

Tema 4:

Arquitecturas de aplicaciones

web

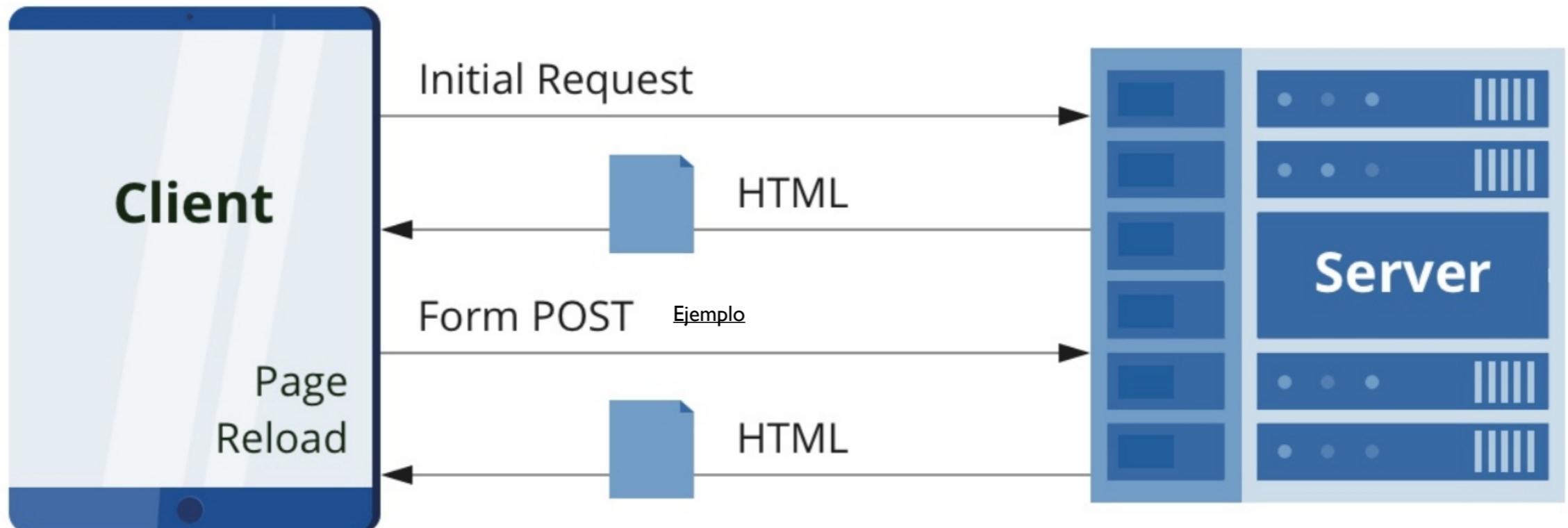
(Según cómo se reparten el trabajo el cliente y el servidor)

Tema 4: Arquitecturas de apps web

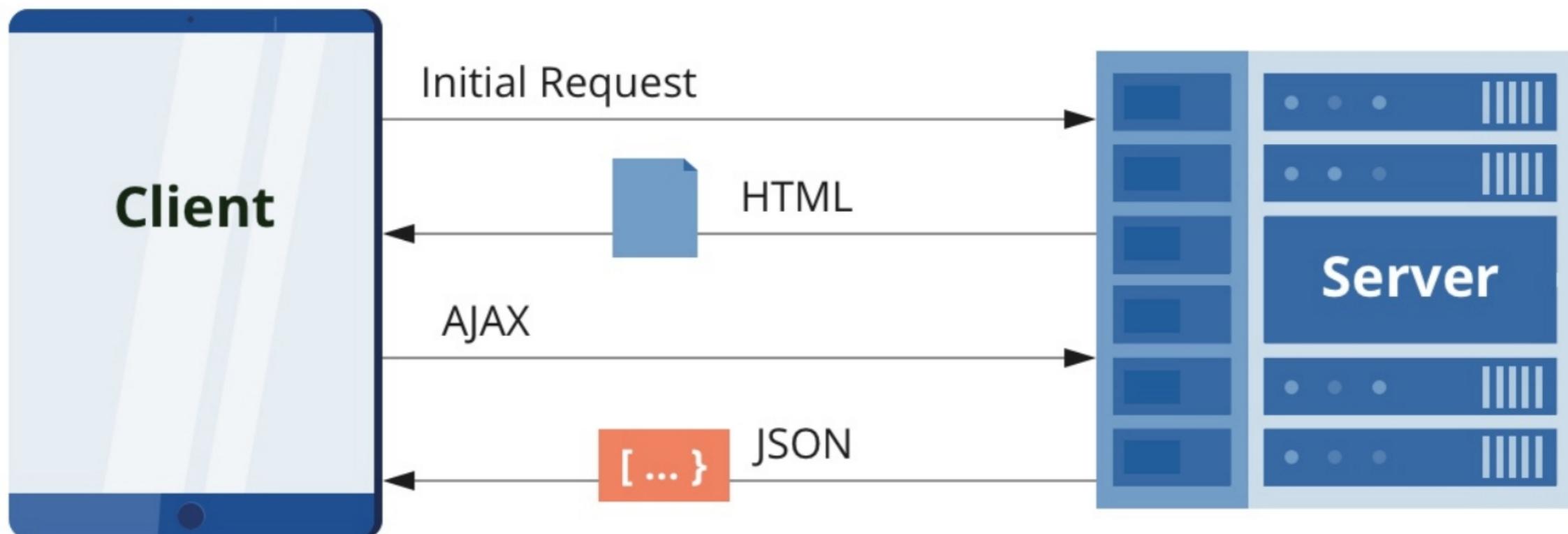
4.1

Introducción: MPAs vs SPAs

Multi Page Apps (MPA)



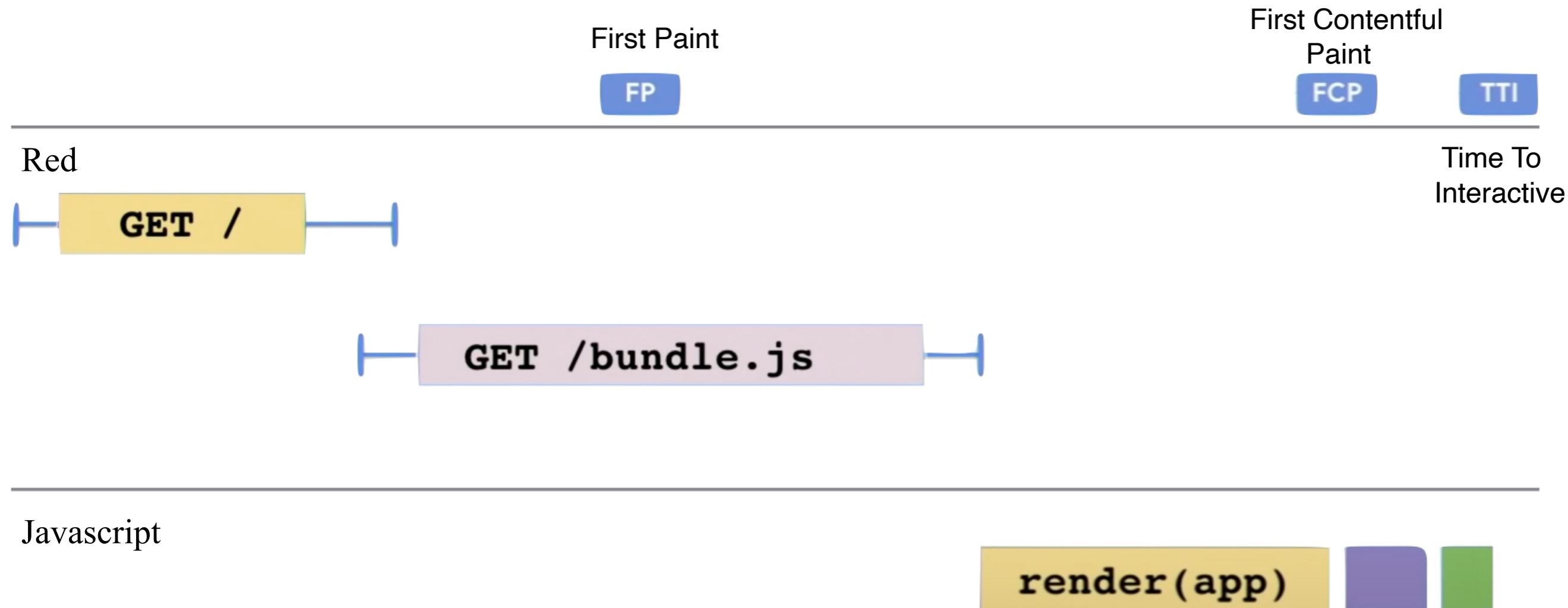
Single Page Apps (SPA)



