
# React Server Components

## Explained for Backend Developers

# ABOUT ME

Jonas Bandi
jonas.bandi@ivorycode.com
Twitter: @jbandi

- Freelancer, in the last 10 years mainly in projects between modern web development and traditional business applications.
- Teaching at Berner Fachhochschule since 2007
- In-House courses & coaching for modern web technologies in the enterprise: UBS, Postfinance, Mobiliar, AXA, BIT, SBB, Elca, Adnovum, BSI ...

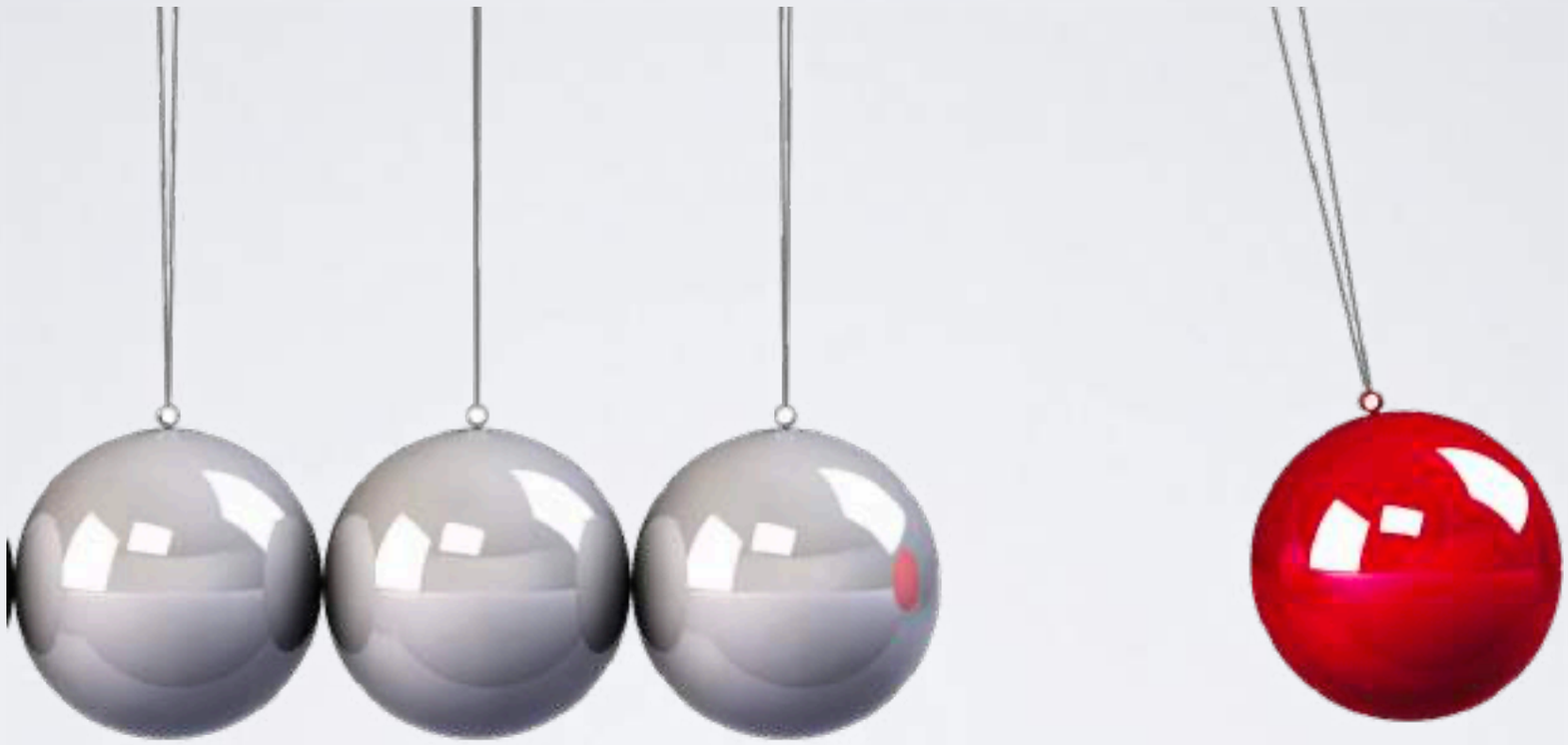
letsboot.ch
swiss dev training

digicomp

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

JavaScript / Angular / React / Vue / Vaadin
Schulung / Beratung / Coaching / Reviews
jonas.bandi@ivorycode.com

# The Pendulum is swinging ...



Server ⟵ ⟶ Client

... will it ever stop?

# React Server Components

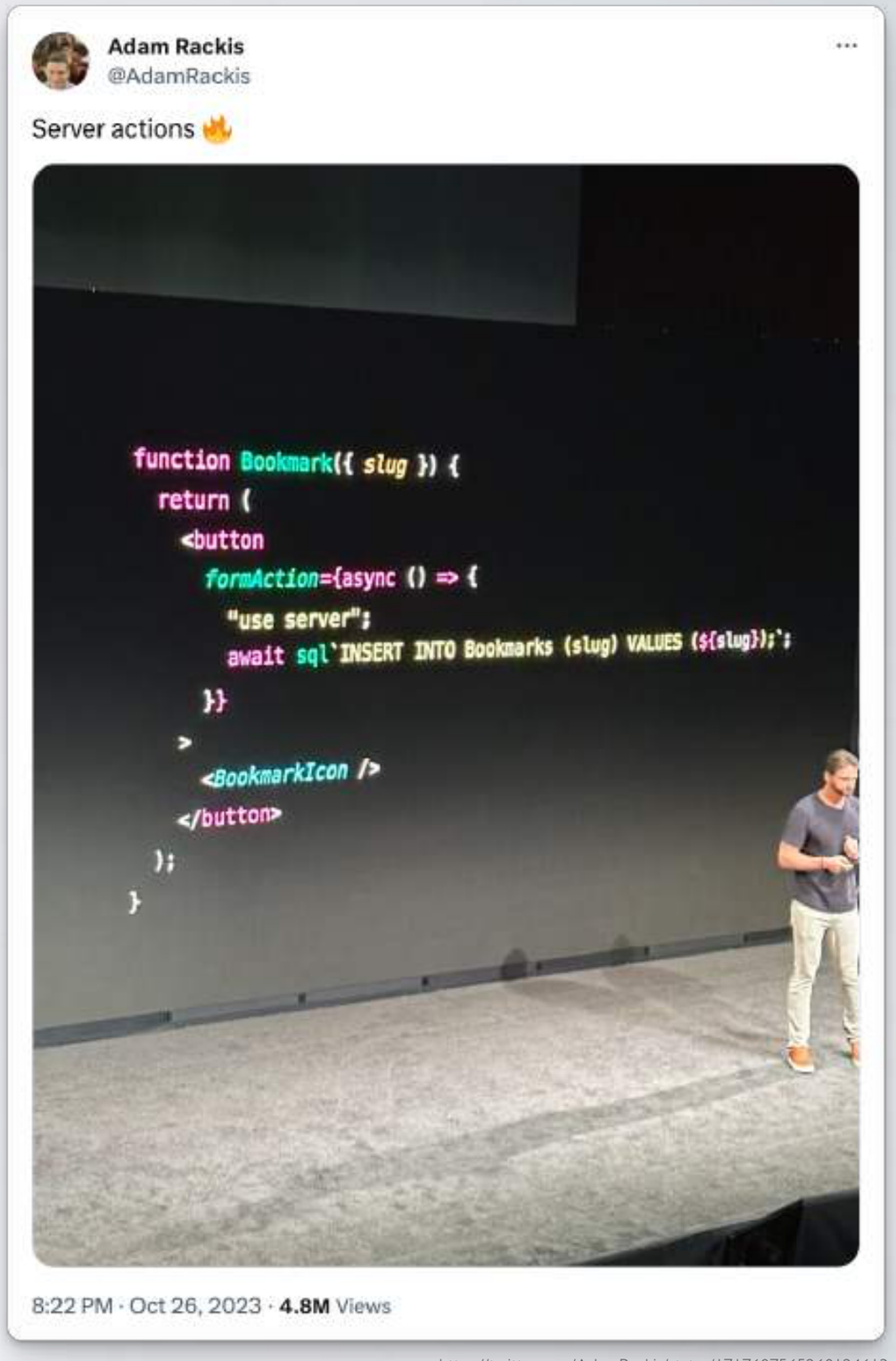

https://twitter.com/sebastienlorber/status/1737380479383277636


Are RSCs and NextJS Really That Bad?

Jack Herrington

21K views • 2 days ago

https://www.youtube.com/watch?v=u0OMdWJfdhg




https://twitter.com/AdamRackis/status/1717607565260124613

Original Talk: https://www.youtube.com/watch?v=9CN9RCzznZc&t

# React Server Components is an Architecture!

(Frameworks that support React Server Components have to implement the "React Architecture")
https://react.dev/

# React Server Components

Stable since React 19 (released December 2024)

- first announced in December 2020
- introduced for production in Next v13 (October 2022)
- today also supportd in Redwood and Waku
- Support announced for React Router v7 (formerly Remix)

React *as a library* provides the "foundation" for React Server Components but you need a Framework to use them (deep integration into Router and Bundler).

The official React documentation is recommending to use a "production-grade full-stack framework" (aka Meta-Framework):
https://react.dev/learn/start-a-new-react-project

Frameworks supporting RSCs:

NEXT.JS Waku

RSCs support announced:

React Router v7 (formely Remix)

RedwoodJS

The new kid on the block:
(no RSC support so far)

TanStack Start

# AGENDA

Where are we? - The Era of SPAs ...

