Tema 3, parte IV: Patrones de renderizado en aplicaciones web

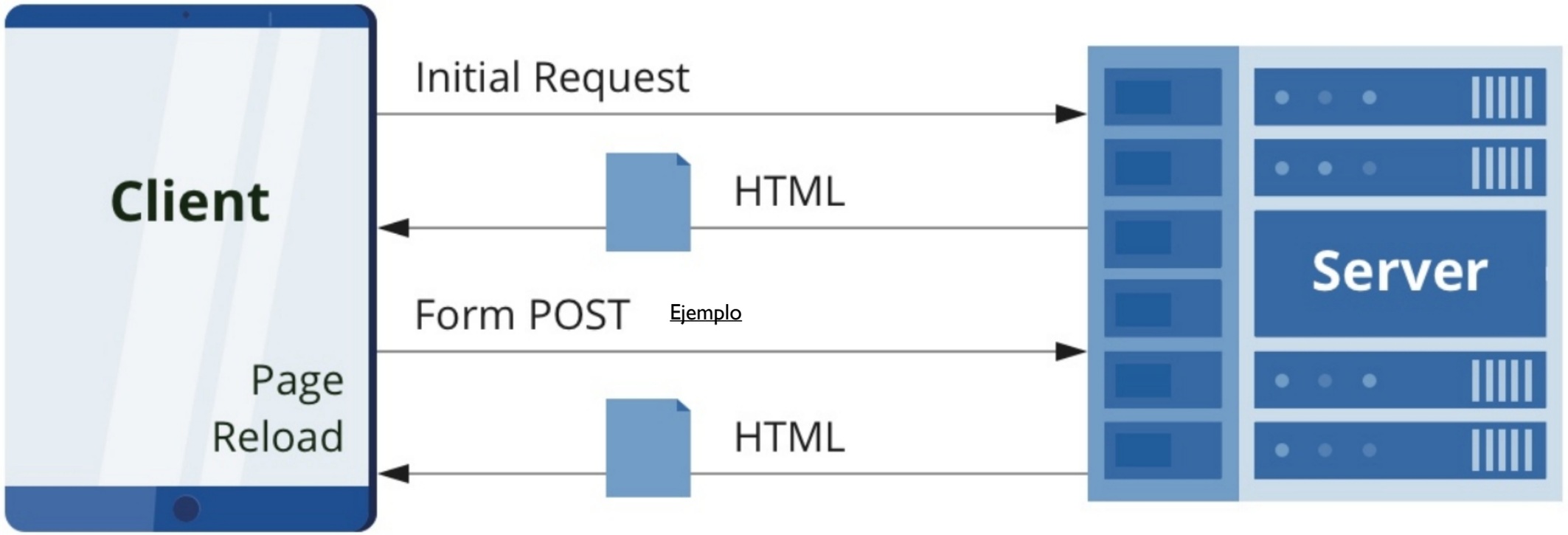
(O cómo se reparten el trabajo de generar el contenido cliente y servidor)

Tema 3, parte IV: Patrones de renderizado

1

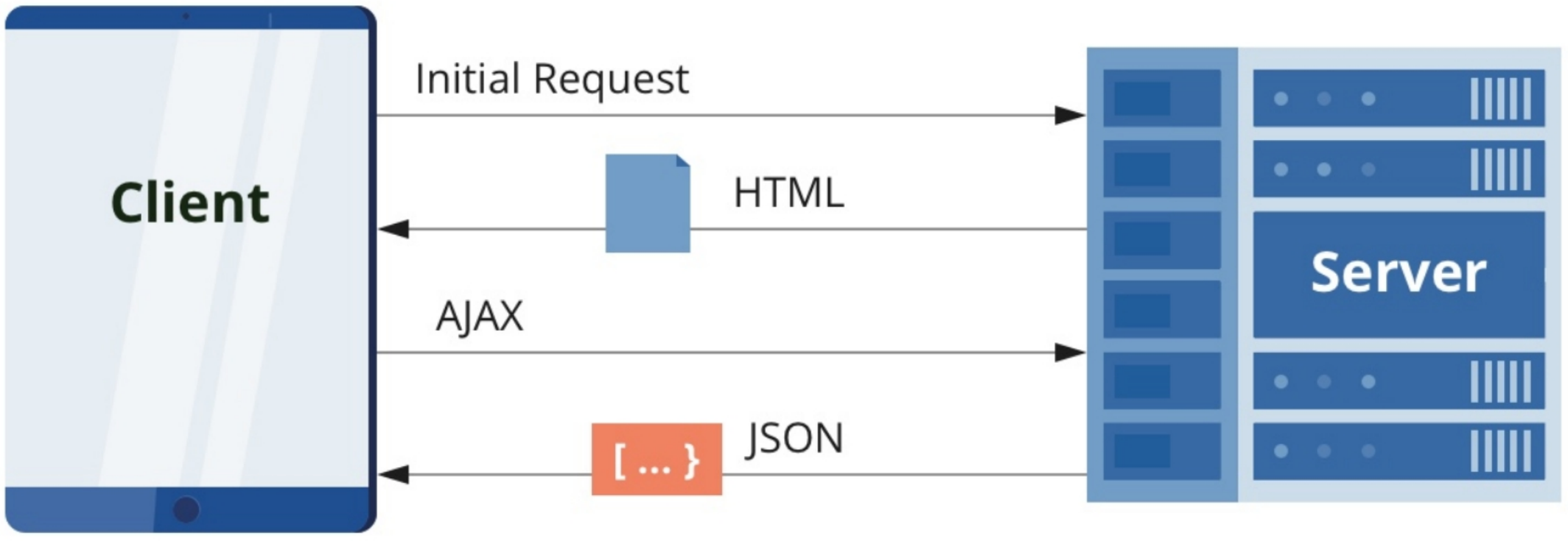
Introducción: MPAs vs SPAs

Multi Page Apps (MPA)



https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli

Single Page Apps (SPA)