Qué pasa en una SPA

```
<html>
  <head>
    <script defer src="bundle.js">
  </head>
<body>
  <div id="app">
    <!-- esto puede estar vacío o tener un placeholder o un spinner -->
  </div>
</body>
</html>
```

Qué pasa en una SPA



[Rendering on the Web: Performance Implications of Application Architecture \(Google I/O '19\)](#)

Explicación de todas estas siglas (¡y más!) en Core Web Vitals (*los “Web Vitals” o métricas web son indicadores de calidad de la experiencia de usuario*)

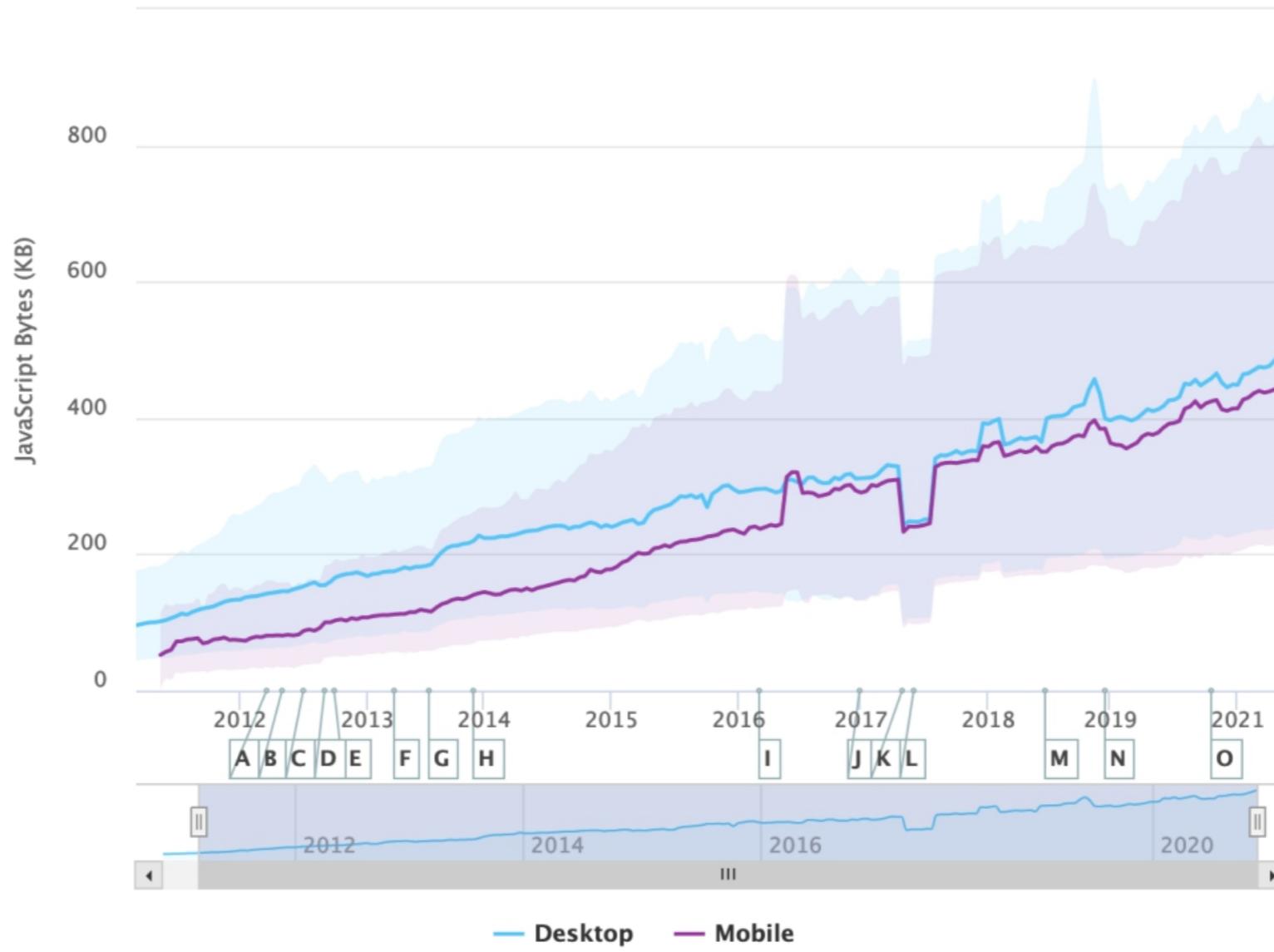
En 2010 Twitter cambió su arquitectura frontend a una SPA, ***y no te imaginas lo que sucedió después...***

[...] Descubrimos que el análisis y la ejecución sin procesar de JavaScript provocaban valores atípicos masivos en la velocidad de renderizado percibida. **En nuestra arquitectura totalmente del lado del cliente, no verá nada hasta que nuestro JavaScript se descargue y ejecute.** El problema se agrava aún más si no tienes una máquina potente o si estás ejecutando un navegador antiguo. **La conclusión es que una arquitectura del lado del cliente conduce a un rendimiento más lento.** [...]

"[Improving performance on Twitter.com](#)" , del *Blog de engineering de Twitter*

A pesar de que la potencia de los dispositivos actuales es mucho mayor que la de los de 2010, también ha crecido el tamaño de las apps JS

<https://httparchive.org/reports/page-weight#bytesJs>



¿Hay que volver entonces a las MPA?

¿No habrá alguna arquitectura que combine el rendimiento en *rendering* (FCP) de las MPA con la interactividad que ofrecen las SPA?