A short React Refresher

React Server Components

Step-by-Step Introduction

Focus: Client - Server Interaction

*Prior Art:*

Island Architecture

RPC-style fetching and actions

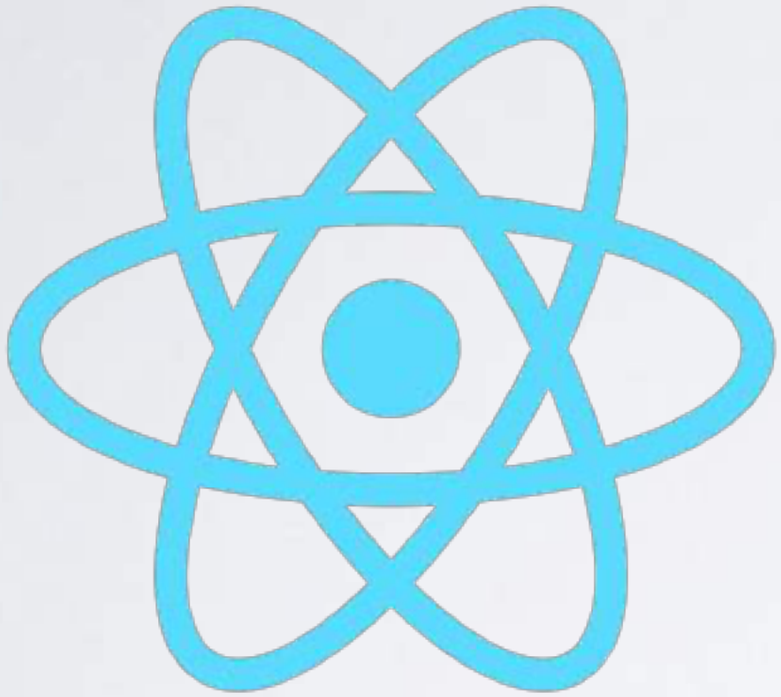
Server-Driven SPA

My goal is to make you feel like this:

# The Era of Single Page Applications

Server → Client

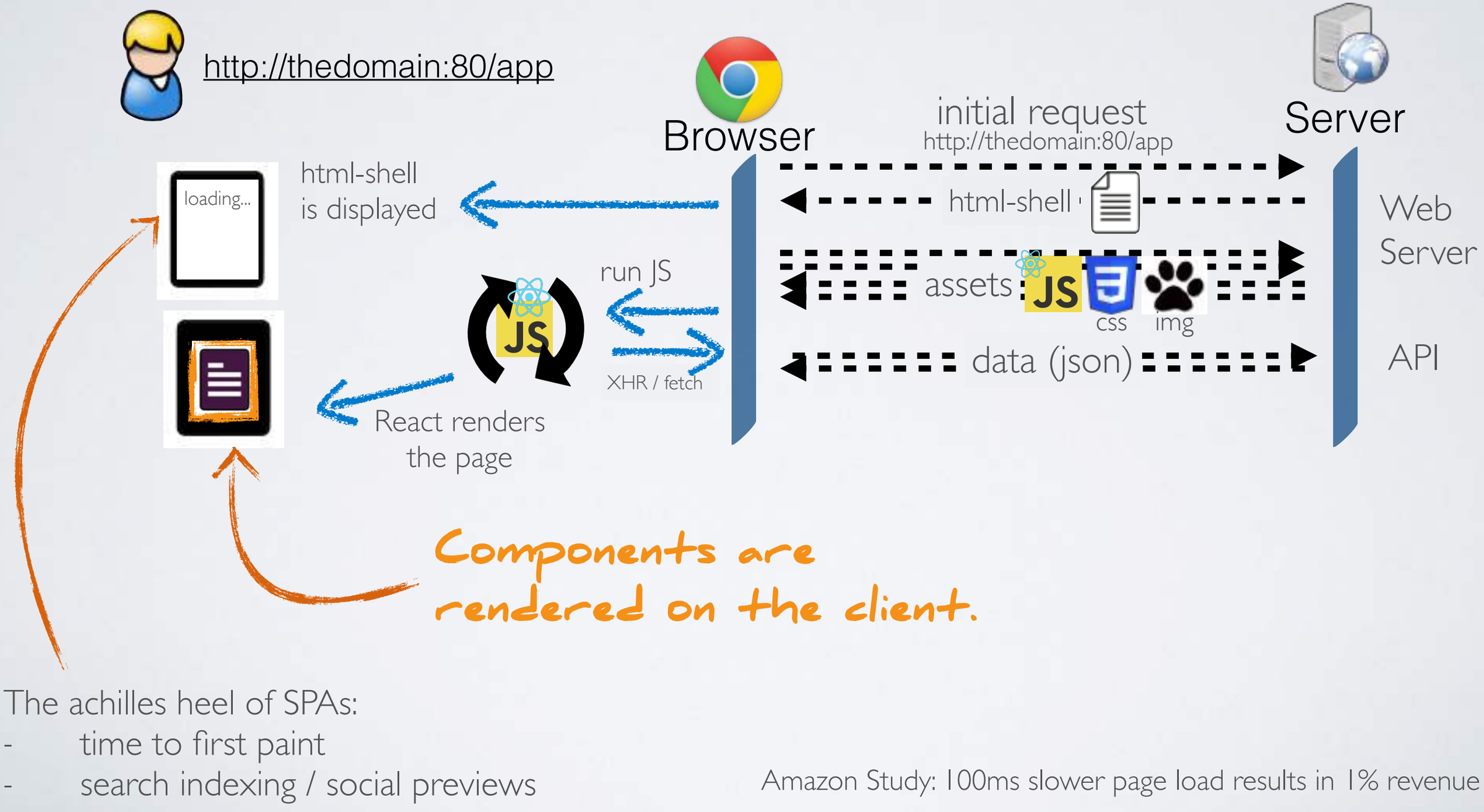
# Pure SPA Demo



Good old React as we know it ...

React is now almost12 years old! (released in May 2013).
jQuery was 7 years old, when React was released ...

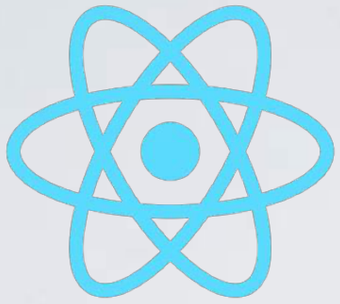
# Traditional SPA:
# Client Side Rendering




The achilles heel of SPAs:

- time to first paint
- search indexing / social previews

Amazon Study: 100ms slower page load results in 1% revenue loss!



# React is Easy!

component

```
export function App() {
   return (
      <div><Greeter/></div>
   );
}
```

component

```
export function Greeter() {
   const {id} = useIdParamFromUrl();
   return <Alert severity="info">This id a demo: {id}!</Alert>;
}
```

Passing Props

3rd party component

custom hook

```
export function useIdParamFromUrl() {
   const { id } = useParams();
   return userId;
}
```

3rd party hook

The power of React is a component model which enables simple & elegant composition ...

**Cory House**
@housecor

I love the simplicity of React's reuse model.

Repeating JSX? Create a component.
Repeating logic? Create a hook.

I can compose these simple building blocks in infinite ways.

1:08 PM · Nov 25, 2023 · **16.7K** Views

https://twitter.com/housecor/status/1728385239611789758

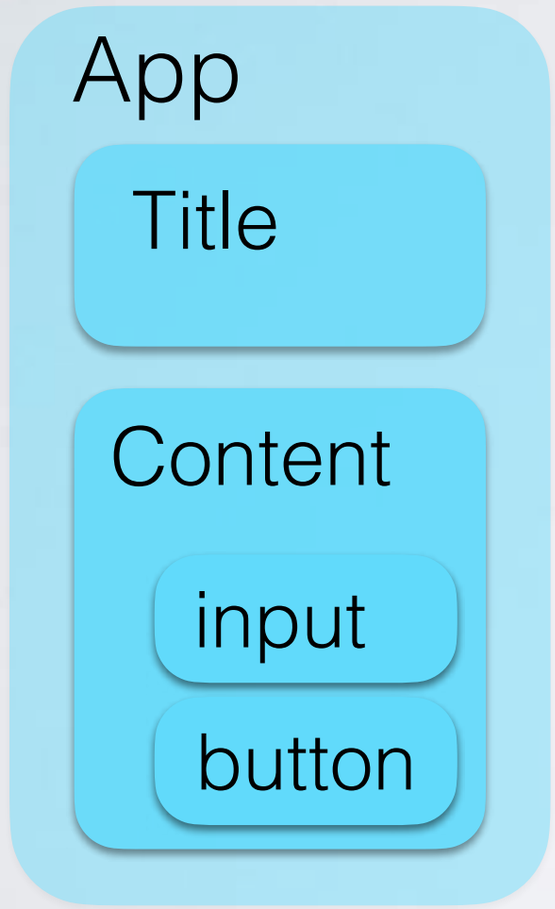
# The Virtual DOM

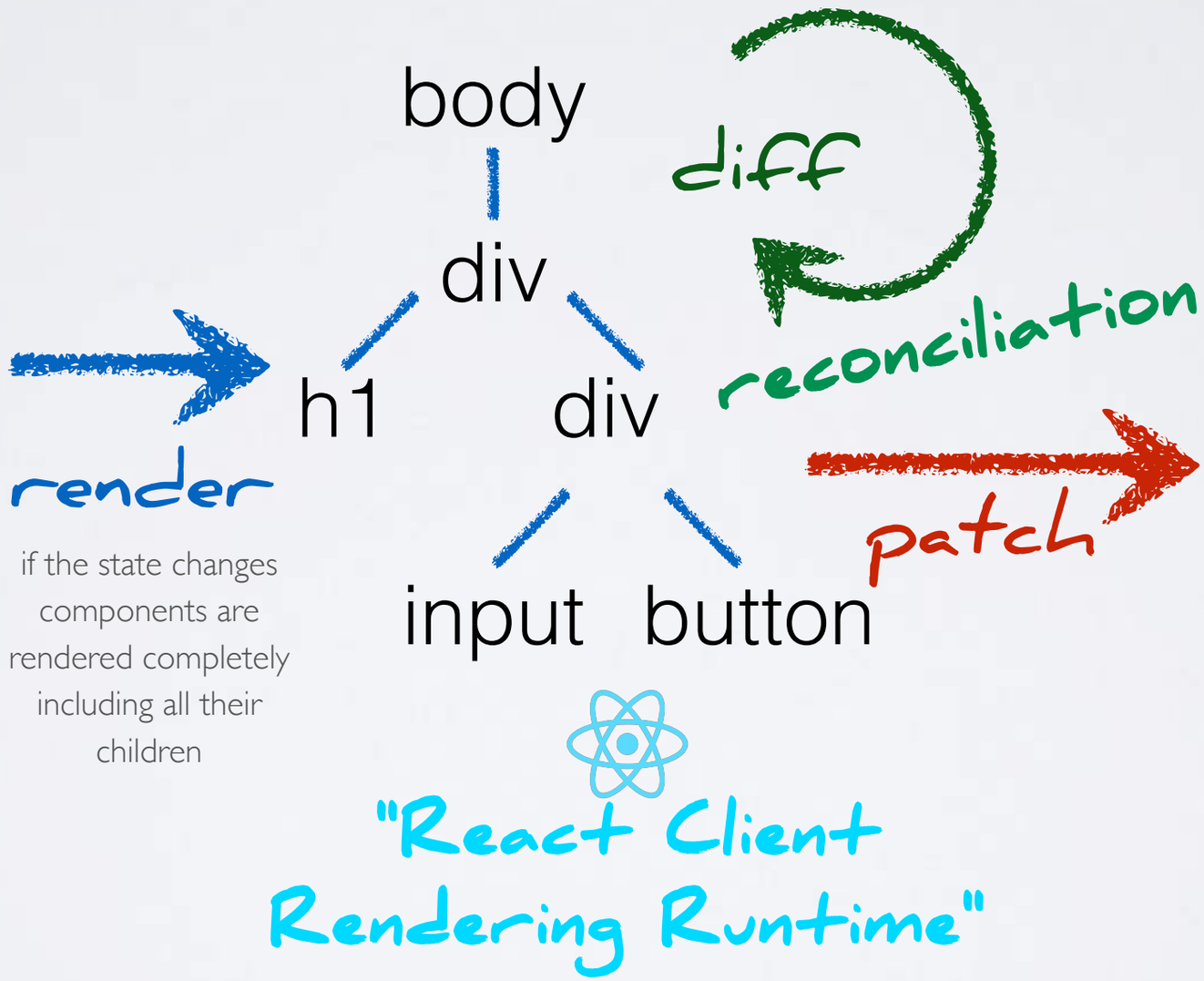
Components

Virtual DOM

In-Memory, implemented in JavaScript

Browser DOM




```
<body>
  <div>
    <h1></h1>
    <div>
      <input/>
      <button>
        </button>
    </div>
  </div>
</body>
```

The Virtual DOM also enables server-side rendering and rendering to iOS/Android UIs.

