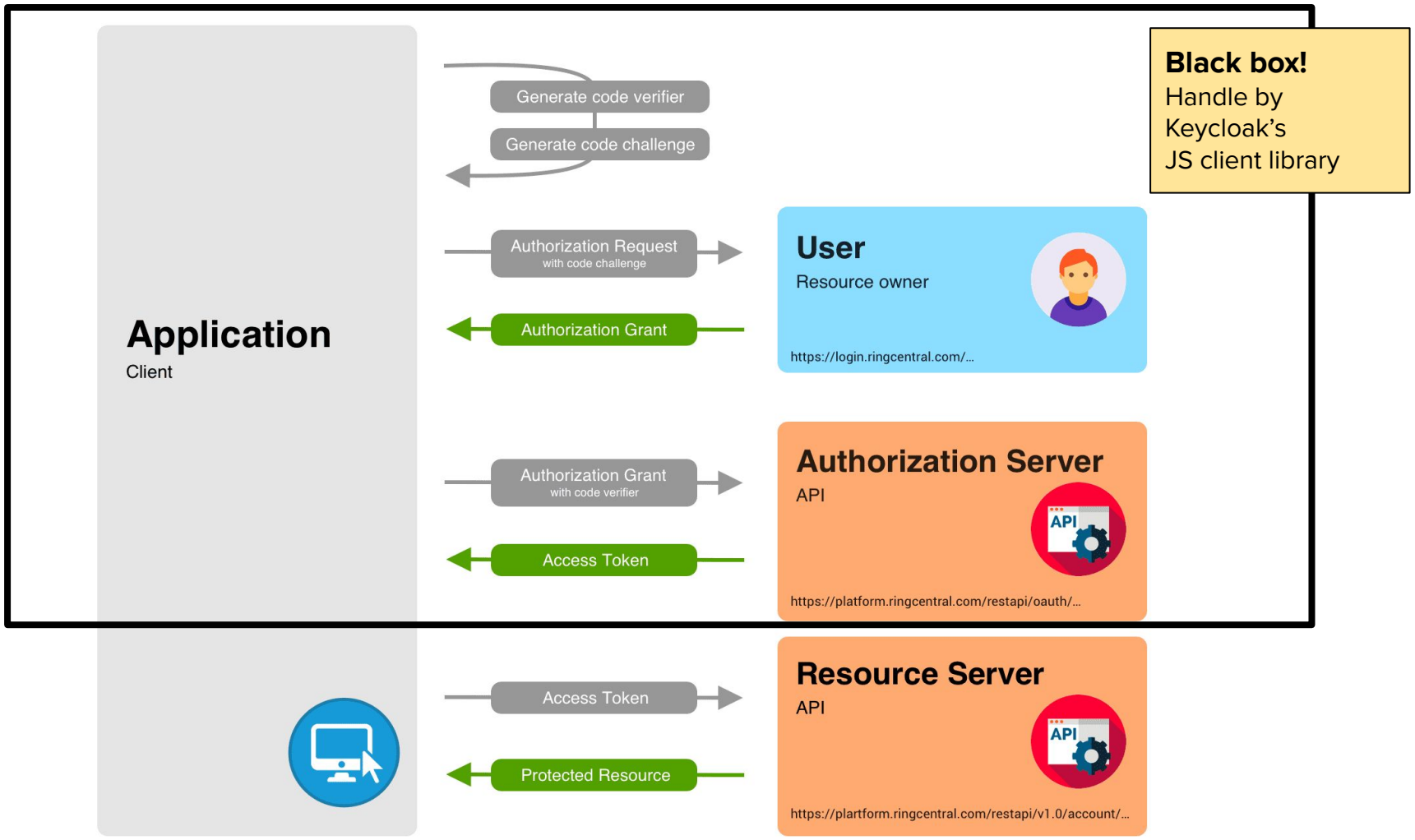
Used to retrieve new access token when this has expired