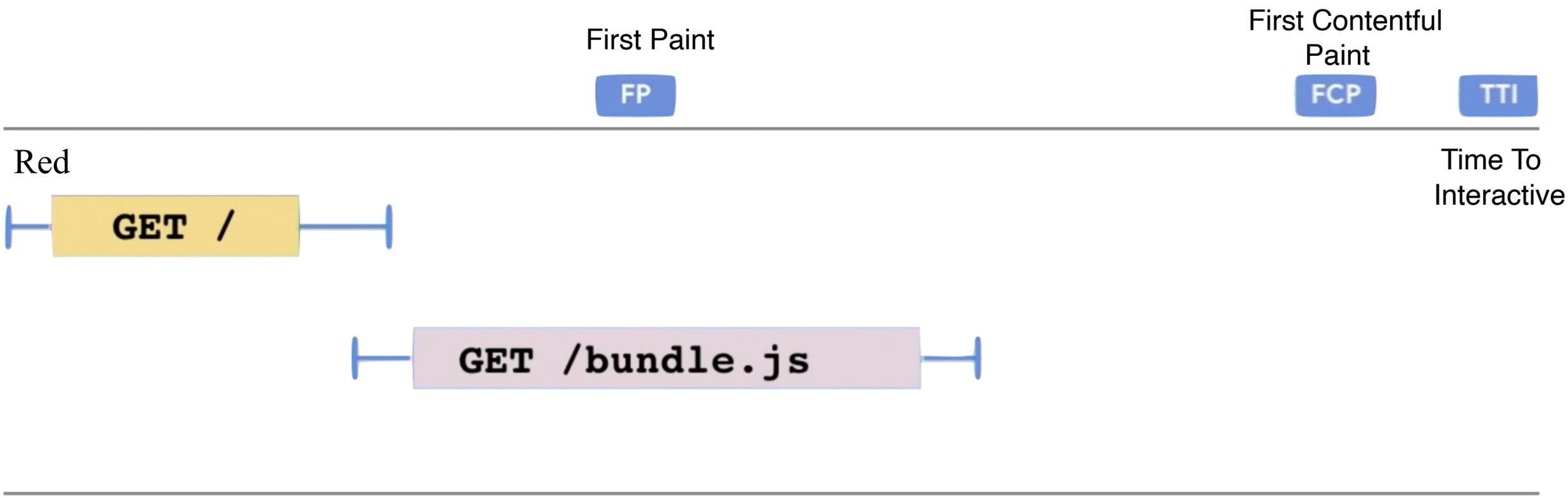


https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli

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)*

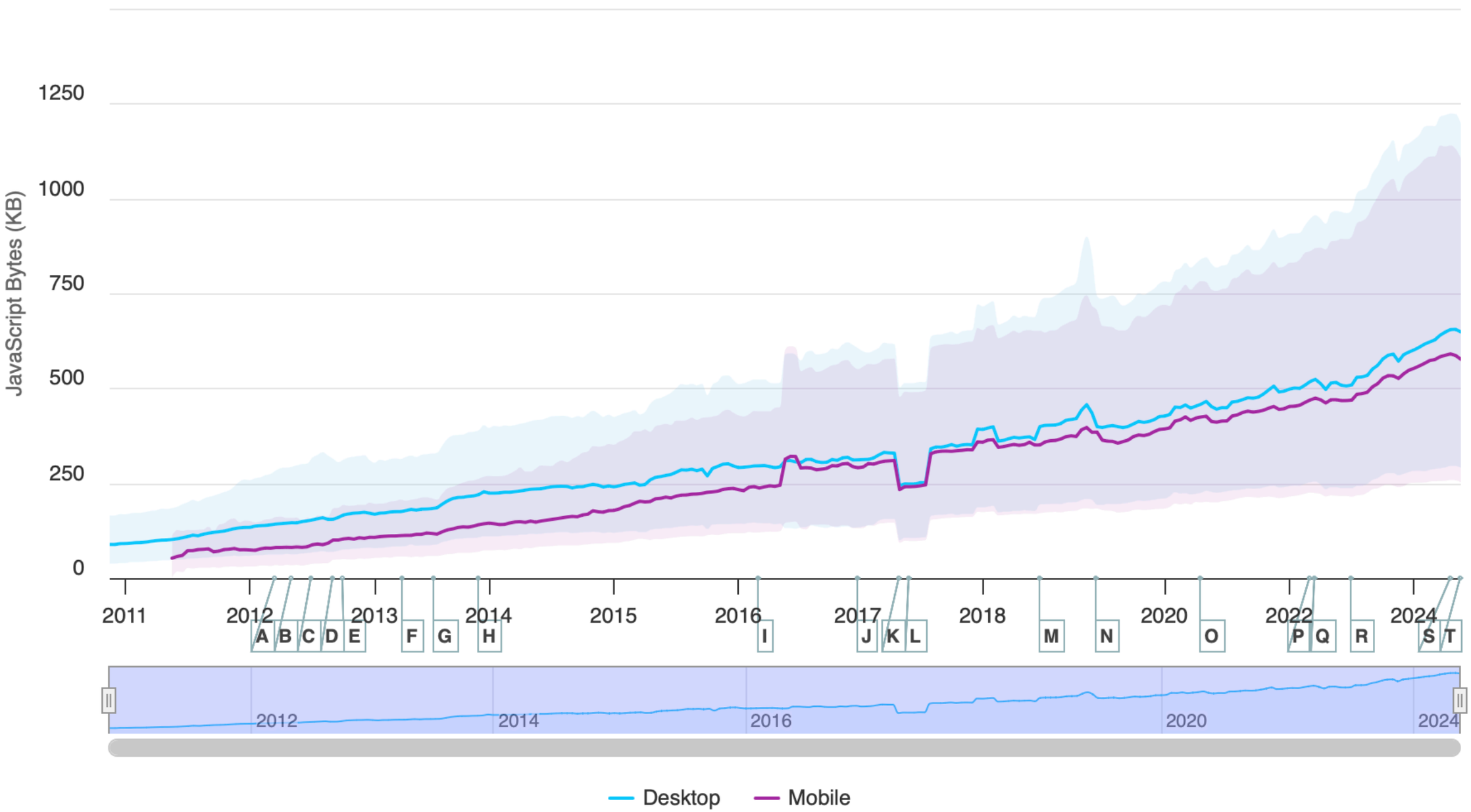
En 2010 Twitter cambió su arquitectura frontend a una SPA, ***y no te imaginas lo que sucedió después…***

> [...] Descubrimos que el *parseado* y la ejecución de JavaScript provocaban valores atípicos masivos en la velocidad percibida de renderizado. **En nuestra arquitectura totalmente del lado del cliente, no verás 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 en 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 3, parte IV: Patrones de renderizado

2

Apps universales o isomórficas

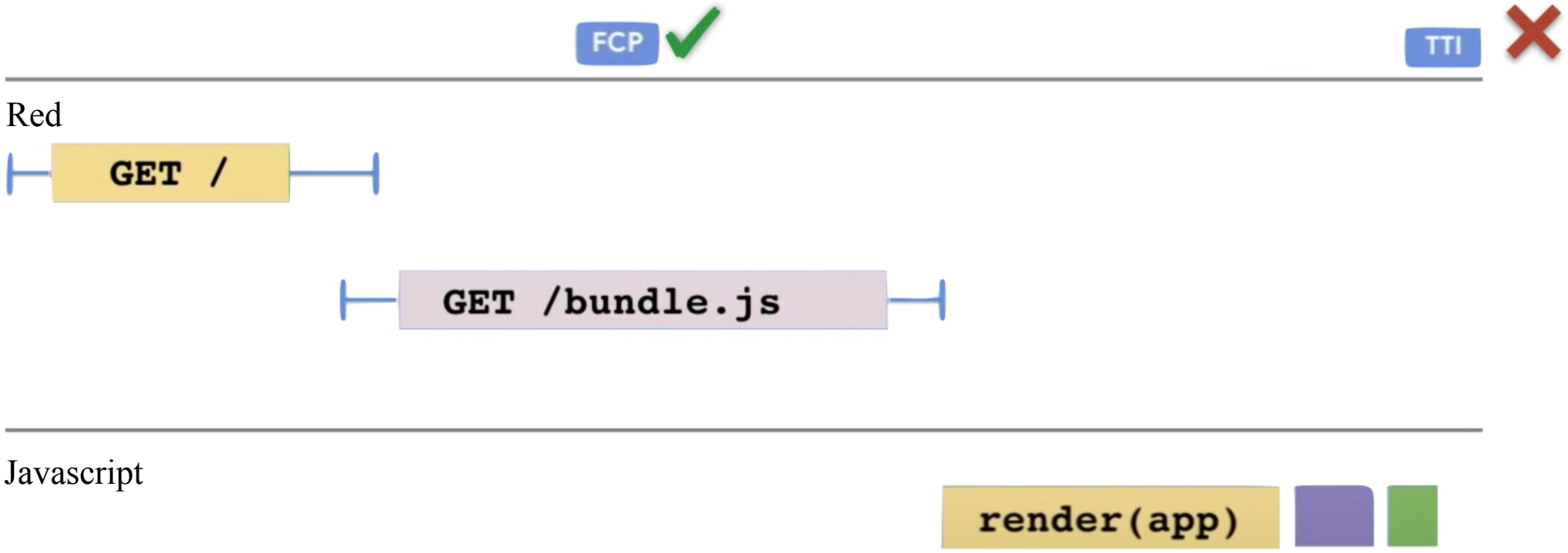
También conocido como SSR (Server Side Rendering)