# React Data-Access

```
export function useApiData() {
  let ignore = false;
  const [data, setData] = useState("");

  useEffect(() => {
      async function fetchData() {
          const messageText = await fetchDataFromApi();
          if (!ignore) {
              setData(messageText);
          }
      }
      fetchData();

      return () => {
          ignore = true;
      };

  }, []);

  return data;
}
```

effect functions can't be async

no loading state, no error state, no caching

ignoring stale responses

cleanup function

effect dependencies

The sad face of React...

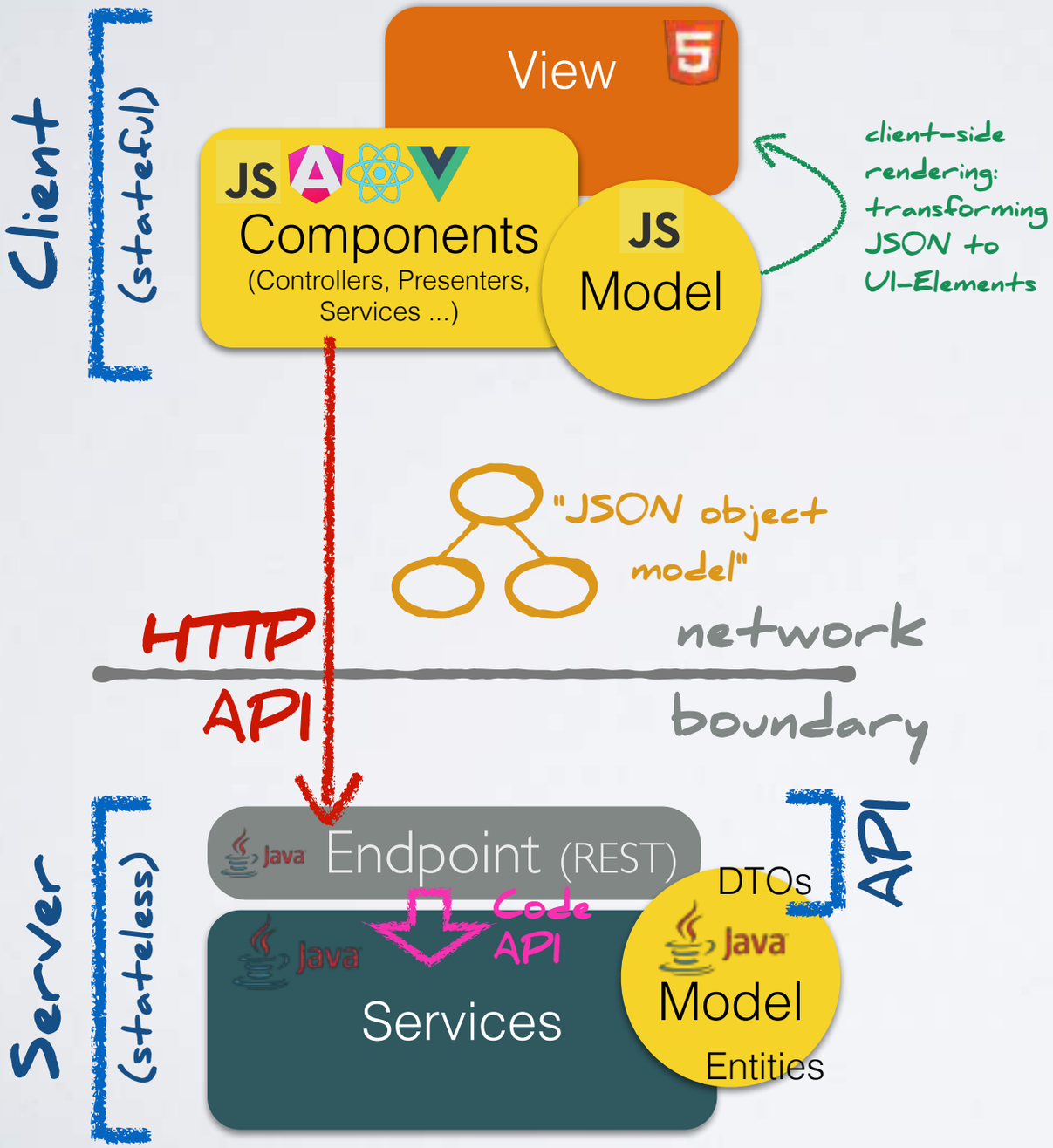
https://react.dev/learn/synchronizing-with-effects#fetching-data

The Story of React Query: https://youtu.be/OrliU0e09io?si=ZVVHRfnUCgm-gCTw&t=127

# SPA Data-Fetching


The architecture for Single Page Applications implied a HTTP API.

# The Role of the API




The rise of SPA development caused a "de-facto" architecture of formalized HTTP/REST-APIs.

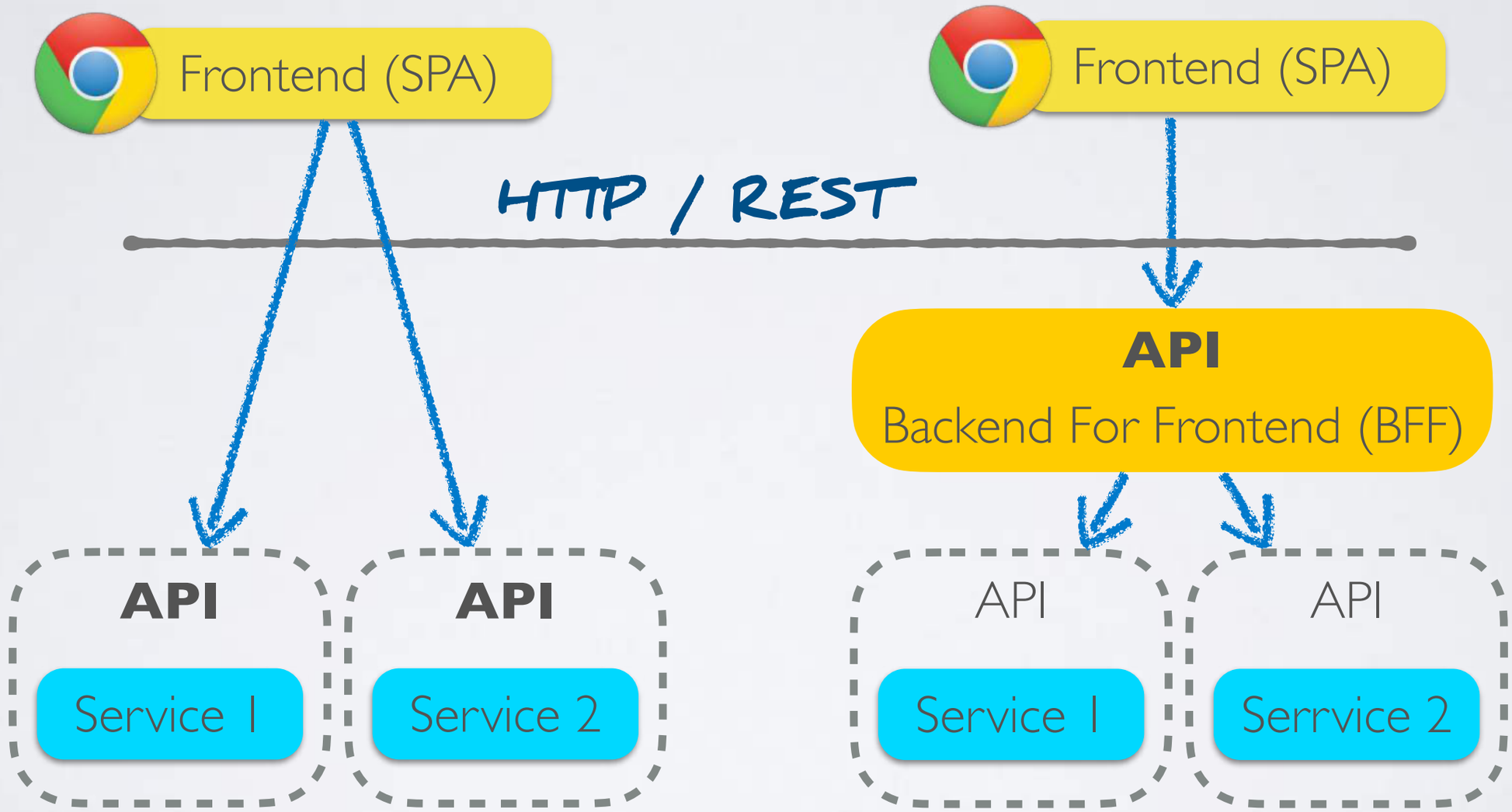
Symptoms:

- "API-First" Design
- "The central role of API-Gateways" (the return of ESBs)
- ...

Creating a formalized API is a non-trivial effort:
Proper design of URLs, Mapping, Serialization, Security ...

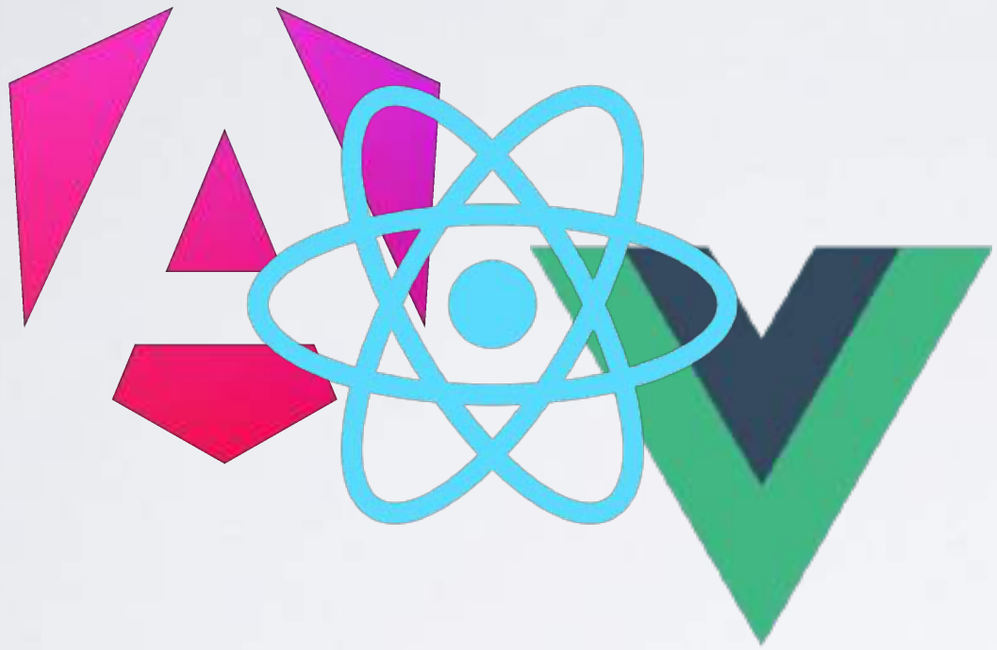
There are advantages in a formalized HTTP API: separation of concern, clearly specified and testable boundaries, reuse, team separation ...