Tema 4: Arquitecturas de apps web

4.2

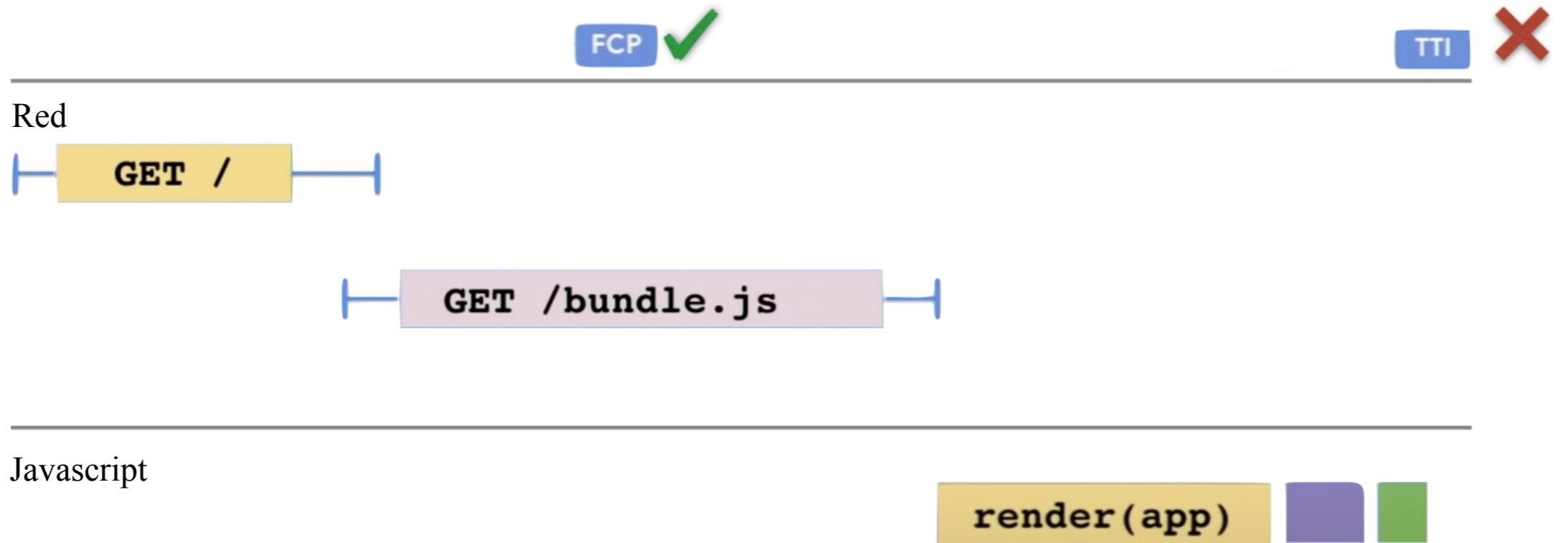
Apps universales o
isomórficas

Apps universales/isomórficas

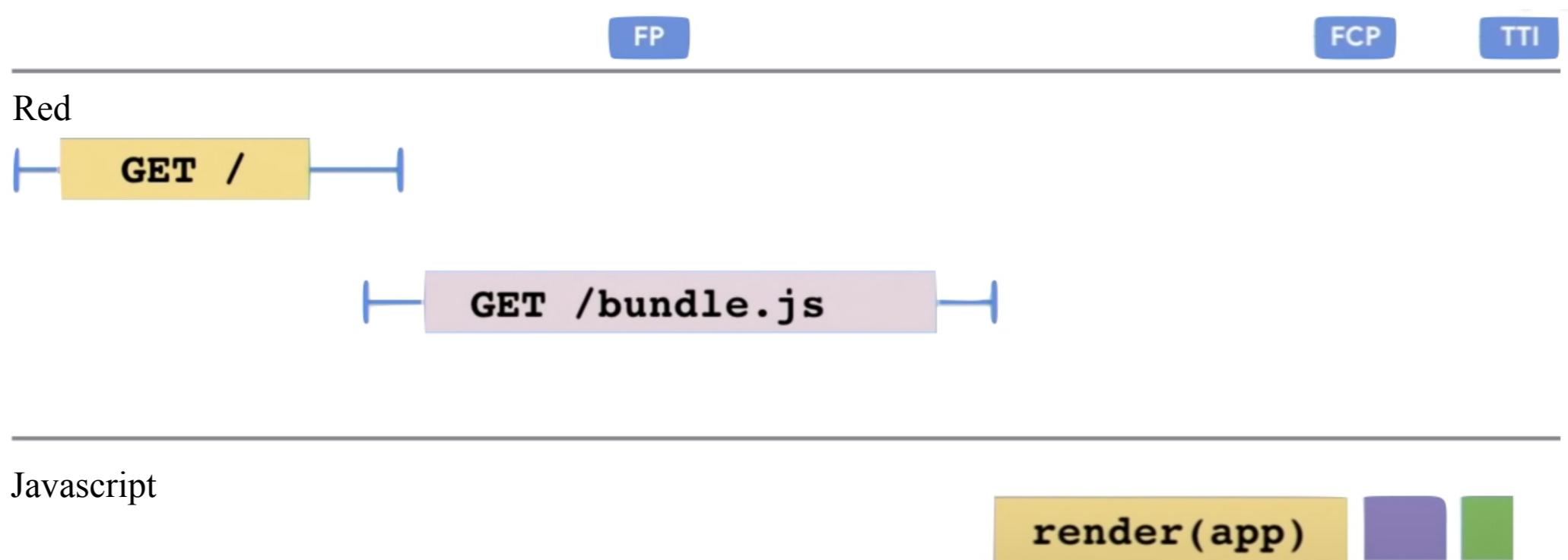
- También conocidas como Server Side Rendering (SSR)
- El HTML se genera en el servidor y se envía al cliente (== MPA)
- Una vez renderizado en el cliente, los componentes se “hidratan” con JS y se convierten en interactivos
- A partir de aquí todo es como en una SPA

```
<head>
  <title>ToDo</title>
  <link rel="stylesheet" href="/bundle.css"></script>
</head>
<body>
  <h1>To Do's</h1>
  <ul>
    <li><input type="checkbox"> Wash dishes</li>
    <li><input type="checkbox" checked> Mop floors</li>
    <li><input type="checkbox"> Fold laundry</li>
  </ul>
  <footer><input placeholder="Add To Do..."></footer>
<script>
  var DATA = {"todos":[
    {"text":"Wash dishes","checked":false,"created":1546464530049},
    {"text":"Mop floors","checked":true,"created":1546464571013},
    {"text":"Fold laundry","checked":false,"created":1546424241610}
  ]}
</script>
<script src="/bundle.js"></script>
</body>
```