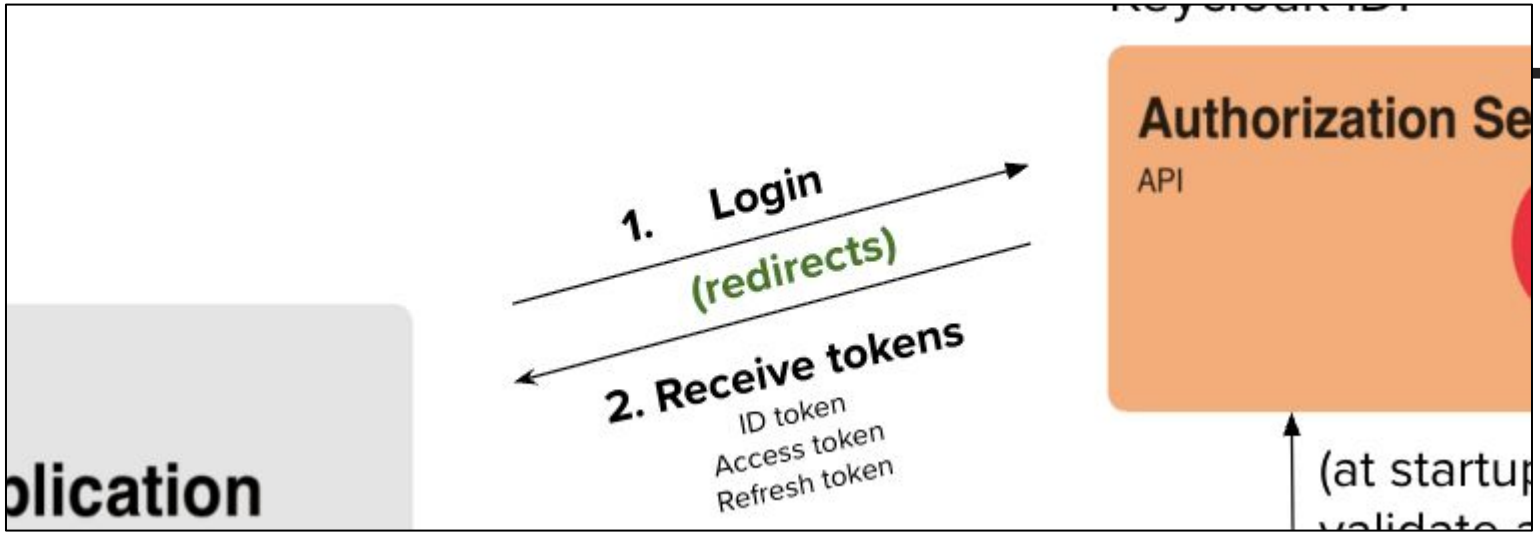
Token flow


Keycloak
Authorization Server
API
API
1. Login
(redirects)
2. Receive tokens
ID token
Access token
Refresh token
(at startup, get certificates to validate access tokens)
Application
Client
Resource Server
API
API
3. Sent Access token
with every HTTP request
4. Validate access token
5. Return data
React client
Spring backend

Complete flow

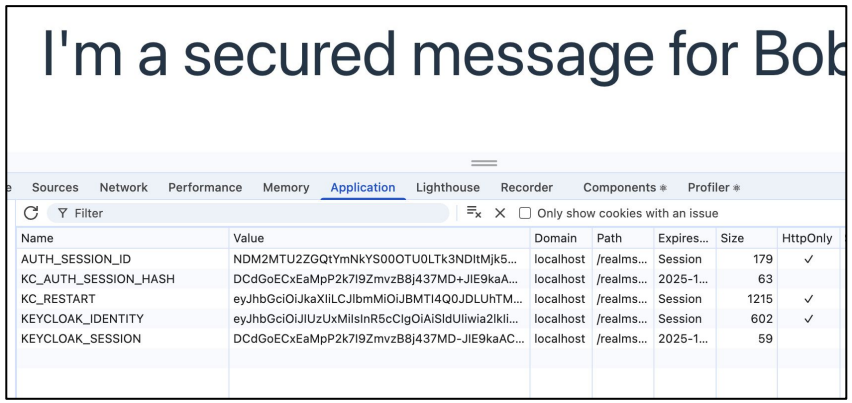
= Authorization Code Flow with PKCE (“pixee”)





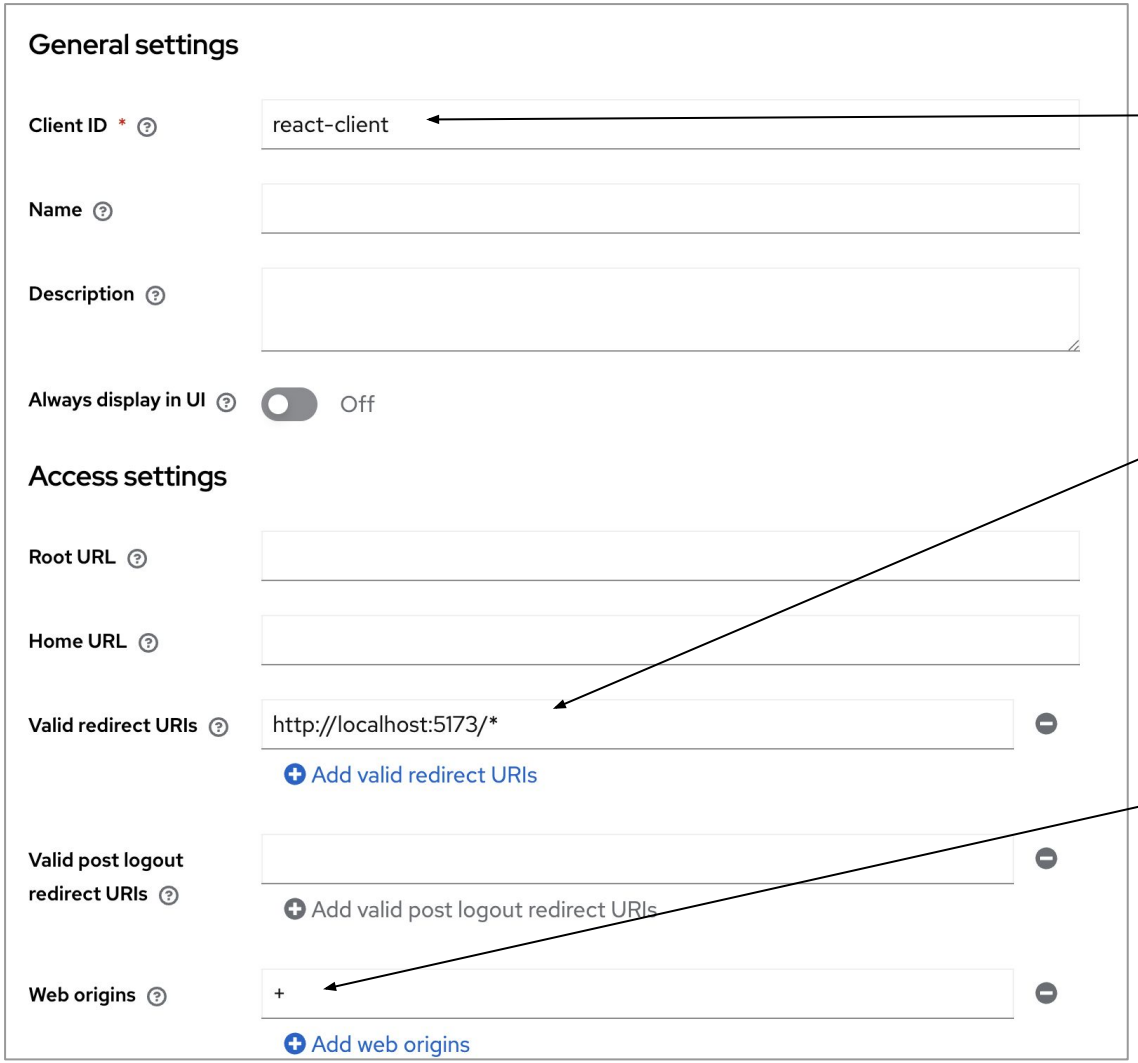
After step 2, a ***secure*** (only sent over https) and ***httpOnly*** (not accessible from JS) session **cookie** exists between the React application and KC.

