Modern Frontend - Back To The Server

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 ...

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

letsboot.ch
swiss dev training

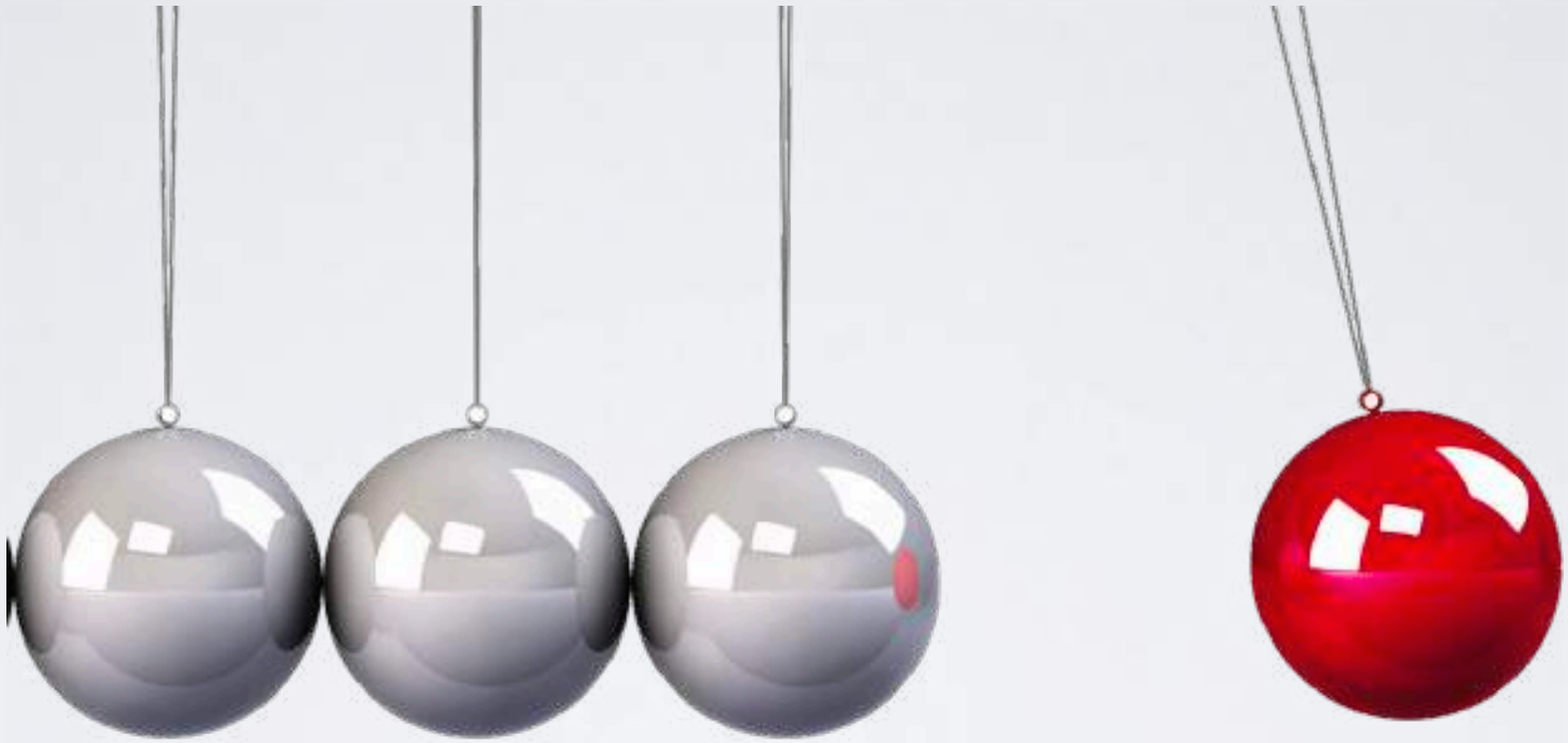
digicomp

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

jonas.bandi@ivorycode.com

What are you using ... ?

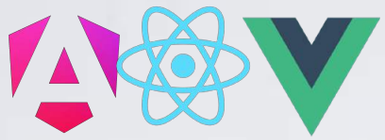
The Pendulum is swinging ...



Server ⟵ ⟶ Client

... will it ever stop?

AGENDA



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

SSR - Server Side Rendering

- Meta-Frameworks
- Hydration

Island Architecture

- HTTP Streaming
- Partial Prerendering & Server Islands

Server Side Data Loading & Mutations

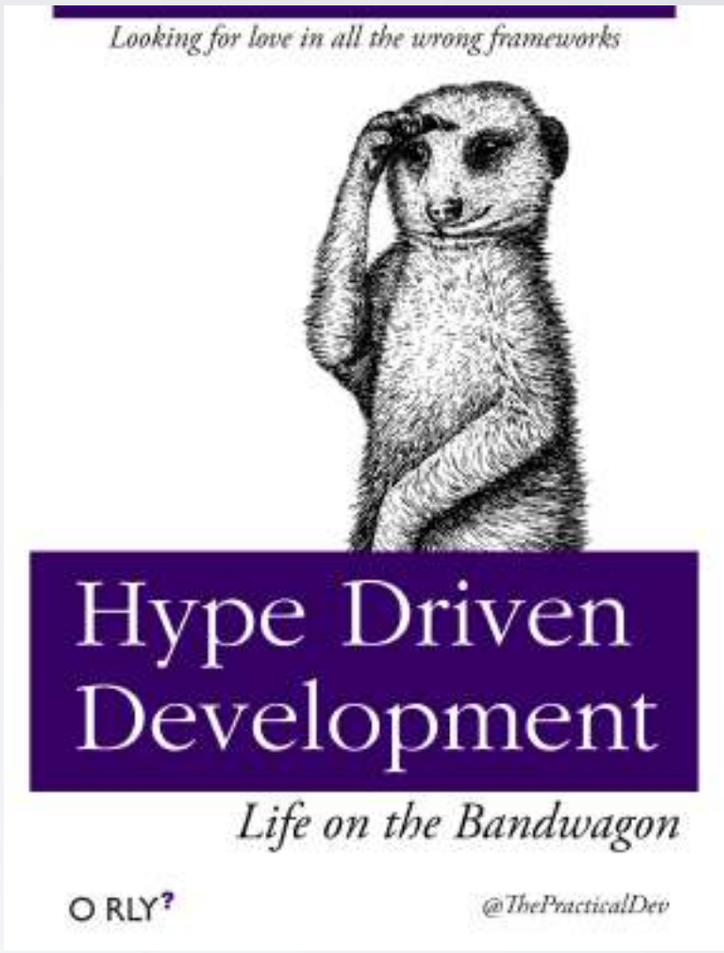
- RPC-style fetching and actions
- React Server Components

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

Slides & Code:

https://github.com/jbandi/baselone-2024

My goal is to make you feel like this:

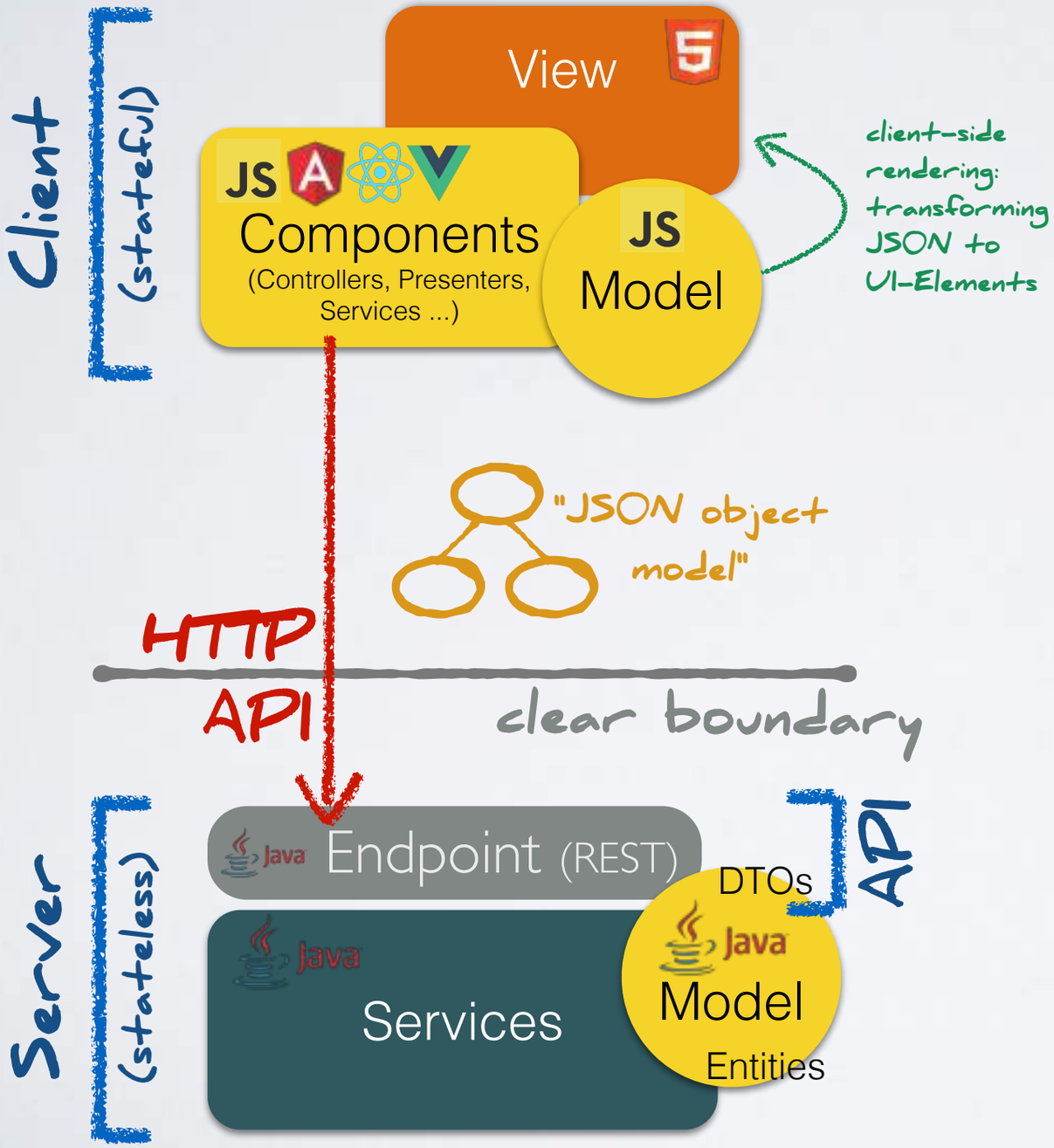


... not about the implementation details but hopefully by looking beyond the current "state of the art".