Apps universales/isomórficas

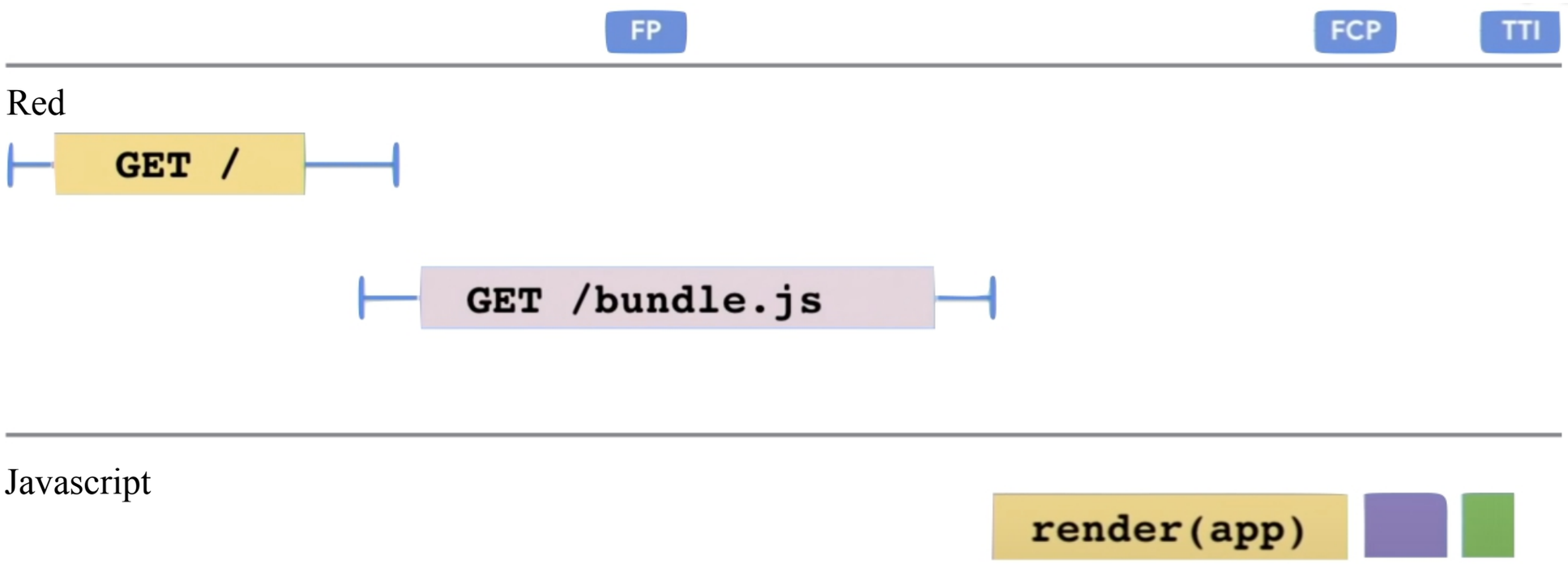
- También conocidas como Server Side Rendering (SSR)
- El HTML se genera en el servidor y se envía al cliente (== MPA)
- Una vez renderizado en el cliente, los componentes se "**hidratan**" con JS y se convierten en interactivos
- A partir de aquí todo es como en una SPA

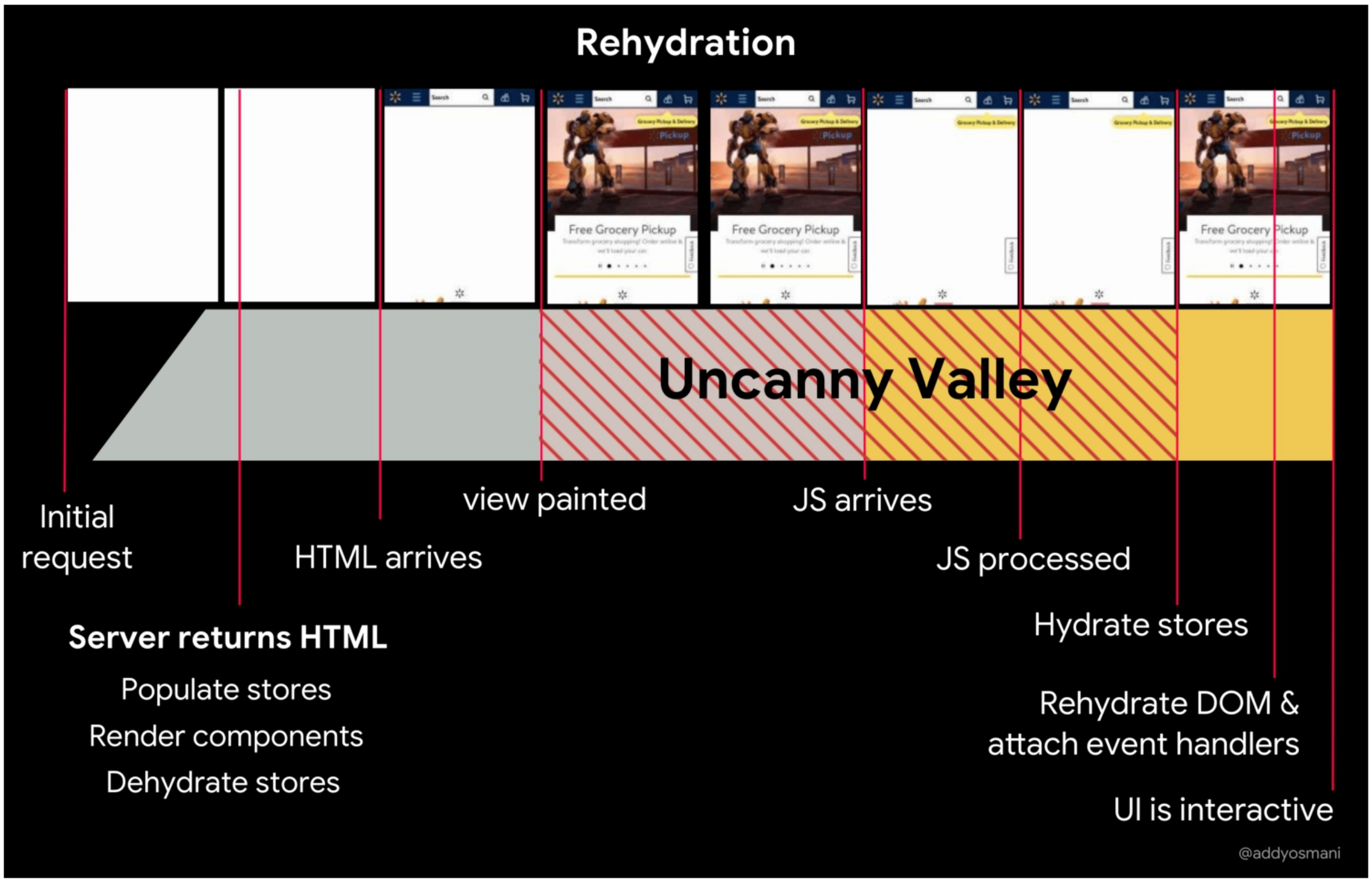
```
<head>
  <title>ToDo</title>
  <link rel="stylesheet" href="/bundle.css"></script>
</head>
<body>
  <h1>To Do's</h1>
  <ul>
    <li><input type="checkbox"> Wash dishes</li>
    <li><input type="checkbox" checked> Mop floors</li>
    <li><input type="checkbox"> Fold laundry</li>
  </ul>
  <footer><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





Addy Osmani, the cost of client side rehydration

¿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');
});
```

Toma una app Vue y devuelve el HTML de la app renderizada en forma asíncrona (**solo servidor**)

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

```
import { ref, createSSRApp } from 'vue';

export function createApp() {
  return createSSRApp({
    setup() {
      const count = ref(0);
      return { count };
    },
    template: `
       <h1>{{count}}</h1>
       <button @click="count++">Incrementar</button>
    `,
  });
}
```

client.js: código ejecutado **solo 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 (y otras) "*out of the box* "

(Vue)

[Demo básica de Nuxt]

(React)