This is used by the KC adapter to check if the user is still logged in (SSO-Single Sign On)



Keycloak configuration

Configure a client in de KC admin panel under the Realm for the application (1/2)



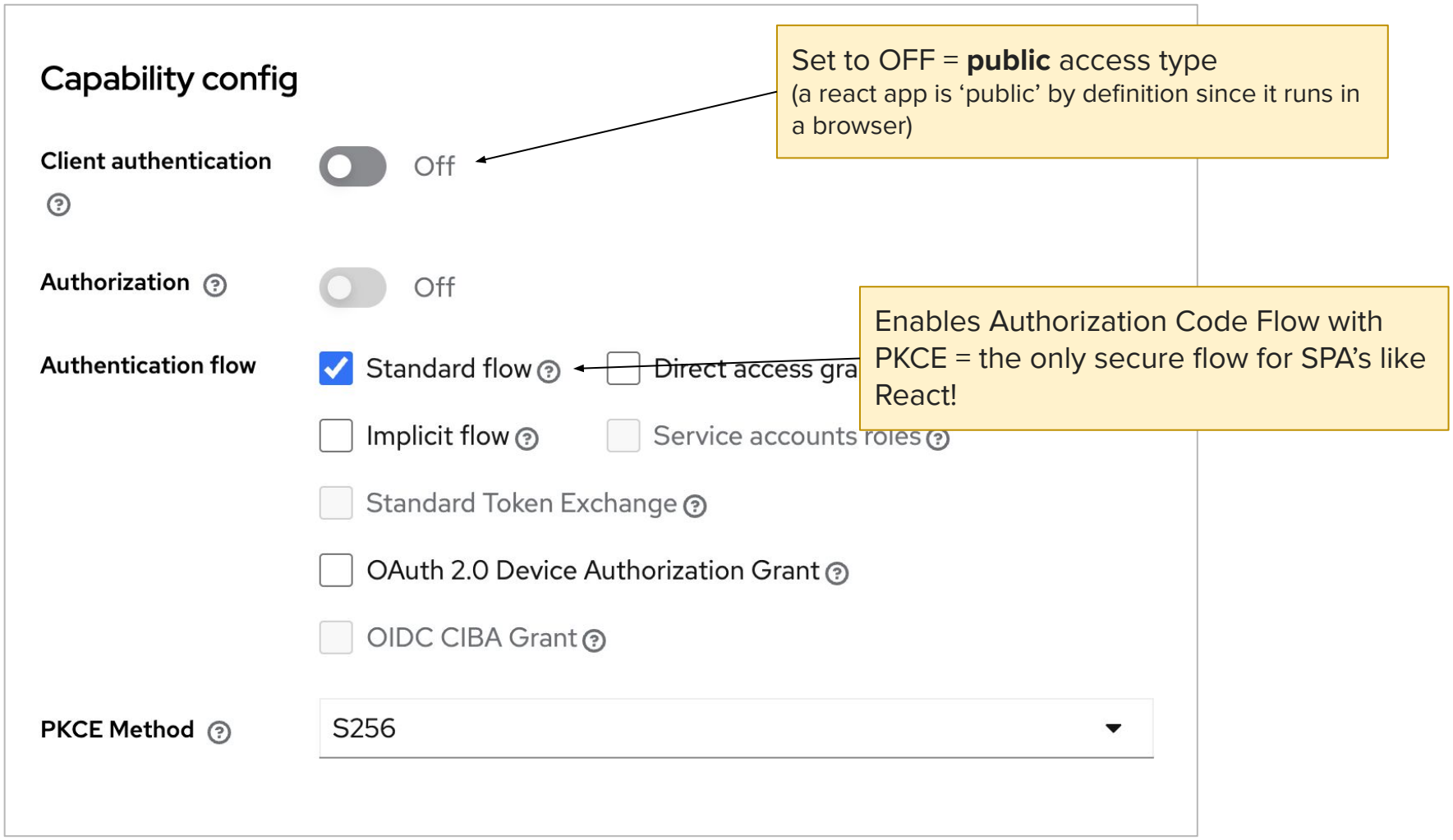
Client ID, see further

URL of the React app

Enables CORS between React and KC
(+ means valid redirect URI's)

