Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Divide et impera: desarrollo modular de aplicaciones web

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Isidro Torregrosa Torralba

Tutor: Patricio Orlando Letelier Torres

Curso 2019 - 2020

Resumen

Resumen en español.

Palabras clave: a, b, c, d.

Resum

Resumen en valencià.

Paraules clau: a, b, c, d.

Abstract

Abstract in English.

Keywords : a, b, c, d.

Tabla de contenidos

Tabla de figuras

1. Introducción

1.1. **Motivación**

El desarrollo web ha experimentado un cambio de tendencia con respecto a sus objetivos durante la última década. La finalidad principal del desarrollo web ha sido, históricamente, la creación de páginas: documentos que aportan contenido gráfico y textual al usuario sin que este necesite interactuar con el mismo. En la actualidad, son cada vez más las empresas u organizaciones que optan por el desarrollo de aplicaciones, las cuales no sólo aportan contenido e información, si no que tienen como principal objetivo ofrecer funcionalidad por medio de la interacción con el usuario. Este cambio de tendencia se debe en parte a que este tipo de aplicaciones están ocupando el lugar de las aplicaciones nativas, tanto de plataformas móviles como de escritorio, pues, entre otras ventajas son más baratas de desarrollar y mejora en varios aspectos la experiencia del usuario final [1].

El aumento de requisitos funcionales a los que las aplicaciones web deben hacer frente supone un incremento en el tamaño de las mismas. Como se aprecia en la figura 1.1, la transferencia total de recursos pedidos por una página creció de media, desde 2010, un 334,4% en dispositivos de escritorio y un 1187,4% en dispositivos móviles.

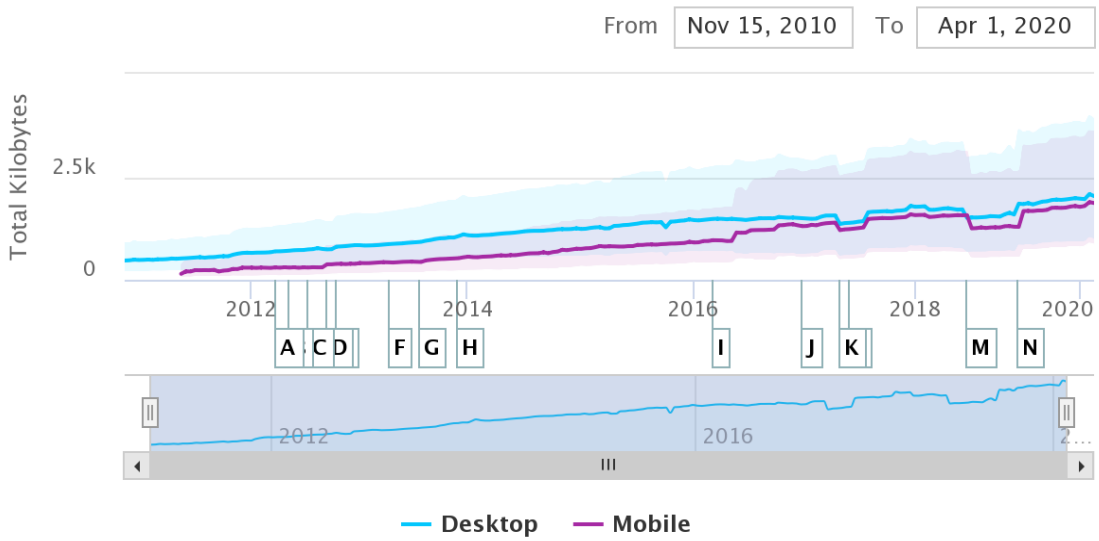

Figura 1.1: serie temporal de la descarga de Kilobytes por página [2]

También es notable el crecimiento del ecosistema con el fin de hacer frente a los nuevos problemas y a las expectativas cada vez más exigentes de los usuarios. La cantidad de nuevas herramientas de JavaScript (JS), el lenguaje de programación predominante en el desarrollo web, ha aumentado de forma drástica comparada con como lo han hecho las herramientas de otros ecosistemas como el de Java o .NET,

plataformas muy populares en el desarrollo de aplicaciones de servidor. La figura 1.2 muestra la cantidad de módulos publicados en los repositorios centrales de los distintos ecosistemas desde el 2012 hasta la actualidad.