# Architecture: APIs for SPAs




https://samnewman.io/patterns/architectural/bff/

# SSR Demo



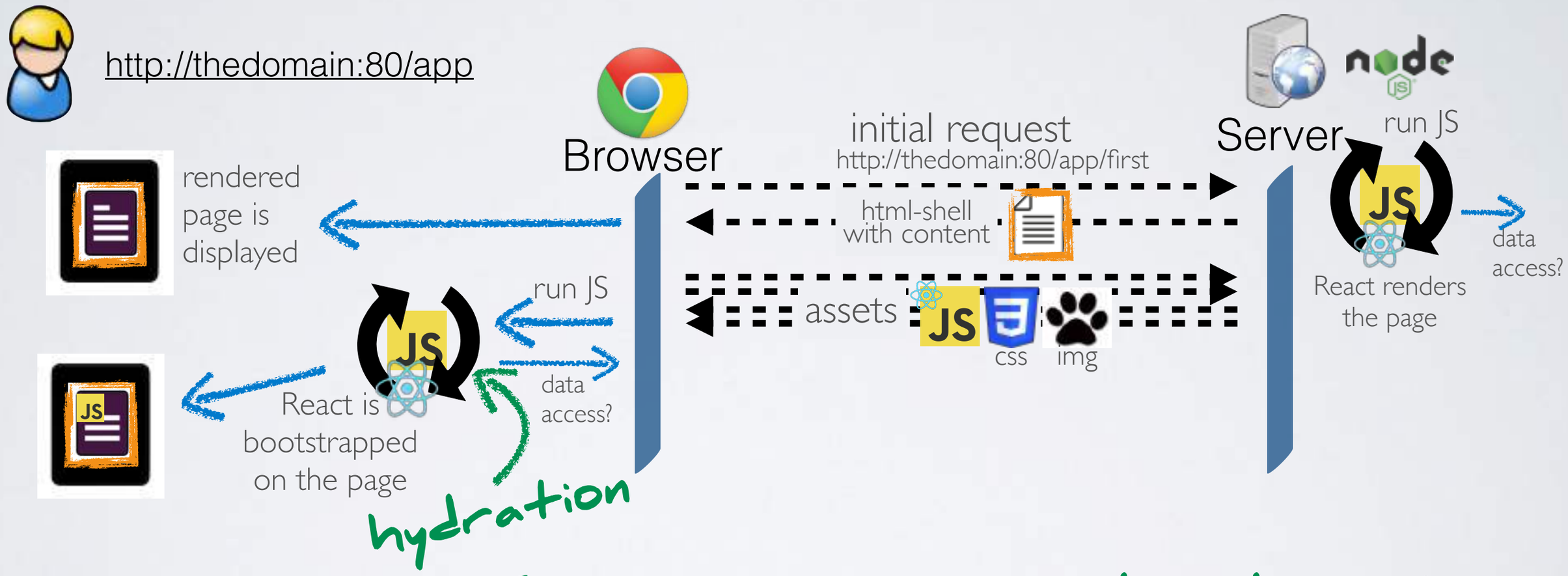
Today every modern frontend framework is capable of server side rendering.

"Production-grade SSR" goes beyond rendering components to a html-string. This requires deeper integrations into bundling as well as data-fetching and routing.

- In React and Vue this one of the reason for "Meta-Framework": Next.js, Remix/ReactRouter v7, Nuxt, Quasar ...
- Angular is capable of SSR "out-of-the-box" since v17 and they are improving SSR in every version.

# SPA with Server Side Rendering (SSR)

(*initial* rendering on the server - hydration on the client)




Components are rendered on the server and the client.

Advantages:

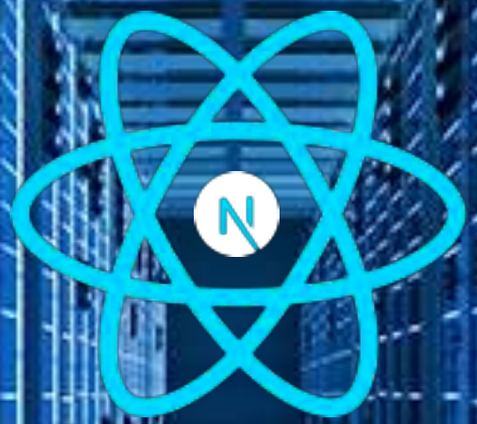
- search indexing / social previews
- improving time to first contentful paint

SSR has its own challenges:

- UX (page is not interactive on first render)
- Data Access (different mechanisms on the client and server)
- Browser APIs (not available on the server)

# Introduction to React Server Components


I want you to forget everything
you know about React!



# Introducing:
# "React Server"

# It's a React component ....

```
export function Greeter() {

  console.log("Rendering Greeter");

  return (
    <div>
      <h1>Display of Greeter.</h1>
    </div>
  );
}
```

... but exclusively rendered on the server!

# It is still a SPA!

Your Code

React Client Runtime

generate a react tree on the client

Client Component

*Virtual DOM*

In-Memory, implemented in JavaScript

*Browser DOM*

render instructions running on the client

```
export function Greeter() {
  return (
    <h1>Hello World!</h1>
  );
}
```

body
div
h1
div
input
button

diff

reconciliation

patch

```
<body>
 <div>
  <h1>...</h1>
  <div>
   <input/>
   <button>
        </button>
  </div>
 </div>
</body>
```

1. render components into the "React Server Component Payload" on the server

2. send "RSC Payload" to the client

3. execute on the client

Server Component

generate a react tree on the server and send "render instructions" to the client

# The RSC Payload


```
1:"$Sreact.fragment"
2:I[6213,[],"OutletBoundary"]
4:I[6213,[],"MetadataBoundary"]
6:I[6213,[],"ViewportBoundary"]
0:{"b":"lGRtPxl8R9zGlIQbEBlF2","f":[["children","01-simple","children","__PAGE__",
["__PAGE__",{}],["__PAGE__",["$","$1","c",{"children":[["$","div",null,
{"className":"component_wrapper__A1SYt","children":["$","div",null,
{"className":"component_server__jZH6G","children":[["$","h1",null,{"children":"Simple
Component"}],["$","div",null,{"children":"11:00:40 PM"}]]}]}],[["$","link","0",
{"rel":"stylesheet","href":"/_next/static/css/
3200fe5086d6c3a8.css","precedence":"next","crossOrigin":"$undefined","nonce":"$undefi
ned"}]],["$","$L2",null,{"children":"$L3"}]]}],{},null,false],["$","$1","h",
{"children":[null,["$","$1","r2xzraK9UhNA7fwDVQYiH",{"children":[["$","$L4",null,
{"children":"$L5"}],["$","$L6",null,{"children":"$L7"}],["$","meta",null,
{"name":"next-size-adjust","content":""}]]}]]}],false]],"S":false}
7:[["$","meta","0",{"name":"viewport","content":"width=device-width, initial-
scale=1"}]]
5:[["$","meta","0",{"charSet":"utf-8"}],["$","title","1",{"children":"Create Next
App"}],["$","meta","2",{"name":"description","content":"Generated by create next
app"}],["$","link","3",{"rel":"icon","href":"/favicon.ico","type":"image/x-
icon","sizes":"16x16"}]]
3:null
```

# Load data on the server!

```
export async function Greeter() {

  const dataFromDb = await queryDataFromDb();

  return (
    <div>
      <h1>{dataFromDb}</h1>
    </div>
  );
}
```

# Asynchronous rendering!

## Making data fetching easy!

# SPA data fetching without HTTP-API!


https://twitter.com/dan_abramov/status/1631887155080429569

# Wouldn't that be tempting?

# Out of Order Streaming

```
<h3>Server Data:</h3>
<Suspense fallback={<Spinner />}>
   <Backend messageId={1} />
</Suspense>
<Suspense fallback={<Spinner />}>
   <Backend messageId={2} />
</Suspense>
<Suspense fallback={<Spinner />}>
   <Backend messageId={3} />
</Suspense>
```

All | Fetch/XHR | Doc | CSS | JS | Font | Img | Media | Manifest | WS | Wasm | Other | cked response cookies

Blocked requests | 3rd-party requests

| Name | Method | Status | Type | Initiator | Size | Time | Waterfall |
|---|---|---|---|---|---|---|---|
| 03-streaming?v=31 | GET | 200 | document | Other | 4.1 kB | 4.07 s | |

```
<div hidden id="S:0">
    <div>
        <h1>Hello from DB!</h1>
        <p>10:14:32 PM</p>
    </div>
</div>
<script>
    $RC("B:0", "S:0")
</script>
```

```
<div hidden id="S:1">
    <div>
        <h1>Hello World!</h1>
        <p>10:14:32 PM</p>
    </div>
</div>
<script>
    $RC("B:1", "S:1")
</script>
```

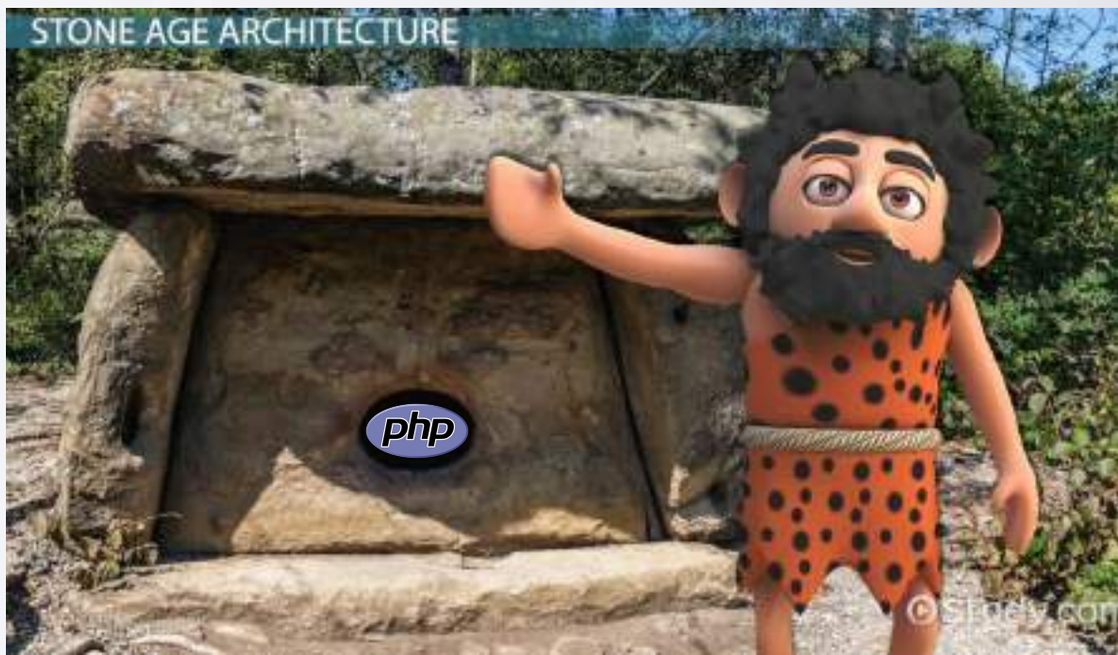
https://www.youtube.com/watch?v=23bHSDJD9y4

Wait!