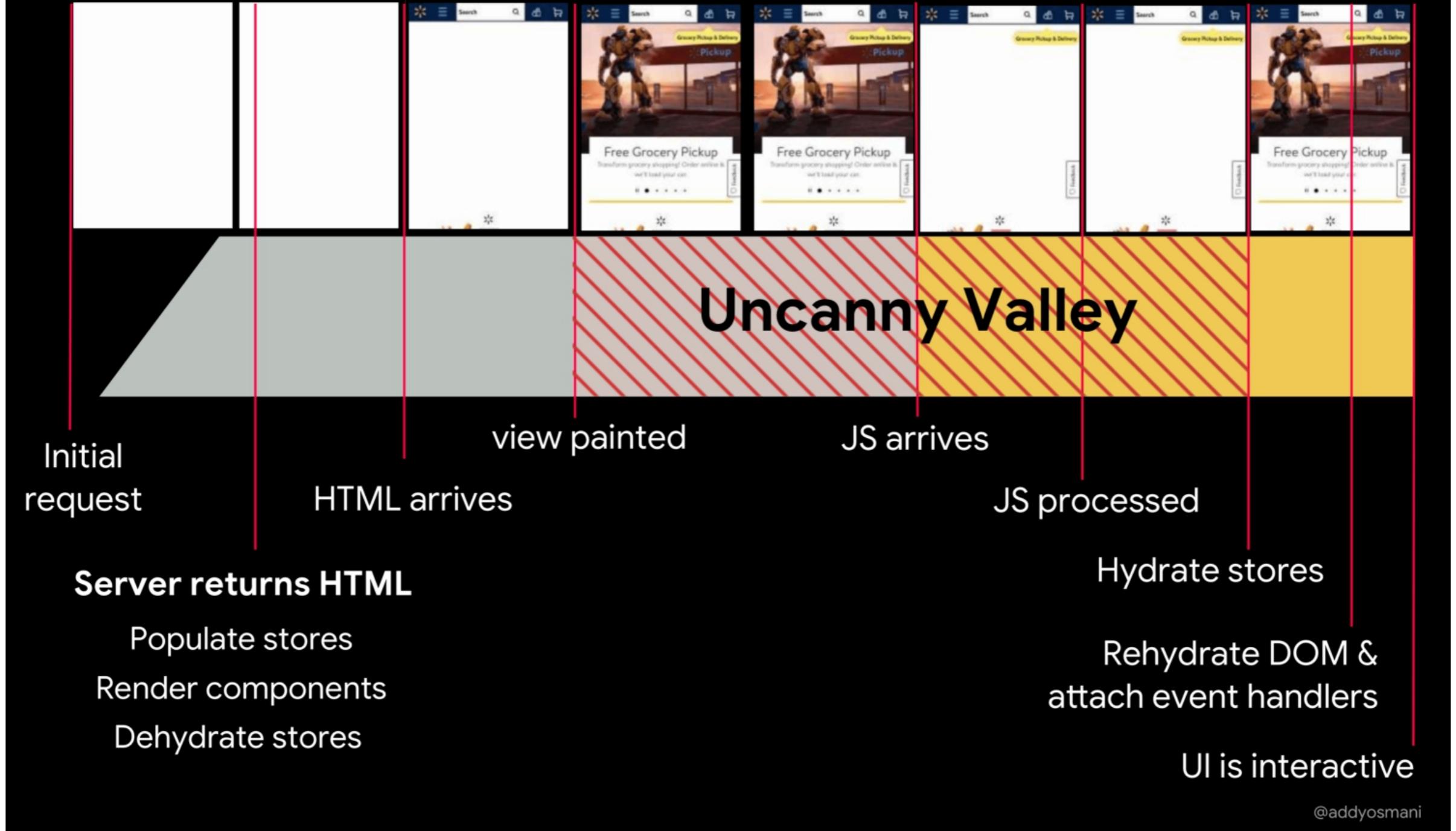
App universal



SPA



Rehydration



Addy Osmani, the cost of client side rehydration

@addyosmani

¿Y cómo se implementa esto?

- Muchos *frameworks* ofrecen soporte, por ejemplo Vue

server.js: código ejecutado por el servidor

```
import express from 'express';
import { renderToString } from 'vue/server-renderer';
import { createApp } from './app.js';

const server = express();

server.get('/', async (req, res) => {
  const app = createApp();
  var html = await renderToString(app)
  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR Example</title>
        //... omitidas algunas líneas por brevedad
        </script>
        <script type="module" src="/client.js"></script>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
  `);
});

server.use(express.static('.'));

server.listen(3000, () => {
  console.log('ready');
});
```

app.js: código ejecutado por el servidor y por el cliente

```
import { createSSRApp } from 'vue'

export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

Toma una app Vue y devuelve el HTML de la app renderizada en forma asíncrona

client.js: código ejecutado por el cliente

```
import { createApp } from './app.js'
createApp().mount('#app')
```

Para que el servidor sirva **client.js** al navegador cuando este se lo pida

De la web [Vue Server Side Rendering Guide](#)
[Ejemplo completo en StackBlitz](#)

Es todavía más complicado en una app real

- Con **Single File Components (SFC)** hay que generar dos versiones de cada uno (cliente y servidor), ya que internamente los templates generan el HTML de modo distinto en cada caso (Virtual DOM vs concatenación de cadenas)
- Hay que gestionar el *routing*, los *stores* (estado centralizado) y las llamadas a APIs también de modo “universal” (distinto en cliente y servidor)
- Nos puede interesar tener algunas rutas que funcionen en modo SPA y otras en modo “universal”

Meta-Frameworks

- Implementan estas funcionalidades “*out of the box* ”



(Vue)

[Demo básica de Nuxt](#)



(React)



(Svelte)

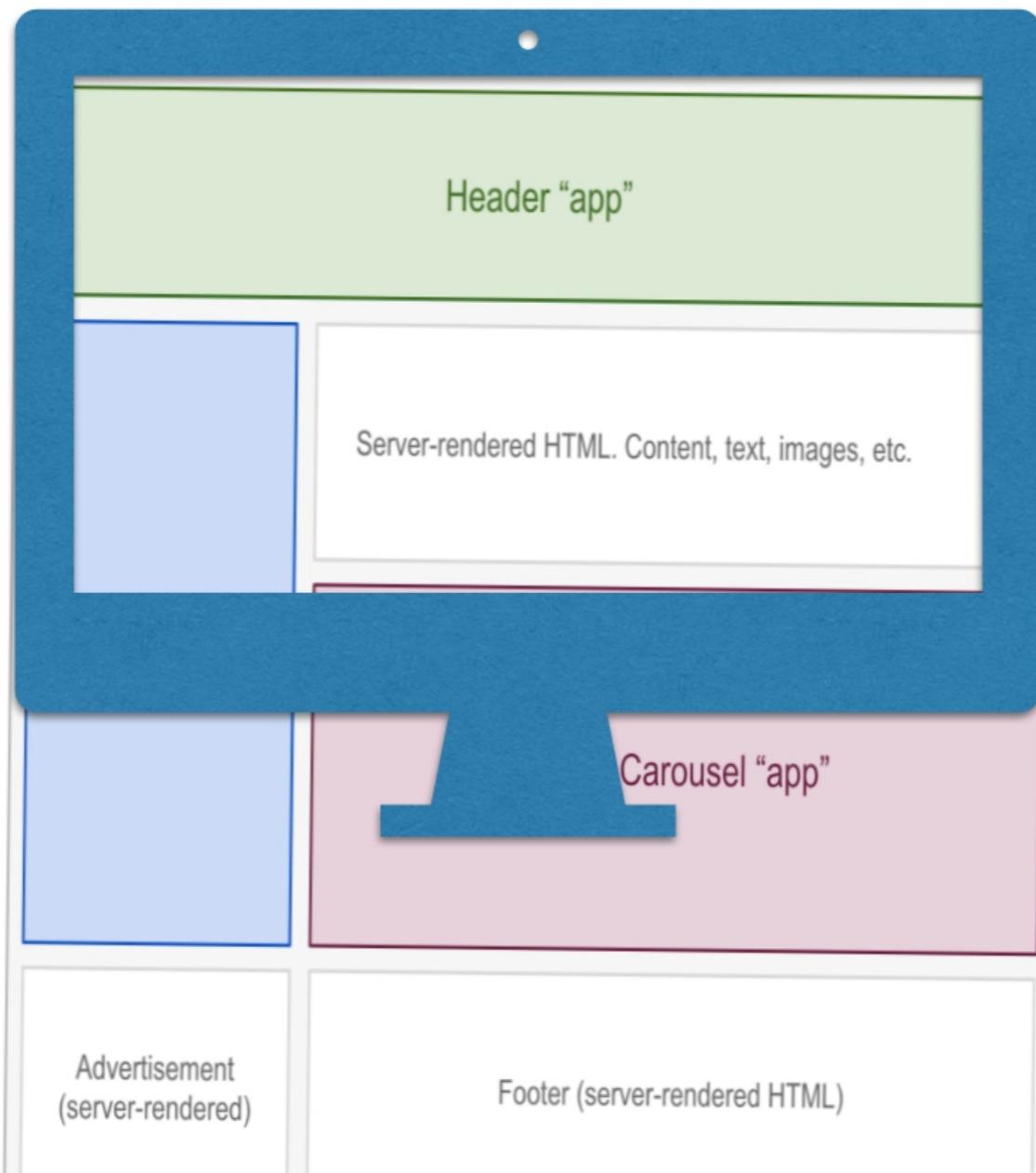
[Demo básica de SvelteKit](#)



(Angular)

Progressive Hydration

- Diferir la “hidratación” de ciertos componentes porque no son tan importantes o porque no son todavía visibles en la pantalla (están muy abajo en la página)



¿Y cómo se implementa esto?

Aunque p.ej. Vue no tiene soporte “por defecto”, hay *plugins* e implementaciones de “terceros”