The Era of Single Page Applications

Architecture for 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)

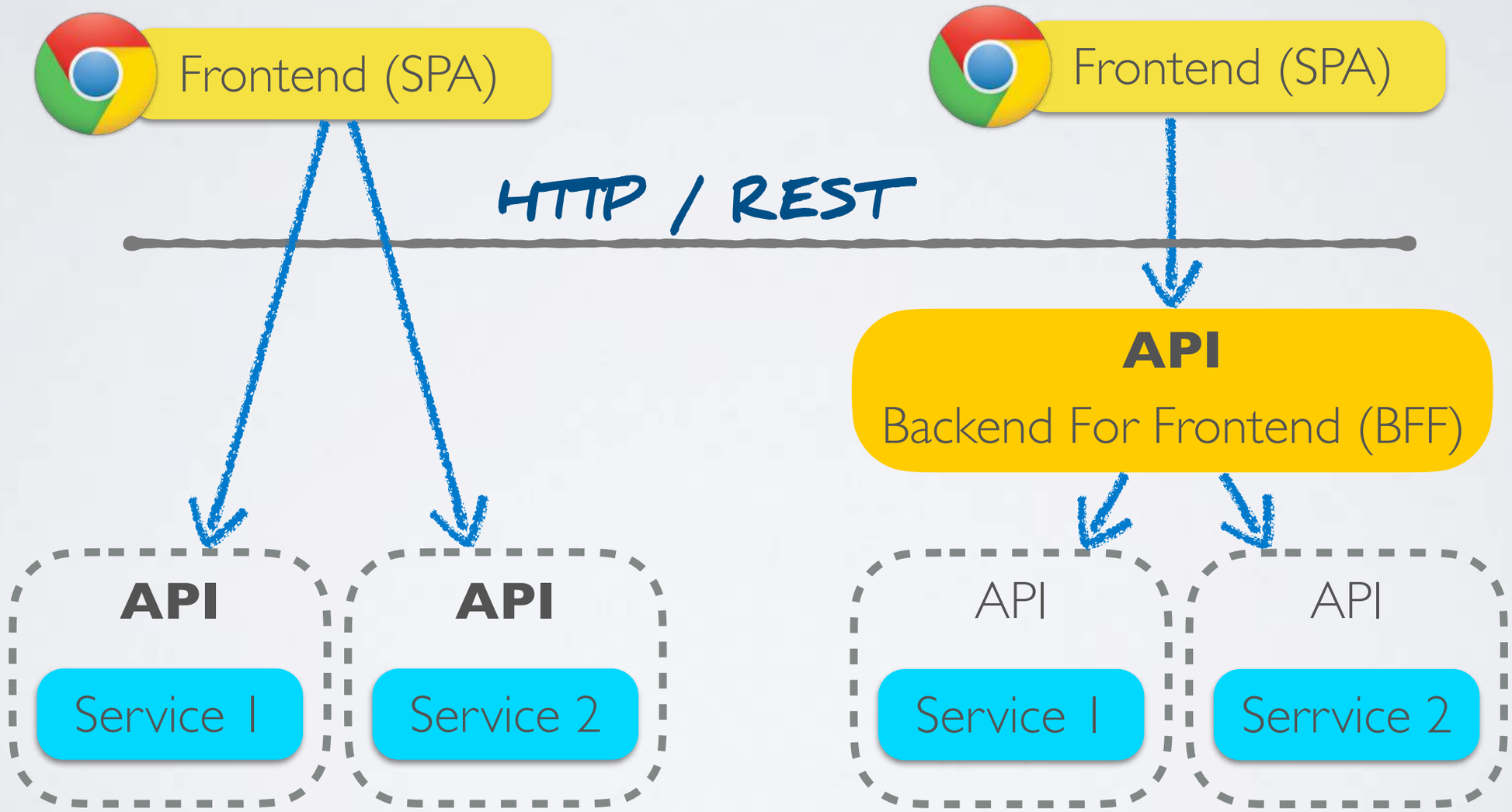
...

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 ...

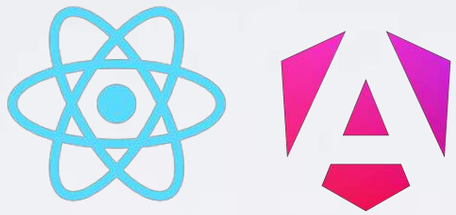
Traditional Architectures for SPAs

Client - API - Server



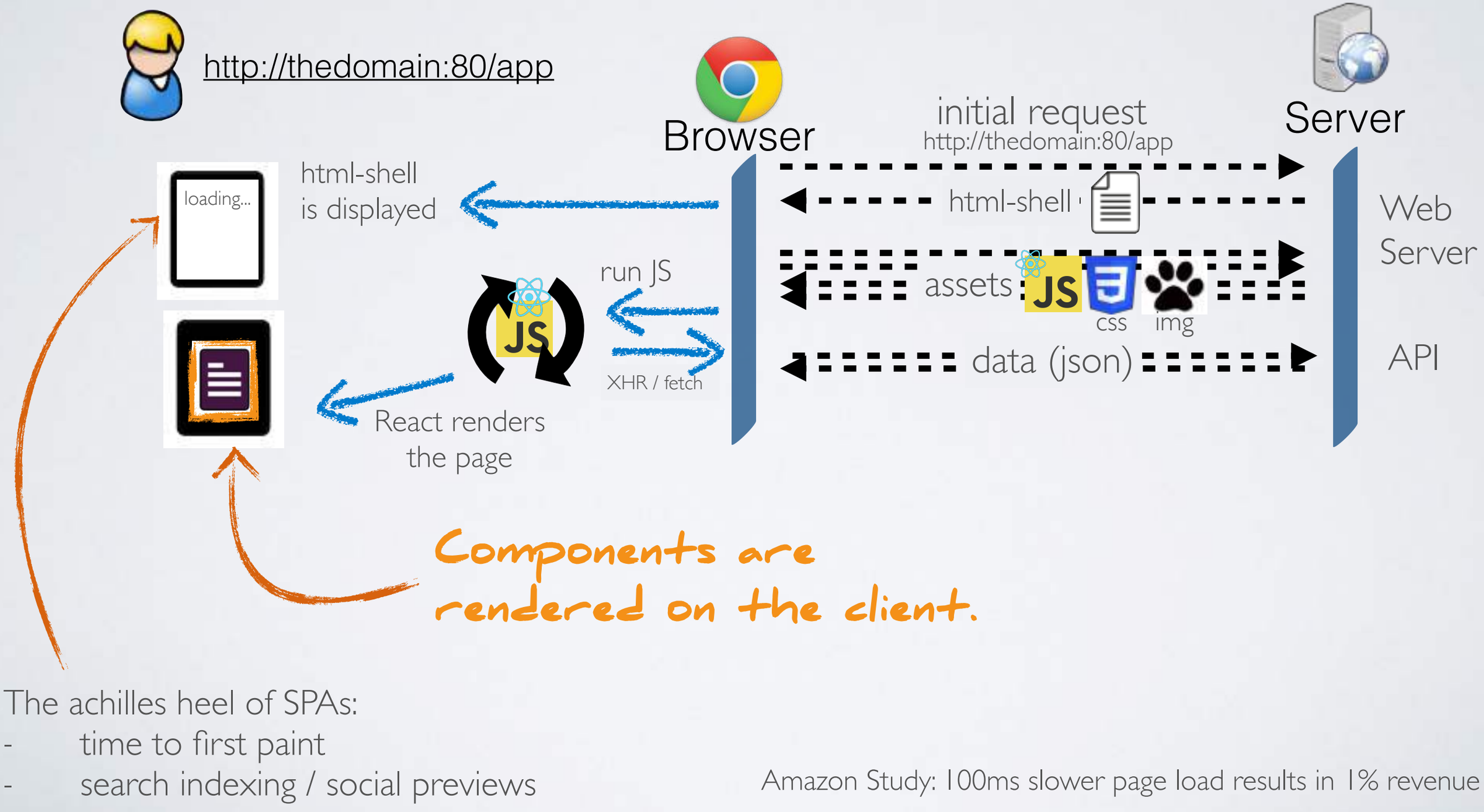
https://samnewman.io/patterns/architectural/bff/

Pure SPA Demo



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

Traditional SPA:
Client Side Rendering

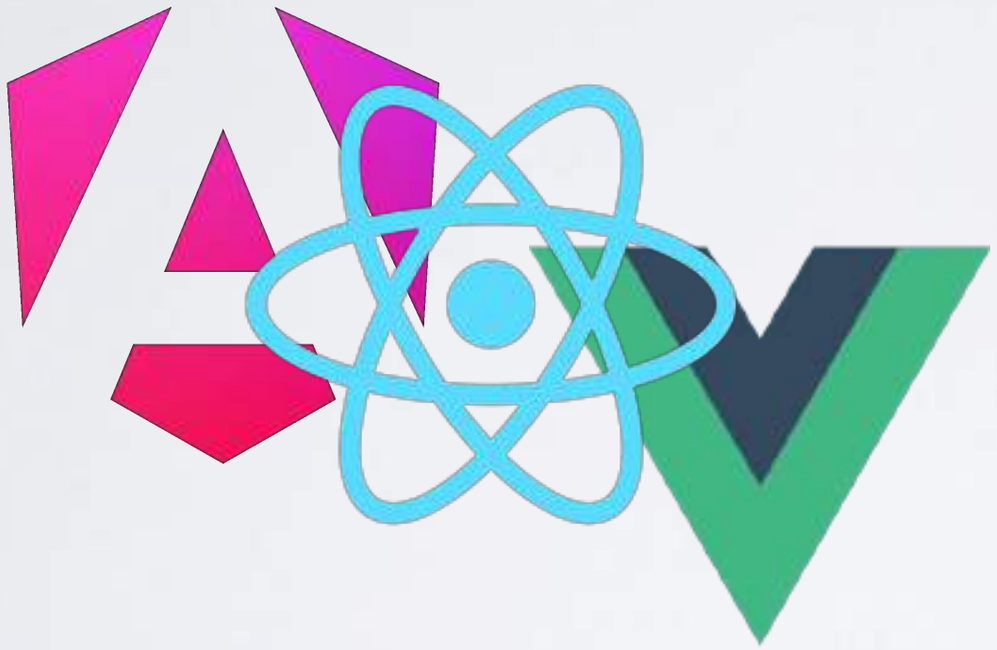


The achilles heel of SPAs:

- time to first paint
- search indexing / social previews

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

Server Side Rendering (SSR)



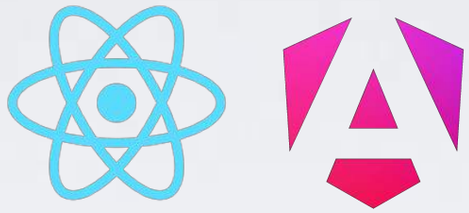
Today every modern frontend framework is capable of server side rendering.

