

BACK
TO THE
Server



Modern Frontend – Back To The Server



#BaselOne24

baselone.ch

ABOUT ME

Jonas Bandi

jonas.bandi@ivorycode.com

Twitter: @jbandi



- Freelancer, in den letzten 10 Jahren vor allem in Projekten im Spannungsfeld zwischen modernen Webentwicklung und traditionellen Geschäftsanwendungen.
- Dozent an der Berner Fachhochschule seit 2007
- In-House Kurse & Beratungen zu Web-Technologien im Enterprise: UBS, Postfinance, Mobiliar, AXA, BIT, SBB, Elca, Adnovum, BSI ...



JavaScript / Angular / React / Vue / Vaadin
Schulung / Beratung / Coaching / Reviews

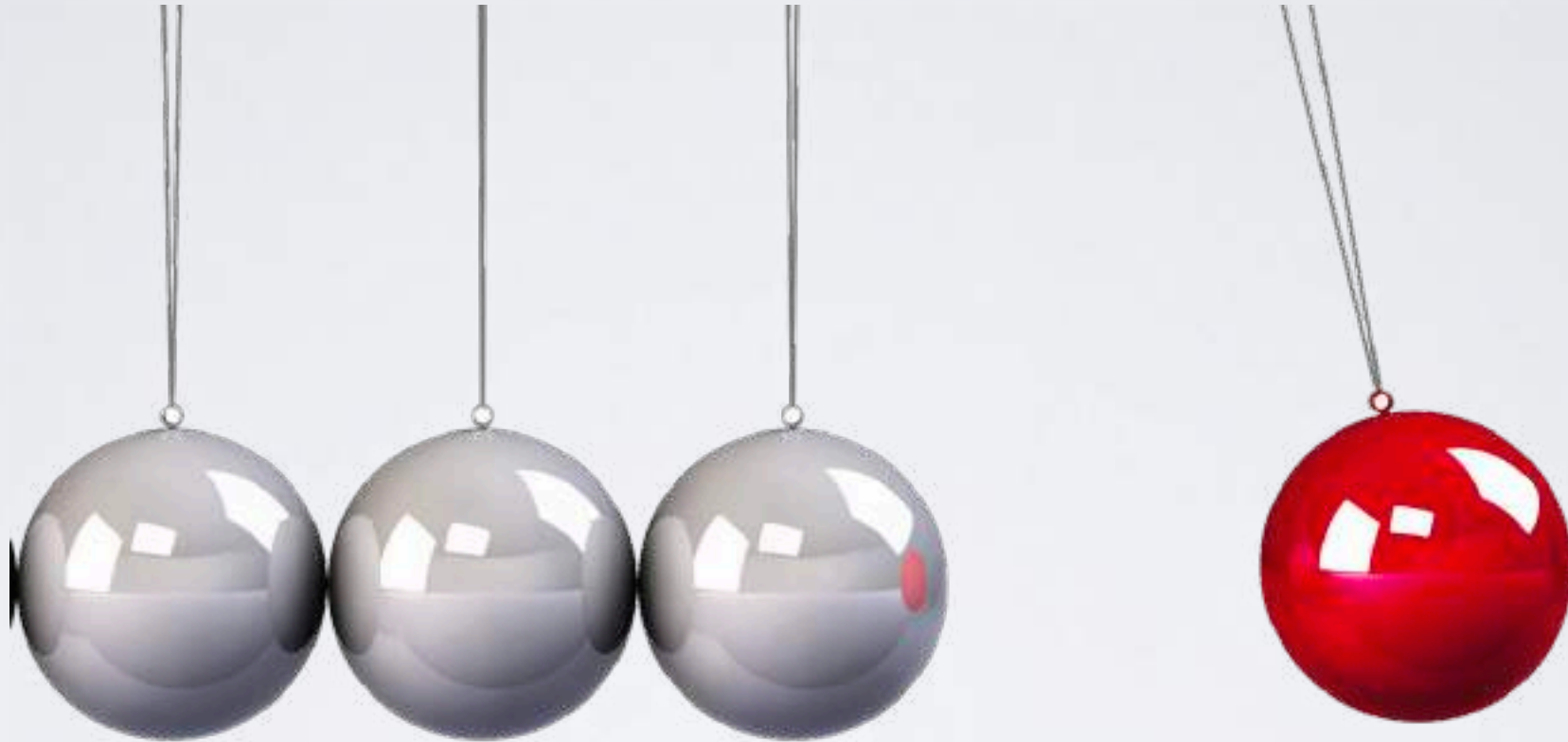
jonas.bandi@ivorycode.com



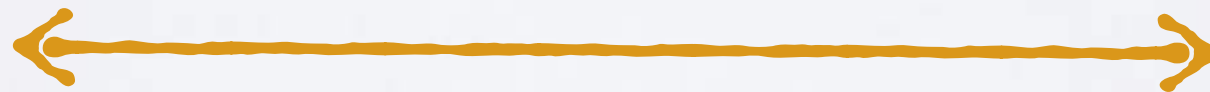


What are you using ... ?

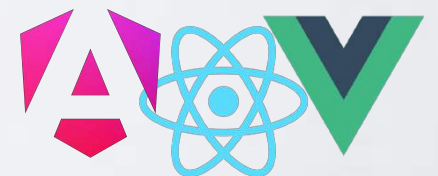
The Pendulum is swinging ...



Server

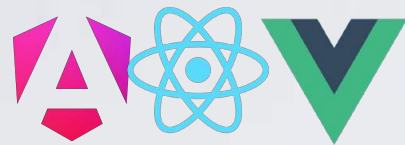


Client



... will it ever stop?

AGENDA



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

NEXT.js **Remix**



SSR - Server Side Rendering

Meta-Frameworks
Hydration



Island Architecture

HTTP Streaming
Partial Prerendering & Server Islands

Remix
NEXT.js

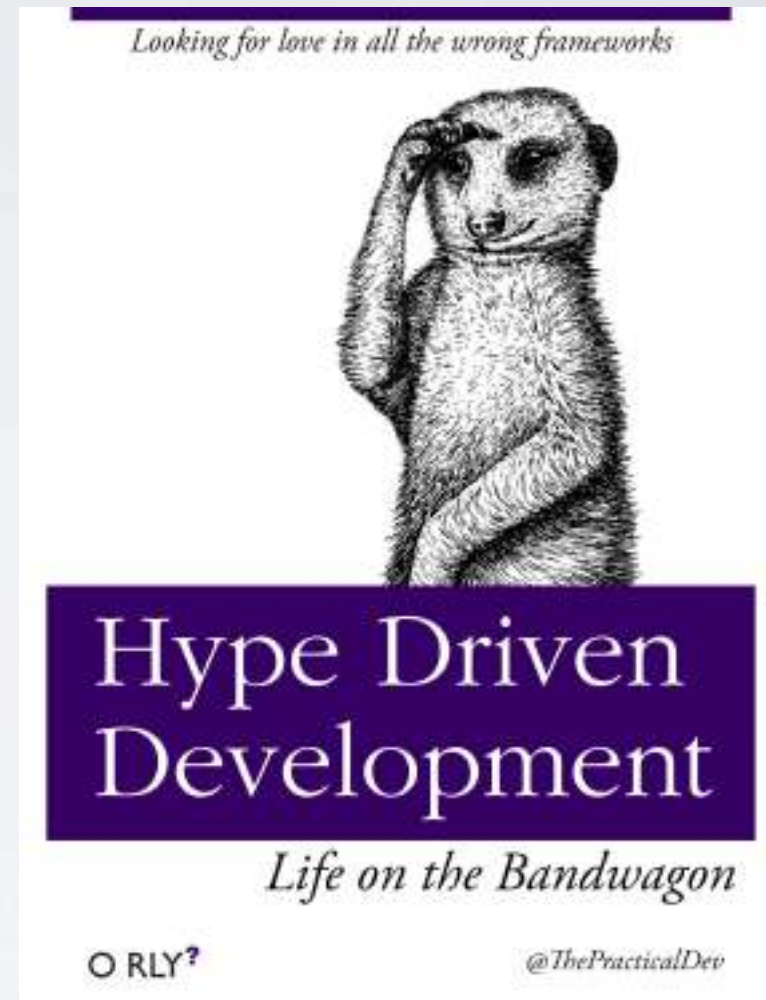
Server Side Data Loading & Mutations

RPC-style fetching and actions
React Server Components



Server-Driven SPA (aka. "Live View")

My goal is to make you feel like this:



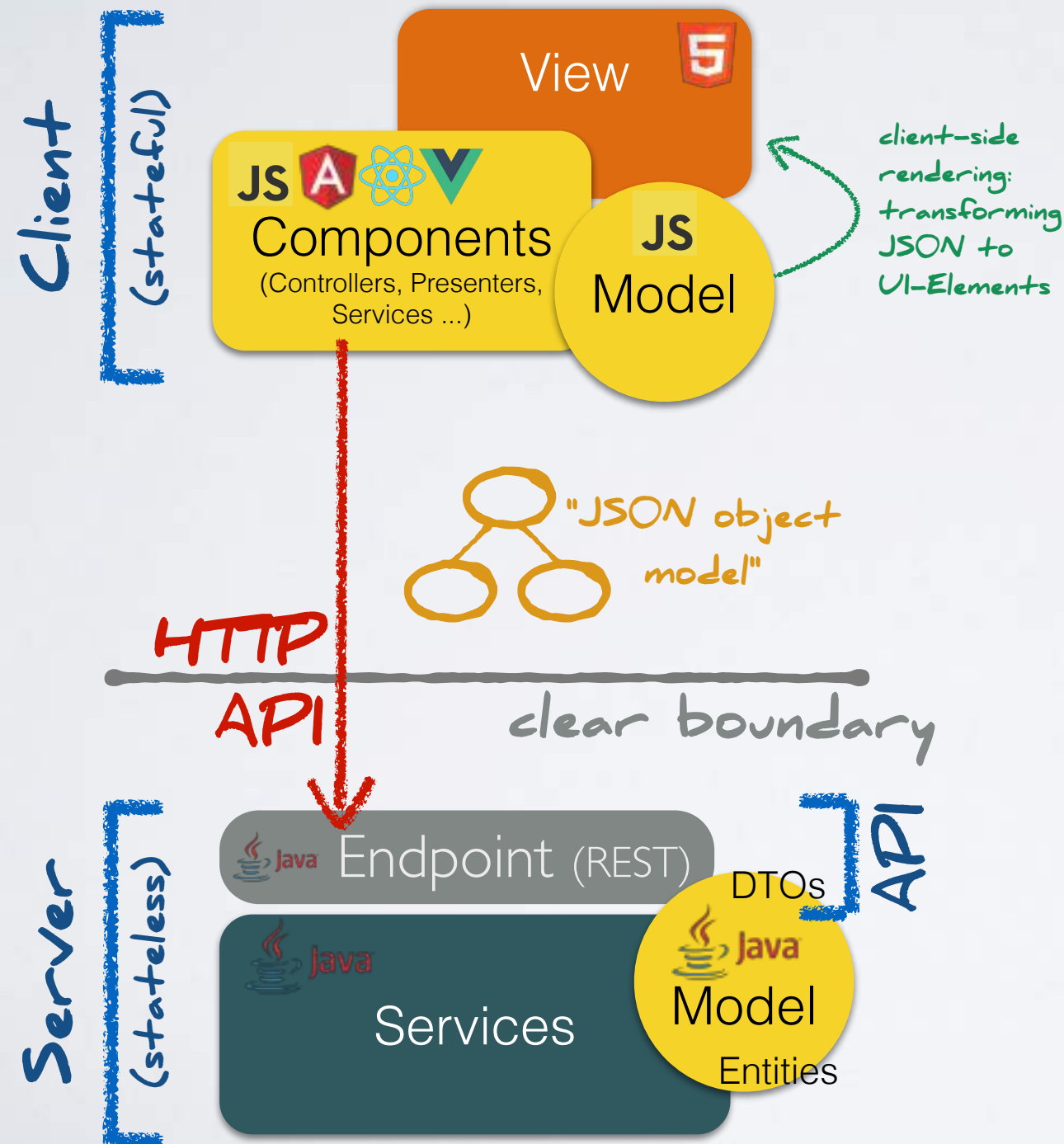
... not about the implementation details
but hopefully by looking beyond the
current "state of the art".



The Era of Single Page Applications

The Role of the API:

traditional client-server boundary



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: 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 for Single Page Applications:

STAR WARS

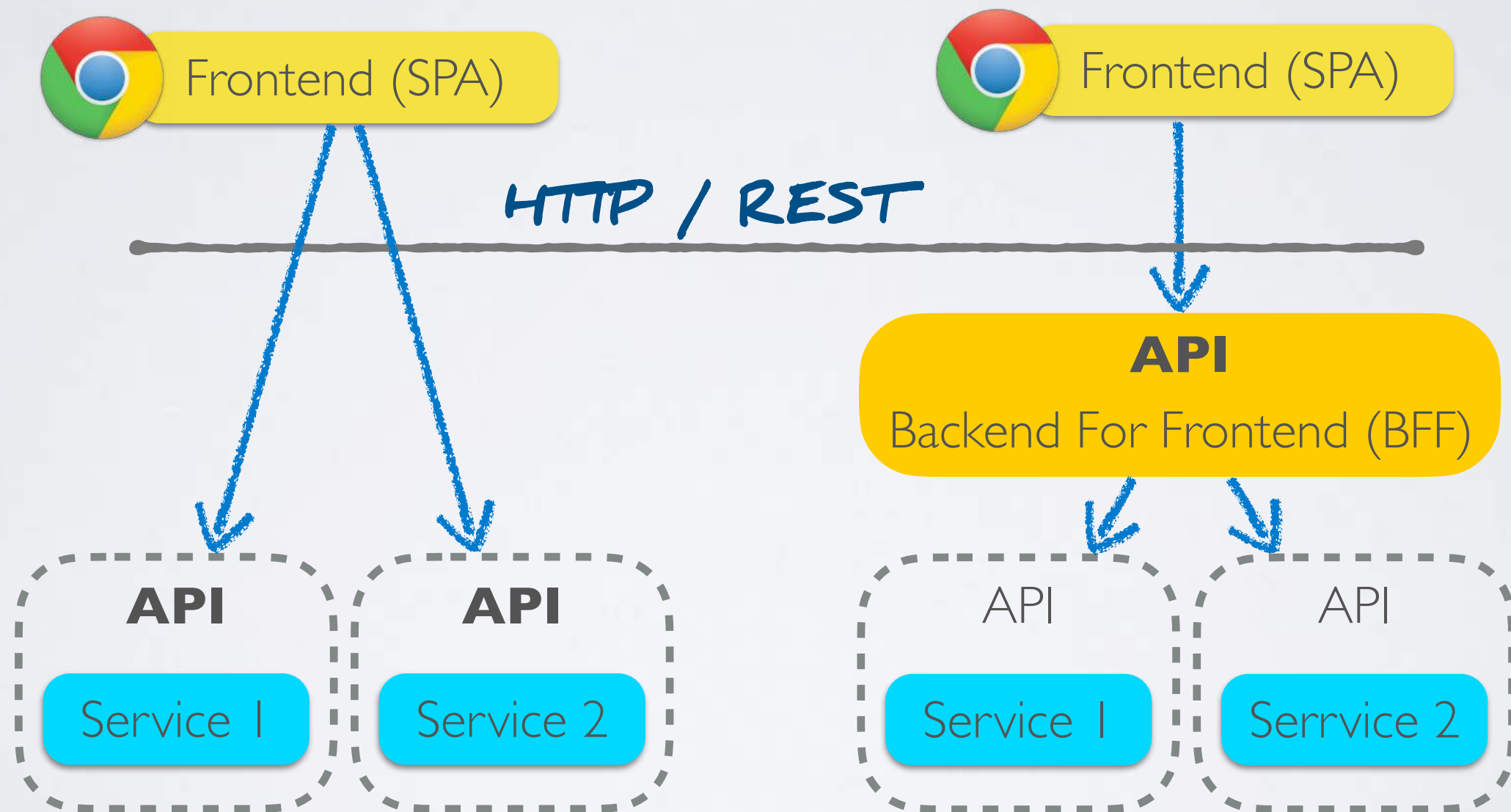
THE RISE OF

THE API

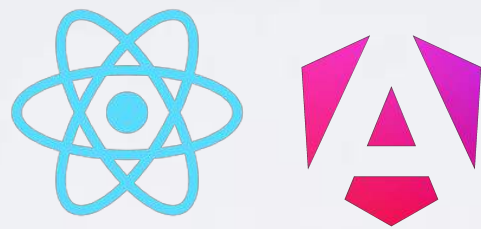
EPISODE IX

Traditional Architectures for SPAs

Client - API - Server

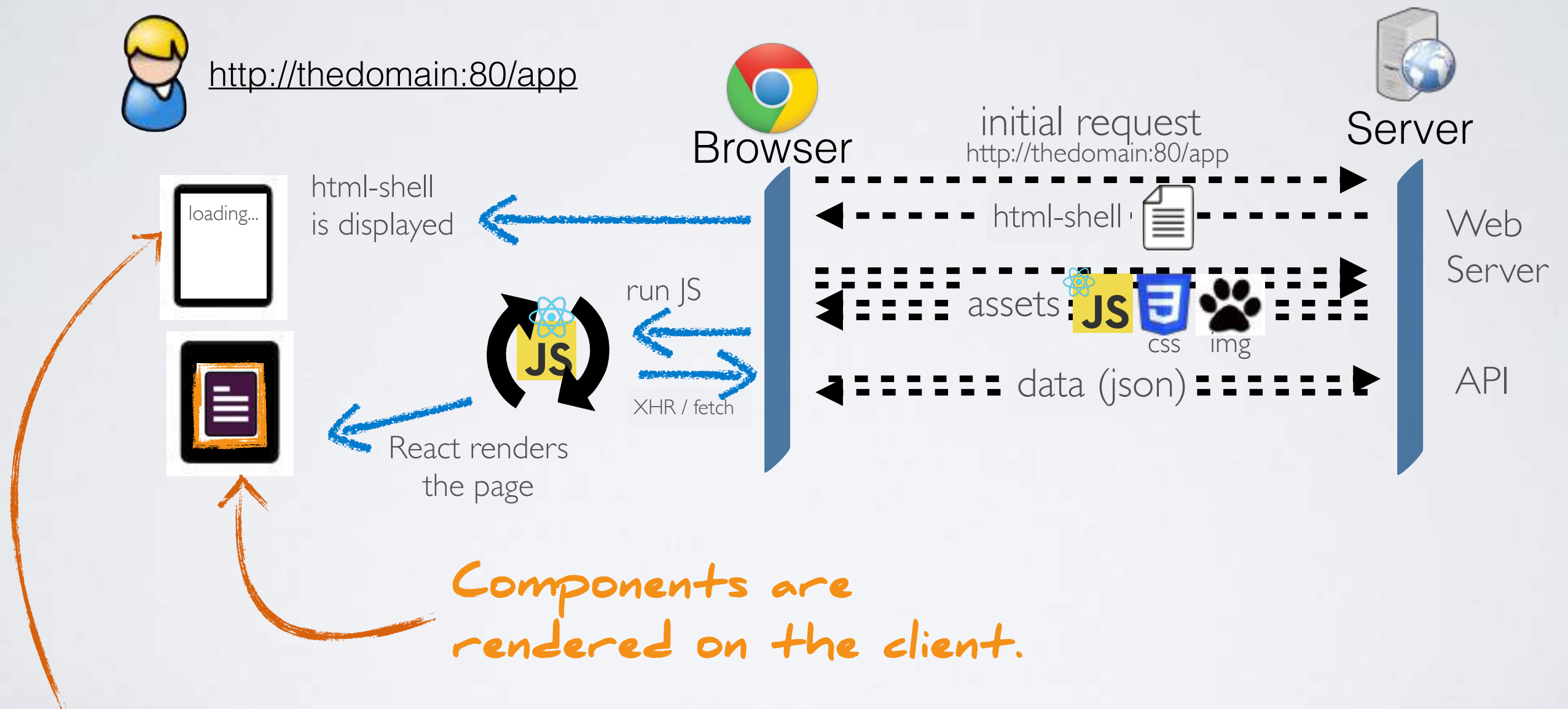


Pure SPA Demo



- html document as "shell"
- navigation without network
- fetching data from API

Traditional SPA: Client Side Rendering

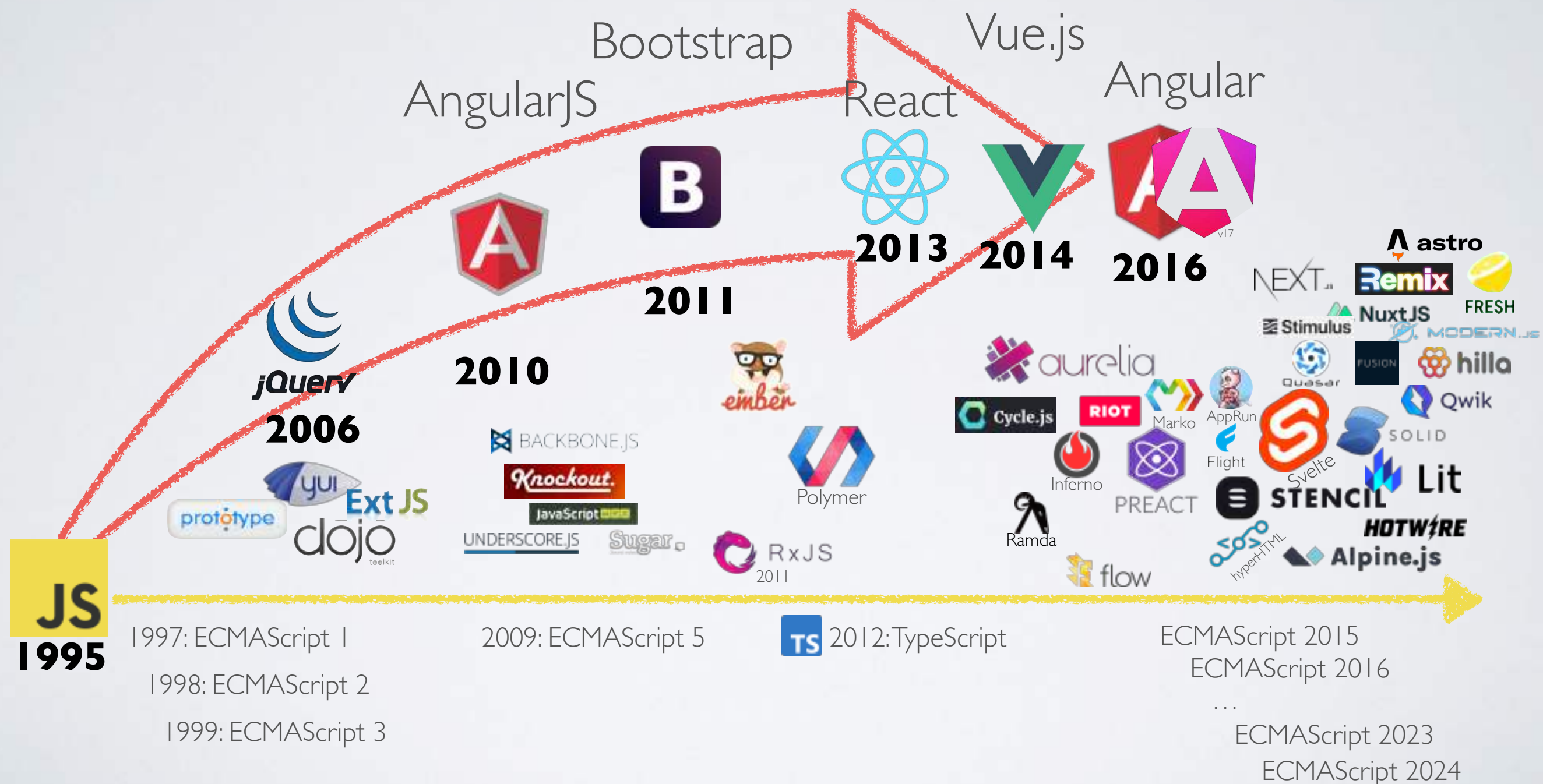


The achilles heel of SPAs:

- time to first paint
- search indexing / social previews

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

The Frontend JavaScript Ecosystem



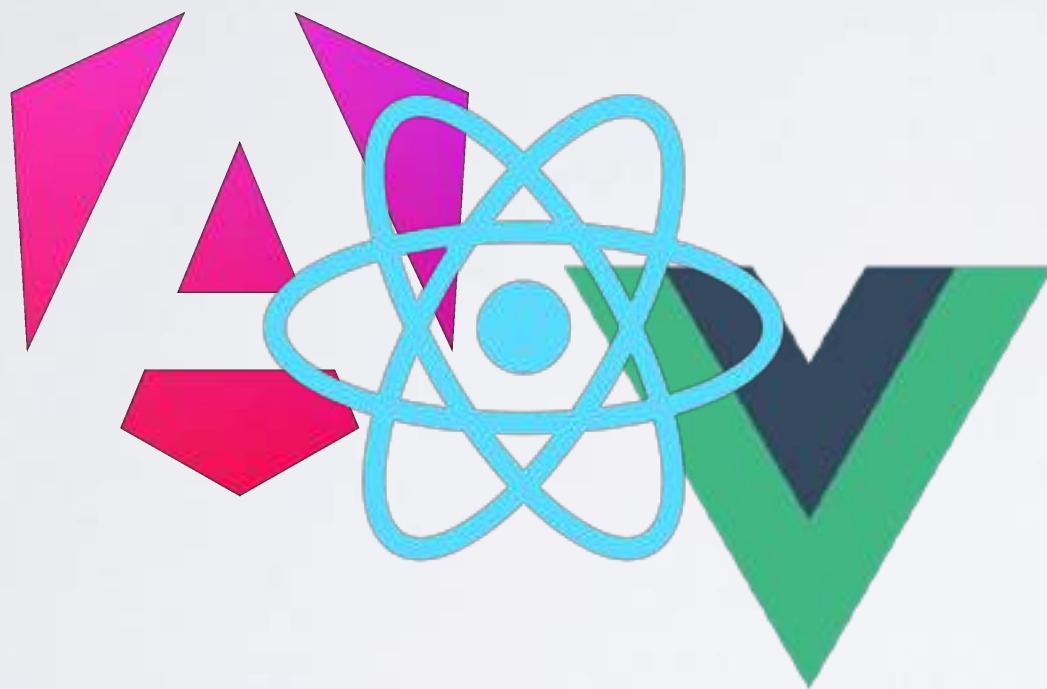
Innovation in pure frontend frameworks is
stagnating.
Frontend frameworks have become very similar.

dotJS 2024 - Minko Gechev - Converging Web Frameworks
<https://www.youtube.com/watch?v=grRH8e46Pso>

The innovation is moving towards the server-side
and the full-stack perspective of web
frameworks.

"Mind The Gap" by Ryan Florence at Big Sky Dev Con 2024
<https://www.youtube.com/watch?v=zqhE-CepH2g>

Server Side Rendering (SSR)



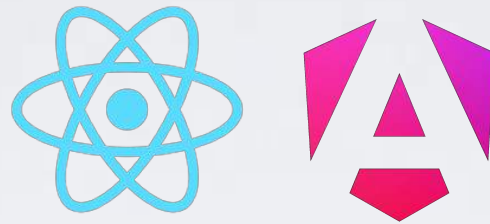
Today every modern frontend framework is capable of server side rendering.

But there are differences when it comes to data fetching ...

Angular supports non-destructive hydration since v16..
The Angular CLI introduced SSR scaffolding in v17.

<https://angular.dev/guide/hydration>
<https://blog.angular.io/angular-v16-is-here-4d7a28ec680>
<https://angular.dev/cli/new>

SSR Demo



- html document has content
 - components are rendered on the server and the client
 - interactivity only after hydration
 - navigation without network
-
- Angular: HttpClient works with SSR!
(note: no client-side fetch on initial load!)

Server Side Rendering (SSR)