```
< template>
  <div class= "IndexPage">
    <!-- ... -->
    <LazyHydrate when-visible>
      <ImageSlider/>
    </LazyHydrate>
    <!-- ... -->
  </div>
</template>

<script>
import LazyHydrate from 'vue-lazy-hydration';
export default {
  name: 'IndexPage',
  components: {
    LazyHydrate,
    ImageSlider: () => import( '../components/ImageSlider.vue' ),
  },
},
</script>
```

Otros frameworks

Algunos frameworks “alternativos” sí lo incluyen, p.ej. Marko, Astro,...

<https://docs.astro.build/core-concepts/component-hydration/>

```
...
// Example: hydrating a React component in the browser.
import MyReactComponent from './components/MyReactComponent.jsx';
...
<!-- "client:visible" means the component won't load any client-side
     JavaScript until it becomes visible in the user's browser. -->
<MyReactComponent client:visible />
```

<MyComponent client:load />

Hydrate the component on page load.



<https://astro.build/>



<https://markojs.com/>

<MyComponent client:idle />

Hydrate the component as soon as main thread is free (uses [requestIdleCallback\(\)](#)).



<https://qwik.builder.io/>

<MyComponent client:visible />

Hydrate the component as soon as the element enters the viewport (uses [IntersectionObserver](#)).

Useful for content lower down on the page.

<https://stackblitz.com/edit/qwik-todo-demo>

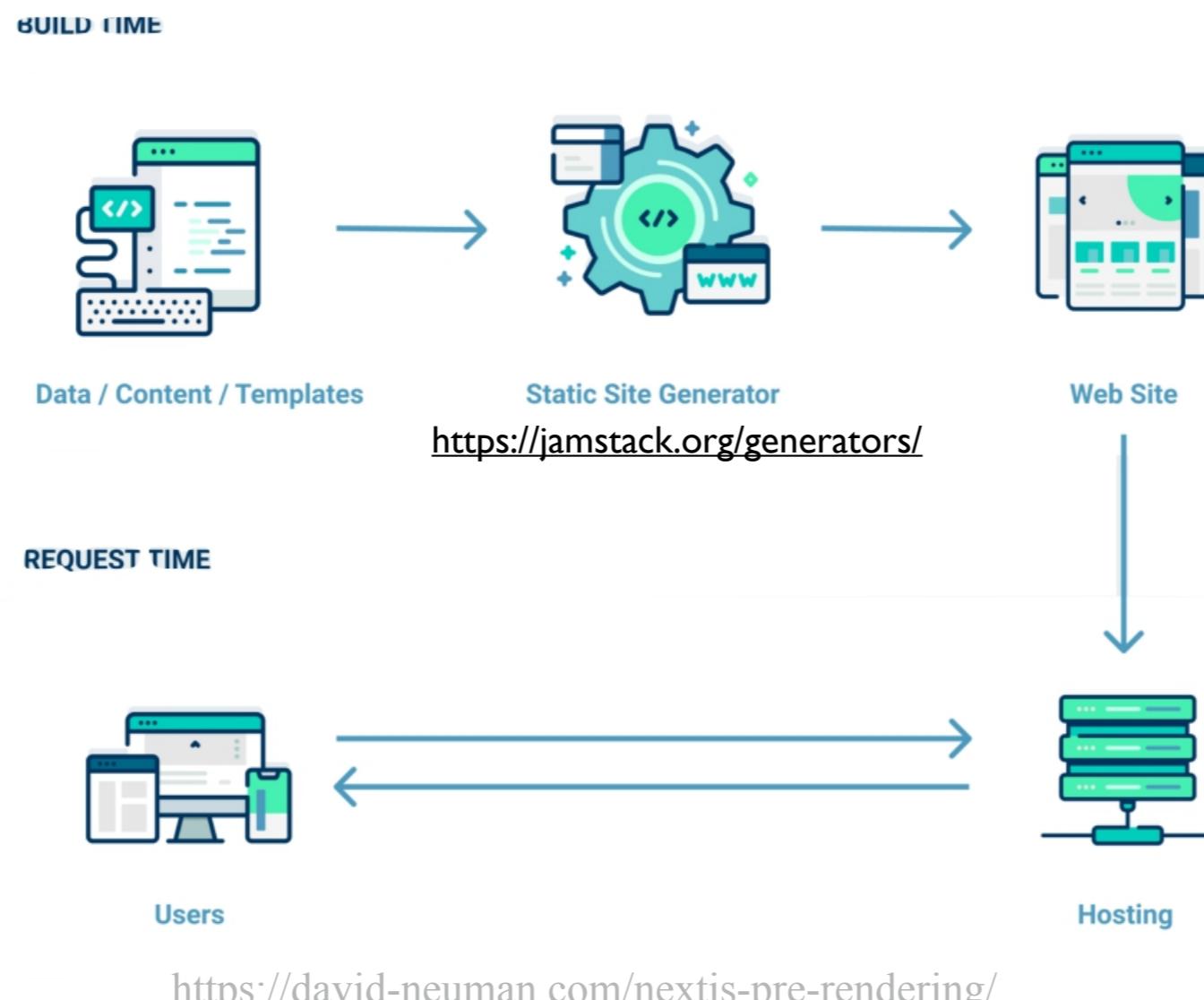
Tema 4: Arquitecturas de apps web

4.3

Prerendering/
Jamstack

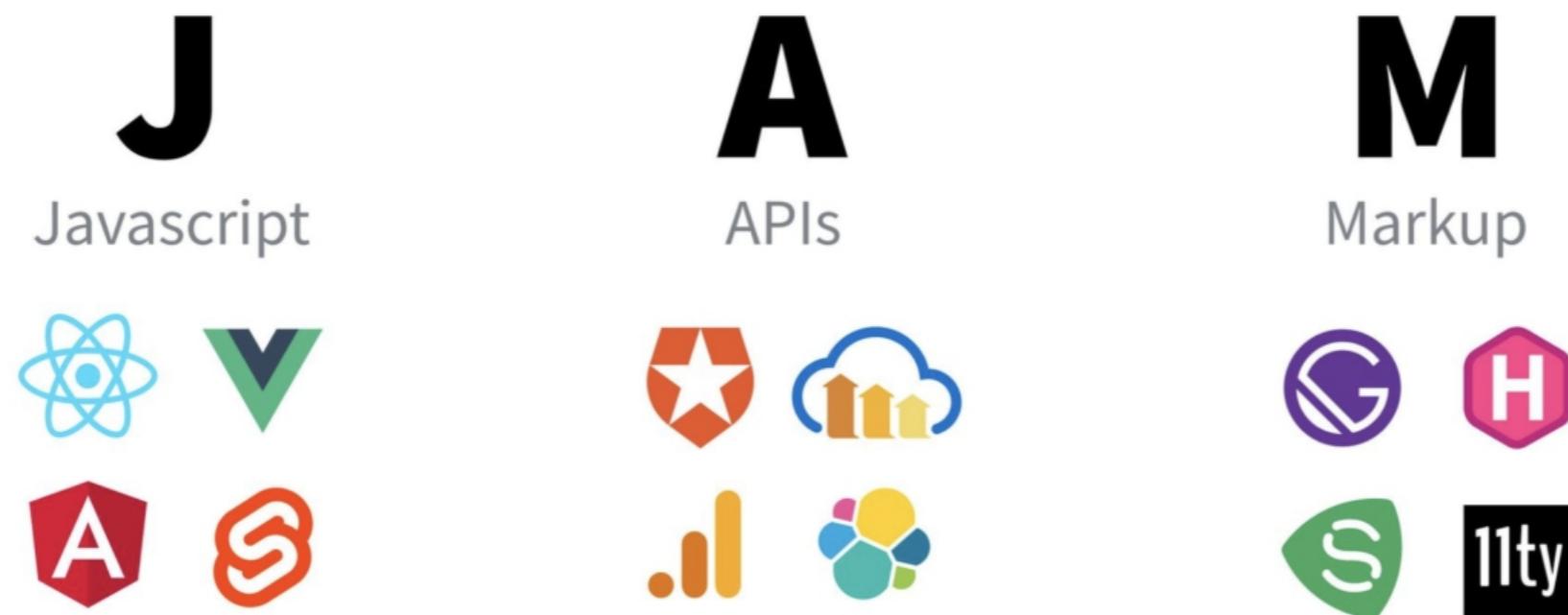
Prerendering

- Generar el HTML desde el servidor en una etapa previa de “build”, y no al acceder a la página