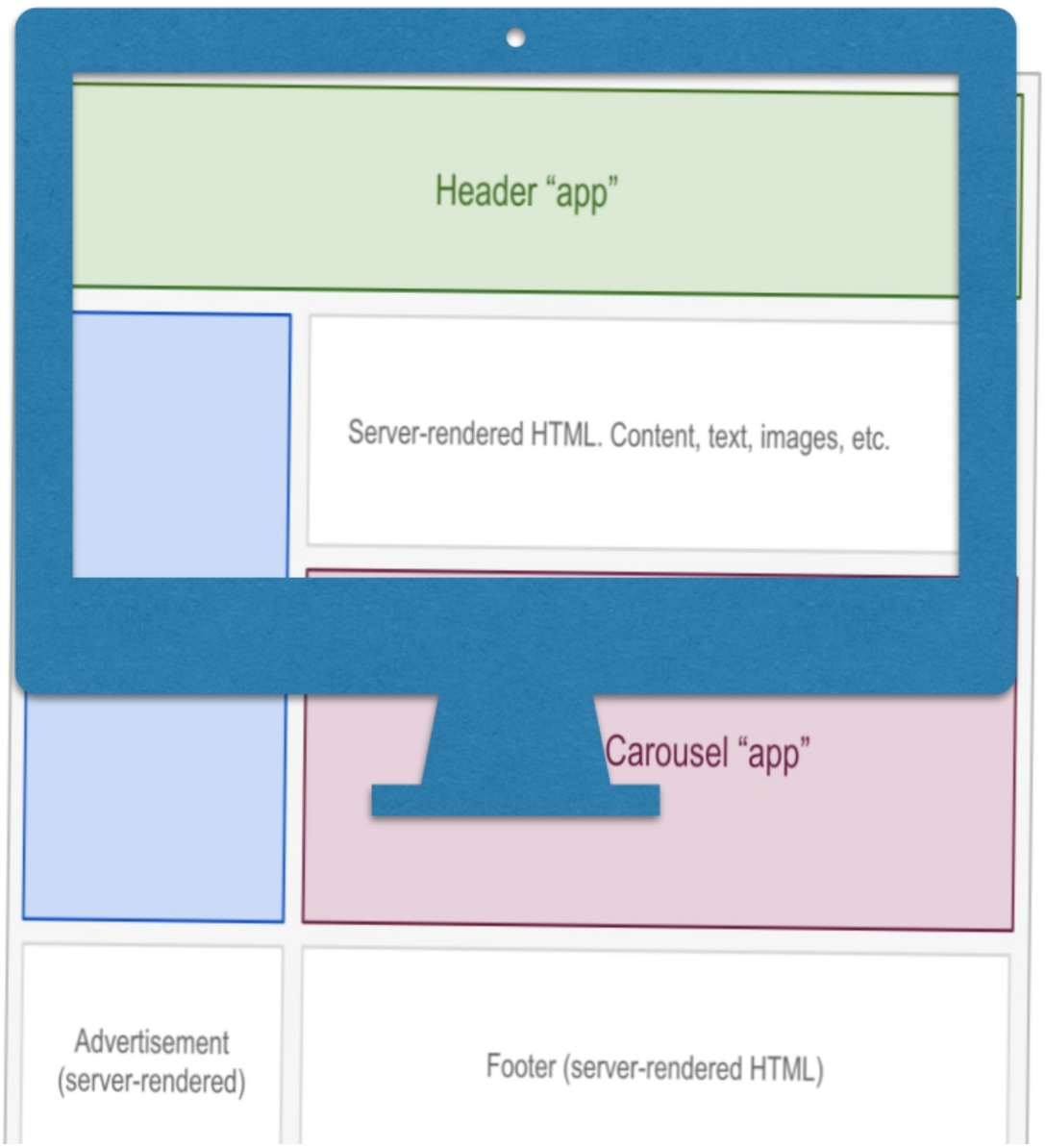
(Svelte)

[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” (idem p.ej en React)

```vue
<template>
  <LazyHydrationWrapper
    :when-visible="{ rootMargin: '50px' }"
    @hydrated="onHydrated"
  >
    <!--
    Content hydrated when one of the root elements is visible.
    All root elements are observed with a margin of 50px.
    -->
  </LazyHydrationWrapper>
</template>

<script setup>
  import { LazyHydrationWrapper } from 'vue3-lazy-hydration';

  function onHydrated() {
    console.log('content hydrated !');
  }
</script>
```