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

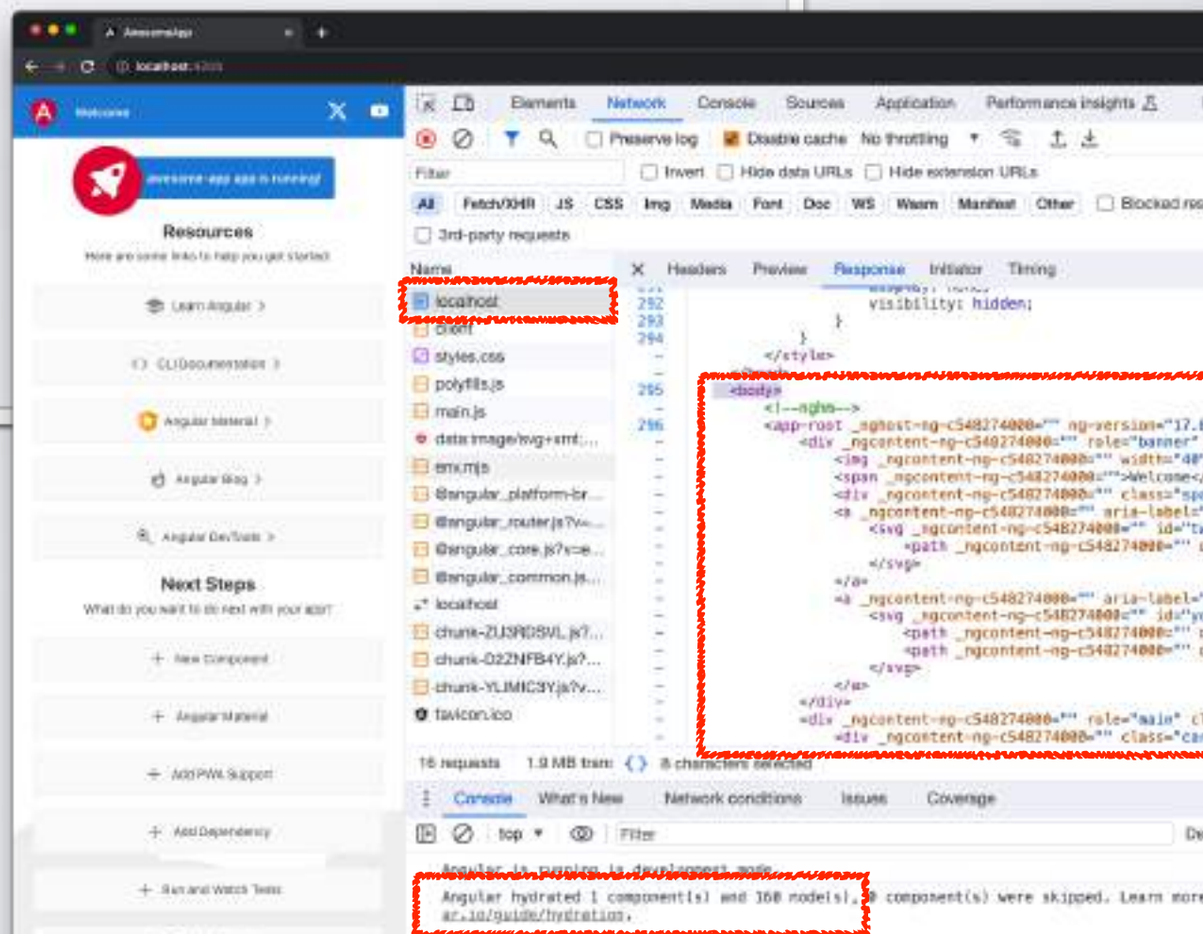
- Angular: HttpClient works with SSR!
  (note: no client-side fetch on initial load!)

Server Side Rendering (SSR)