...


It looks like PHP

from 25 years ago!



# Prepare for more ...

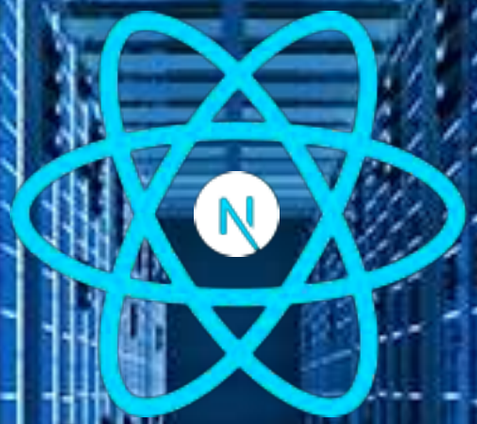

# Limitations

# React Server Components

*... are rendered on the server only*
*... are are stateless*

- Can't use hooks:
  no state: `useState`, `useReducer`, `useContext`
  no lifecylce: `useEffect`
- Can't handle DOM events:
  `onClick`, `onBlur` ...
- Can't use browser APIs:
  `localstorage`, `geolocation` ...

# Restore Memory ...



Introducing:

# "React Server"
# + (traditional) Components

# React Client Components

*... are rendered on the client and also initially on the server.*

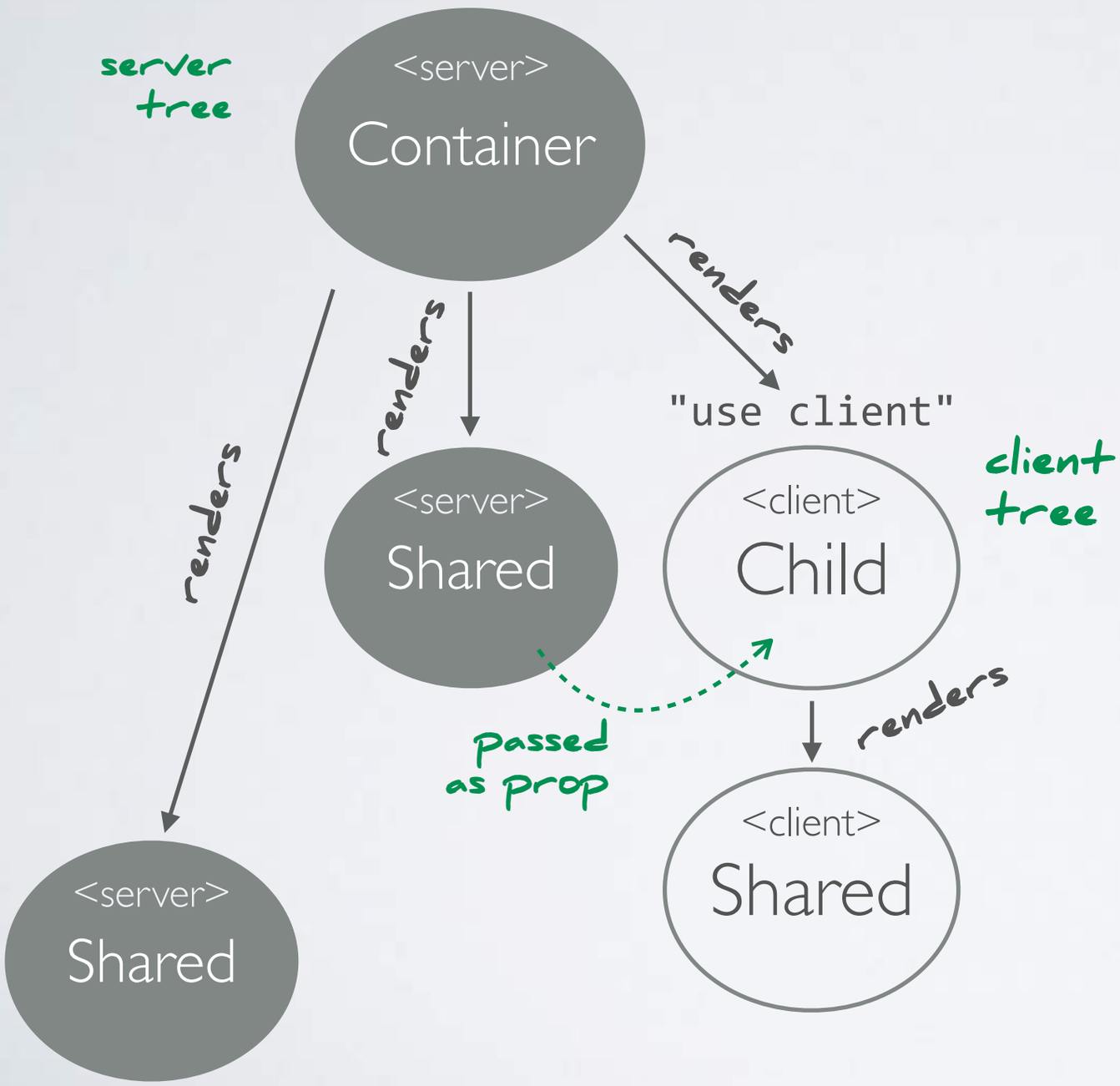
```
"use client"
export function Clock() {
  const [time, setTime] = useState(new Date())

  useEffect(() => {
    setInterval(() => setTime(new Date())}, 1000);
  }, []);

  return (
    <div>
      <h1>{time.toLocalTimeString()}</h1>
    </div>
  );
}
```

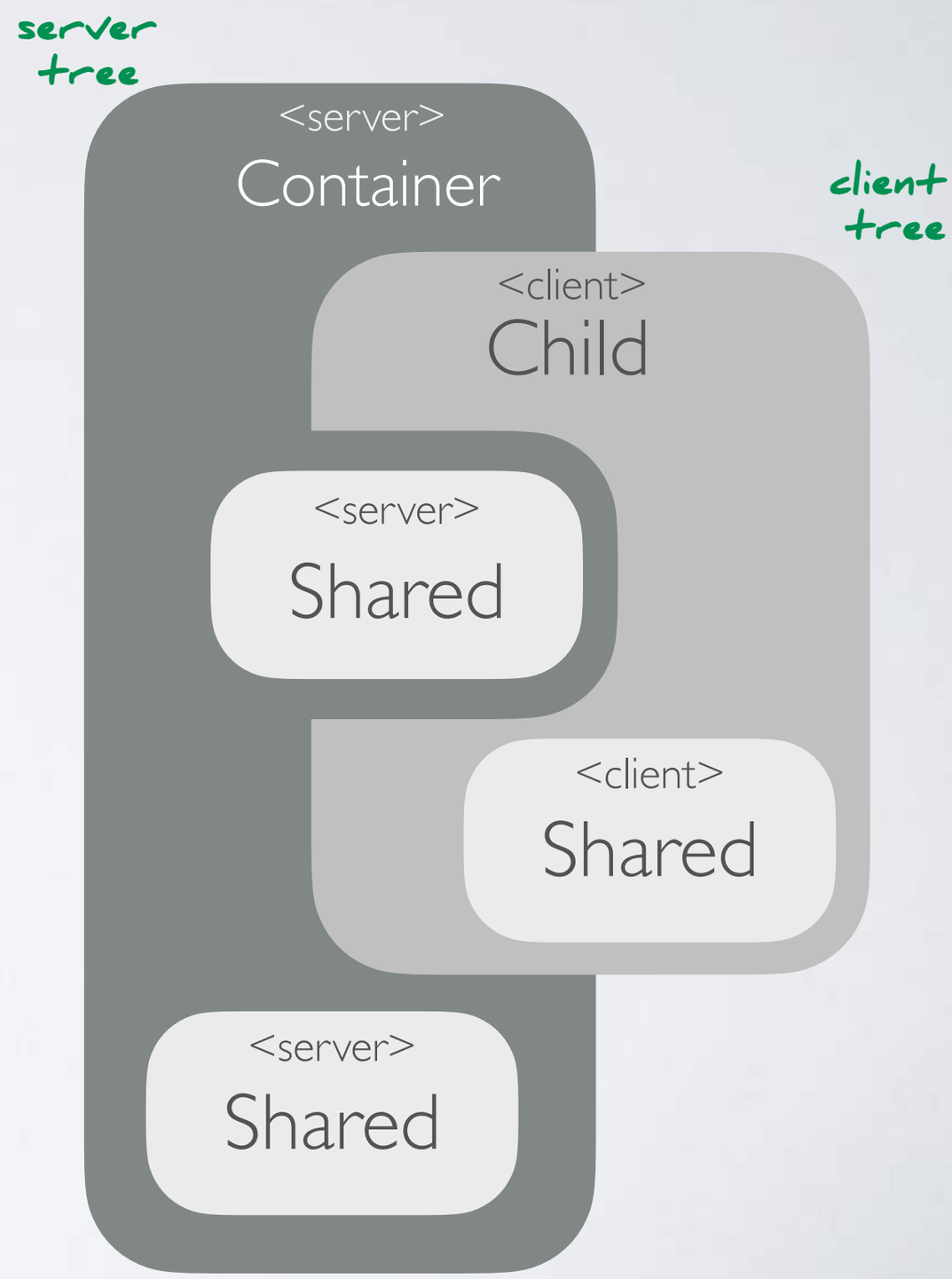
Client Components are "opt in".
Per default a component is a Server Component.