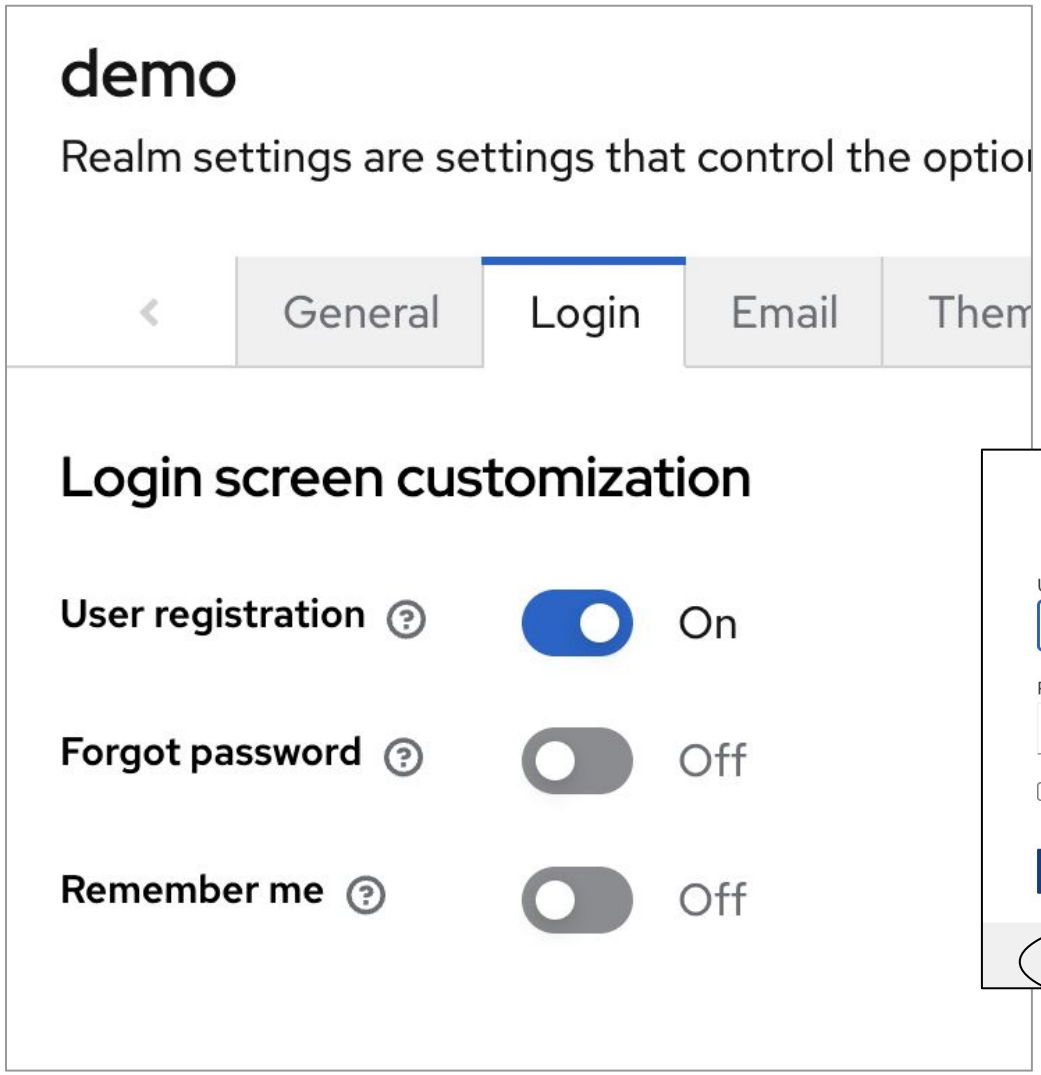
Keycloak configuration

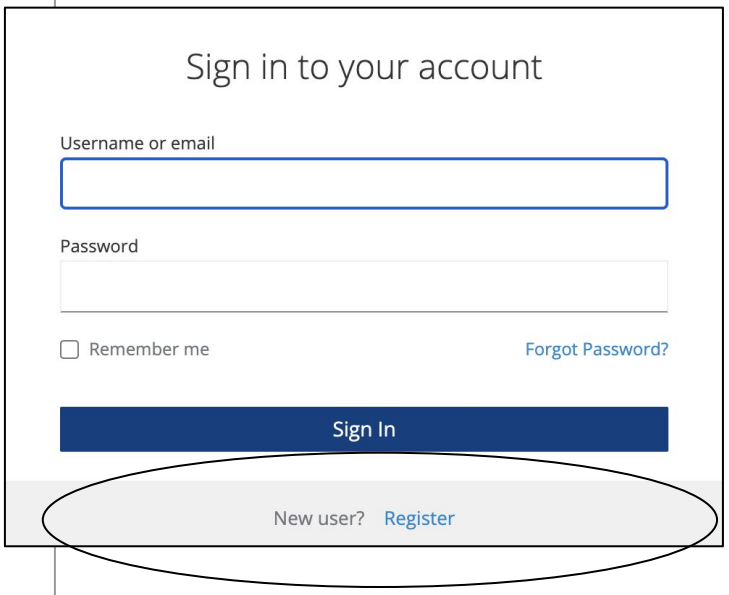
Configure a client in de KC admin panel under the Realm for the application (1/2)



Keycloak configuration

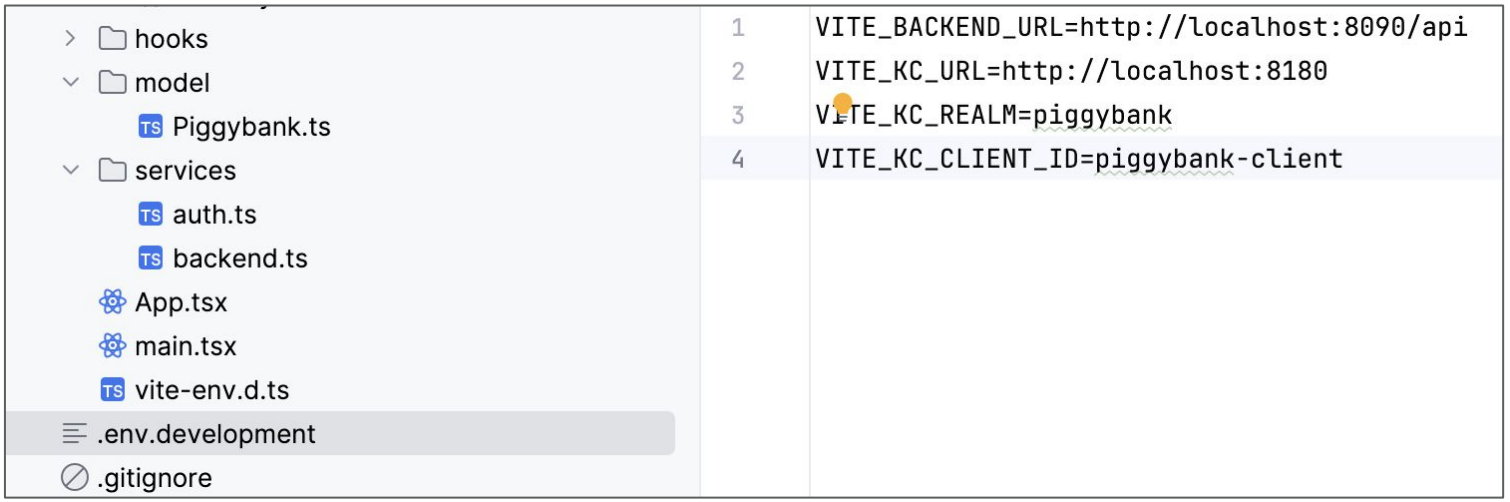
If needed, add roles, users, user registration etc...





React

Setup the necessary variables pointing to your KC instance, backend, realm and client-id. You can use vite's .env files for this...



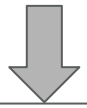
Install the KC Javascript adapter

React

In **main.tsx**, remove strict mode
In strict mode, all components are initialised/rendered twice to detect possible bugs. The init method of Keycloack.js cannot cope with this...

(there are other workarounds for this like checking this with a ref, ommitted here)

```
createRoot( container: document.getElementById( elementId: 'root')!).render(
  children: <StrictMode>
    <App />
  </StrictMode>,
)
```



```
createRoot( container: document.getElementById( elementId: 'root')!).render(
    children: <App/>
)
```

React

Typically a **context** is used to init the client library and provide security info to your components
(see example code on Canvas)

```
const keycloakConfig = {
    url: import.meta.env.VITE_KC_URL,
    realm: import.meta.env.VITE_KC_REALM,
    clientId: import.meta.env.VITE_KC_CLIENT_ID,
}

const keycloak: Keycloak = new Keycloak( config: keycloakConfig)

export default function SecurityContextProvider({children}: PropsWithChildre
    const [loggedInUser, setLoggedInUser] = useState<User | undefined>( initial
    const [isInitialised, setIsInitialised] = useState( initialState: false)

    useEffect( effect: () : void => {
        keycloak.init( initOptions: {onLoad: 'check-sso'})
    }, deps: [])

    keycloak.onReady = () : void => {
        setIsInitialised( value: true)
    }

    keycloak.onAuthSuccess = () : void => {
        addAccessTokenToAuthHeader( token: keycloak.token)
        updateUserFromToken()
    }

    keycloak.onAuthLogout = () : void => {
        removeAccessTokenFromAuthHeader()
    }

    keycloak.onAuthError = () : void => {
```

There are two onLoad options

- **login-required** will force the user to the KC login page
- **check-sso** will only check if the user is logged in