... even Angular can do that now 😇

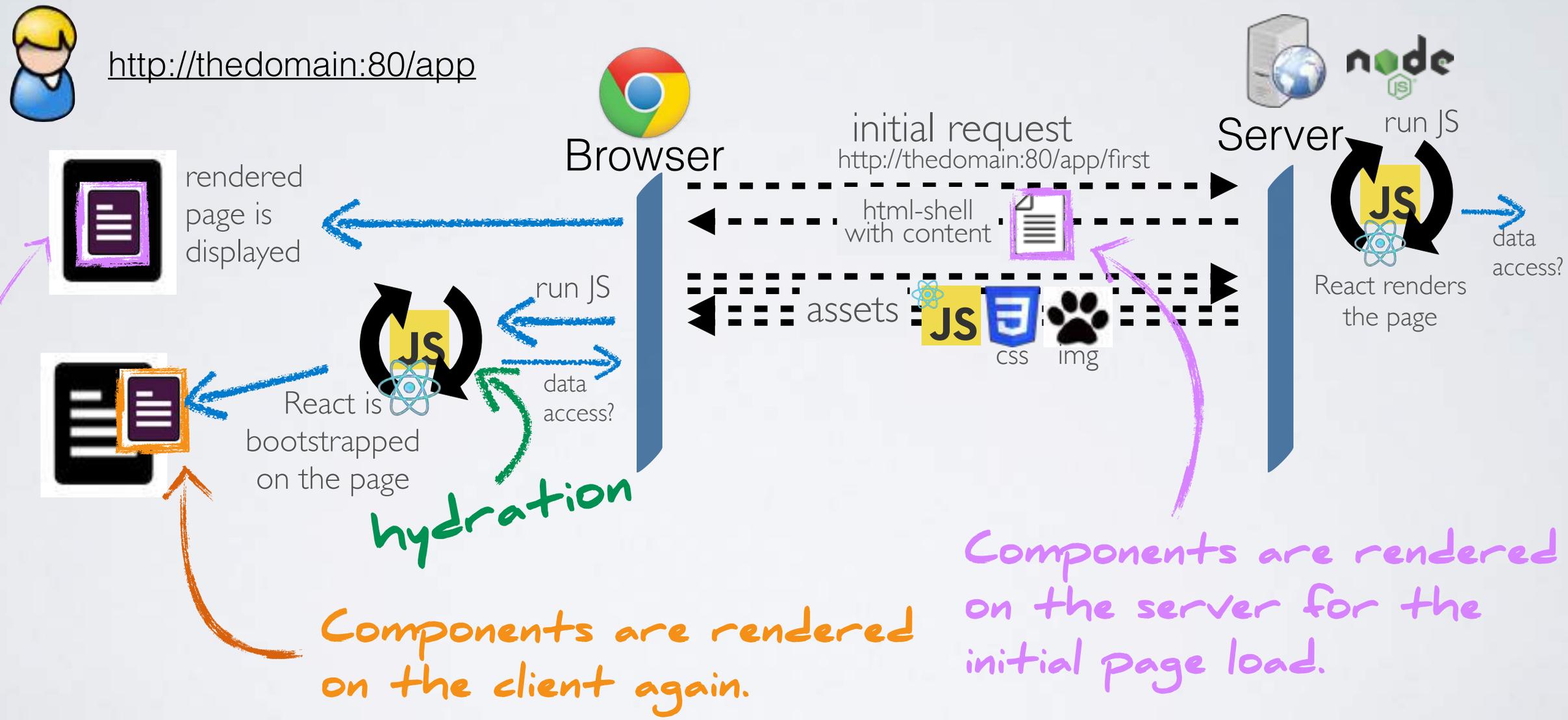
Using Angular CLI v17 or later

```
> 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)

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?

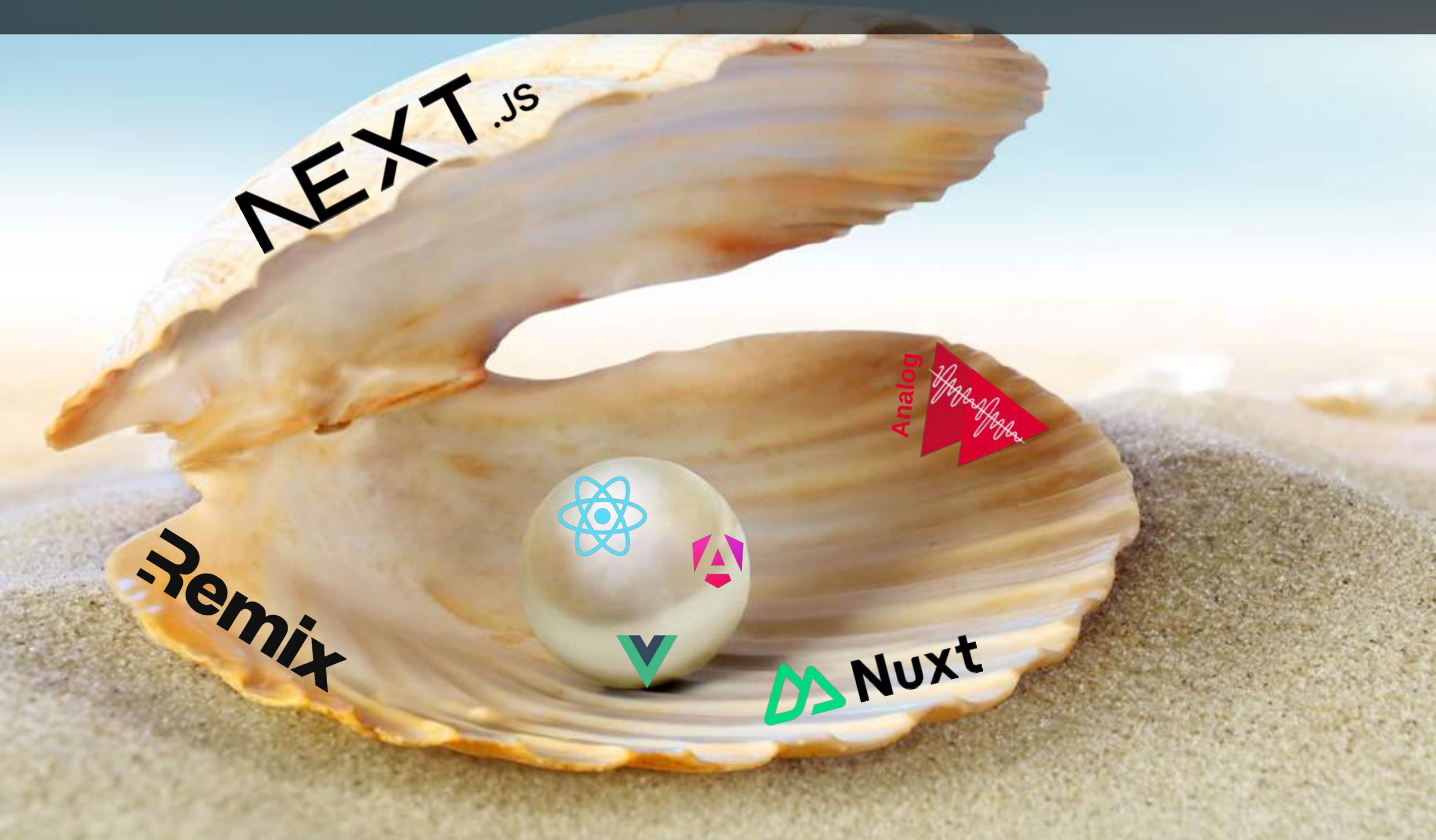
The traditional component model of frontend frameworks imposes synchronous rendering.

Data fetching in JavaScript is inherently asynchronous.

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

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

Meta Frameworks



Meta Frameworks

NEXT.JS

Remix

RedwoodJS

TanStack Start

Nuxt

Analog

SOLIDSTART

SVELTEKIT

qwik Qwik City

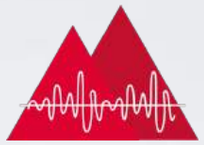
A common goal of meta-frameworks is to simplify the project setup of frontend applications.

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" and there is less need for a Meta-Framework ...

Meta Framework Demos:

- Remix (React) 
- 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

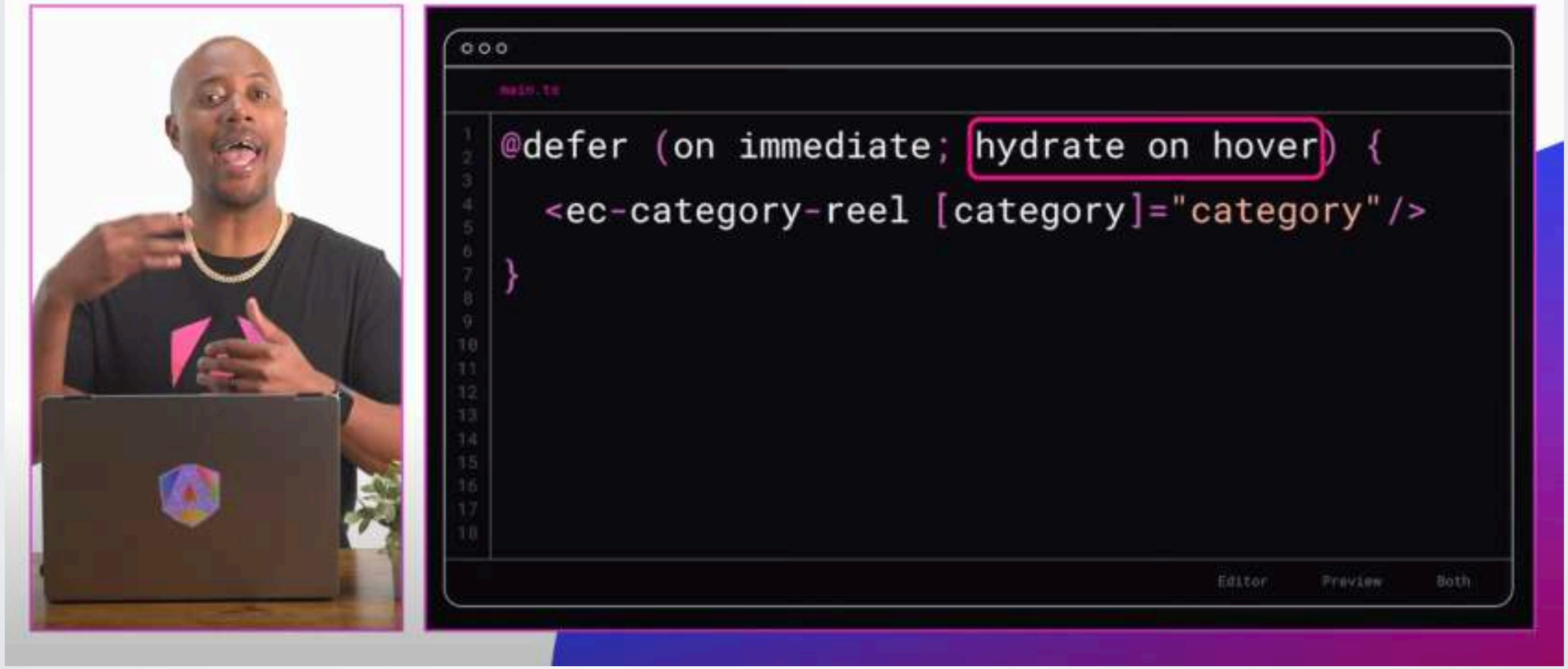
Island Demo with Astro

https://astro.build/

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

Incremental Hydration in future Angular

Integrated with the **`@defer`** templating feature for lazy loading components:



https://www.youtube.com/watch?v=I4n1IcZ3vRs

Streaming Rendering

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

Streaming Demos

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

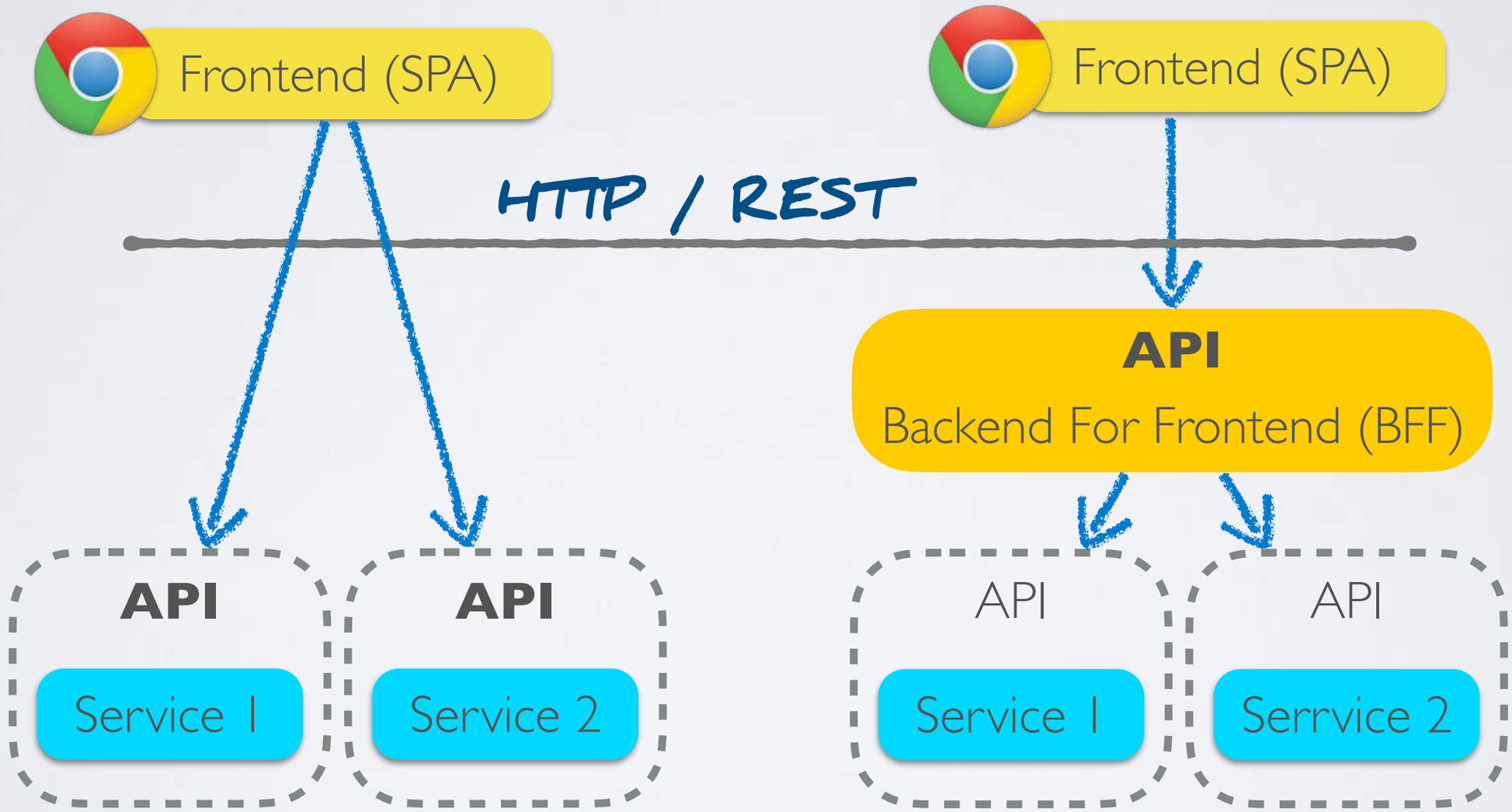
Loading the static UI from a CDN and dynamically loading chunks of UI from the server.

Demo: https://server-islands.com/
Announcement: https://astro.build/blog/astro-4120/

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

Traditional Architectures for SPAs

Client - API - Server



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

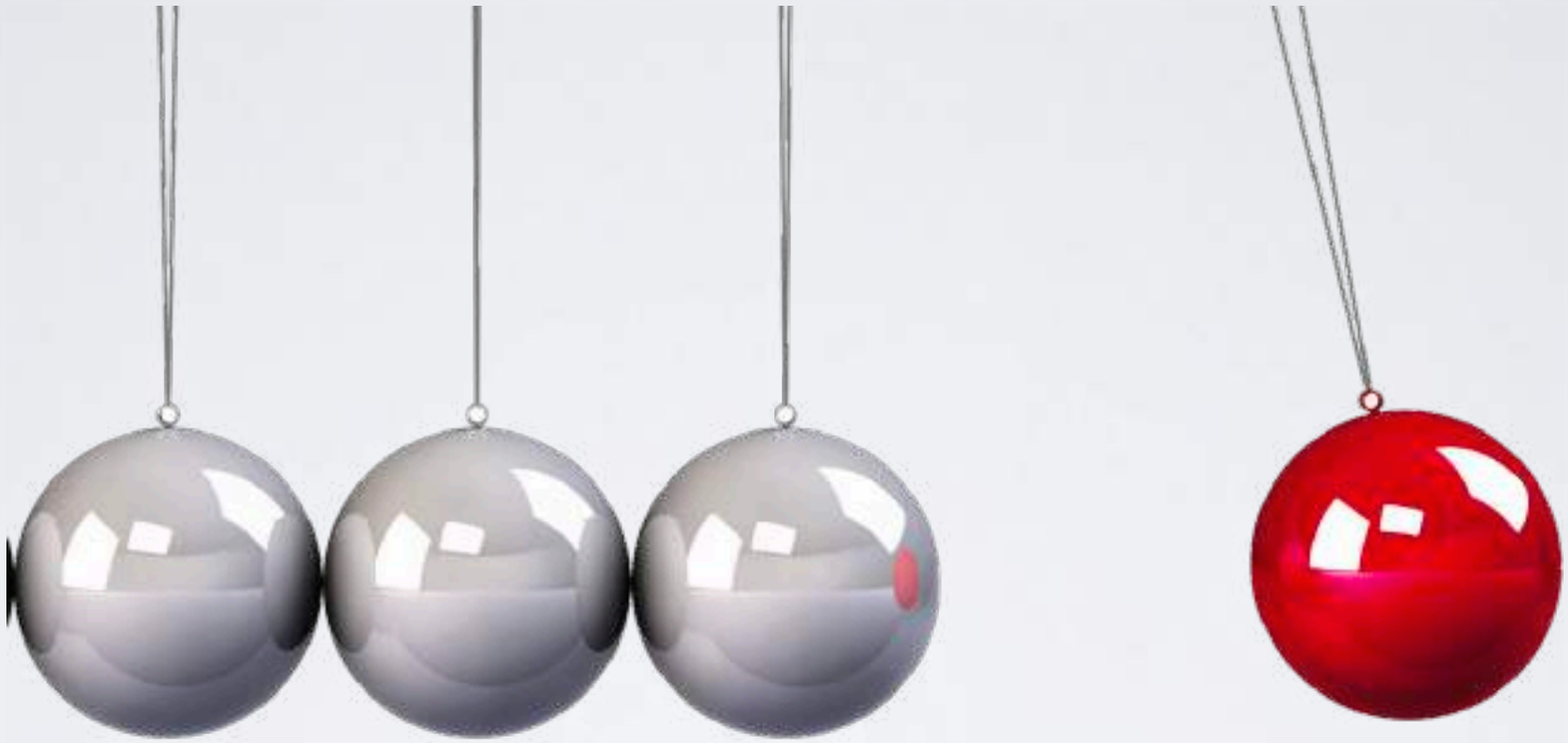
Breaking the Boundary - Why?

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