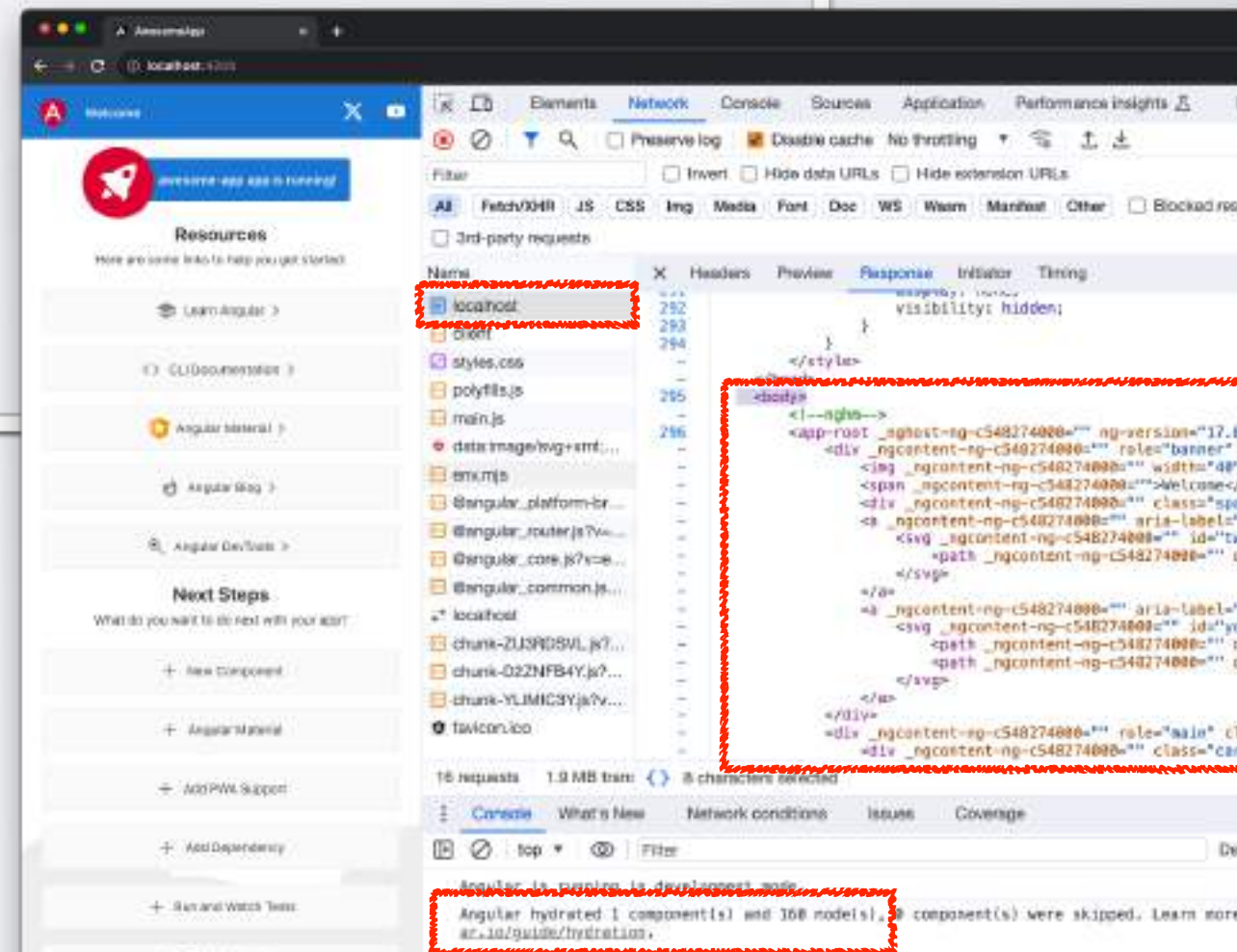
... even Angular can do that now 😊

Using Angular CLI v17 or later

```
> npx @angular/cli@next new ng-ssr
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR)
  and Static Site Generation (SSG/Prerendering)? Yes
```

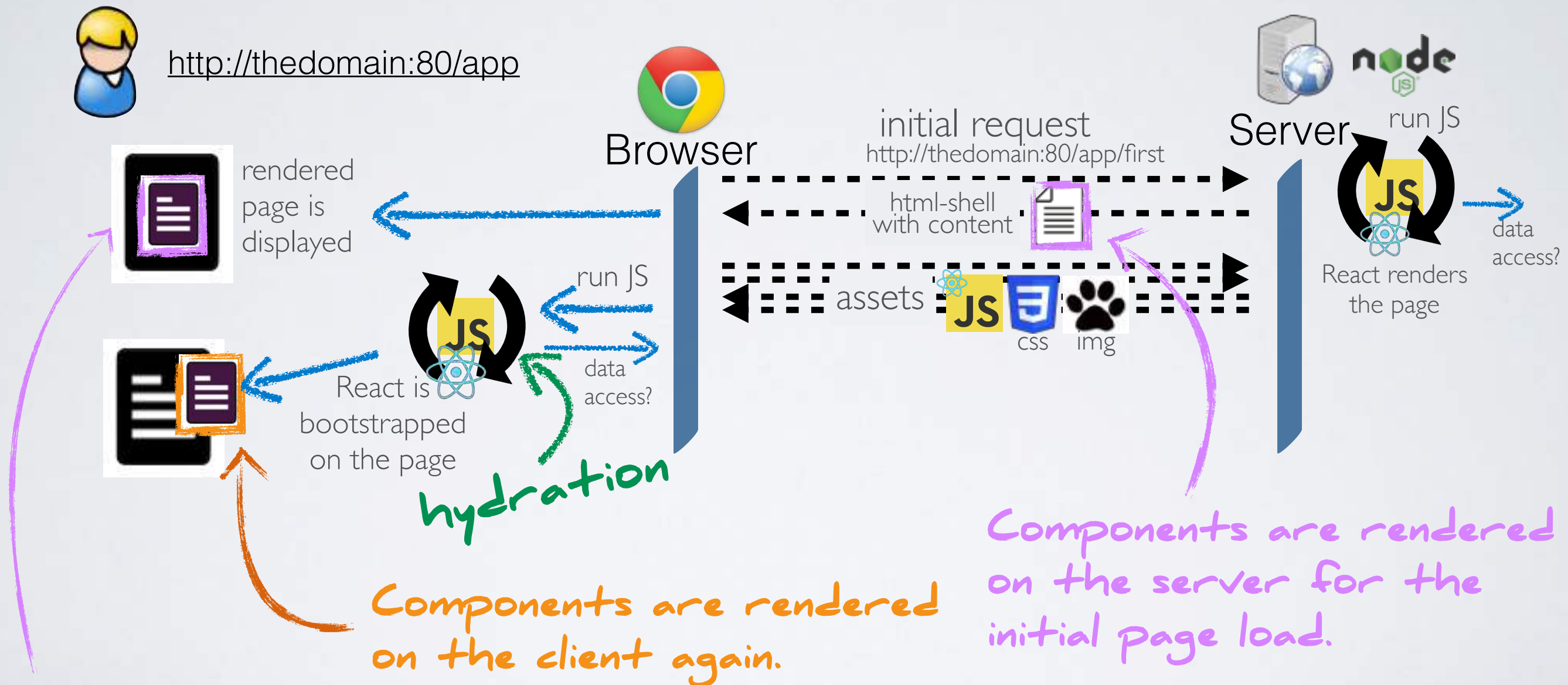
...

✓ Packages installed successfully.
cd ng-ssr
npm start



SPA with Server Side Rendering (SSR)

(initial rendering on the server - hydration on 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)

Hydration

[https://en.wikipedia.org/wiki/Hydration_\(web_development\)](https://en.wikipedia.org/wiki/Hydration_(web_development))

Hydration is the process of converting static HTML to a dynamic web page by attaching event handlers to the DOM.

Traditional frontend frameworks implement hydration by rebuilding the whole component tree on the client.

... but to build the component tree, you also need the data for the components ...

Fetching Data on the Server?

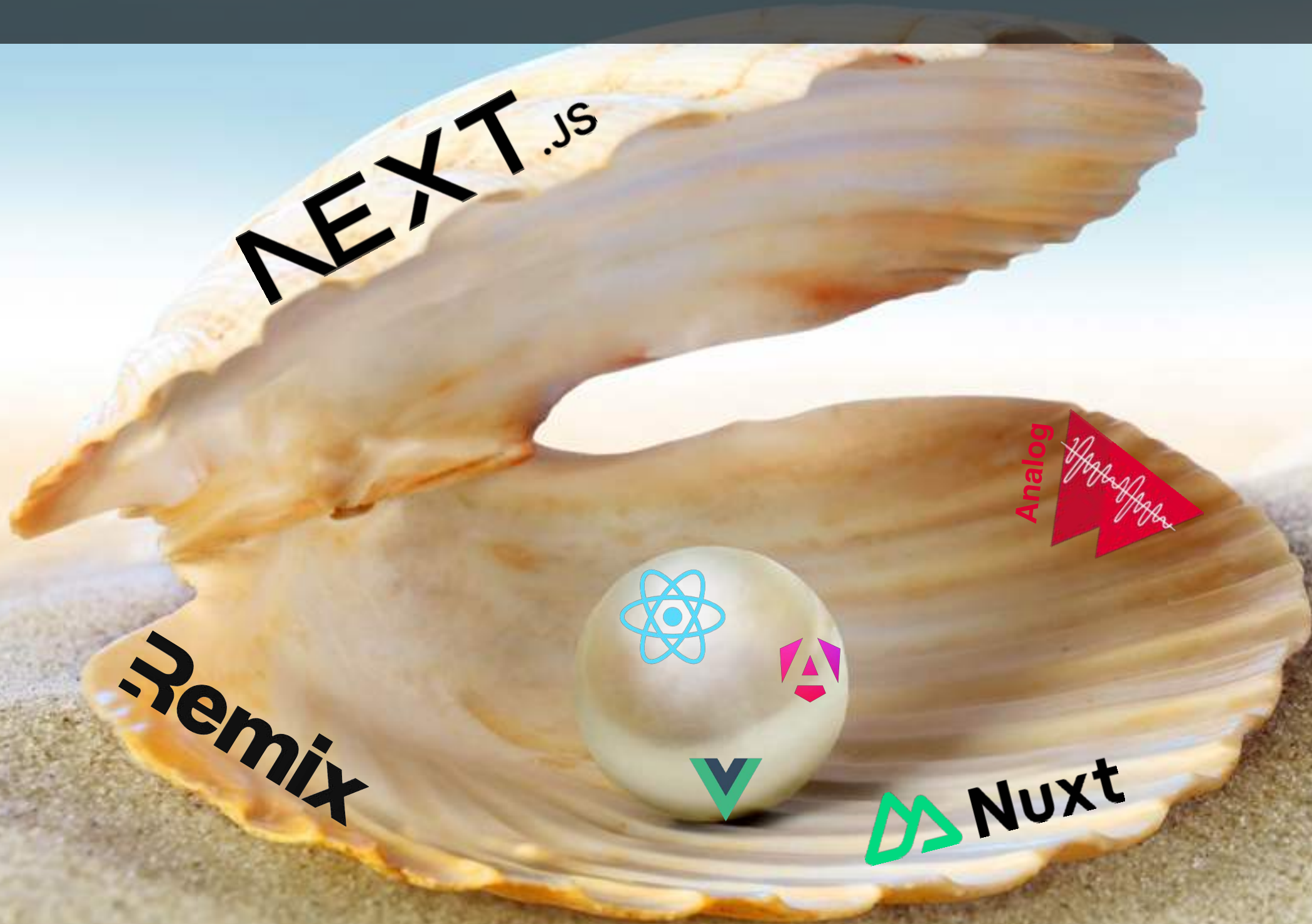
The traditional component model of frontend frameworks imposes synchronous rendering.

Data fetching in JavaScript is inherently asynchronous.

=> data fetching on the client involves several "render cycles" of component instances (stateful)

=> this is in conflict with server-side rendering which must be stateless ...

Meta Frameworks



Meta Frameworks

NEXT.js **Remix**




A common goal of meta-frameworks is to simplify the project setup of frontend applications.

Typically using convention over configuration.

- routing setup - file based routing
- server-side rendering & static site generation
- best practices for build & deployment
- api endpoints - file based
- server-side data fetching

=> compared to other frameworks, Angular is much more "complete", therefore there is less need for a Meta-Framework ...