https://github.com/freddy38510/vue3-lazy-hydration

Otros frameworks

Algunos frameworks sí lo incluyen de forma nativa, 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 />
```

astro

https://astro.build/

`<MyComponent client:load />`

Hydrate the component on page load.

`<MyComponent client:idle />`

Hydrate the component as soon as main thread is free (uses requestIdleCallback()).

marko

https://markojs.com/

`<MyComponent client:visible />`

Hydrate the component as soon as the element enters the viewport (uses IntersectionObserver).
Useful for content lower down on the page.

Otros frameworks

- Frameworks como Qwik llevan esta idea “al límite” permitiendo que el JS se cargue justo a medida que se necesita

https://qwik.builder.io/

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

(del README.md)

1. The TODO app should have come up in the stackblitz IDE.
2. Open it in the inspector and notice that the app was delivered as HTML
3. Notice that very little JS got downloaded/executed with the application
4. Open the network tab, and notice that more code is downloaded as you interact with the application.
5. Look for `on:click`/`on:keyup` element attributes which define event listener

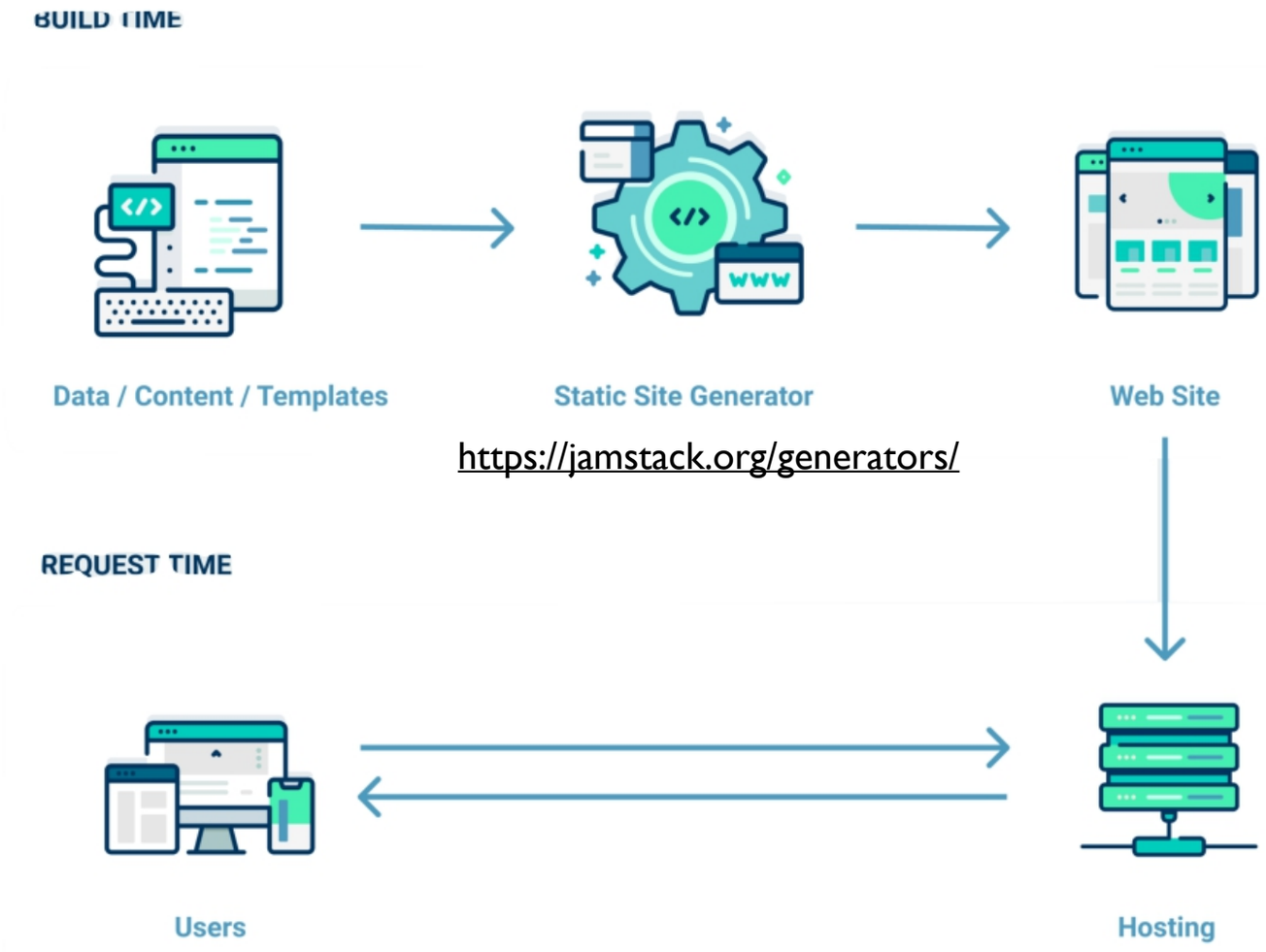
```
<input class="toggle"
       type="checkbox"