Once a frontend framework runs on the server, it is no longer constrained by the server-client separation of the traditional SPA architecture.

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

The Pendulum is swinging ...



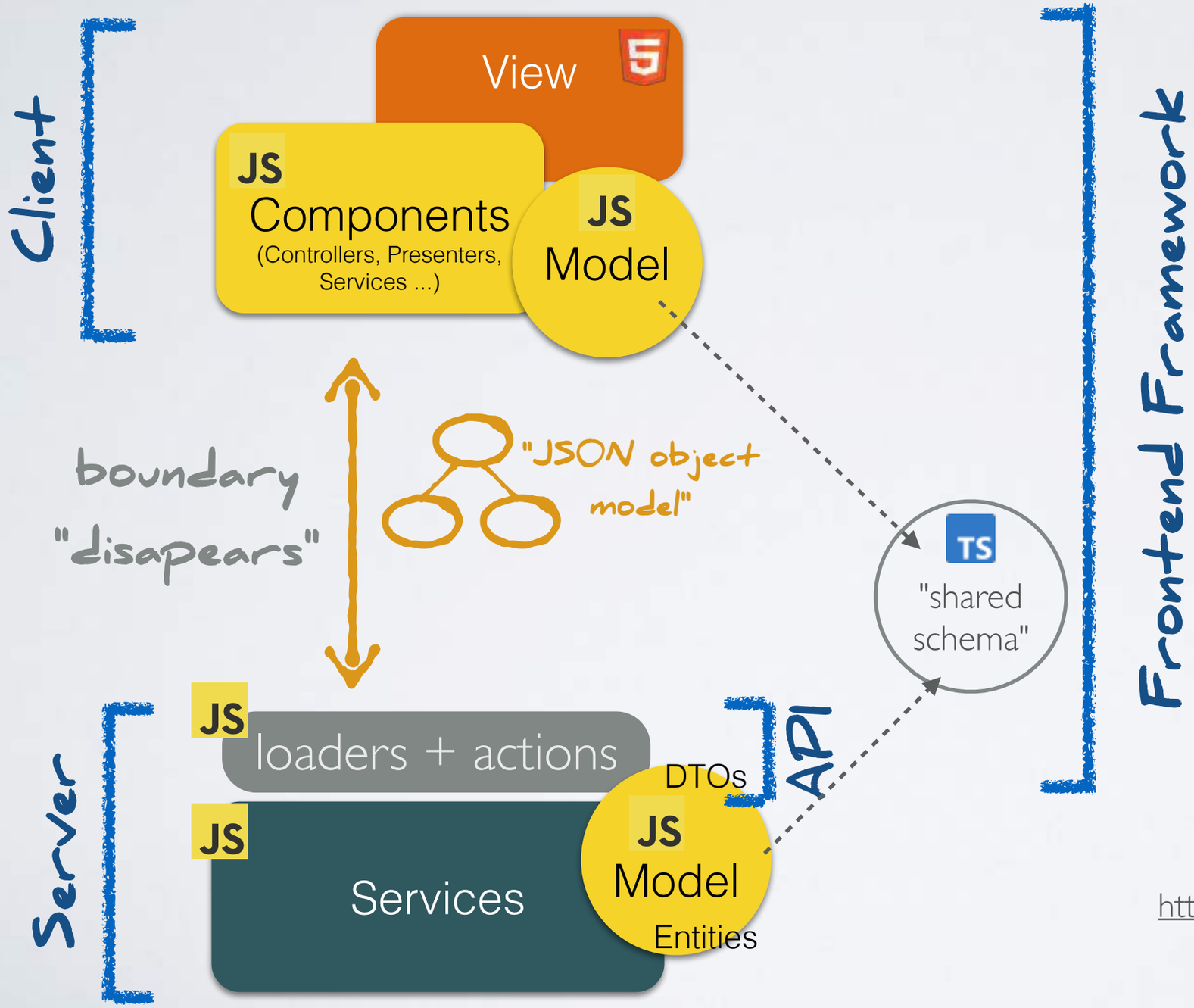
Server ⟵ ⟶ Client

NEXT.JS

Remix

SVELTEKIT

Full-Stack Frontend Frameworks



Server-Side data loading and actions

Remix

https://remix.run/

SOLIDJS Start

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/

"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

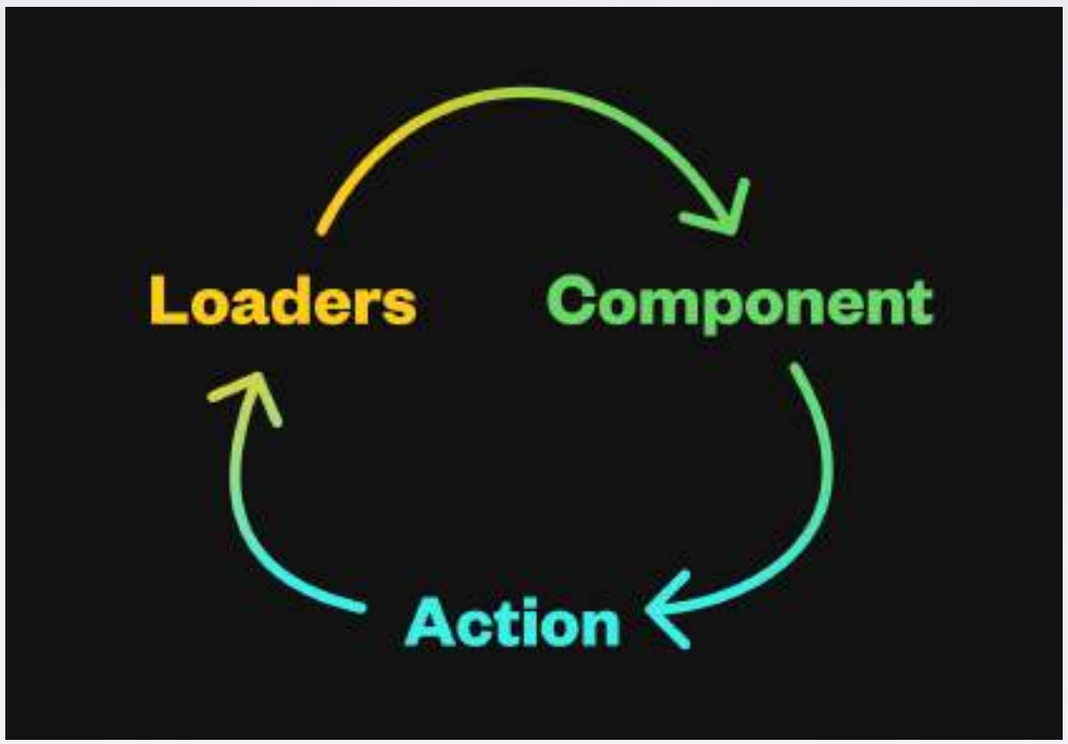
- colocation of client and server code
- hooks for client-side
- streaming
- full-stack data-flow

- Analog (Angular) 

"Transparent" Server-Side Actions

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

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

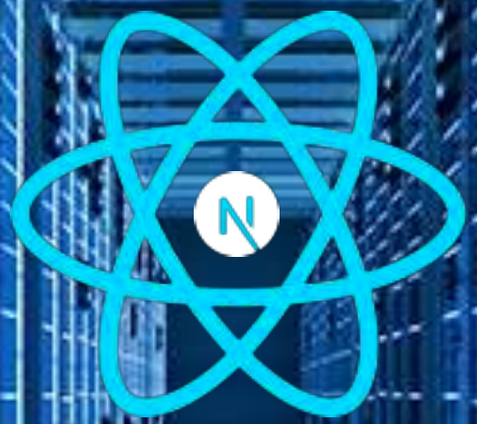
qwik Qwik City
https://qwik.builder.io/docs/route-loader/
https://qwik.builder.io/docs/action/
https://qwik.builder.io/docs/server$/

Similar RCP concepts:

Hilla
https://hilla.dev/

https://trpc.io/

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



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!

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

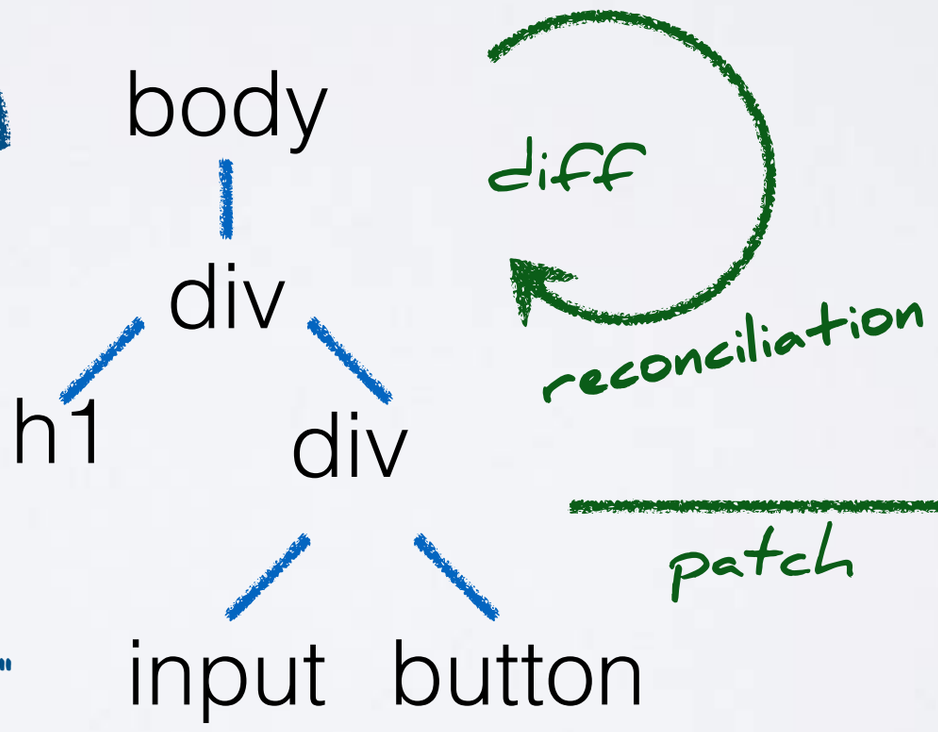
render instructions running on the client

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

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

2. send "RSC Payload" to the client

3. execute on the client




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

Server Component

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

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 | 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

React Client Components

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

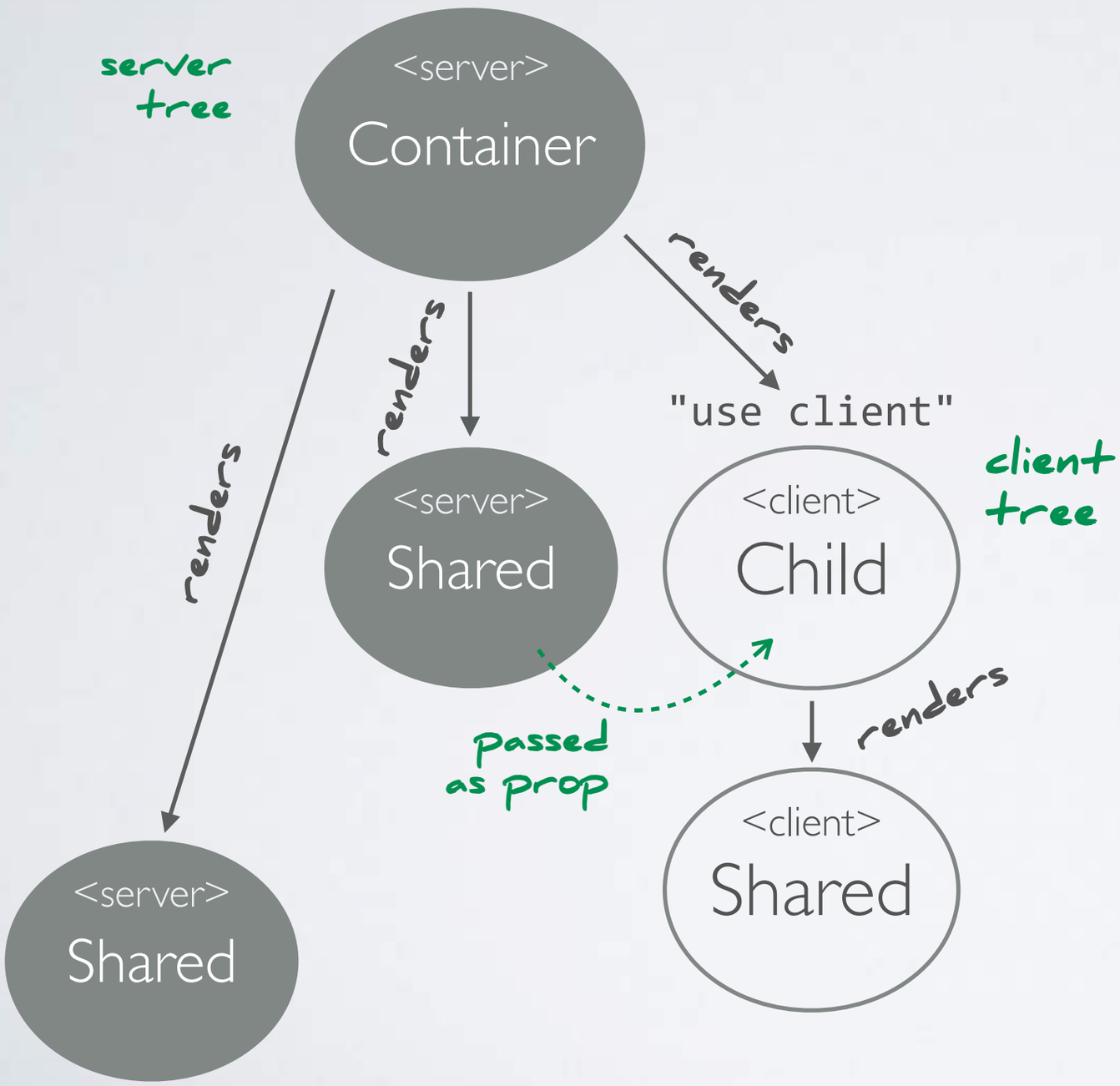
```
"use client"
export function Clock() {
  const [time, setTime] = useState(new Date())