Meta Framework Demos:

- Remix (React) **Remix**
 - Analog (Angular) 
- file-based routing
 - data-fetching:
 - data is fetched on the server
 - on initial load data is rendered into html & passed for hydration
 - on navigation data is transported via http
 - hydration: components rendered on server & client

<https://remix.run/docs/en/main/file-conventions/routes>

<https://remix.run/docs/en/main/discussion/routes>

<https://analogjs.org/docs/features/routing/overview>

Hydration



Miško Hevery (AngularJS/Angular/Qwik) ✓

@mhevery



Hydration is a horrible workaround because web frameworks don't embrace how browsers actually work.

Yet somehow we have turned it into a virtue.

Hydration is a hack to recover the app&fw state by eagerly executing the app code in the browser.

That is why your app is slow.

5:46 AM · Apr 13, 2022

<https://twitter.com/mhevery/status/1514087689246568448>

Hydration is Pure Overhead:

<https://www.builder.io/blog/hydration-is-pure-overhead>

Beyond traditional Server Side Rendering

Island Architecture
Streaming Server Side Rendering

also:

Progressive Hydration

Resumability (Qwik, Wiz)

<https://www.patterns.dev/vanilla/islands-architecture/>

<https://www.patterns.dev/react/streaming-ssr>

<https://www.patterns.dev/react/progressive-hydration/>

<https://qwik.dev/docs/concepts/resumable/>

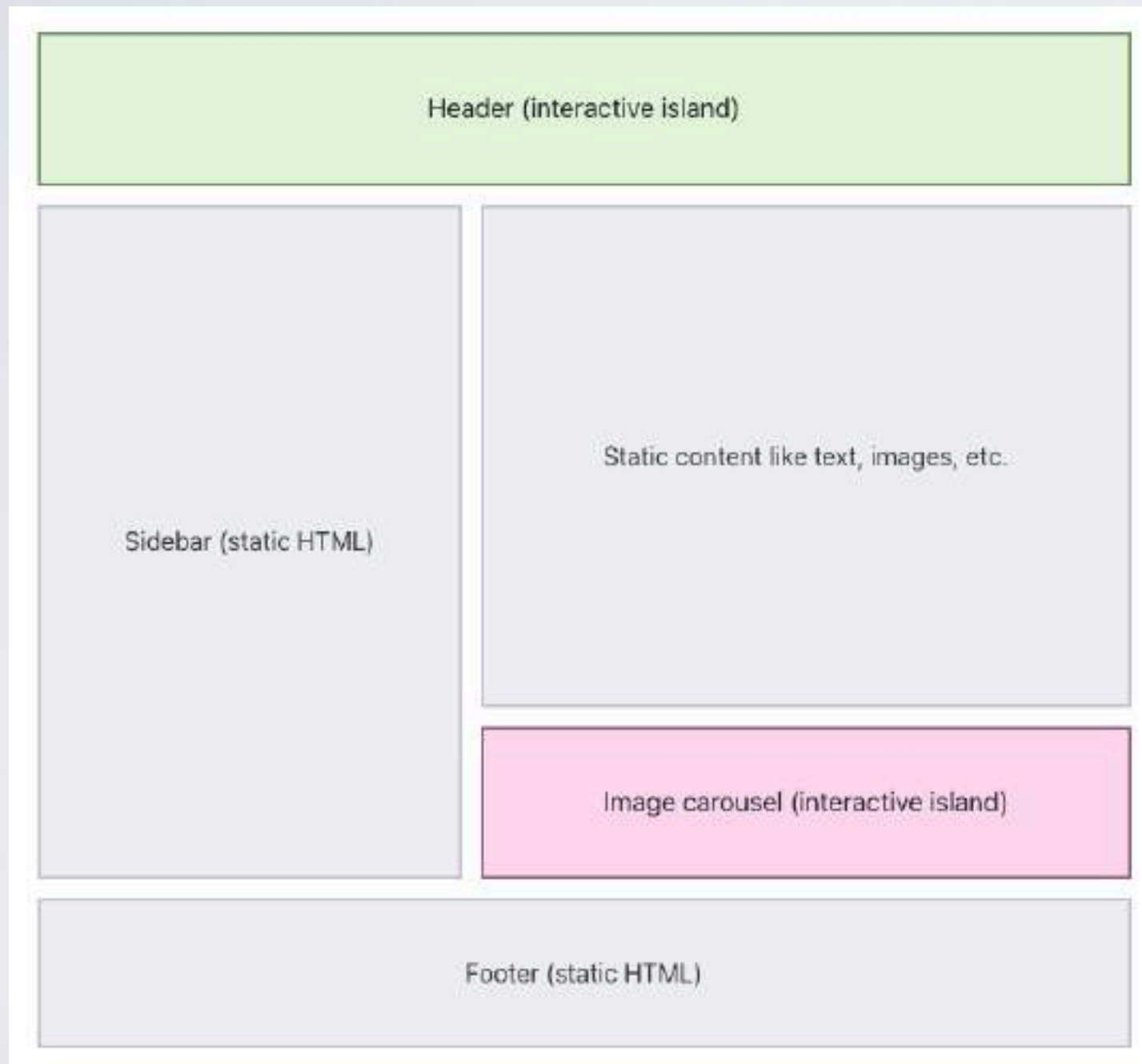


Island Architecture

(aka partial hydration)

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

Island Demo with Astro



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



Streaming Rendering

deliver chunks of ui as soon as available
rather than waiting for the entire content

Streaming Demos

Simple Streaming Demo

Streaming in Astro

Out of Order Streaming in Next.js

Online: <https://streaming-function.vercel.app/>

Evolution of Streaming Rendering:

Next.js Partial Prerendering

Delivering the static UI from a CDN and streaming the dynamic parts from a server.

Demo: <https://www.partialprerendering.com/>

Doc: <https://nextjs.org/learn/dashboard-app/partial-prerendering>

Astro Server Islands

Loading the static UI from a CDN and dynamically loading chunks of UI from the server.

Demo: <https://server-islands.com/>

Announcement: <https://astro.build/blog/astro-4120/>



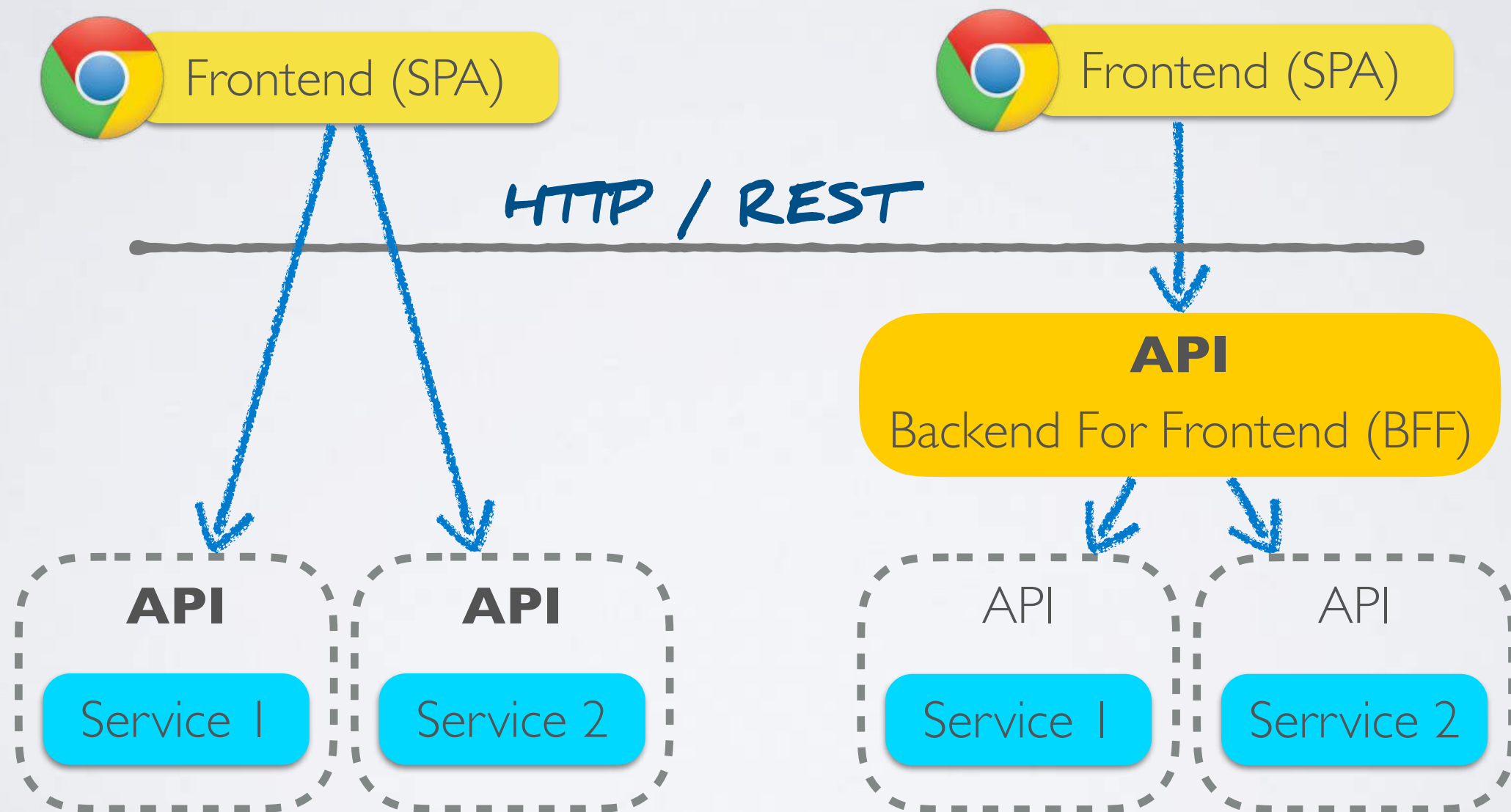
So far the traditional Client-Server
Boundary is still intact ..