# Composition

"logical perspective"

"composition perspective"



# Full-Stack Data Flow

"composition perspective"

server tree

<server>
Container

props

<client>
Child

props

<server>
Shared

client tree

props

<client>
Shared

props

<server>
Shared

"phyisical perspective"

RSC Payload

React Client Runtime

Container

virtual dom tree
renderd by RSC payload
(including props)

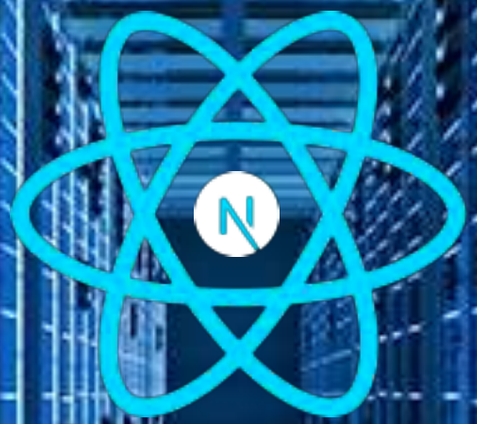
props

virtual dom tree
renderd by client
component

# A component tree that spans between client and server!


https://twitter.com/dan_abramov/status/1631887155

# In case you did not believe it the first time ...



Introducing:

# "React Server"

# + Client Components

# + Server Functions

# Server Functions

Browser

```
function ServerComponent(){
  return (
    <form action={serverActionRpc}>
      <button>Submit</button>
    </from>
  )
}
```

RSC Payload sent to the browser

*first scenario:* server component calling server function

rendering

React Client Runtime

(virtual dom tree)

RPC endpoint

Server Function:

```
"use server";
export async function serverActionRpc(arg) {
  await updateDb(arg);
  revalidatePath("/");
}
```

API call

rendering

*second scenario:* client component calling server function

```
"use client"
function ClientComponent(){
  return (
    <button onClick={serverActionRpc}>
      Update
    </button>
  )
}
```

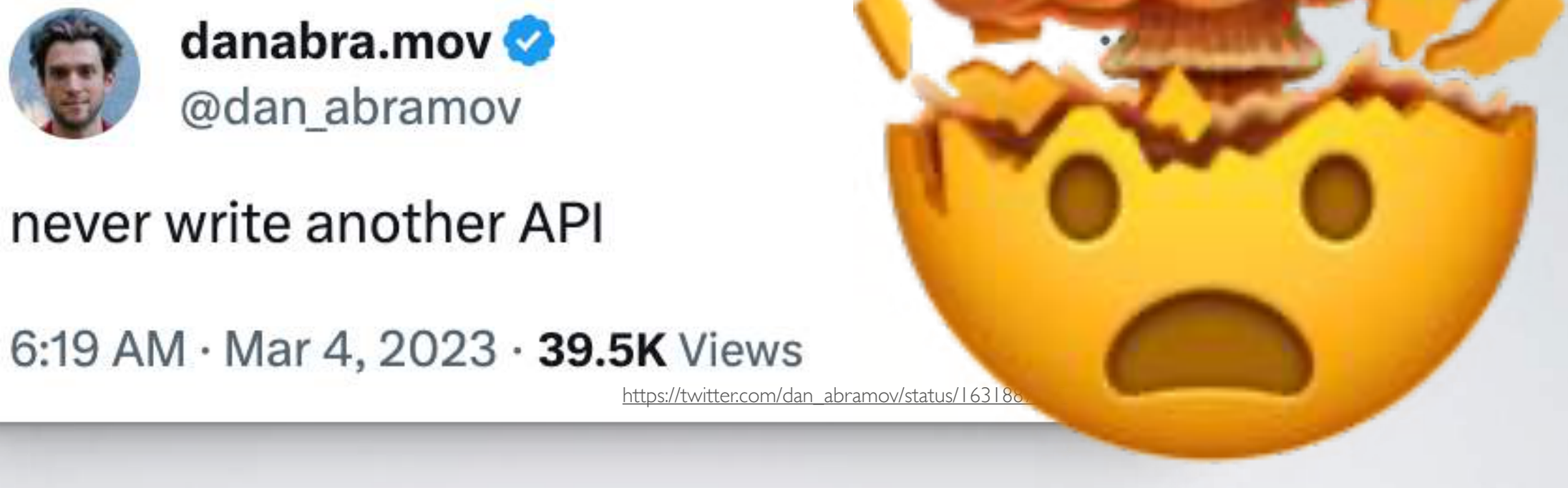
Server

JavaScript bundle loaded by the browser

JS

Network

https://react.dev/reference/rsc/server-functions



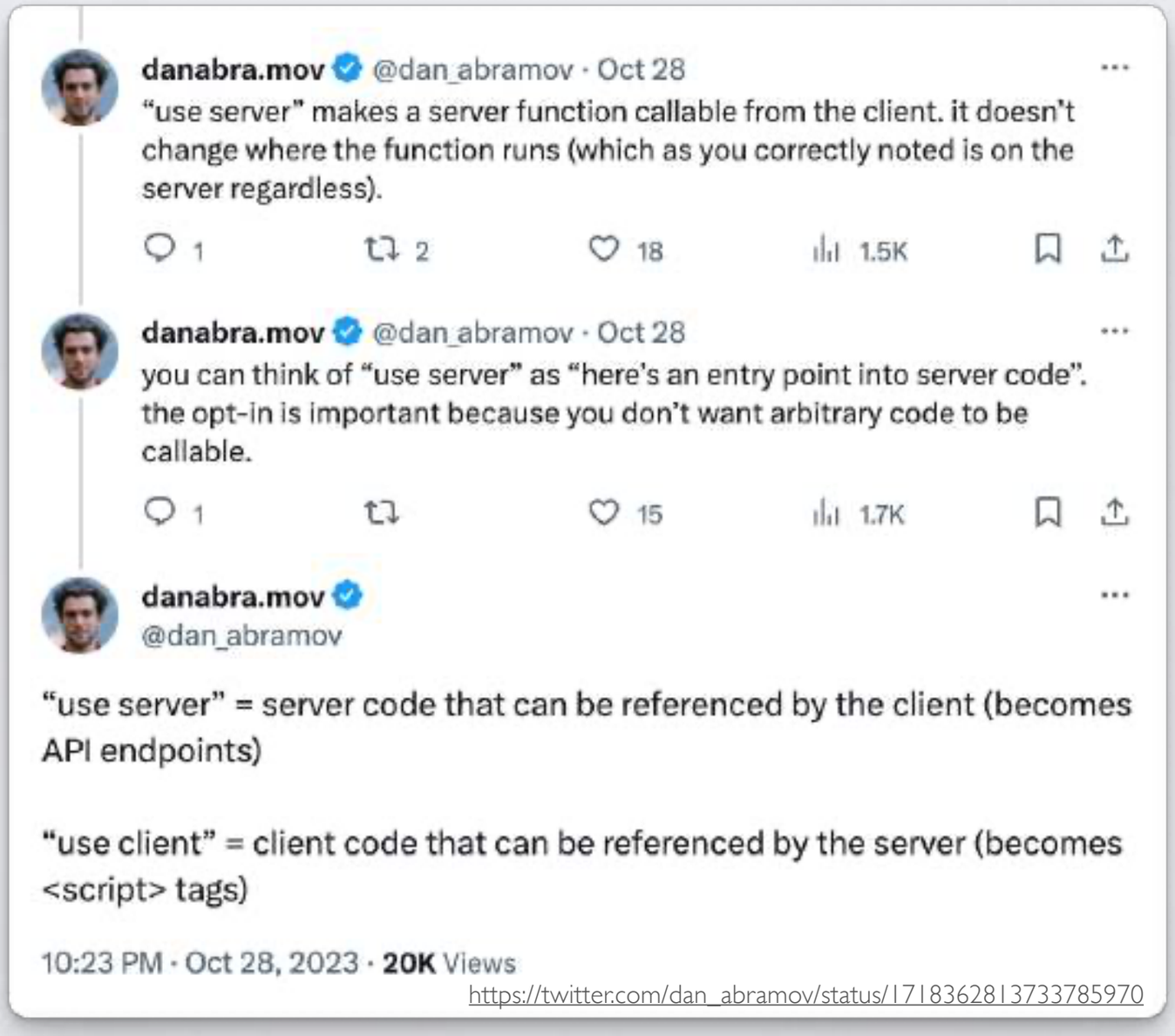

Also no HTTP-API for data mutations ....

# Server … Client … it's confusing …

```
'use client'
'use server'
```

network boudary,
js bundle shipped to client

network boudary,
RPC endpoint called by client



**danabra.mov** @dan_abramov · Oct 28

"use server" makes a server function callable from the client. it doesn't change where the function runs (which as you correctly noted is on the server regardless).

1 2 18 1.5K

**danabra.mov** @dan_abramov · Oct 28

you can think of "use server" as "here's an entry point into server code". the opt-in is important because you don't want arbitrary code to be callable.

1 15 1.7K

**danabra.mov**
@dan_abramov

"use server" = server code that can be referenced by the client (becomes API endpoints)

"use client" = client code that can be referenced by the server (becomes <script> tags)

10:23 PM · Oct 28, 2023 · **20K** Views

https://twitter.com/dan_abramov/status/1718362813733785970

# React Server Funtions

STAR WARS

RETURN OF THE RPC

## RPC-Style data fetching & mutations

# New Hooks

There are new hooks for managing async operations with Server Function in Client Components:

- `useActionState`
  https://react.dev/reference/react/useActionState
- `useFromStatus`
  https://react.dev/reference/react-dom/hooks/useFormStatus
- `useOptimistic`
  https://react.dev/reference/react/useOptimistic