- Si os gustan las siglas, SSG (Server Side Generation) (vs. SSR)
- Mejora el FCP y la escalabilidad del servidor
- No es aplicable a todos los contenidos



JamStack

- Abreviatura de **J**avascript, **A**PIs y **M**arkup
- Principios básicos: *prerendering* + servicios (APIs externos)



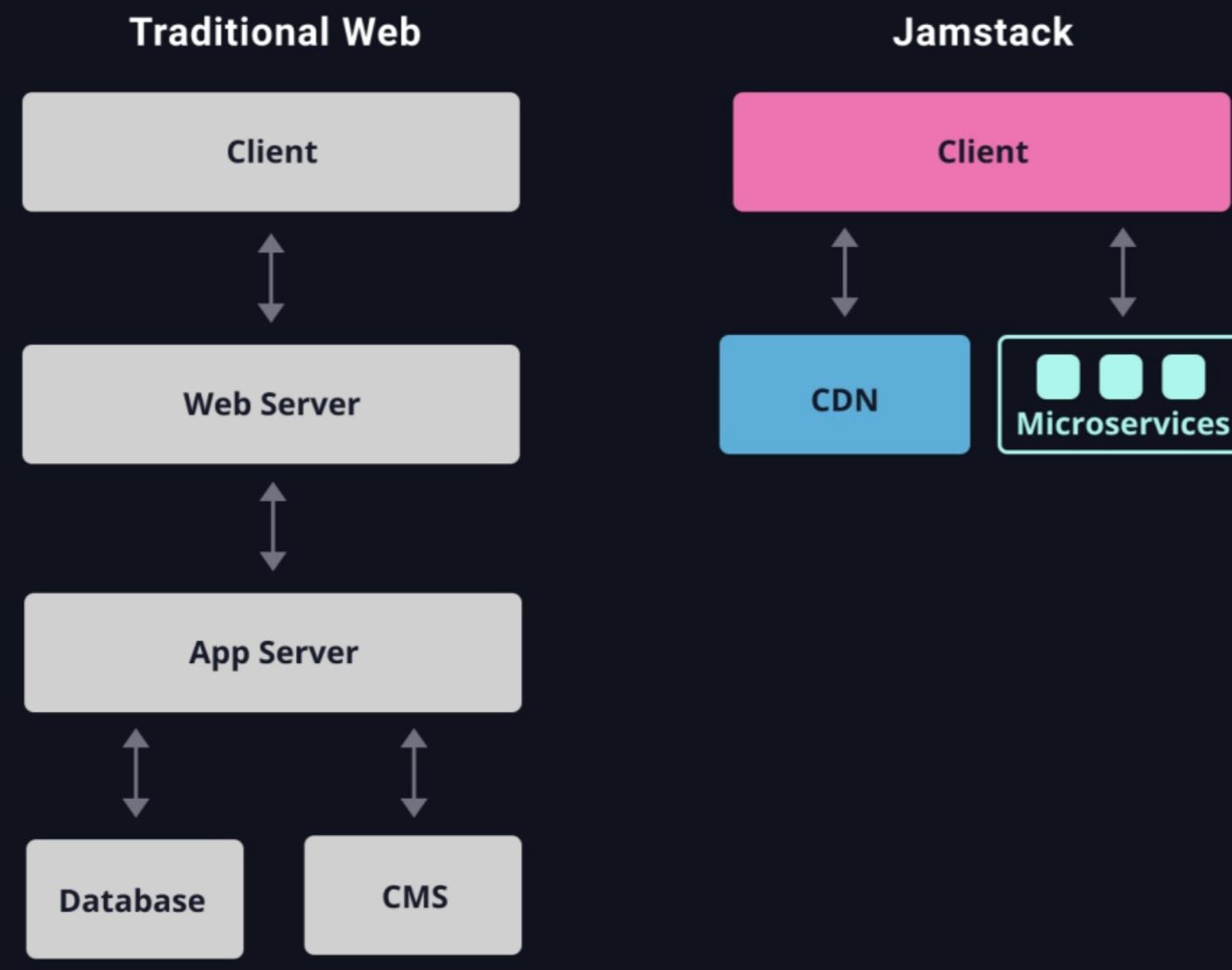
<https://jamstack.org/what-is-jamstack/>

JamStack vs. stack tradicional

The future is highly distributed

Jamstack is the new standard architecture for the web. Using Git workflows and modern build tools, pre-rendered content is served to a CDN and made dynamic through APIs and serverless functions.

Technologies in the stack include JavaScript frameworks, Static Site Generators, Headless CMSs, and CDNs.



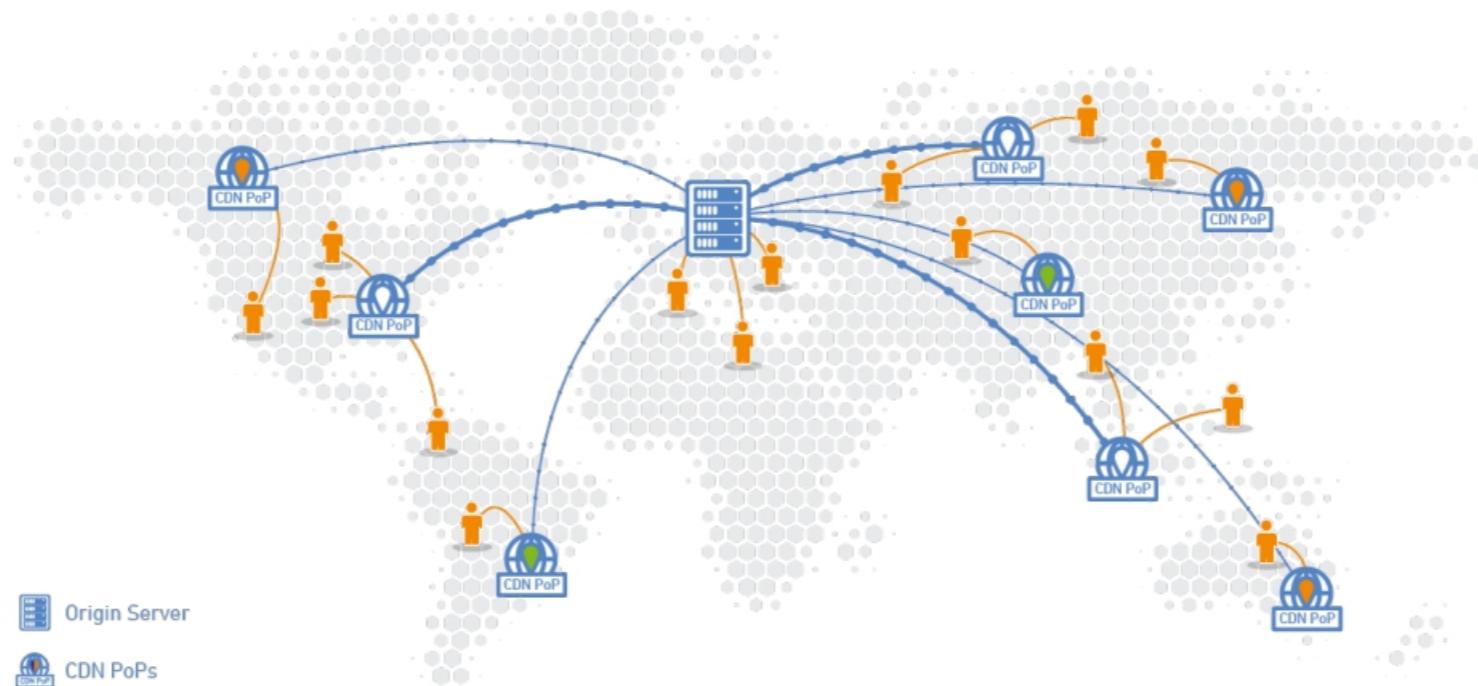
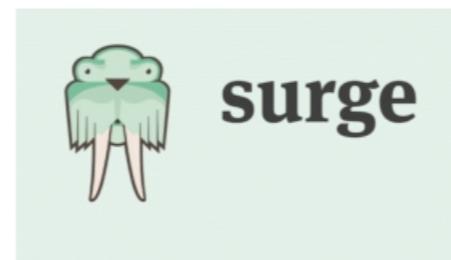
<https://jamstack.org/>

Despliegue

- Servidores “estáticos” que actúan como una CDN



netlify



Ejemplo: Google web.dev

The screenshot shows a browser window displaying the 'Learn PWA!' course on the <https://web.dev/learn/pwa/> page. The page features a sidebar on the left with various icons for connectivity, notifications, and download status. The main content area has a breadcrumb navigation path: web.dev > Learn > Learn PWA!. A search bar is at the top right. The main title 'Learn PWA' is displayed prominently. Below it, a description states: 'A course that breaks down every aspect of modern progressive web app development.' A 'On this page' section is visible. The main content area contains