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

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

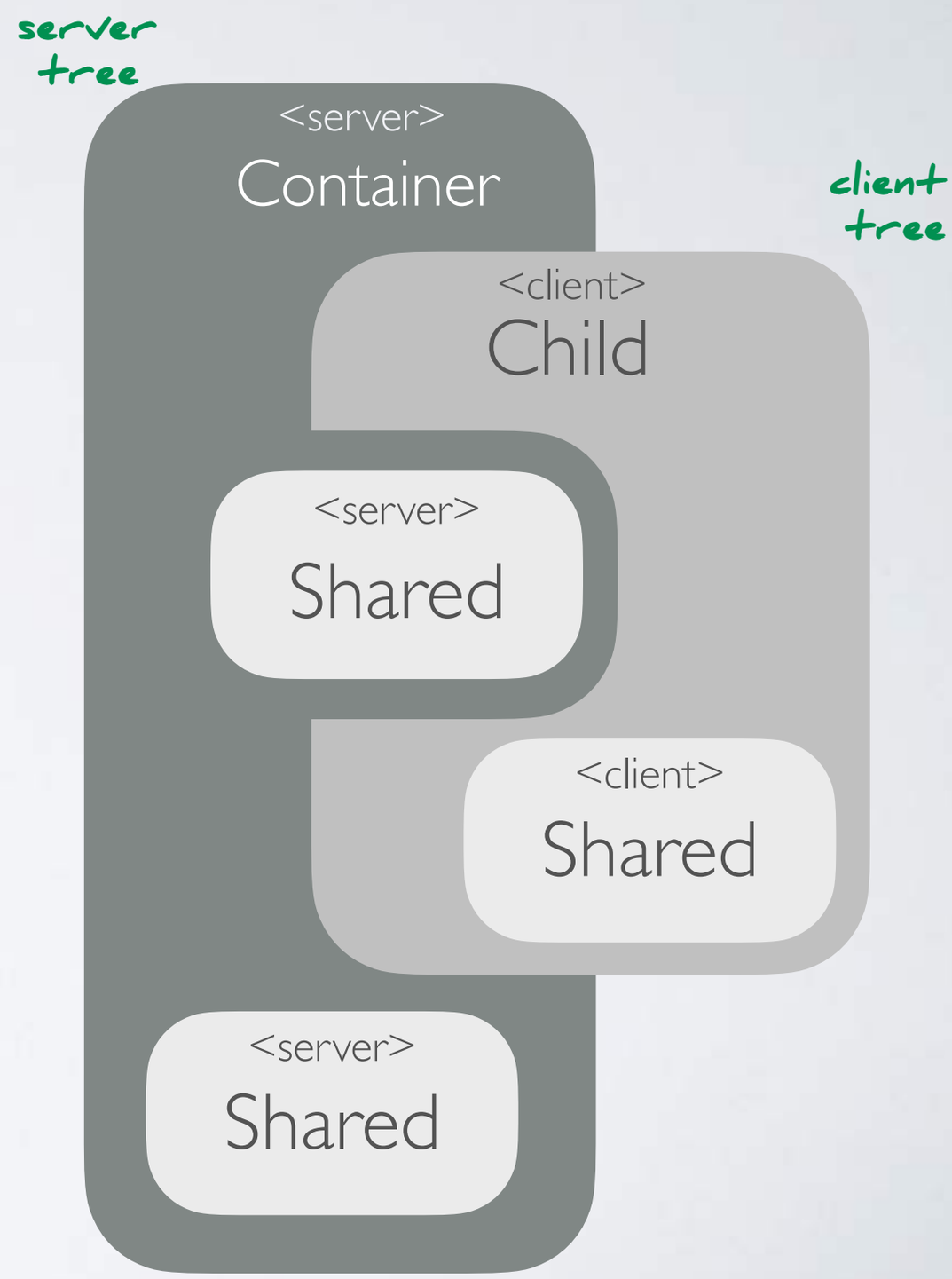
Client Components are "opt in".
Per default a component is a Server Component.

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!

danabra.mov
@dan_abramov

never write another API

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

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

Browser

Network

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

RSC Payload sent to the browser

rendering

first scenario: server component calling server function

React Client Runtime

(virtual dom tree)

RPC endpoint

API call

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

Server

JavaScript bundle loaded by the browser

JS

second scenario: client component calling server function

rendering

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

React Server Components

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

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 programmig model and
no need for a REST API.

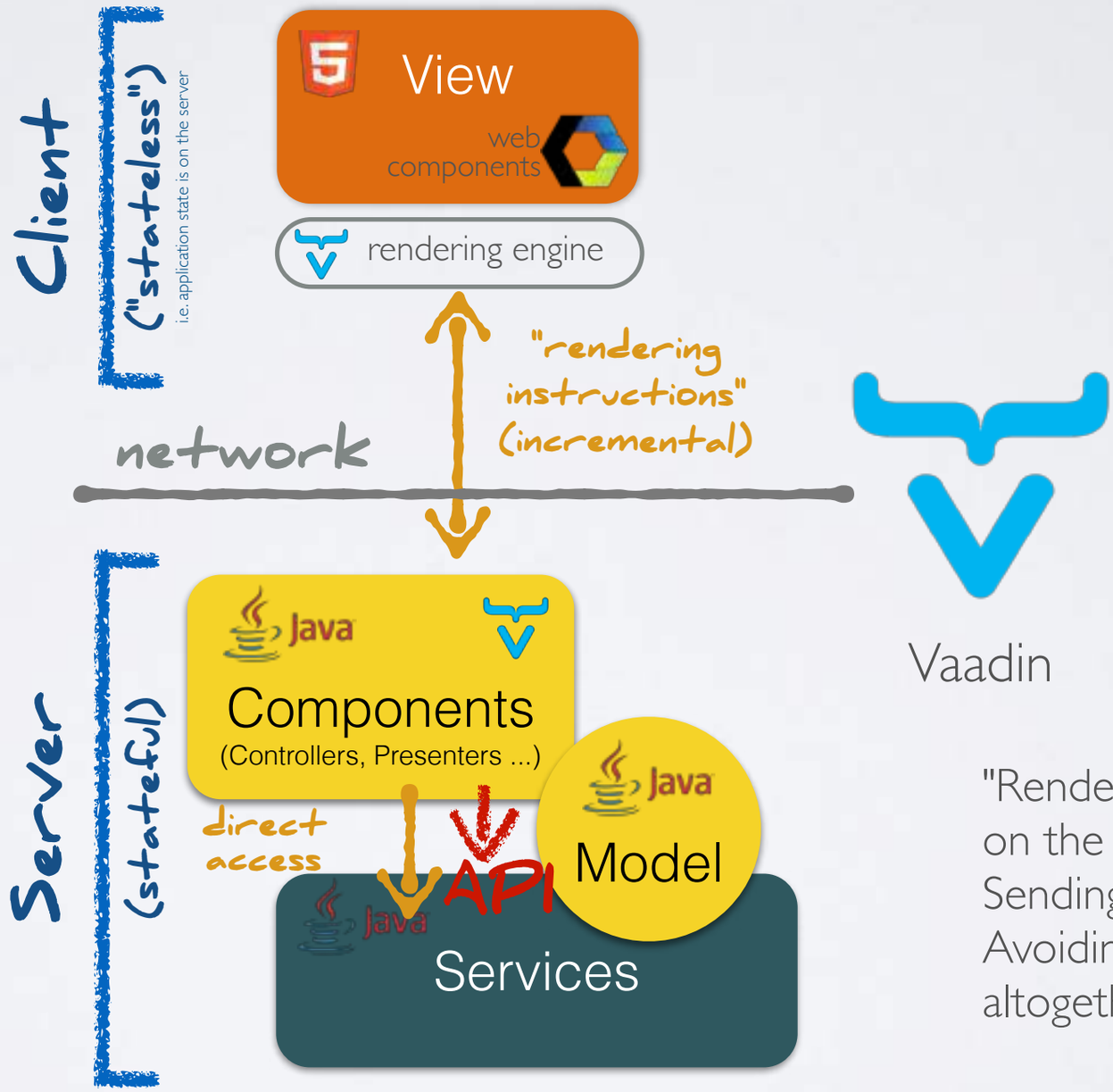
Demos:

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"



"Rendering" always happens on the server.
Sending UI-diffs to the client.
Avoiding Hydration altogether.

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 ...

Conclusion

Optimized rendering strategies require the frontend framework to run on the server *and* the client.

Once a frontend framework runs on the server, it is no longer constrained by the traditional server-client separation of the traditional SPA architecture.

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

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

The pendulum is swinging back to the server ... but modern full-stack development is very different to web development from 15 years ago.

Frontend technologies have a (heavy) influence on application architecture.

Thank you!

Slides & Code: https://github.com/jbandi/modern-web-cudos

Questions? Discussions ...

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

The JavaScript Dependency "Situation"

```
>npx create-react-app react-project

...

added 1909 packages from 732 contributors
found 0 vulnerabilities

> du -hs react-project/node_modules/
252M    react-project/node_modules/
```

```
>ng new angular-project

...

added 1600 packages from 1278 contributors
found 0 vulnerabilities

> du -hs angular-project/node_modules/
523M    angular/node_modules/
```

```
> vue create vue-project

...

added 1324 packages from 987 contributors
found 0 vulnerabilities

> du -hs vue-project/node_modules/
175M    vue-project/node_modules/
```

Maintenance

A "small" Angular project from 2018:

```
>cd angular-project-from-2018
>nvm use 8          <-- use old node version
>npm install

...

added 1288 packages from 1314 contributors
found 1396 vulnerabilities
(985 low, 18 moderate, 391 high, 2 critical)
```

Real-World example:
In-house React component library, 2 years untouched:

```
>npm audit

...

found 165426 vulnerabilities (109880 low, 526 moderate, 55018 high, 2 critical) in 4149 scanned packages
  run `npm audit fix` to fix 164670 of them.
  678 vulnerabilities require semver-major dependency updates.
  78 vulnerabilities require manual review. See the full report for details.
```

However npm audit vulnerabilities might not be a good metric: https://overreacted.io/npm-audit-broken-by-design/

Architecture:

STAR WARS

THE RISE OF THE API

EPISODE IX

Asynchronity ...

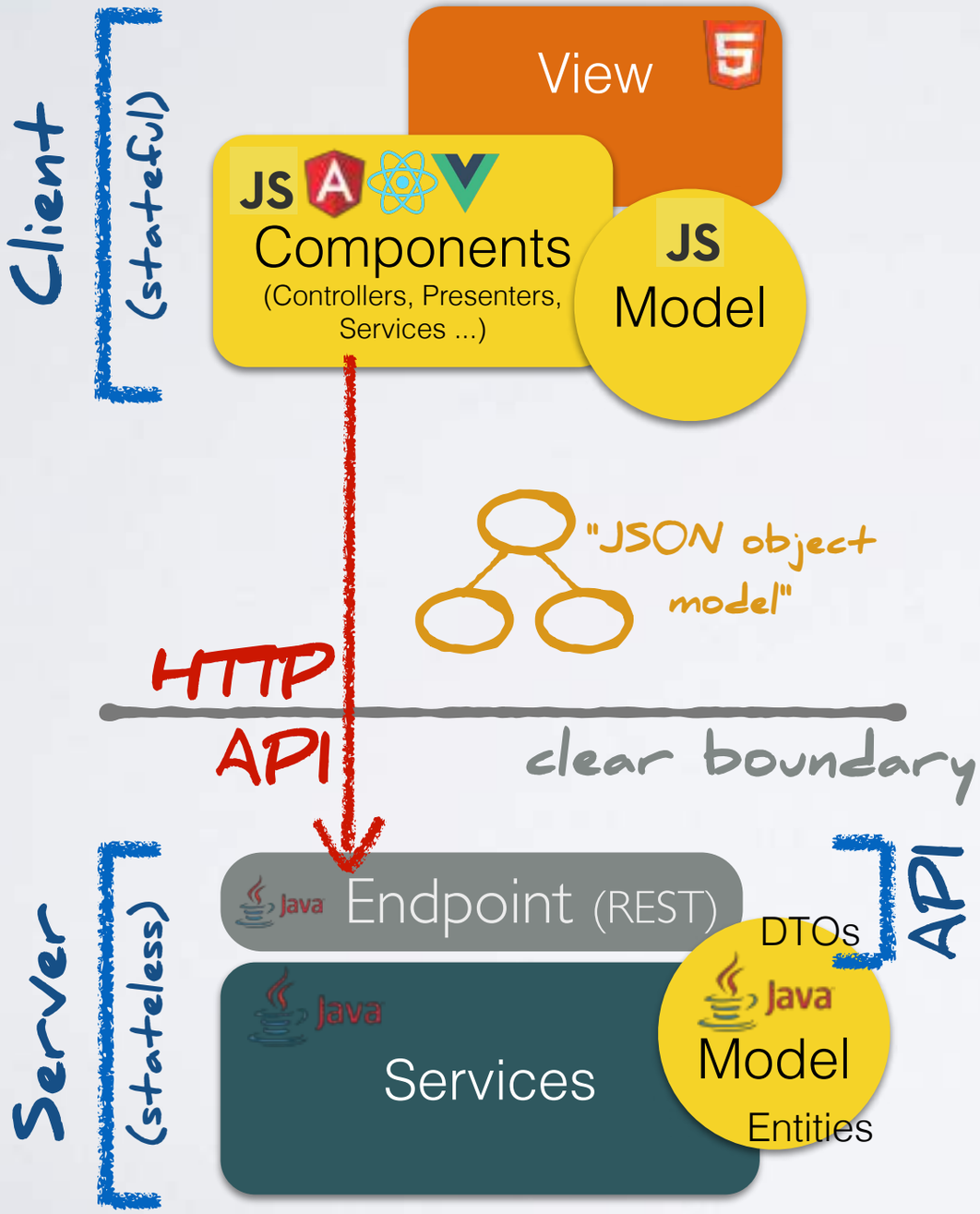
... there is no silver bullet ...

Asynchron programming is omnipresent in JavaScript, because of its singel-threaded nature and async APIs. Frontend frameworks can't hide this asynchronity.

The backend programming model is primarily synchronous. HTMX hides asynchronity behind declarative html enhancements.

The default programming model is synchronous. Asynchronous scenarios can be implemented with Java mechanisms.

The Role of the API



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

Symptoms:

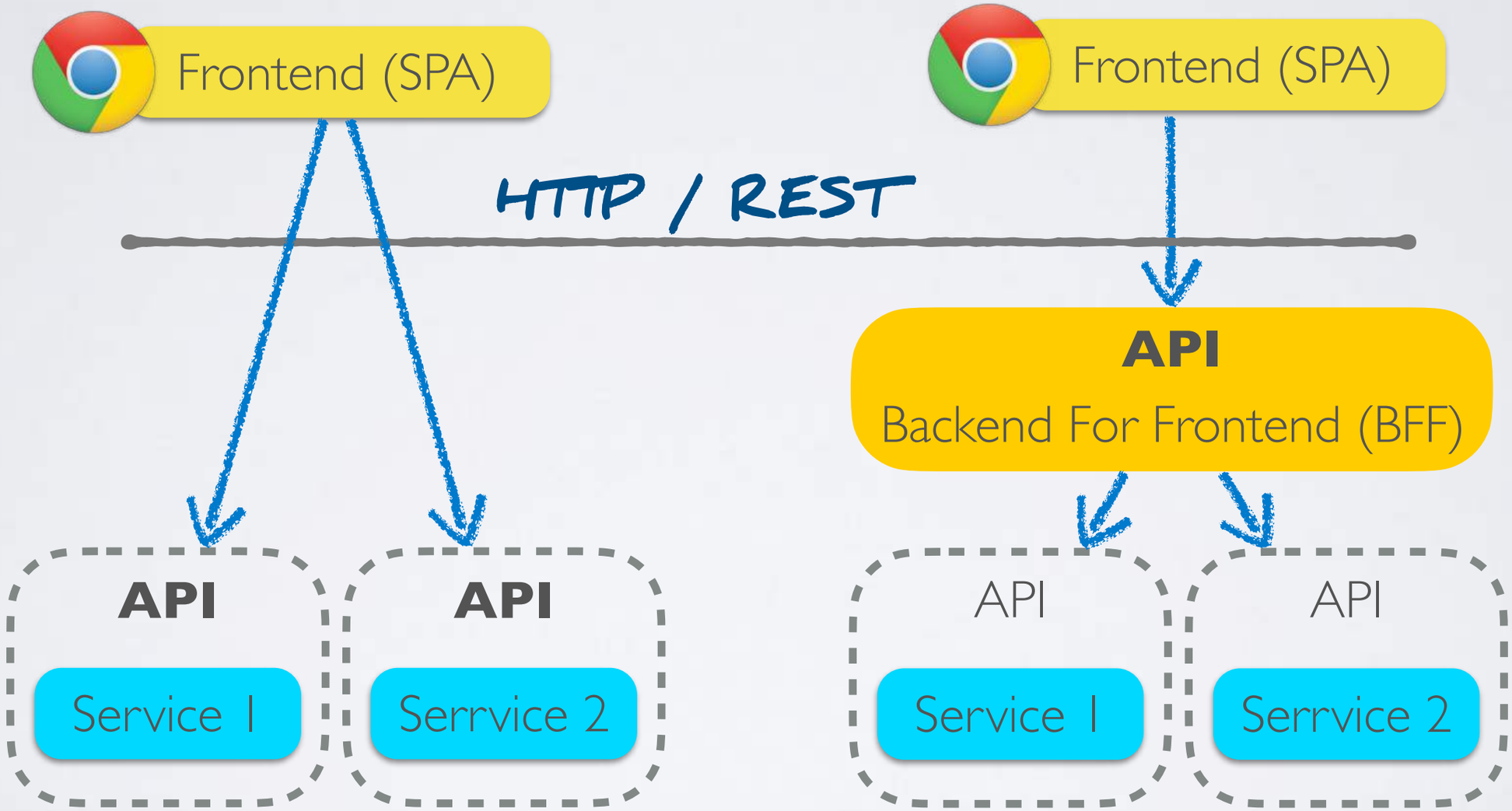
- "API-First" Design
- "The central role of API-Gateways" (the return of ESBs)

...