# The new Era of Full-Stack Frontend Development?

NEXT.JS

React Router v7

Remix

RedwoodJS

TanStack Start

Nuxt

SOLIDSTART

SVELTEKIT

Analog

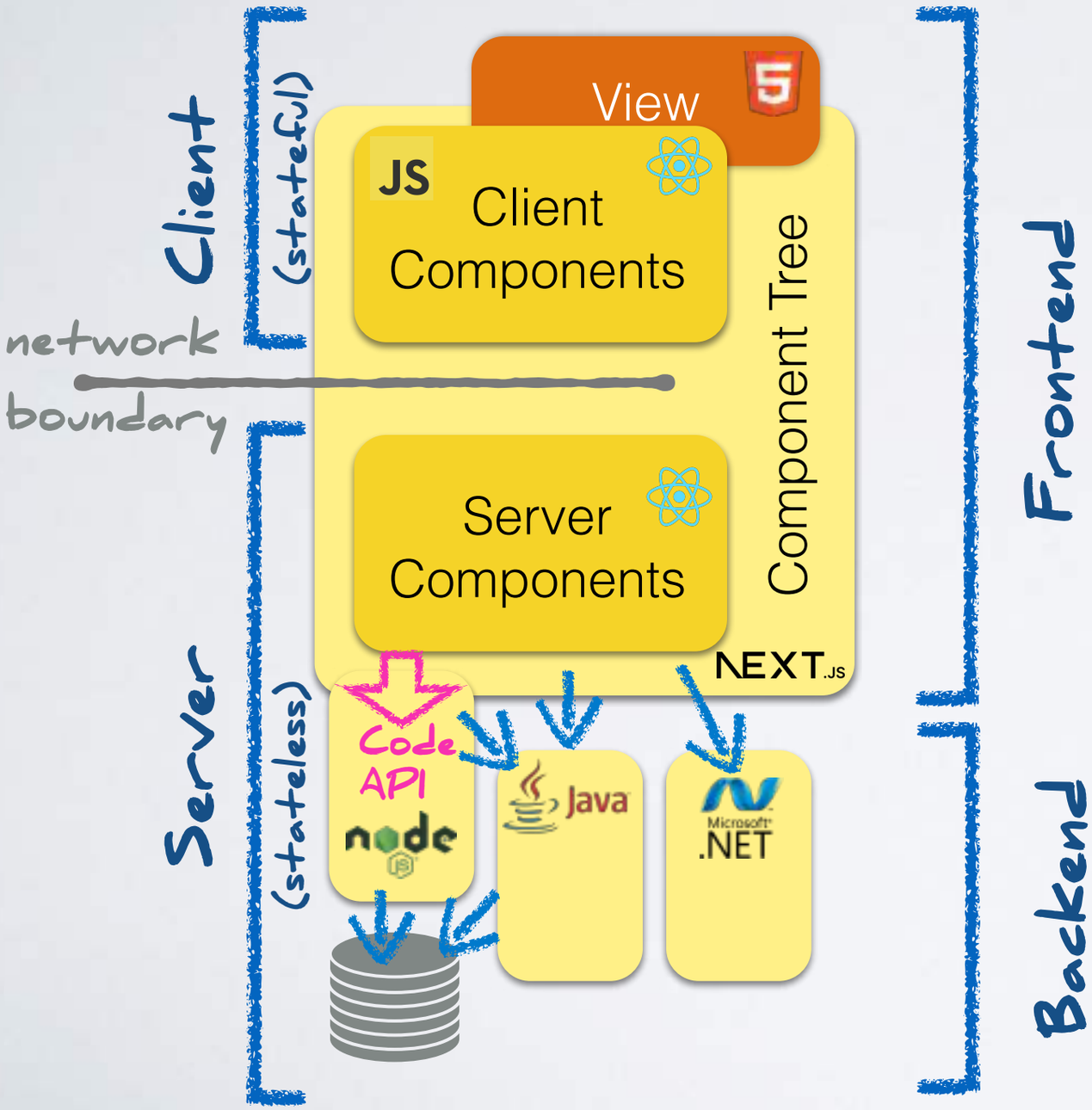
Server ← Client

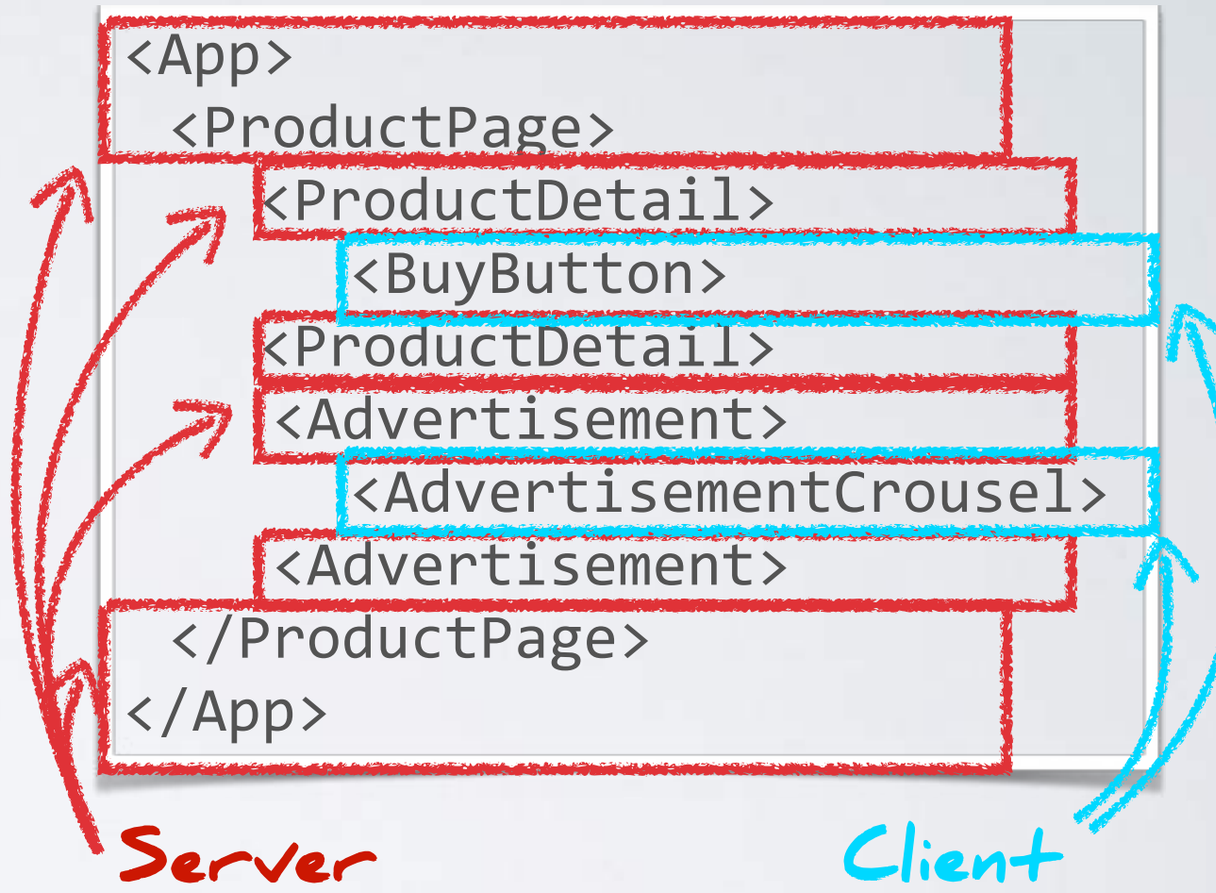
# Architectural Considerations

- Server Components allow for large dependencies to remain on the server side.
- Server Components are located much closer to the data sources — e.g., databases or file systems — they need to generate code.
- Server Components safely keep sensitive data and logic away from the browser.
- The rendering results can be cached and reused between subsequent requests and even across different sessions.

https://www.smashingmagazine.com/2024/05/forensics-react-server-components/

# The Frontend Boundary



# Summary

"React Server Components"

- is a full-stack architecture
- is based on the proven component model of React
- extends the composability patterns of React to the server-side
- solves client-server communication with a consistent programming model based on components with transparent RPC
- is the answer for data-fetching and mutations in React
- has huge potential for an ecosystem of 3rd party full-stack components

- also improves performance by enabling smaller JS bundles and streaming server responses.

# Disclaimer

NEXT.JS

The demos in this talk are based on Next.js.
Next.js is currently the only mature framework that implements React Server Components.
https://nextjs.org/

In reality it is sometimes difficult (and frustrating) to draw the boundary between features of React Server Components and Next.js.

Waku

Waku is an minimal experimental framework that implements RSCs: https://waku.gg/

RedwoodJS

RedwoodJS announced RSC integration:
https://redwoodjs.com/

React Router Remix v7

React Router v7 (formerly Remix) announced RSC integration in a future version: https://reactrouter.com/

# Thank you!

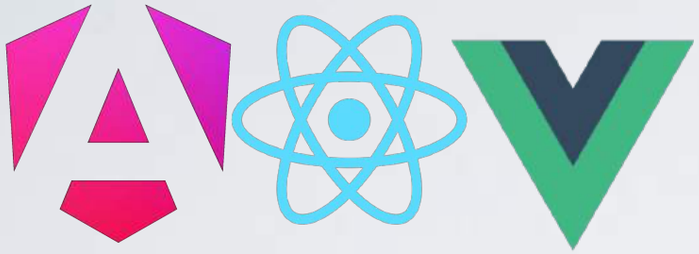
Slides & Code: https://github.com/jbandi/voxxedcern-rsc

# Questions?


Jonas Bandi
JavaScript / Angular / React / Vue / Vaadin
Schulung / Beratung / Coaching / Reviews
jonas.bandi@ivorycode.com

# Prior Art

# The Role of the HTTP API



Traditional SPAs

HTTP API is mandatory because of the strong separation (technical and organisational) of client and server.

"de facto setup":
two projects,
two teams,
API as a contract


Traditional MPA

Server Driven SPA


Modern Full Stack JS

A HTTP API is not needed for the web frontend.
"Full-Stack" project setup is possible.

HTTP APIs are optional
ie. for third-party clients.

# Island Architecture

(aka partial hydration)

avoiding client-side rendering where possible
enabling client-side interaction where needed

*Prior Art:*

# Island Architecture

The islands architecture encourages small, focused chunks of interactivity within server-rendered web pages.

astro

https://astro.build

FRESH

https://fresh.deno.dev

https://docs.astro.build/en/concepts/islands/

https://www.patterns.dev/posts/islands-architecture

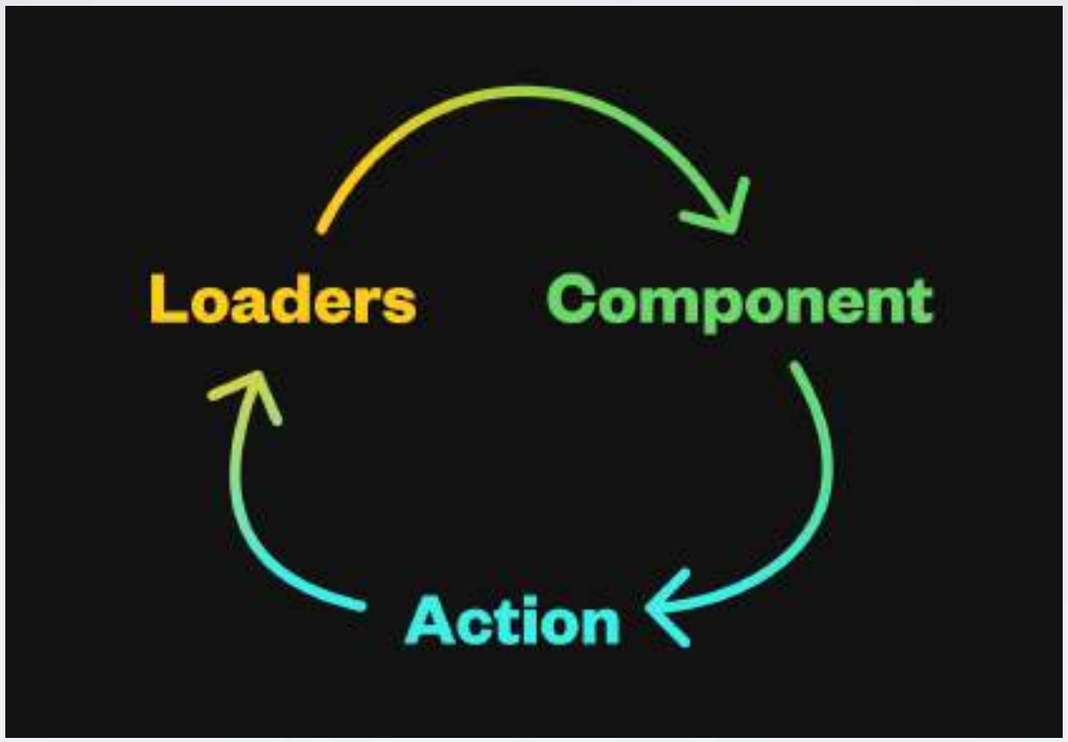
# Island Demo with Astro


https://astro.build/

https://docs.astro.build/en/concepts/islands/

# "Transparent"
# Server-Side Data Fetching & Actions

# Remix




Simplifying state state management with automatic server synchronization. The client can directly use server state without manually managing client state.