a large heading 'Welcome to Learn Progressive Web Apps!' followed by descriptive text about the course's purpose and content. The sidebar lists eight modules: 'Learn PWA', 'Progressive Web Apps', 'Getting started', 'Foundations', 'App design', 'Assets and data', 'Service workers', 'Caching', and 'Serving', each with a 'COMING SOON' note.

Module	Status
Learn PWA	000
Progressive Web Apps	001
Getting started	002
Foundations	003
App design	004
Assets and data	005
Service workers	006
Caching	007
Serving	008

Welcome to Learn Progressive Web Apps!

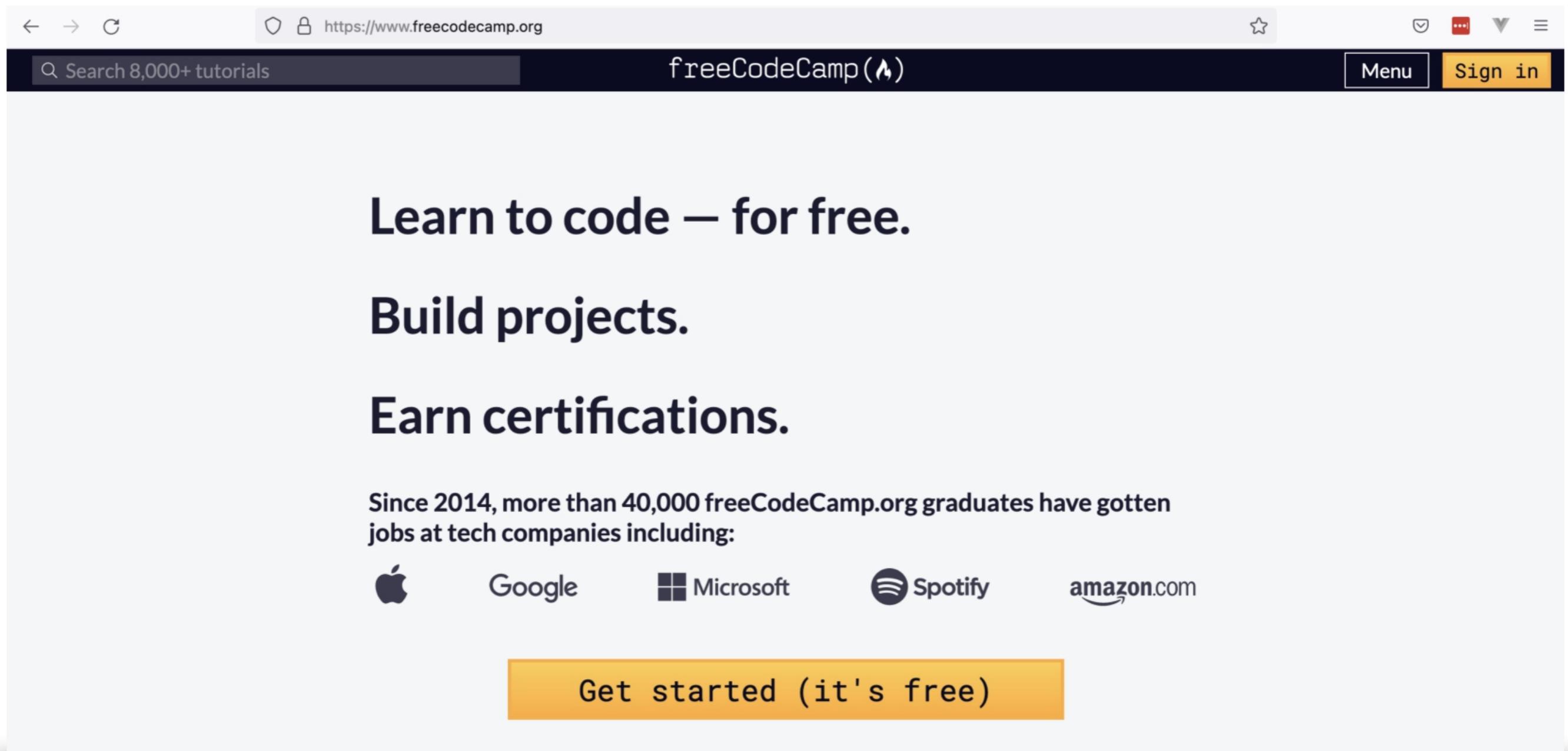
Welcome to Learn Progressive Web Apps!

This course covers the fundamentals of Progressive Web App development into easy-to-understand pieces. Over the following modules, you'll learn what a Progressive Web App is, how to create one or upgrade your existing web content, and how to add all the pieces for an offline, installable app. Use the menu pane by the "Learn PWA" logo to navigate the modules.

You'll learn PWA fundamentals like the Web App Manifest, Service Workers, how to design with an app in mind, what's different from a classic web app, how to use other tools to test and debug your

<https://github.com/GoogleChrome/web.dev>

Ejemplo: Freecodecamp



The screenshot shows the homepage of freeCodeCamp.org. At the top, there is a navigation bar with icons for back, forward, search, and user account. The URL https://www.freecodecamp.org is displayed. Below the navigation bar, there is a search bar with the placeholder "Search 8,000+ tutorials". The main