Creating a formalized API is a non-trivial effort: 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/

Few organizations are in the business of delivering APIs.

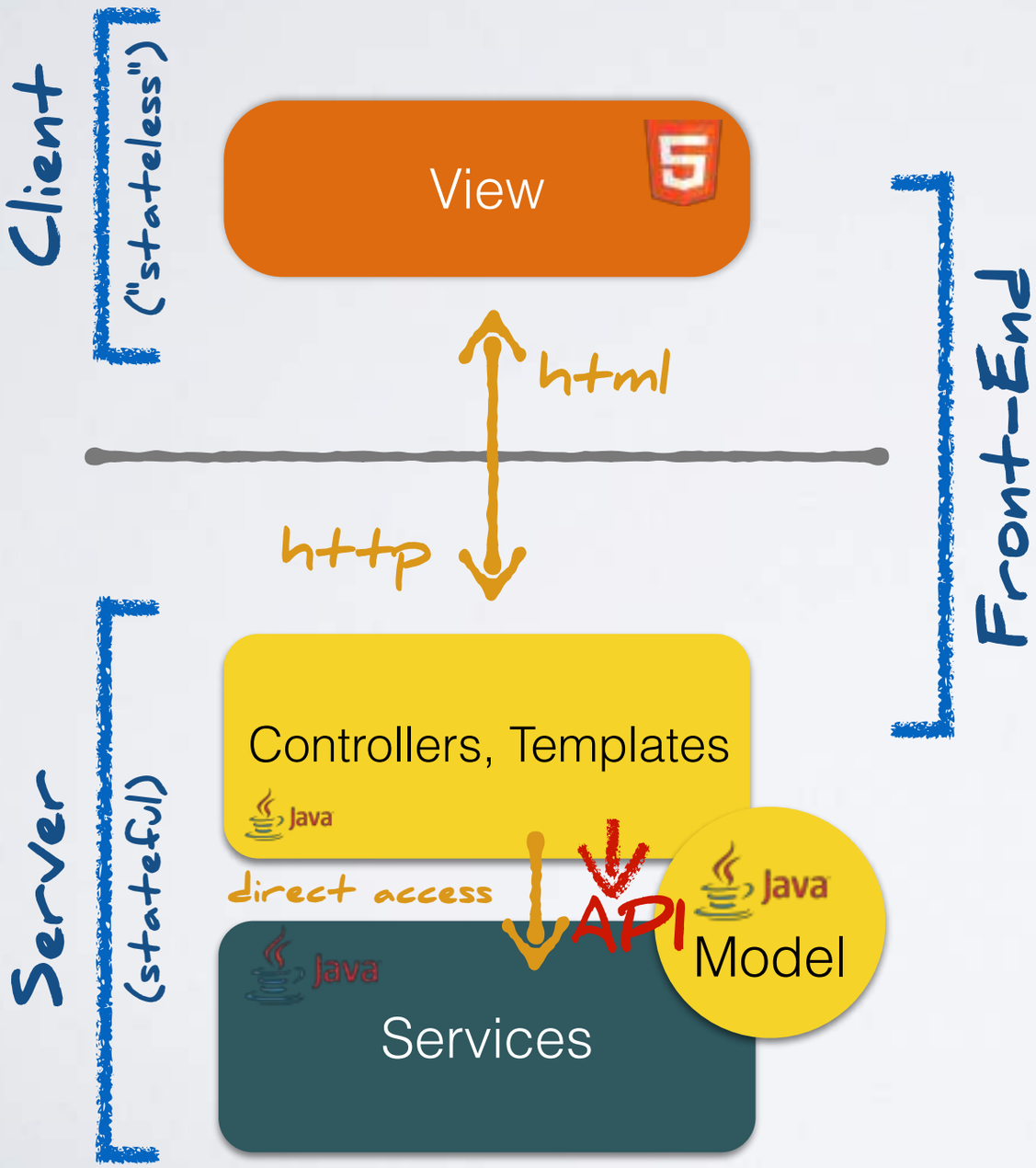
- Stefan Tilkov: Wait, what!? Our microservices have actual human users?
https://www.youtube.com/watch?v=pU1gXA0rfwc

... maybe you just want access to server-side data & functionality?

From the perspective of building a full-stack application (web-frontend with a server backend) the API is an implementation detail. What you care about, is accessing the server-side data and functionality ...

If your web-frontend is the only client to your API, a formalized REST API might just be accidental complexity.

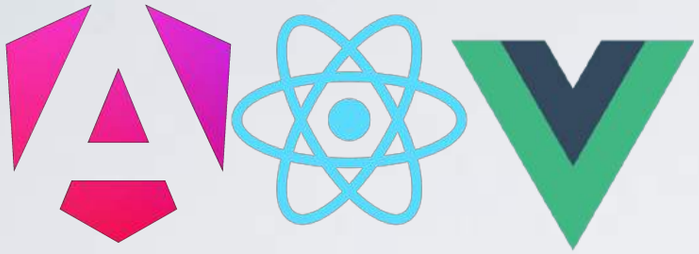
Traditional Web Applications



Traditional server-rendered applications also have an "API" to backend functionality.

But there is much less ceremony involved than with HTTP APIs ...

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.

STAR WARS

RETURN OF THE RPG

Remix

https://remix.run/docs/en/main/route/loader
https://remix.run/docs/en/main/guides/data-writes#remix-mutation-start-to-finish

NEXT.JS
React Server Components

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

Hilla
https://hilla.dev/

qwik Qwik City

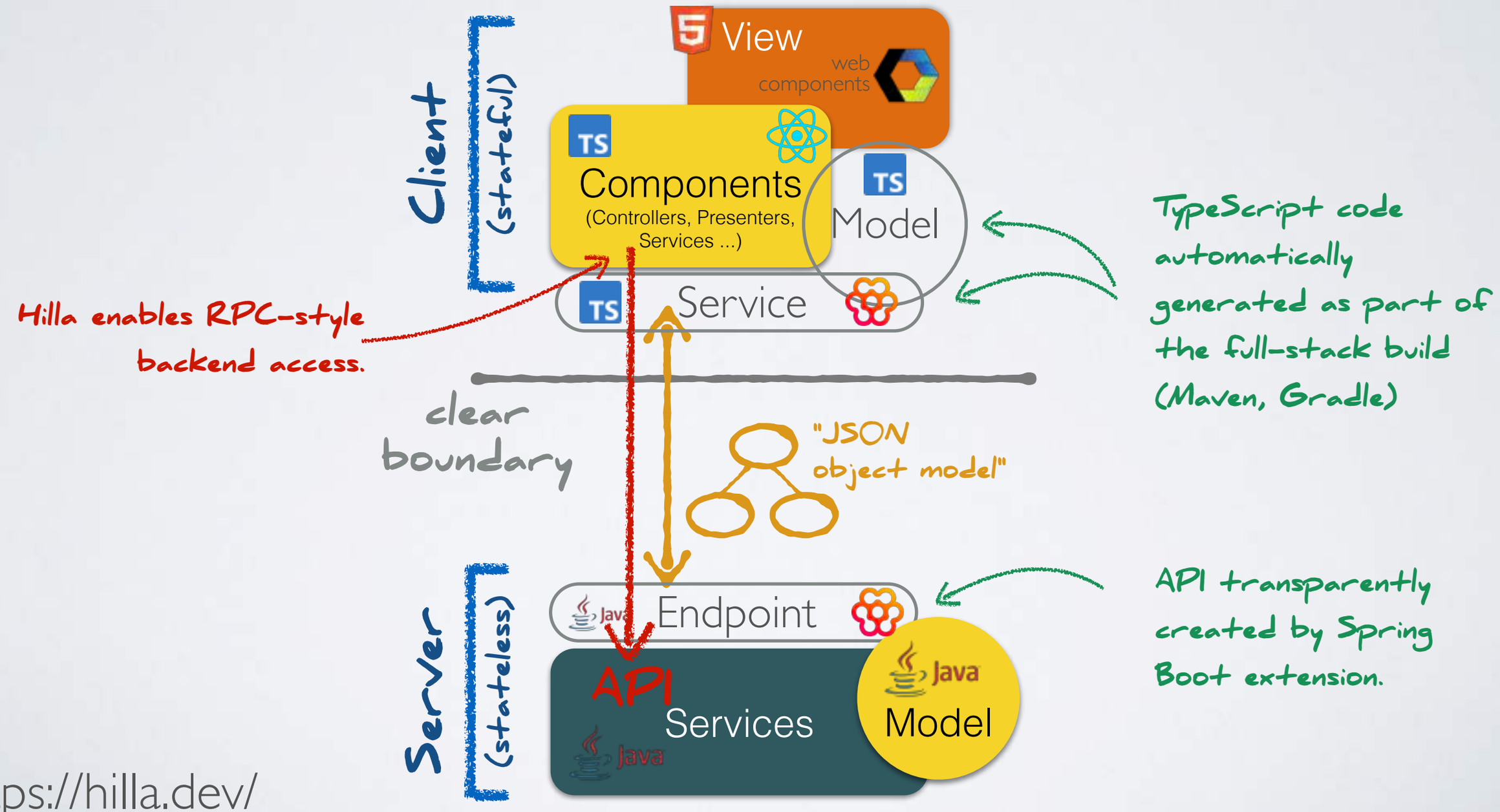
https://qwik.builder.io/docs/route-loader/
https://qwik.builder.io/docs/action/
https://qwik.builder.io/docs/server$/

Also:
https://trpc.io/

Hilla

(formerly known as Vaadin Fusion)

Hilla seamlessly connects a React SPA frontend with as Spring Boot backend.
It also exposes the Vaadin web-components as React components.



https://hilla.dev/

Why Full-Stack is the Future of Web Application Development? OVER

Thursday, September 19th 2024 - 7:00 AM (CST)

Share event

27:15 / 59:08