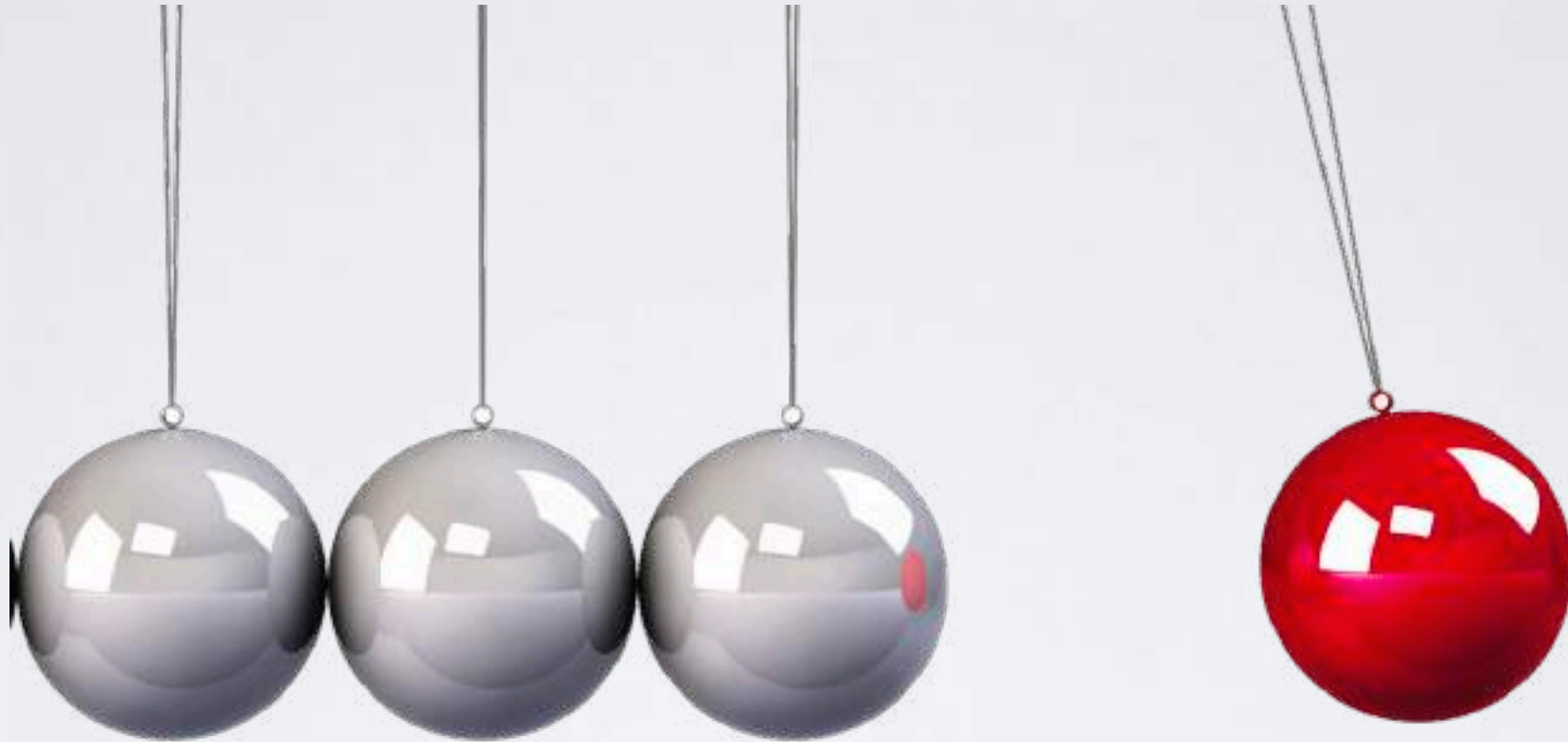
... now we are going to break this
boundary!

Traditional Architectures for SPAs

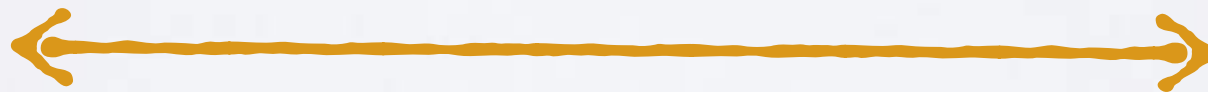
Client - API - Server



The Pendulum is swinging ...



Server

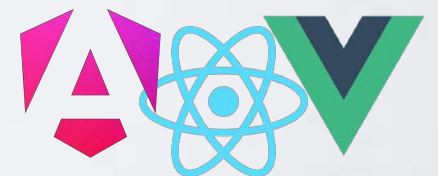


Client

NEXT.js

Remix

SVELTEKIT



... but why?

Blurring the Boundary - Why?

Depending on the rendering strategy, data needs to be fetched on the server and/or on the client.

Advantages of Full-Stack Development:

reducing complexity (technical *and* organisational)

- single programming language: reusability, type-safety
- one tech stack: maintainability, developer knowhow
- improving development speed, reducing team-dependencies

some "ui-logic" is not suited for the client

(i.e. rendering Charts or Markdown)

"Transparent" Server-Side Data-Fetching

The vanishing network – no HTTP-API needed!

"Mind The Gap" by Ryan Florence at Big Sky Dev Con 2024:
<https://www.youtube.com/watch?v=zqhE-CepH2g>

Abracadabra: The vanishing network — Kent C. Dodds | React Universe Conf 2024
<https://www.youtube.com/watch?v=E8LLty9rTWw>

Meta Framework Demos:

- Remix (React) **Remix**

- colocation of client and server code
- hooks for client-side
- streaming
- full-stack data-flow

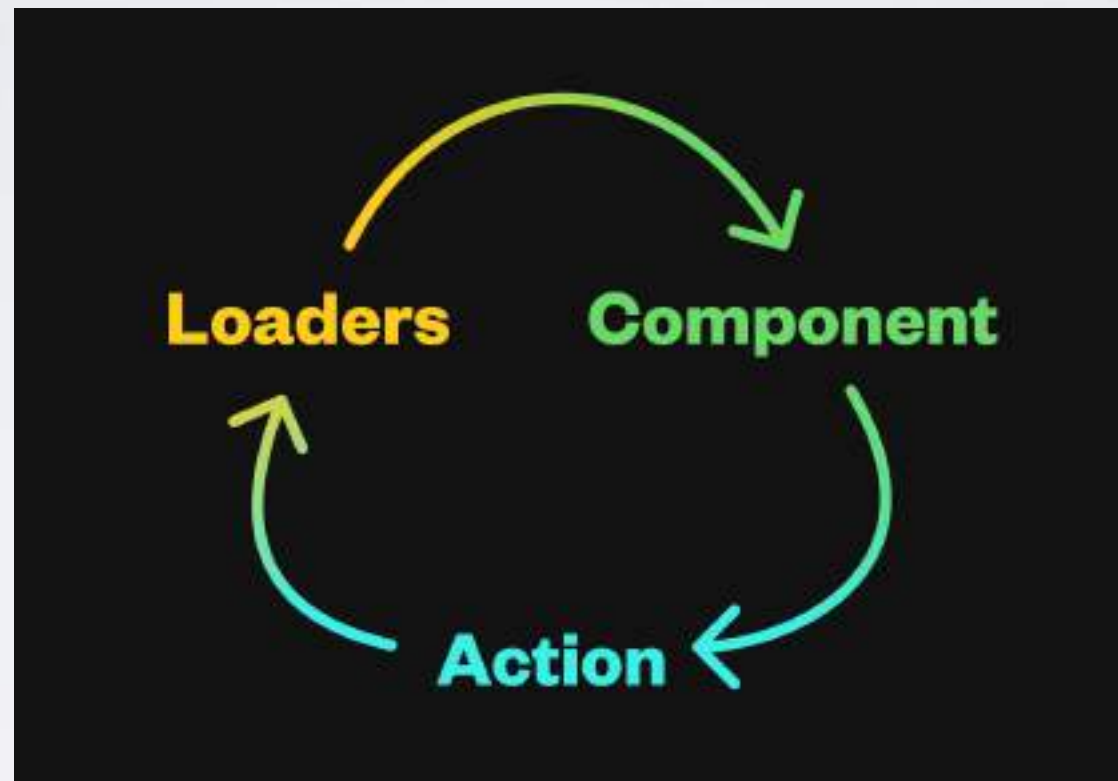
- Analog (Angular)

"Transparent" Server-Side Actions

STAR WARS
RETURN^{OF}_{THE} **RPC**

Demo:

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>



qwik Qwik City

<https://qwik.builder.io/docs/route-loader/>
<https://qwik.builder.io/docs/action/>
[https://qwik.builder.io/docs/server\\$/](https://qwik.builder.io/docs/server$/)



SVELTEKIT

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

Similar RCP
concepts:



Hilla

<https://hilla.dev/>

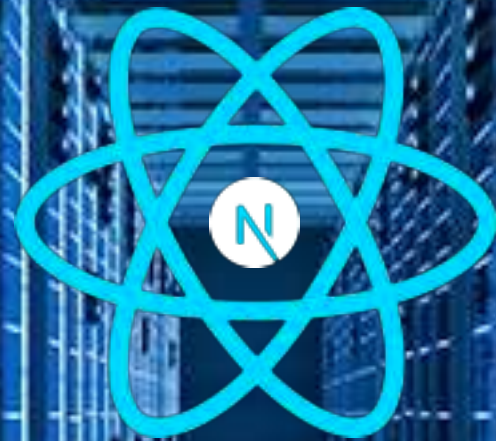


<https://trpc.io/>



TanStack Start

<https://tanstack.com/start/>



React Server Components

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!

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!