heading "freeCodeCamp(🔥)" is centered above three large, bold, dark text blocks: "Learn to code – for free.", "Build projects.", and "Earn certifications.". Below these, a paragraph states: "Since 2014, more than 40,000 freeCodeCamp.org graduates have gotten jobs at tech companies including:" followed by logos for Apple, Google, Microsoft, Spotify, and Amazon.com. At the bottom, a prominent yellow button with the text "Get started (it's free)" is visible.

Learn to code – for free.

Build projects.

Earn certifications.

Since 2014, more than 40,000 freeCodeCamp.org graduates have gotten jobs at tech companies including:

Apple Google Microsoft Spotify amazon.com

Get started (it's free)



How freeCodeCamp Serves Millions of
Learners Using the JAMstack

Version 1: duct tape



Version 2

Our own Node servers

Our own MongoDB cluster

Our own ElasticSearch

It was expensive as heck. A lot of things could go wrong

Version 3

A single Node/Loopback webserver with MLab cluster

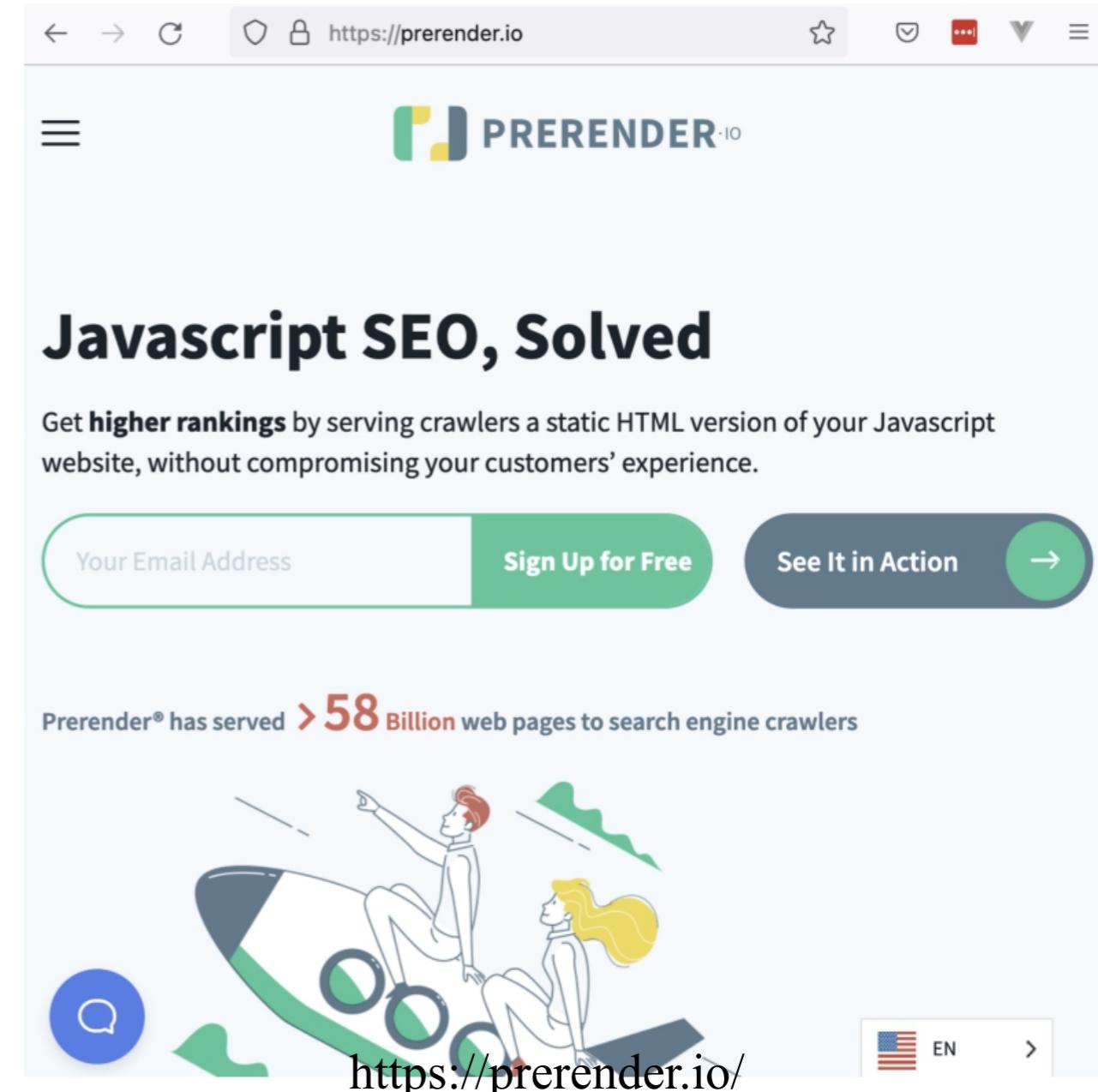
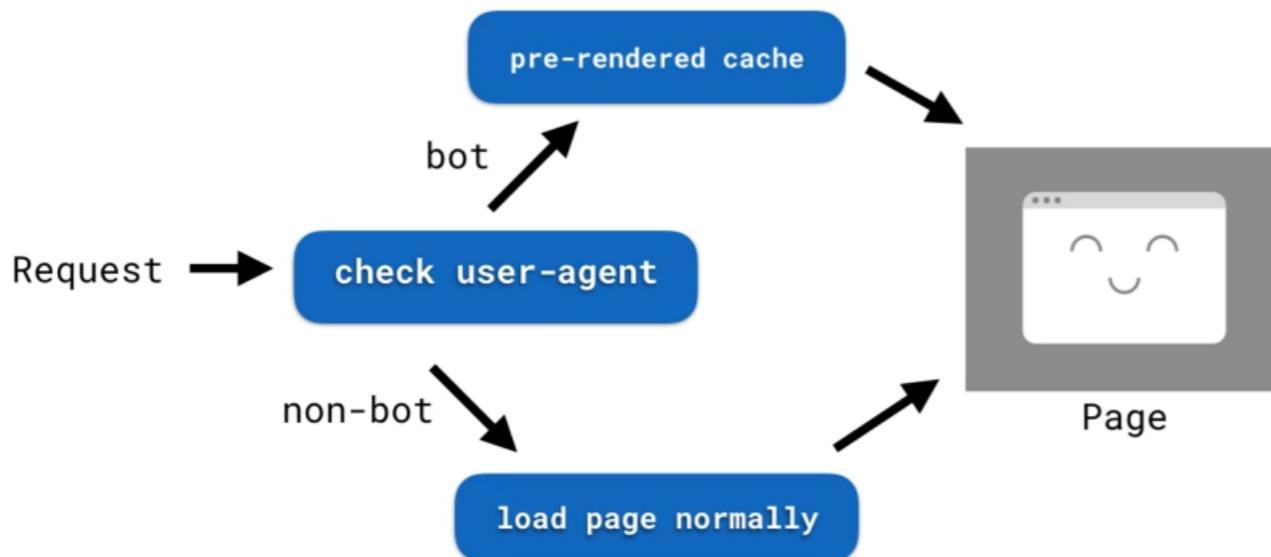
Algolia for search

Auth0 for auth

6,000+ static pages served through Netlify

Prerendering y SEO

- Si lo usamos solo por motivos de SEO se hará prerendering si el cliente que está accediendo es un bot como Google



Algunas referencias

-  [Rendering on the Web: Performance Implications of Application Architecture \(Google I/O19\)](#)
-  [Rendering on the Web , Jason Miller & Addy Osmani, web.dev](#)
-  Javascript design patterns, [rendering patterns](#), Lydia Halle, Addy Osmani y otros
-