React

Pass the received access token as a bearer to all outgoing HTTP calls

```
export function addAccessTokenToAuthHeader(token: string | undefined) : void  {
    if (token) axios.defaults.headers.common['Authorization'] = `Bearer ${token}`
    else {
        removeAccessTokenFromAuthHeader()
    }
}
```

```
export function removeAccessTokenFromAuthHeader() : void  {
    delete axios.defaults.headers.common['Authorization']
}
```

React

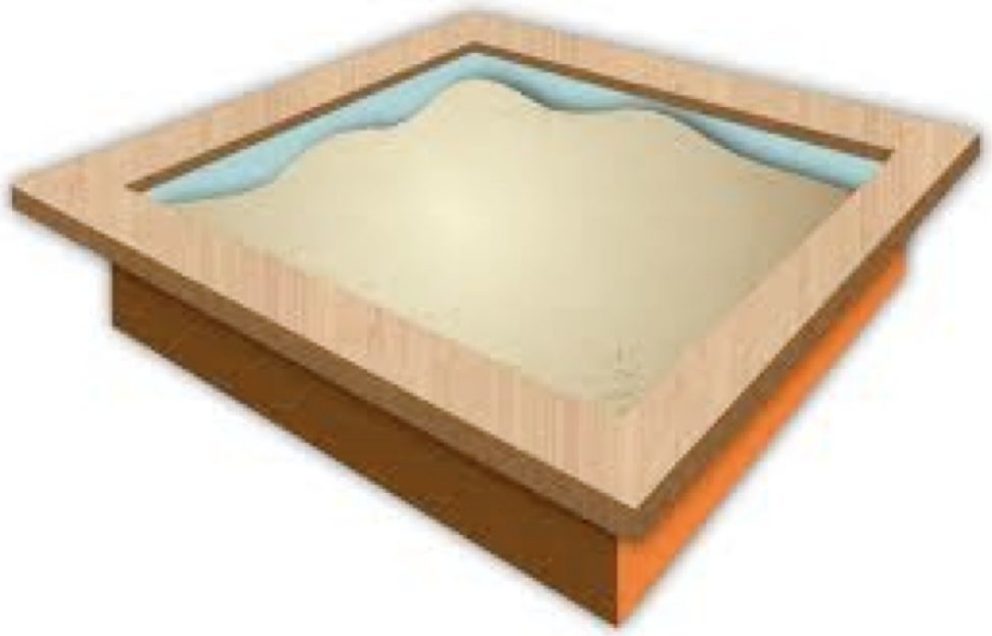
Protect your routes by (for instance) wrapping them in a guard

```
<QueryClientProvider client={queryClient}>
    <SecurityContextProvider>
        <BrowserRouter>
            <Routes>
                <Route path="/securedPage" element={<RouteGuard><SecuredPage/></RouteGuard>}/>
                <Route path="/publicpage" element={<PublicPage/>}/>
                <Route path="/" element={<Navigate to="/publicpage"/>}/>
            </Routes>
        </BrowserRouter>
    </SecurityContextProvider>
</QueryClientProvider>
```

```
export function RouteGuard({children}: PropsWithChildren) : string | number | bigint | boolean | Elem...
    const {isInitialised, isAuthenticated, login} = useContext( context: SecurityContext)

    useEffect( effect: () : void  => {
        if (isInitialised && !isAuthenticated()) {
            login()
        }
    }, deps: [isAuthenticated, login])

    if (!isAuthenticated()) {
        return <div>Authenticating</div>
    }

    return children
}
```