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

☐ Blocked 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>
```


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())

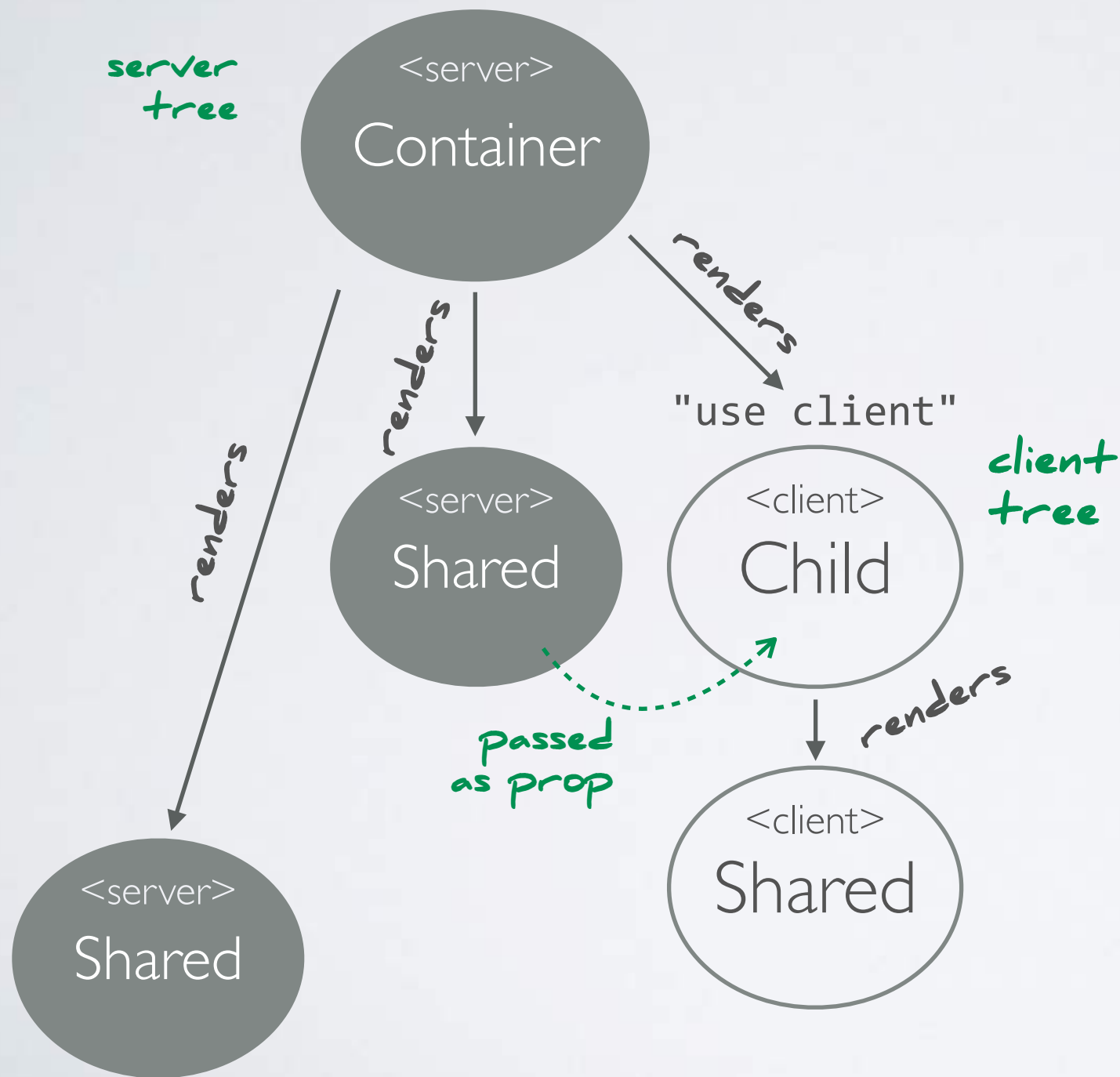
  useEffect(() => {
    setInterval(() => setTime(new Date())), 1000);
  }, []);

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

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

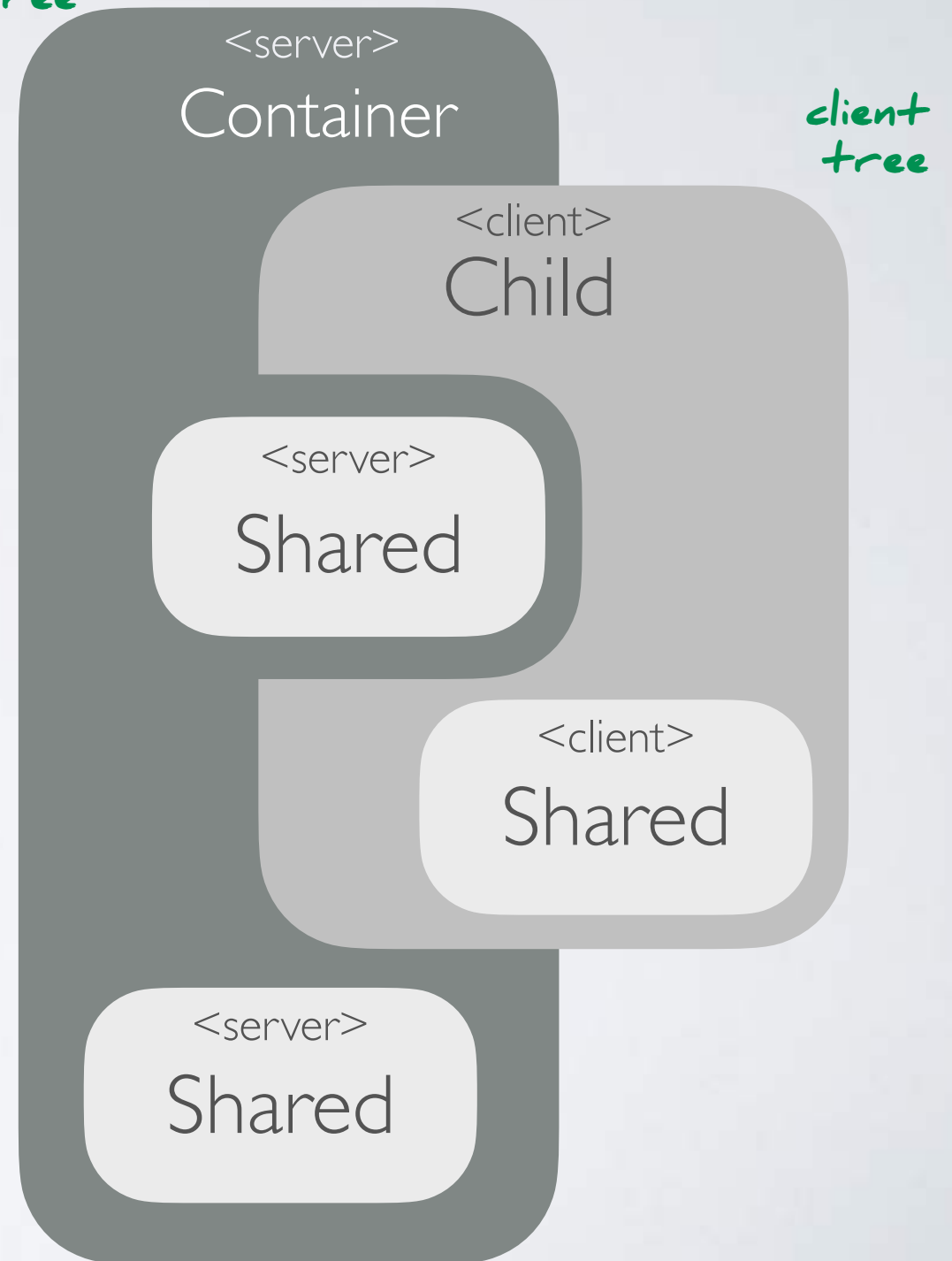
Composition

"logical perspective"



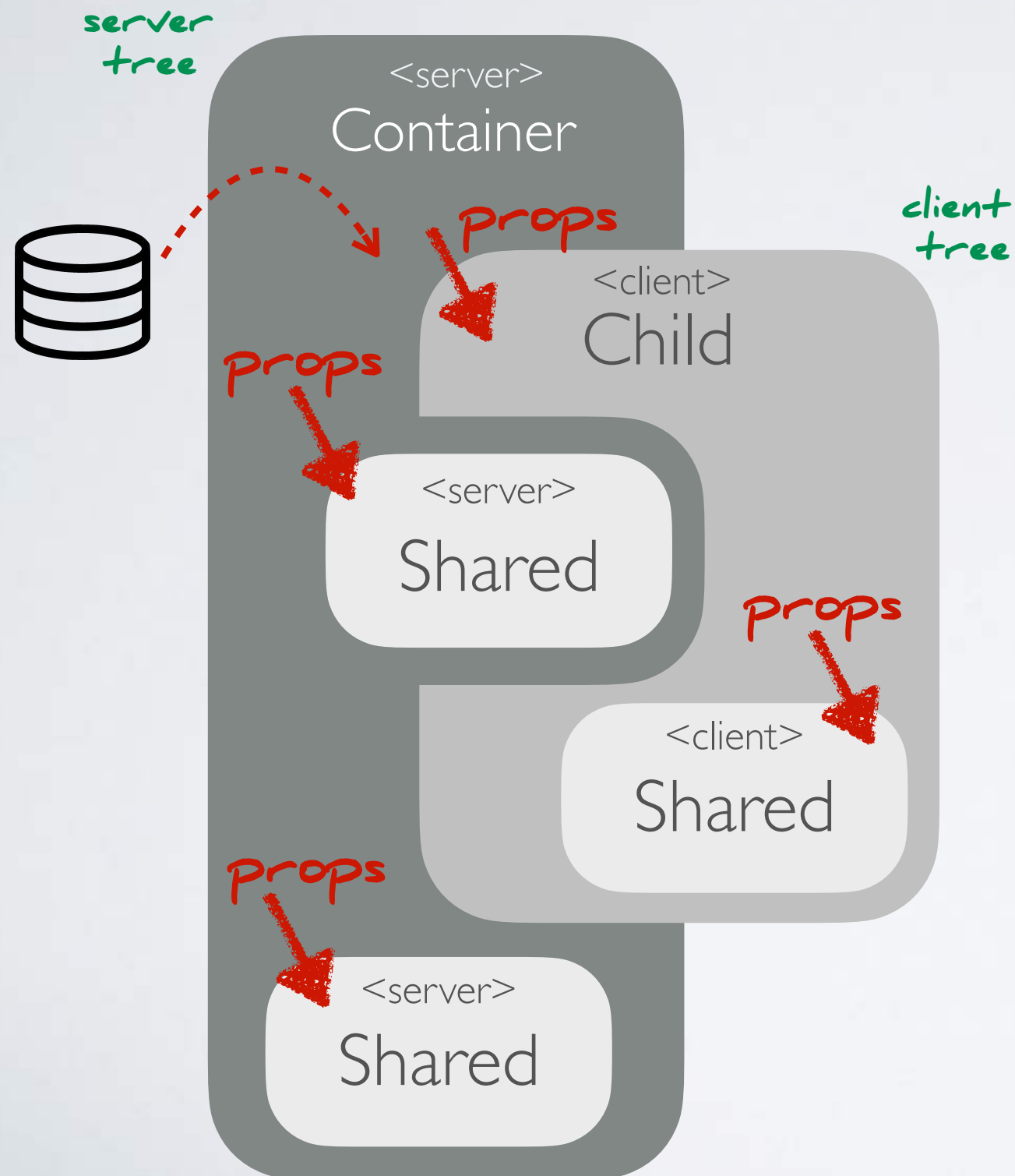
"composition perspective"

server tree

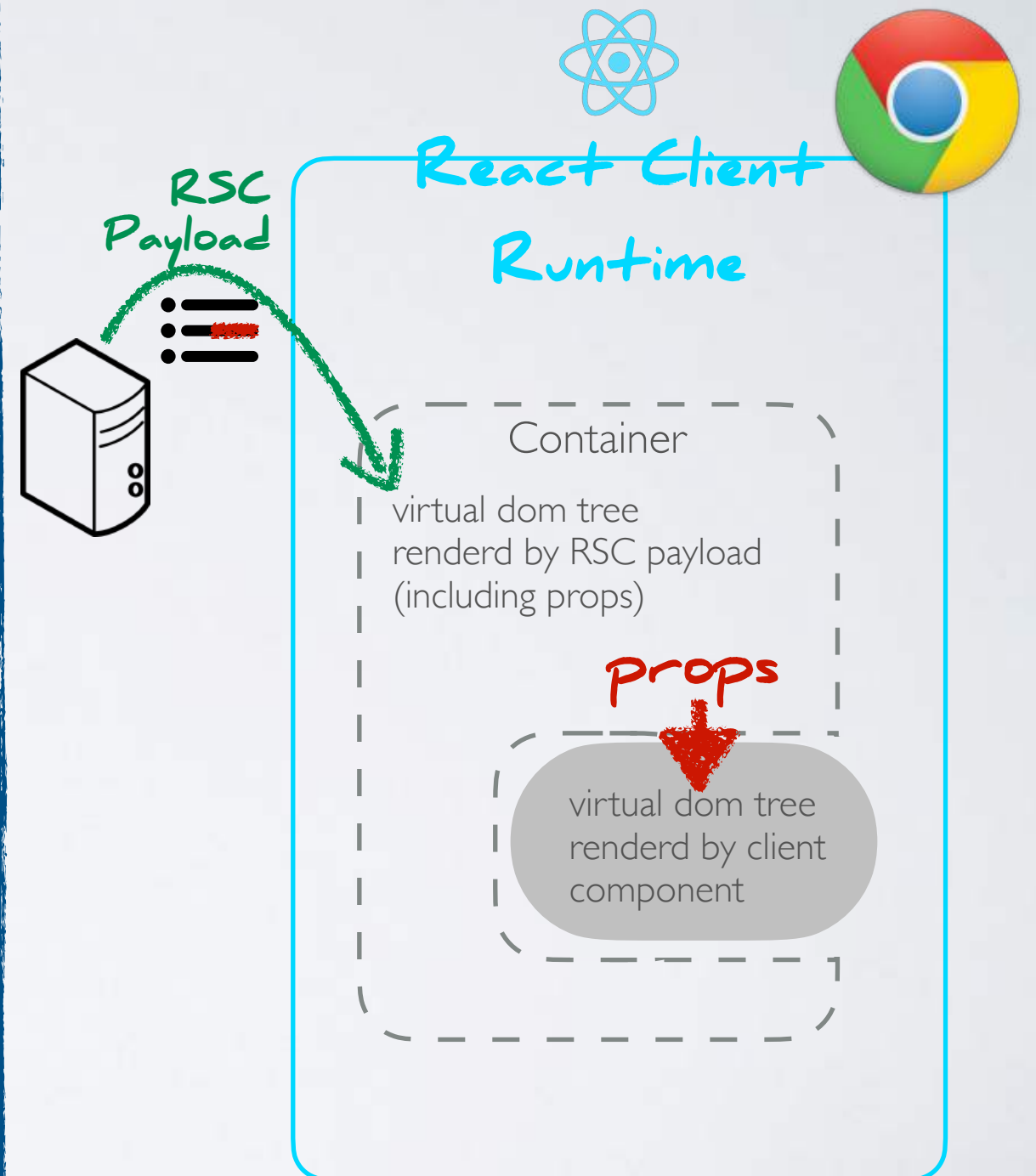


Full-Stack Data Flow

"composition perspective"



"physical perspective"



A component tree that spans
between client and server!



danabra.mov 

@dan_abramov

never write another API

6:19 AM · Mar 4, 2023 · **39.5K** Views

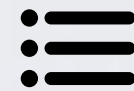
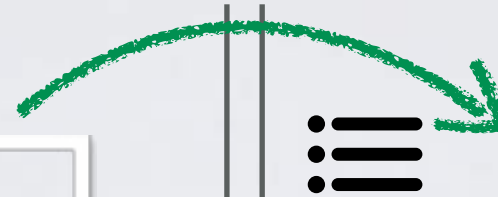
https://twitter.com/dan_abramov/status/1631887155000000000



Browser



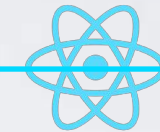
Network



RSC Payload sent to the browser

first scenario: server component calling server function

rendering



React Client Runtime

(virtual dom tree)

second scenario: client component calling server function

rendering

API call



RPC endpoint

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

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

JS

JavaScript bundle loaded by the browser