       checked={item.completed}
       on:click={QRL`ui:/Item_toggle#?toggleState=.target.checked`}/>
<label on:dblclick={QRL`ui:/Item_edit#begin`}>{item.title}</label>
<button class="destroy" on:click={QRL`ui:/Item_remove#?itemKey=${itemKey}`}></button>
```

Tema 3, parte IV: Patrones de renderizado

3 Prerendering/ Jamstack

Prerendering o SSG

- 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 (solo los no personalizados)

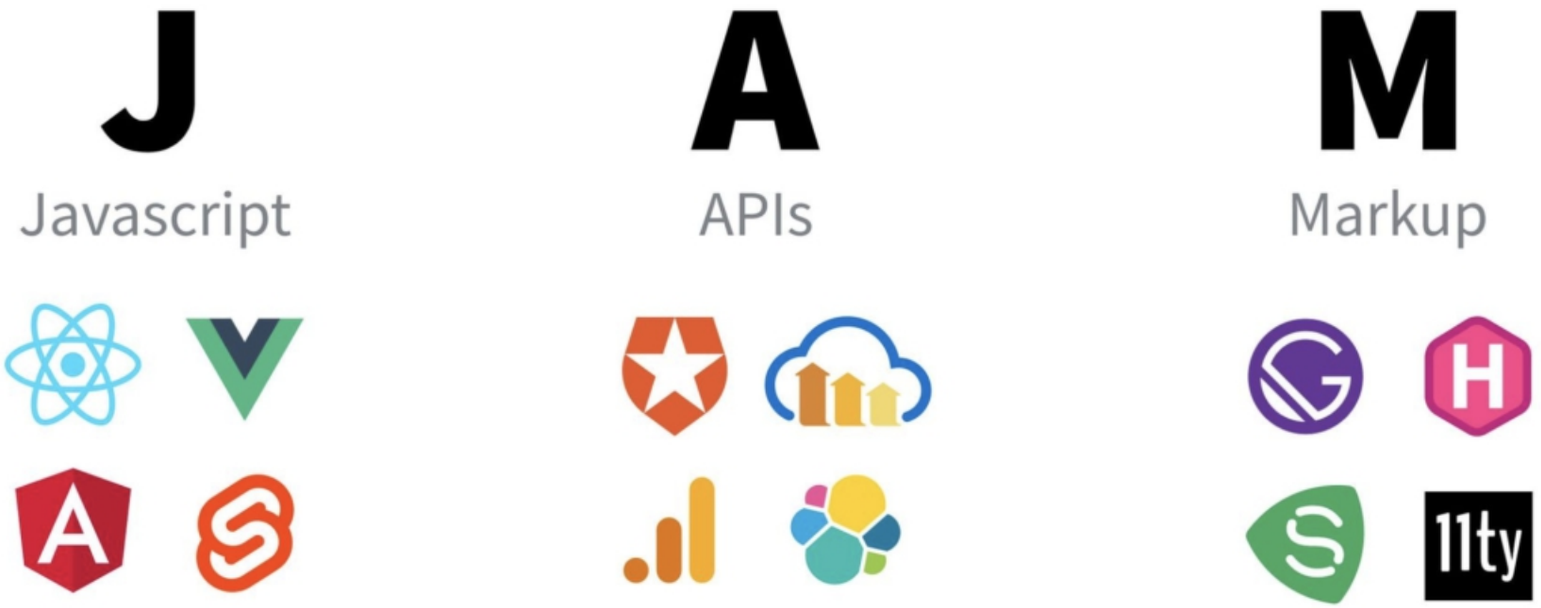


https://jamstack.org/generators/

https://david-neuman.com/nextjs-pre-rendering/

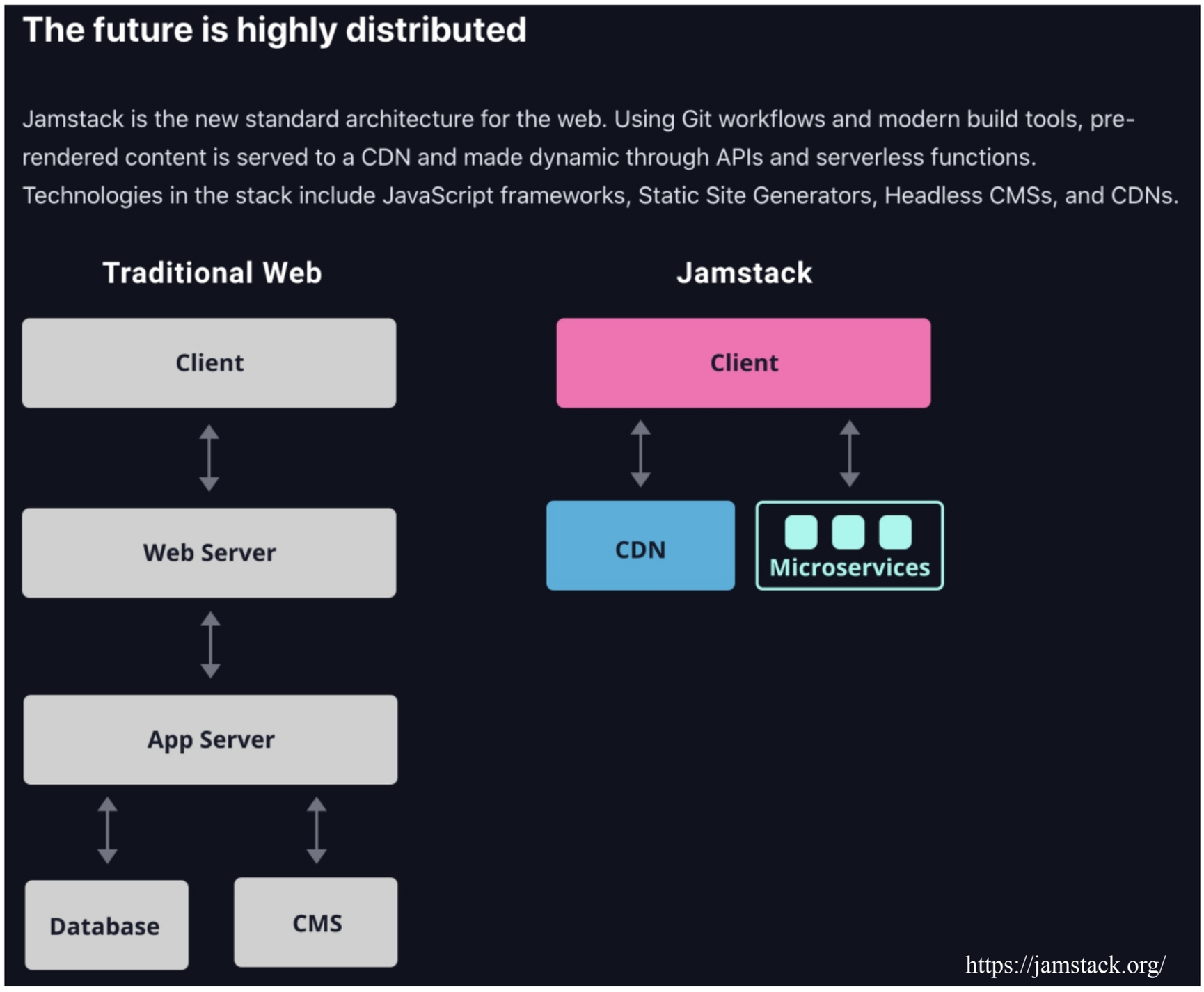
JamStack

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



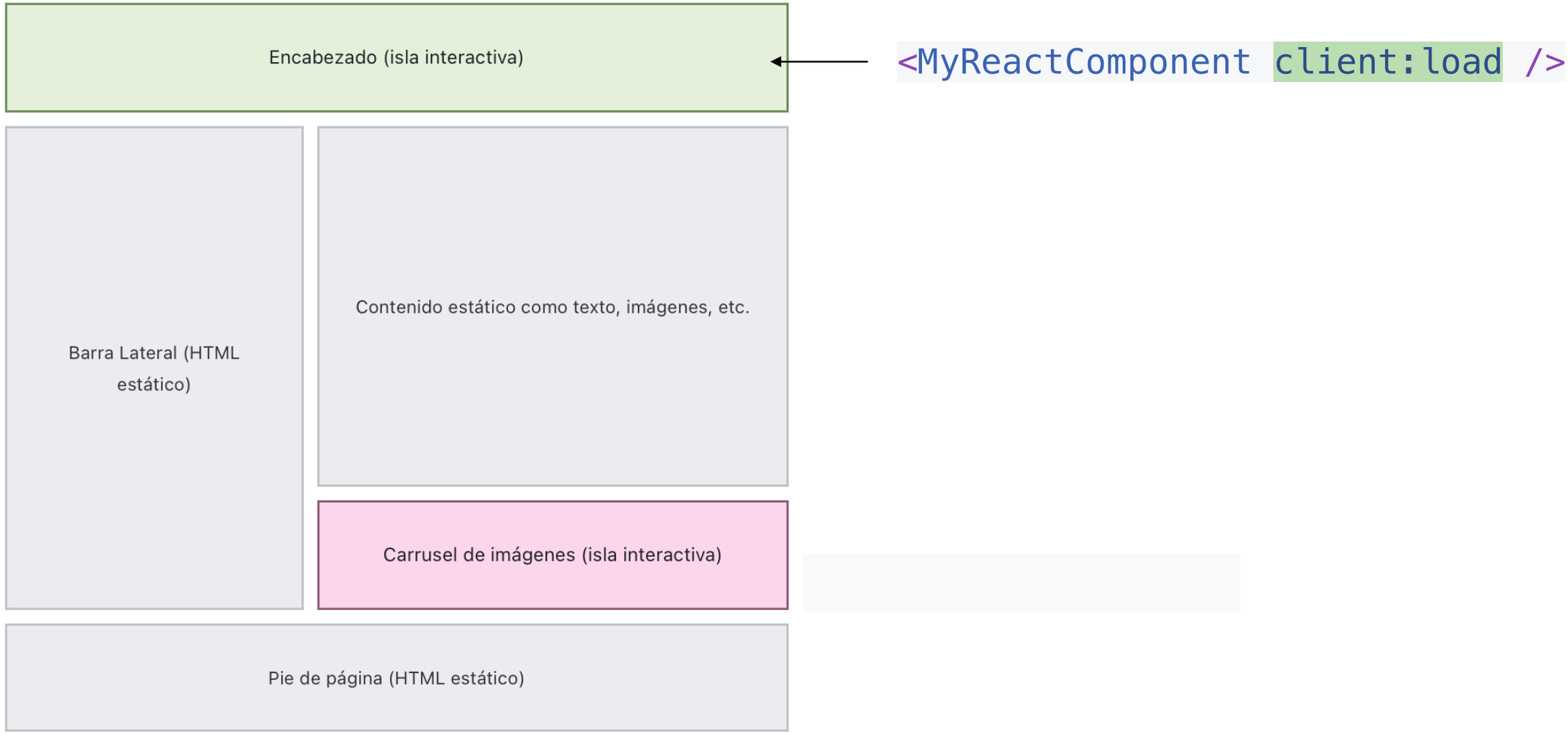
https://jamstack.org/what-is-jamstack/

JamStack vs. stack tradicional



Mezclando ideas

- Por supuesto una página puede tener **partes pre-generadas y otras dinámicas** y personalizadas para el usuario actual
- Esta idea se popularizó en el framework **Astro** como "**Arquitectura de islas**" donde una "isla" es un componente dinámico y el resto es pre-renderizado



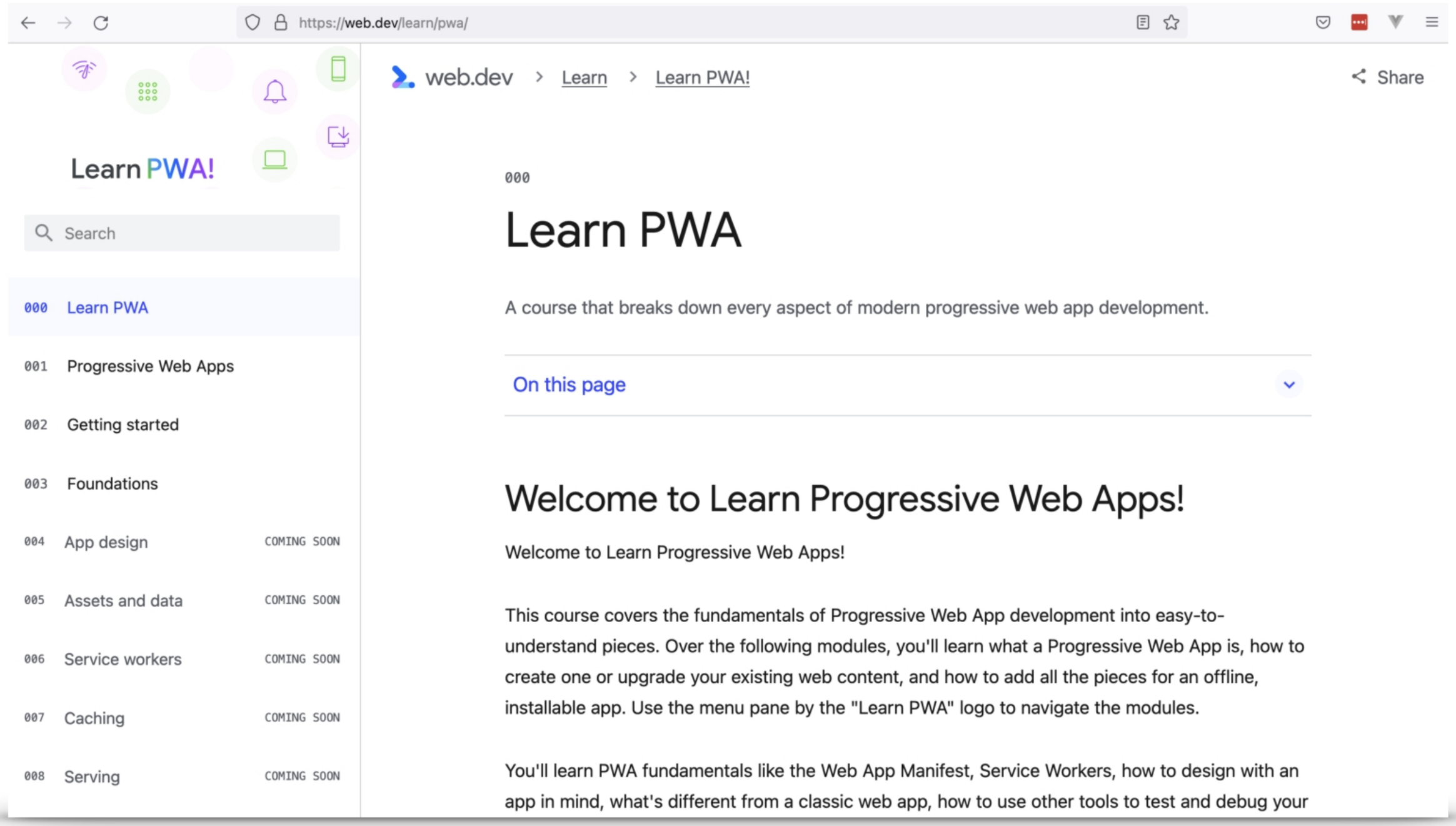
Despliegue

Servidores mayormente “estáticos” que actúan como una CDN

Vercel netlify surge

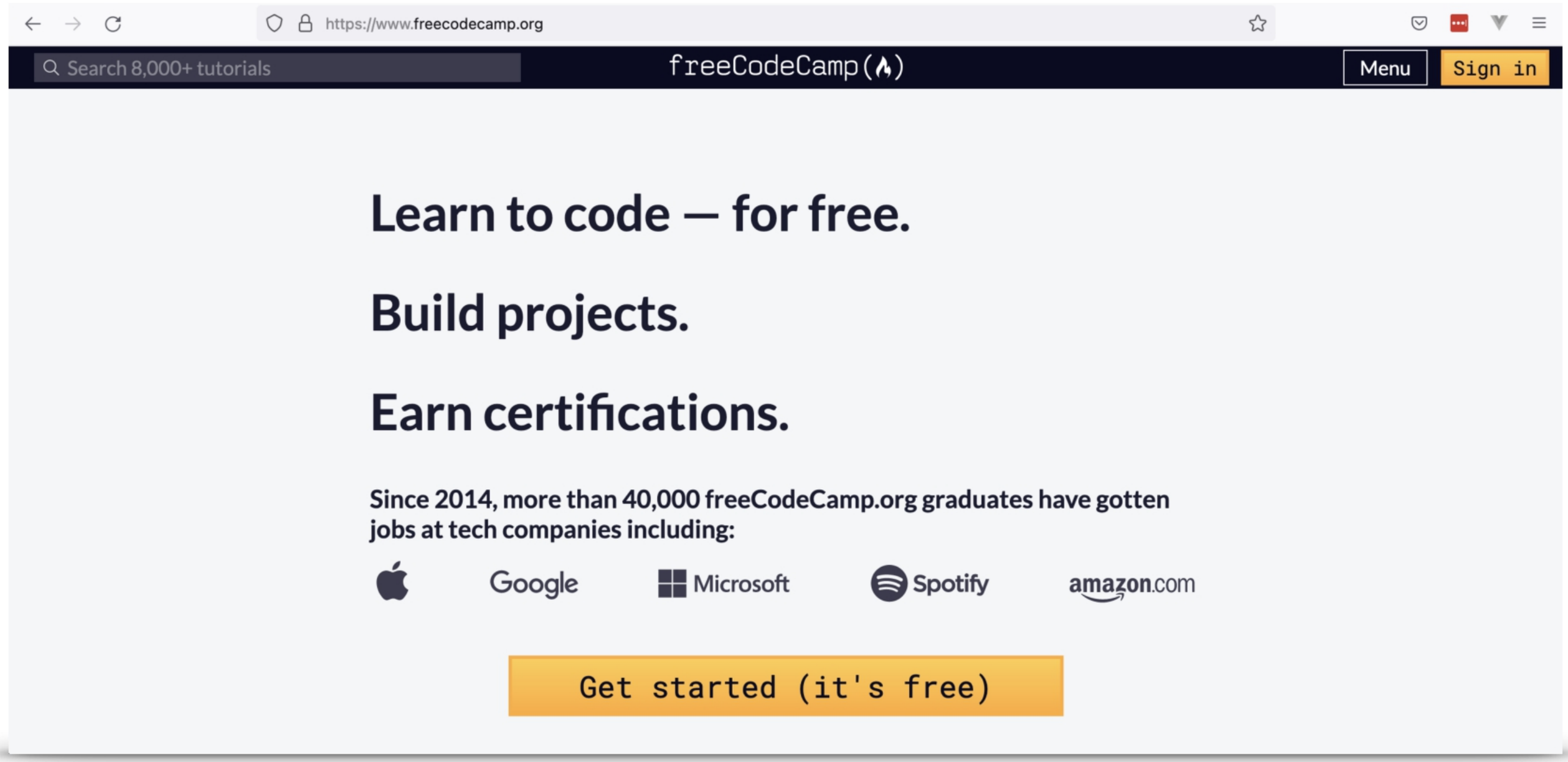
Edge rendering: el contenido dinámico se procesa en servidores físicamente próximos al usuario (en el *edge*), mejorando la latencia