Fullstack Data Flow: https://remix.run/docs/en/main/discussion/data-flow

State Management https://remix.run/docs/en/main/discussion/state-management

Many modern frameworks provide the same concepts of server-side `loader` and `action` methods:

Remix
https://remix.run/docs/en/main/route/loader
https://remix.run/docs/en/main/route/action

SOLIDSTART
https://start.solidjs.com/core-concepts/data-loading
https://start.solidjs.com/core-concepts/actions

SVELTEKIT
https://kit.svelte.dev/docs/load
https://kit.svelte.dev/docs/form-actions

qwik Qwik City
https://qwik.builder.io/docs/route-loader/
https://qwik.builder.io/docs/action/
https://qwik.builder.io/docs/server$/

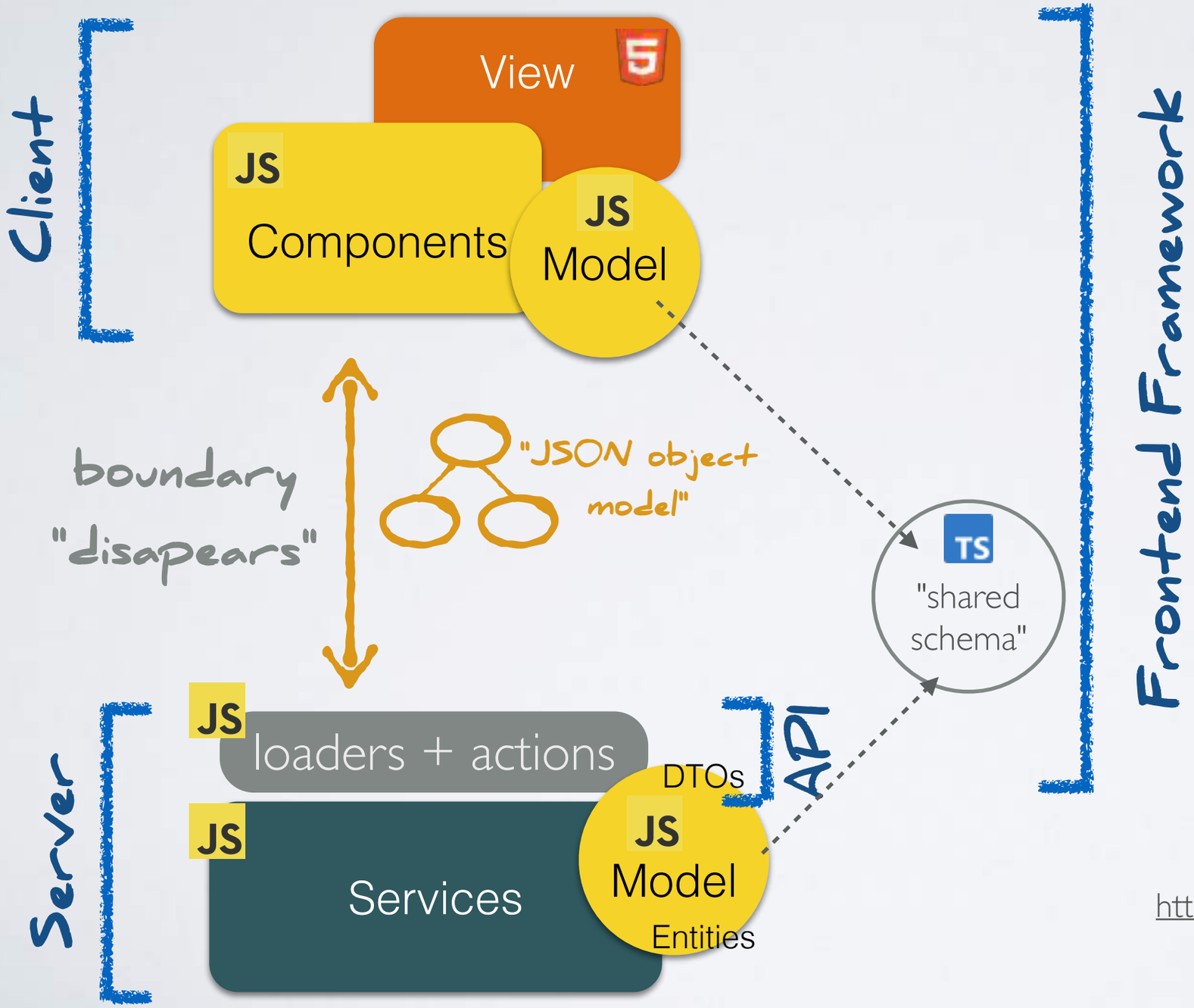
Similar RCP concepts:

Hilla
https://hilla.dev/

https://trpc.io/

TanStack Start
https://tanstack.com/start/

# Full-Stack Frontend Frameworks




Server-Side data loading and actions

Remix

https://remix.run/

SOLIDSTART

SVELTEKIT

qwik

https://remix.run/docs/en/main/route/loader
https://remix.run/docs/en/main/route/action
https://start.solidjs.com/core-concepts/data-loading
https://start.solidjs.com/core-concepts/actions
https://kit.svelte.dev/docs/load
https://kit.svelte.dev/docs/form-actions
https://qwik.builder.io/docs/route-loader/

# Server Driven SPA

# Server Driven SPAs

aka "Live View"

Vaadin

Blazor Server

Phoenix Framework

Enabling SPAs with a server-side programmig model and no need for a REST API.

Demos:

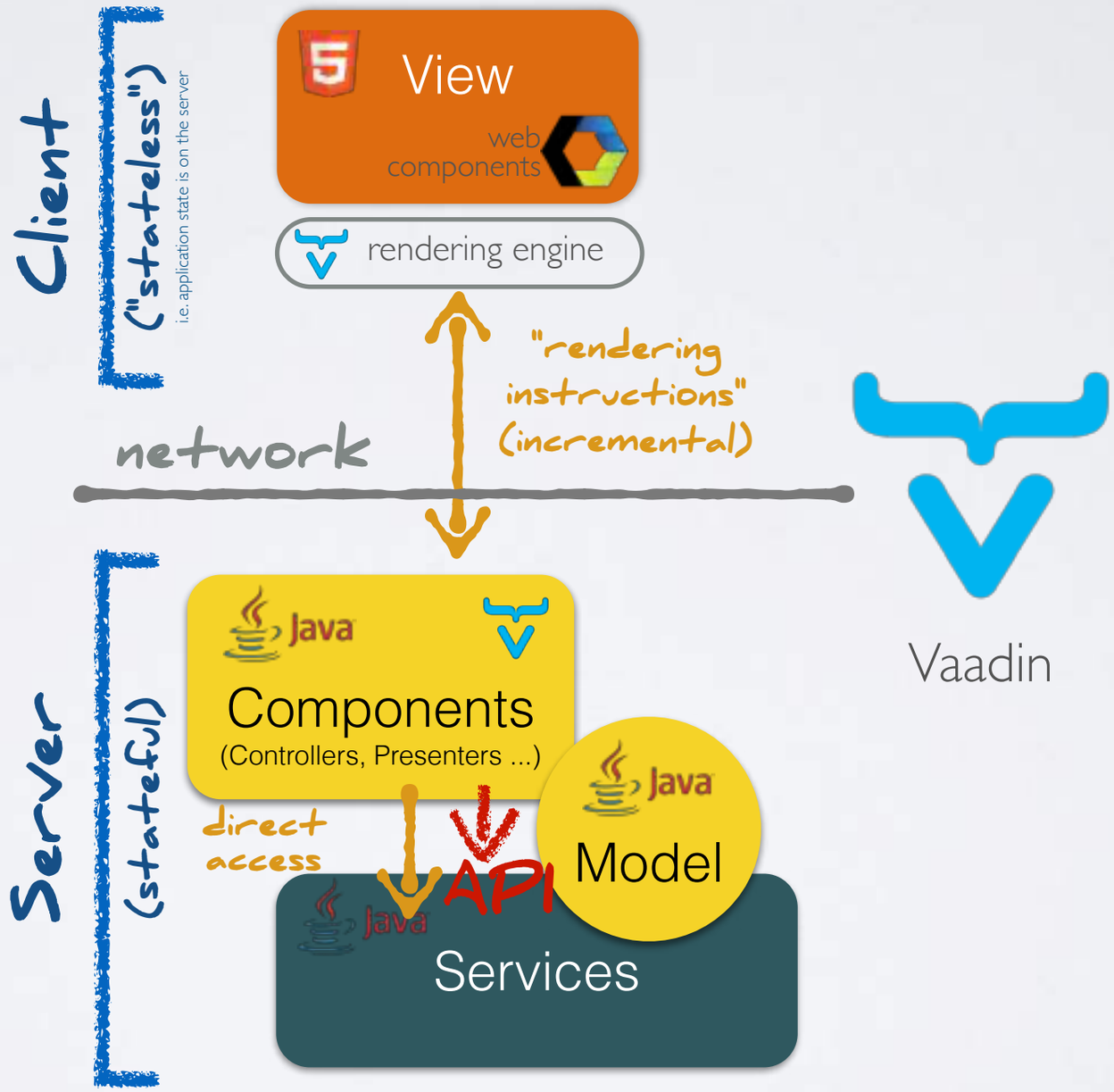
https://labs.vaadin.com/business/

https://blazor.syncfusion.com/demos/datagrid/overview?theme=bootstrap4

Term definition: https://github.com/dbohdan/liveviews

# Vaadin Architecture

using the browser just as a "rendering engine"




Same concepts are used in .NET with Blazor Server

https://vaadin.com/

# Two Perspectives on "Full-Stack"

The frontend ecosystem approaches "Full-Stack" by making server-access transparent.

The backend ecosystems (Java, .NET) approach "Full-Stack" by treating the browser as (remote) render-engine.

In both cases the "network disappears" ...

... data is automatically available in the frontend ...

... the UI automatically reflects updated on the server ...

# Thank you!

Slides & Code: https://github.com/jbandi/voxxedcern-rsc

# Questions?


Jonas Bandi
JavaScript / Angular / React / Vue / Vaadin
Schulung / Beratung / Coaching / Reviews
jonas.bandi@ivorycode.com