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



Server

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 difficult (and frustrating) to draw the boundary between features of React Server Components and Next.js.

Waku

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



Remix announced RSC integration in a future version.
<https://remix.run/>

Server Driven SPA



Server Driven SPAs

aka "Live View"



Vaadin



Blazor Server



Phoenix Framework

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

Demos:

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

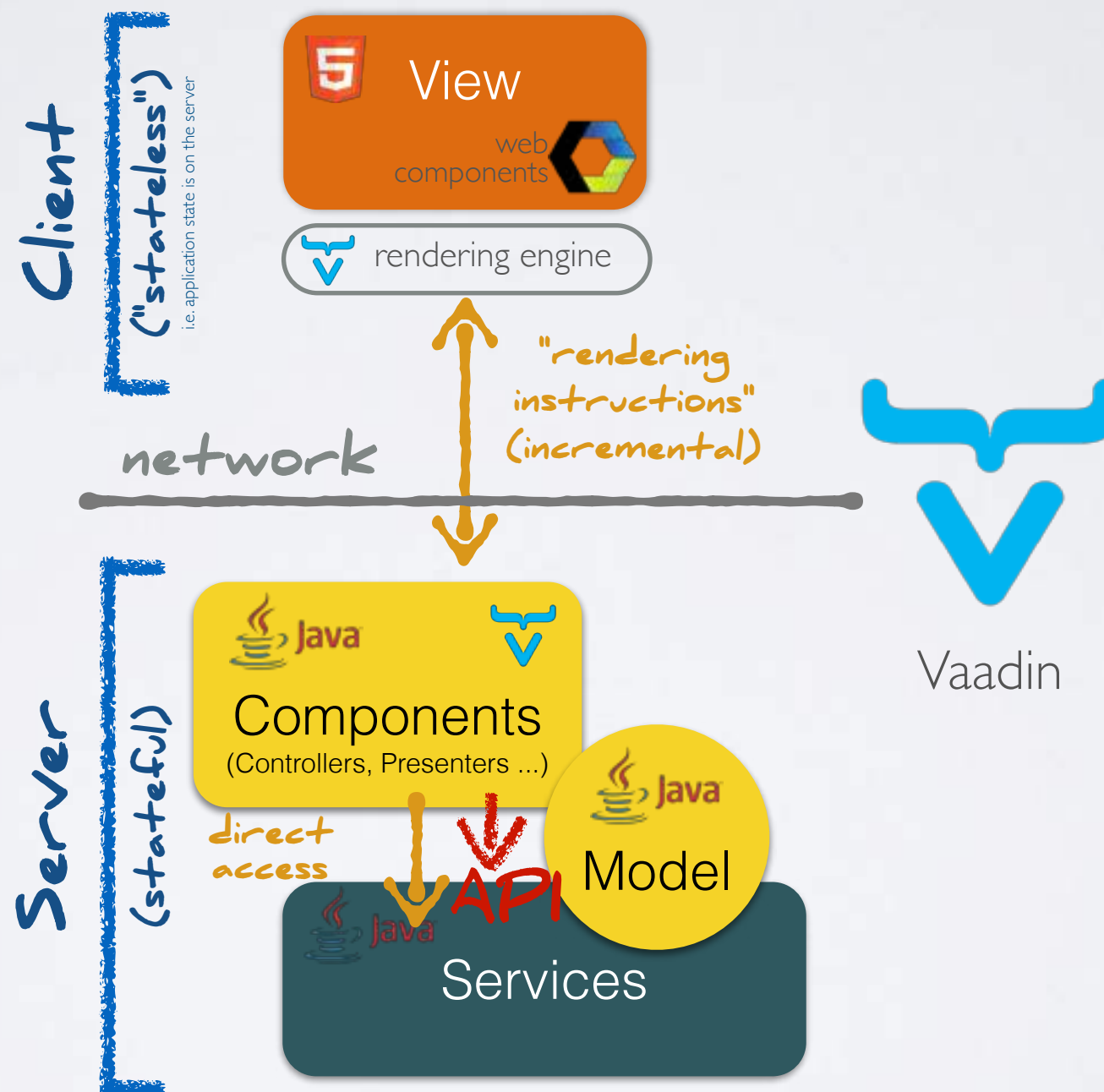
<https://blazor.syncfusion.com/demos/datagrid/overview?theme=bootstrap4>

<https://liveview.zorbash.com/>

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

Vaadin Architecture

using the browser just as a "rendering engine"



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 ...

Conclusion

Frontend technologies have a heavy influence on application architecture.

There are many flavors of "Full-Stack" development.

The frontend ecosystem has entered a new cycle of innovation focused on "Full-Stack" development.

Thank you!

Slides & Code: <https://github.com/jbandi/baselone-2024>

Questions? Discussions ...



Jonas Bandi

JavaScript / Angular / React / Vue / Vaadin

Schulung / Beratung / Coaching / Reviews

jonas.bandi@ivorycode.com



#BaselOne24

baselone.ch