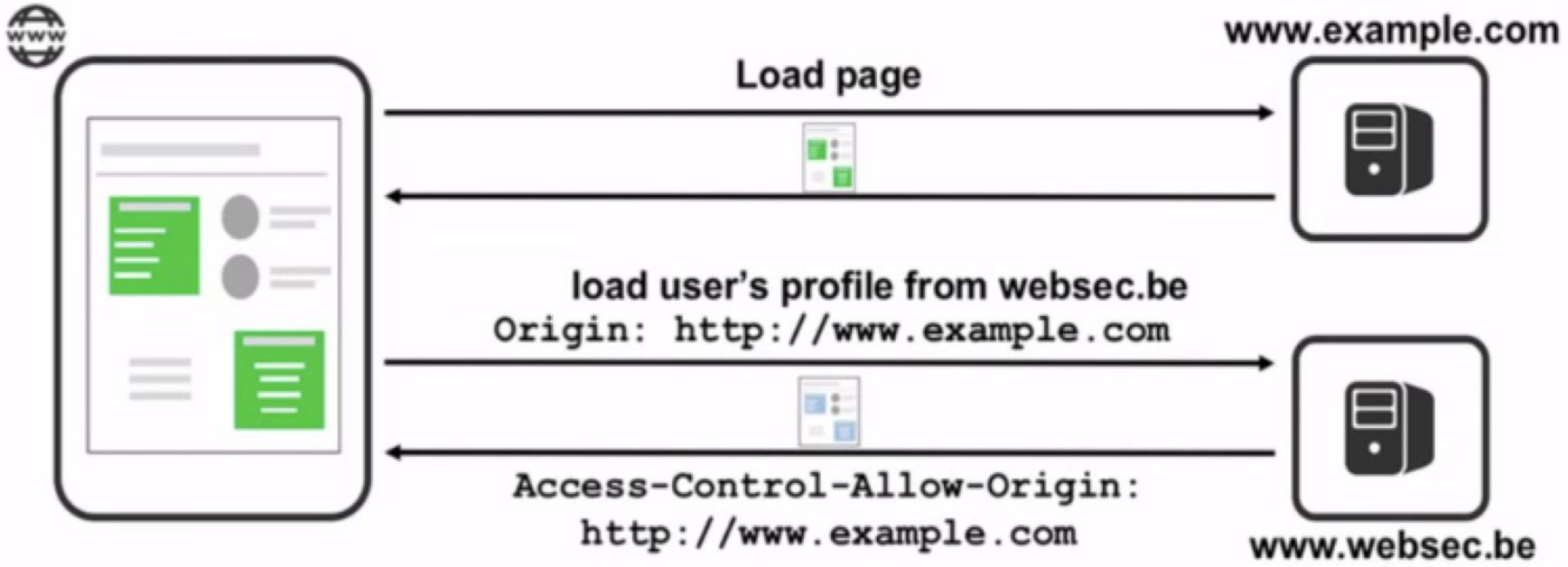
CORS

Browsers run web pages in a **security sandbox**



- Shielded from **local resources** (disk, camera,...)
- Shielded from **remote resources** that come from a **different domain** ('origin') than the domain from which the page was loaded
  = **SAME ORIGIN POLICY (SOP)**

How can you use data from another domain in your web app?
(or even from your domain but on a different subdomain or port)



www.websec.be needs to respond with a **Access-Control-Allow-Origin** header
(set to 'www.example.com' or to '*' to allow all domains)

Should be **configured in the backend** (Spring,...)

CSP

The src en href properties of <img>, <a>, <audio>,... tags are not restricted by CORS.

<img src=”url to another domain”> is perfectly valid HTML (although you can not access the pixels in the loaded image)

If you want to restrict this further, you can use Content Security Policy (CSP)

For example if you set this header, images can be loaded from any domain but media and scripts only from specific domains

```
Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com;
script-src userscripts.example.com
```

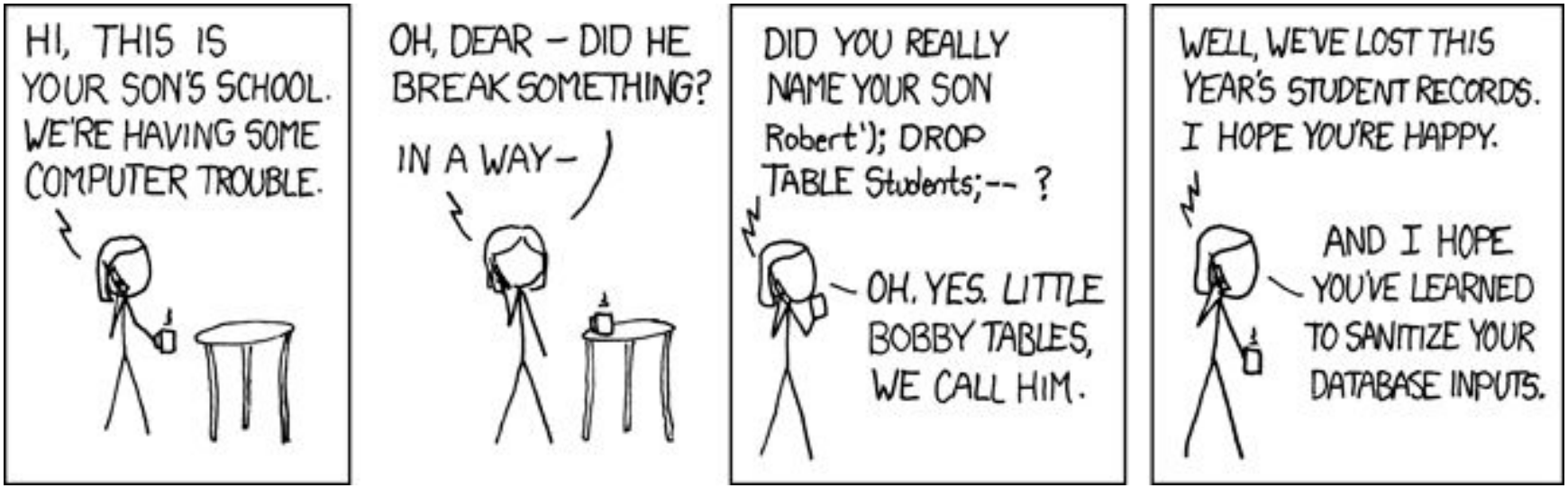
This helps you to reduce the risk on XSS attacks (see further)