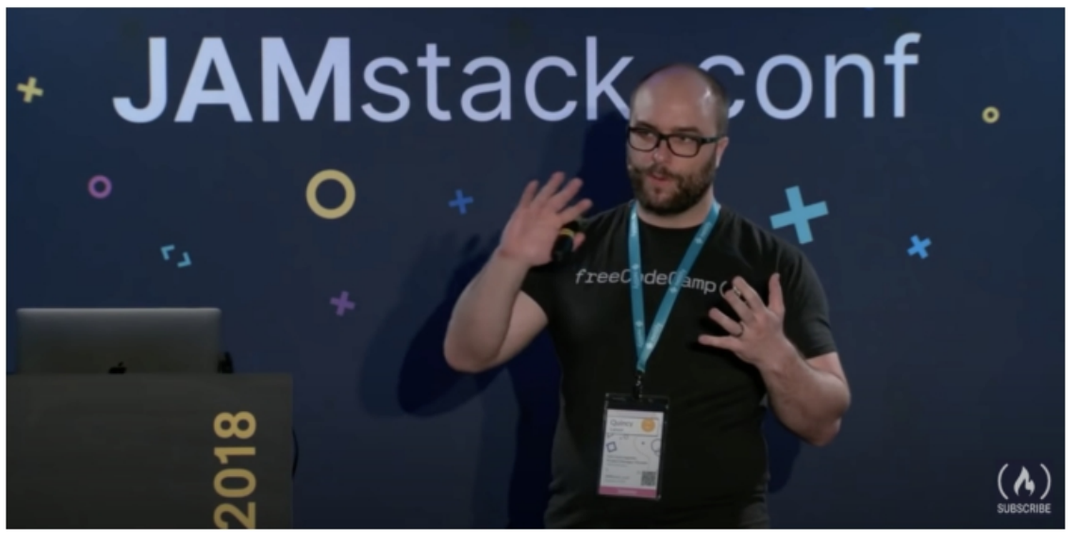
Ejemplo: Google web.dev



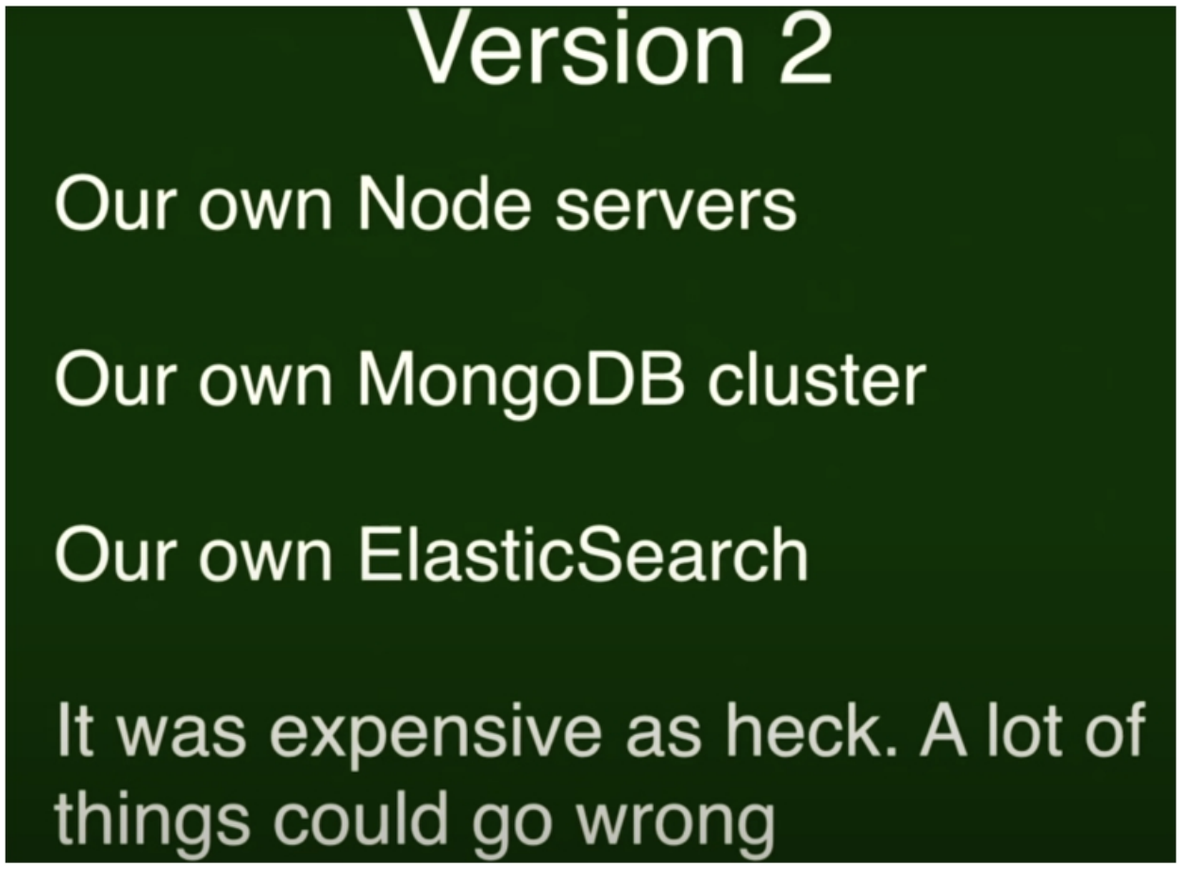
https://github.com/GoogleChrome/web.dev

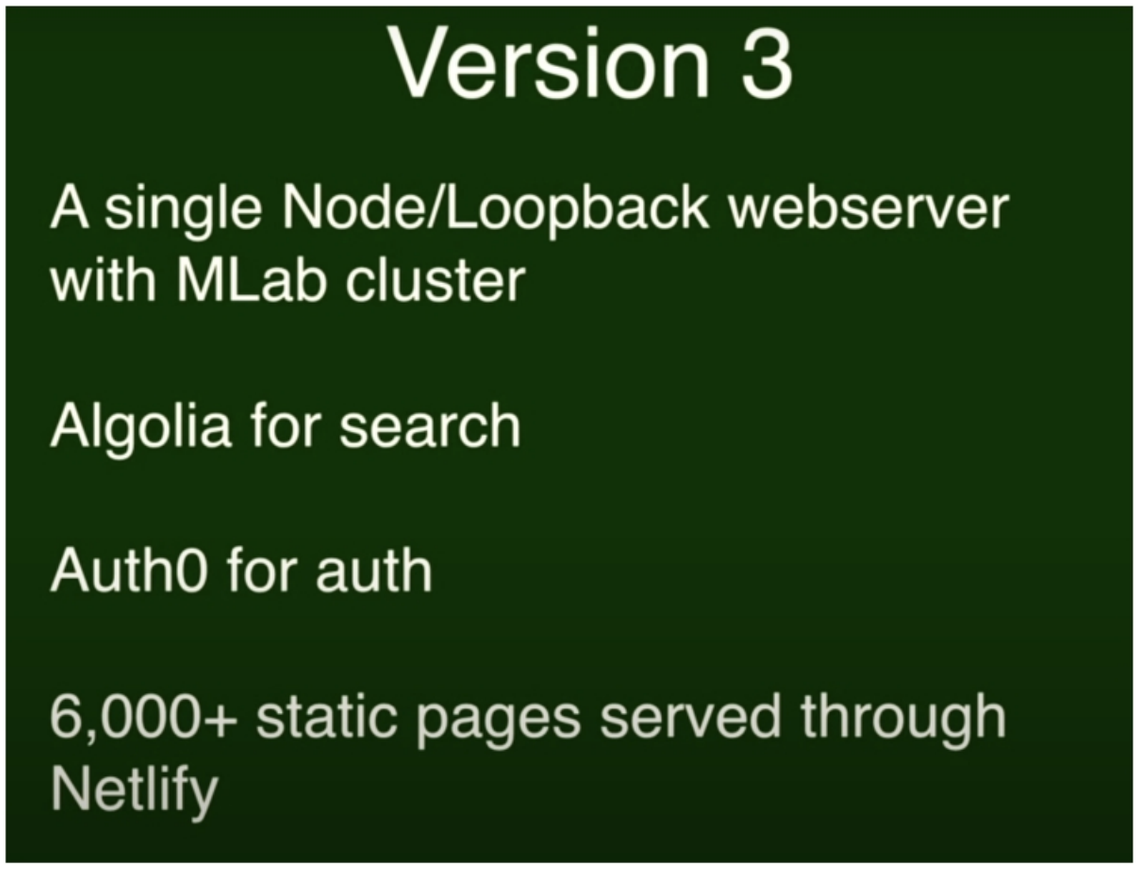
Ejemplo: Freecodecamp





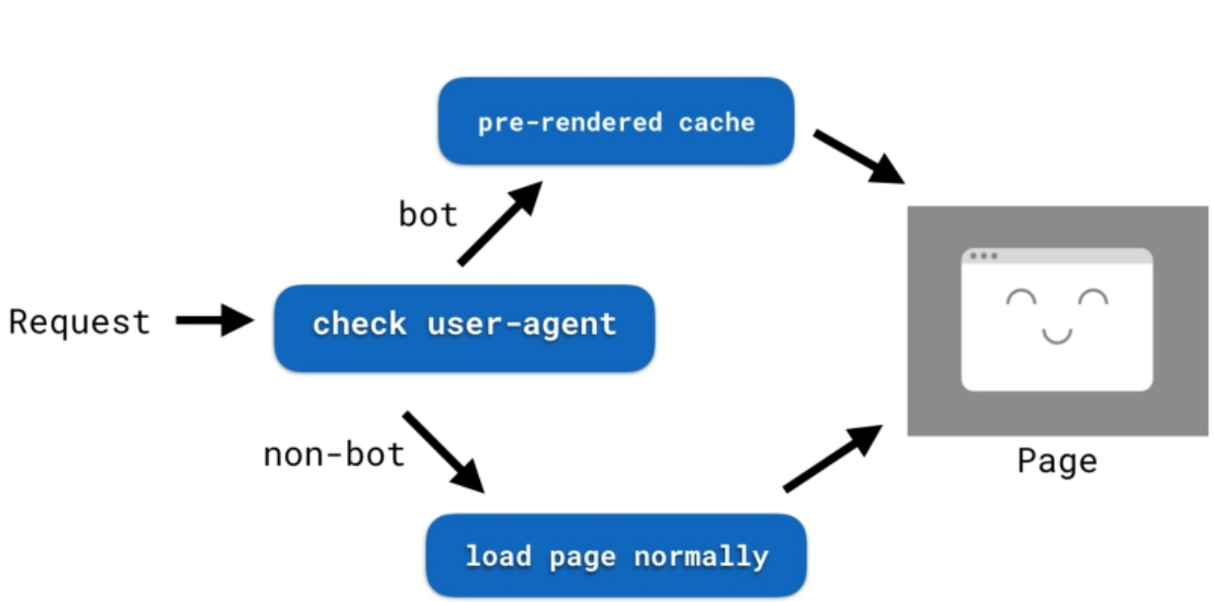
[How freeCodeCamp Serves Millions of Learners Using the JAMstack]



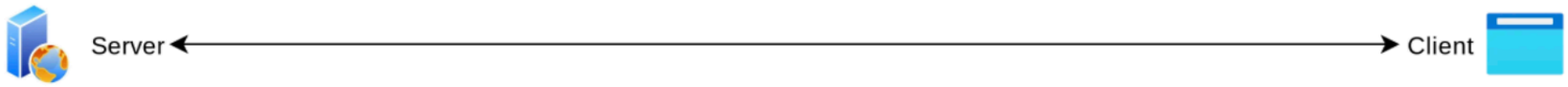


Prerendering y SEO

Los bots de indexado como Google se “llevan mal” con el contenido generado por el cliente. Por motivos de SEO conviene usar *prerendering* si es posible



https://prerender.io/



	Classic SSR	SSR with Hydration	Streaming	Progressive Hydration	Static Generation	Incremental Static Generation	CSR
HTML generated on	Server	Server	Server	Server	Build Server	Build Server	Client
JavaScript for Hydration	No Hydration	JS for all components to be loaded for hydration	JS is streamed with HTML	JS is loaded progressively	Minimal JS	Minimal JS	No Hydration but JS for all components is required for rendering and interactivity
SPA Behaviour	Not Possible	Limited	Limited	Limited	Not Possible	Not Possible	Extensive
Crawler Readability	Full	Full	Full	Full	Full	Full	Limited
Caching	Minimum	Minimum	Minimum	Minimum	Extensive	Extensive	Minimum
TTFB	High	High	Low and consistent across page sizes	High	Low	Low	Low
TTI : FCP	TTI = FCP	TTI > FCP	TTI > FCP	TTI > FCP	TTI = FCP	TTI = FCP	TTI >> FCP
Implemented Using	Server side scripting languages like PHP	React for Server, Next.js	React for Server (React 16 onwards)	Full fledged React solution under development	Next.js	Next.js	CSR frameworks like React, Angular etc
Suitable For	Static content pages like news or encyclopedia pages	Mostly static pages with few interactive components. E.g., comments section of a blog	Mostly static pages that can be streamed in chunks. E.g., search results listing pages	Interactive pages where activation of some components may be delayed. E.g., Chatbot	Static content that does not change often. About Us or Contact us pages of websites	Huge quantities of static content that may change frequently. Blog listing or Product listing pages.	Highly Interactive apps where user experience is critical. E.g., Social media messaging and commenting

https://www.patterns.dev/vanilla/rendering-patterns/

Algunas referencias

- Rendering on the Web: Performance Implications of Application Architecture (Google I/O19)
- Renderización en la Web , Jason Miller & Addy Osmani, `web.dev`
- Javascript design patterns, rendering patterns, Lydia Halle, Addy Osmani y otros
- 10 rendering patterns for web apps, Canal de Youtube “Beyond Fireship”