XSS

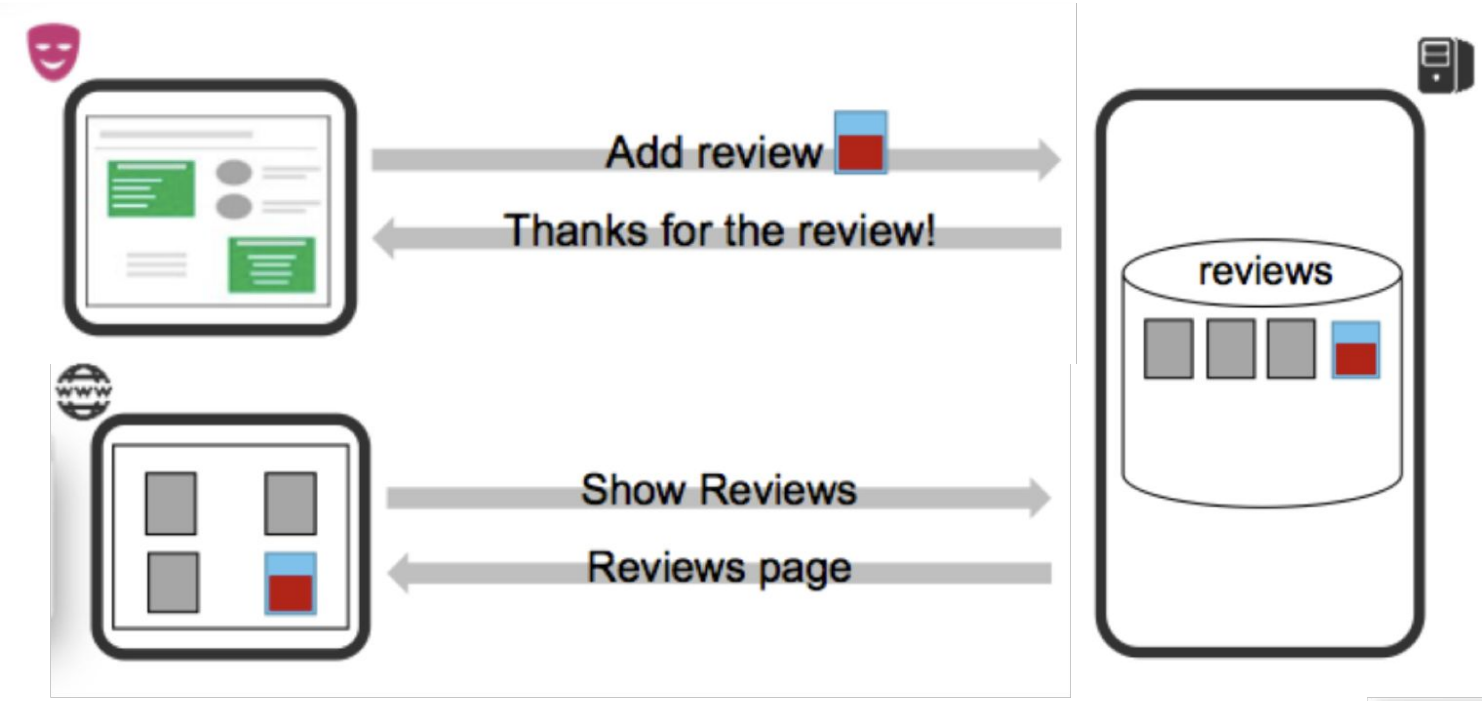
XSS

Users are security risks!

- Each ‘input’ is a possible attack vector
- What if a user inserts JS?
  - JS gets inserted where it wasn’t expected
  - A render of the unexpected input triggers the execution of the JS embedded in the input
  - XSS attack!



I can really recommend product X. It is awesome!
<script>alert('Never gonna let you down!')</script>

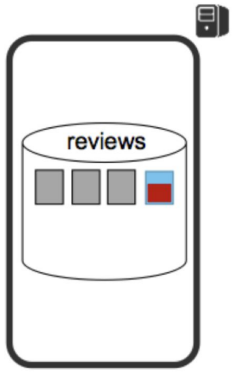


<html><body>… …</body></html>

XSS

1. Attacker adds this review

```
I can really recommend product X. It is awesome!
<script>alert('Never gonna let you down!')</scrip
```



2. Review gets stored unsanitized

3. Innocent user asks for reviews

4. Script code is executed in the context of the user

Attacker could use this security hole to steal user information or perform actions on their behalf

XSS and React

JSX protects against XSS attacks by default: it escapes values embedded in JSX ➜ It is **safe to render user input in JSX**.

<script>alert("XSS Attack")</script&gt

There is one exception!
Sometimes we need to render HTML defined by users or external systems (Rich Text Editor, API that returns HTML,...)

React has the '**dangerouslySetInnerHtml()**' function for that, but this loses the React DOM escaping!

Use this with care!

dangerouslySetInnerHtml

- The name is explicitly chosen to indicate danger
- React's replacement for 'innerHtml()' (don't use this)
- This is NOT a XSS safe method!
- Docs
- If you do need to use this method, use a library to **sanitize you HTML!**
  - DOMPurify is a good one
- Best practice:
  - Create a component that is responsible for sanitizing and rendering the HTML
  - When rendering user input that might contain HTML, use only this component!
  - That way, there is only one call to dangerouslySetInnerHtml in your code

dangerouslySetInnerHtml

```
import DOMPurify from 'dompurify'

export function SanitizedText({ input }: { input: string }) {
    return (
        <span
            dangerouslySetInnerHTML={{
                __html: DOMPurify.sanitize(input),
            }}
        />
    )
}
```

Check the **XSS** demo on Canvas!


That's all Folks!