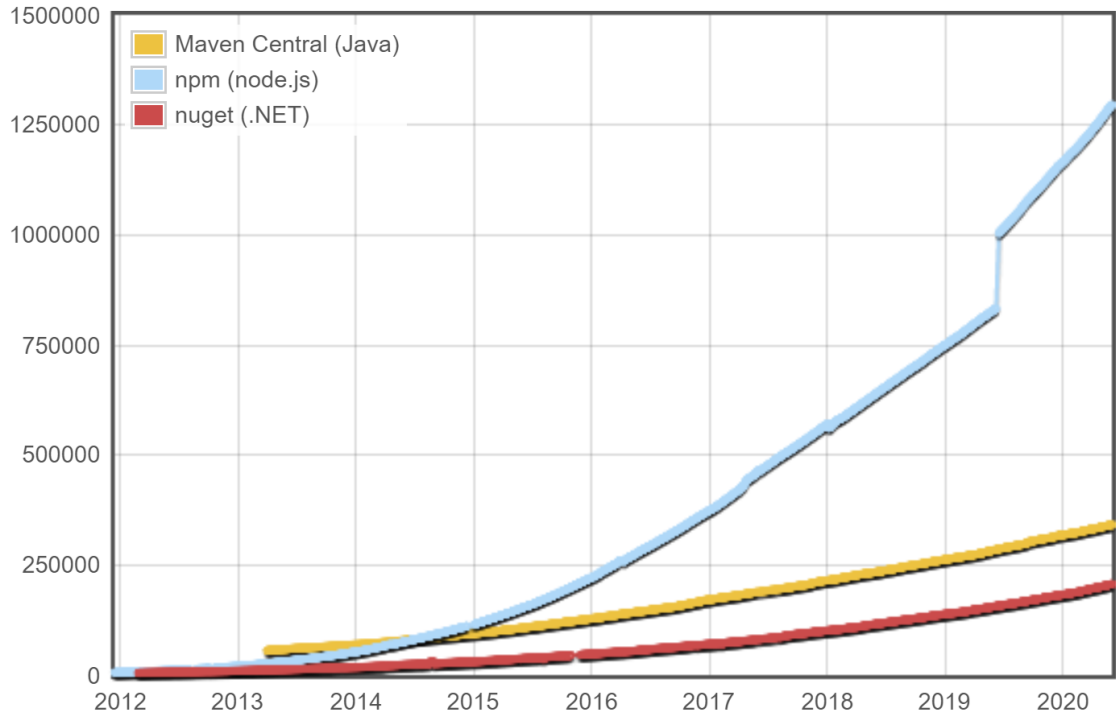

Figura 1.2: número de módulos por año por ecosistema [3]

Si bien la introducción de nuevas herramientas en el desarrollo y su sofisticación hacen que muchas de las tareas sean más simples, no por ello el trabajo del desarrollador resulta más sencillo. "*Cuando tu campo cambia tan rápido que las buenas prácticas quedan obsoletas en un plazo de dos años, tienes que correr para mantenerte al día*" [4].

Así pues, el aumento de los requisitos funcionales, que necesariamente conlleva un aumento en el tamaño del código fuente, sumado al cambio constante del ecosistema, tiene un impacto directo en la complejidad de las aplicaciones, repercutiendo de forma negativa en los procesos de desarrollo y mantenimiento [5]. Esto a su vez tiene una correlación con el aumento del tiempo transcurrido entre la implementación de una funcionalidad y su puesta en producción [6], es decir, el plazo de lanzamiento, y con el aumento de la posibilidad de introducir errores a la hora de realizar cambios en la funcionalidad existente.

Esta memoria pretende proponer un marco de trabajo en el que el incremento en la complejidad de la aplicación no suponga un incremento en la complejidad de su desarrollo, mejorando la satisfacción del usuario final y del equipo de desarrollo en el proceso.

1.2. **Objetivos**

El objetivo de este proyecto, a alto nivel, es la implementación de un marco de trabajo que permita desarrollar una aplicación web de forma sencilla y mantenible, independientemente de la envergadura de la misma. Los objetivos específicos son:

- Desarrollar un marco de trabajo en el que la complejidad sea estable a medida que aumenta la cantidad de funcionalidad ofrecida por la aplicación.
- Estudiar diferentes alternativas tecnológicas a la hora de implementar este marco de trabajo.
- Evaluar los resultados obtenidos desarrollando una aplicación de banca electrónica utilizando este marco de trabajo.

Cabe mencionar que el alcance de este trabajo se limita a la parte de cliente de las aplicaciones web y por tanto no abarca la parte de servidor.

1.3. **Estructura**

Repaso breve sobre lo que sigue y en qué partes de alto nivel se divide el proyecto.

2. Desarrollo de aplicaciones web

Las decisiones a considerar a la hora de desarrollar una aplicación web son numerosas, y a menudo complejas, debido al gran número de opciones disponibles. Así mismo, existen ciertos problemas que toda aplicación debe resolver, derivados, en gran medida, de la rápida evolución de la industria y la falta de soluciones estándar.

En este capítulo se realiza un resumen detallado, a modo de introducción, sobre todos estos aspectos a tener en cuenta. Dando así una visión de alto nivel sobre el reto que constituye el desarrollo web en la actualidad, desde cuestiones arquitectónicas hasta aspectos relacionados con los sistemas de control de versión, pasando por las distintas alternativas tecnológicas disponibles para resolver los problemas más comunes del desarrollo de aplicaciones web.

2.1. Arquitecturas de aplicaciones web

Actualmente existen diferentes alternativas a la hora de construir una aplicación web en función de dónde reside la lógica de la interfaz de usuario, de cómo el navegador obtiene los recursos cuando el usuario visita una página y de qué ocurre en las subsiguientes navegaciones [7].

2.1.1. *Multi Page Application (MPA)*

MPA es el nombre que recibe la arquitectura utilizada tradicionalmente en el desarrollo de aplicaciones web, basado en confeccionar una aplicación a partir de múltiples páginas. Se caracteriza por ser el servidor de la aplicación el que realiza la mayor parte de la lógica y el que ante cada navegación genera los contenidos de la página y los envía al cliente.

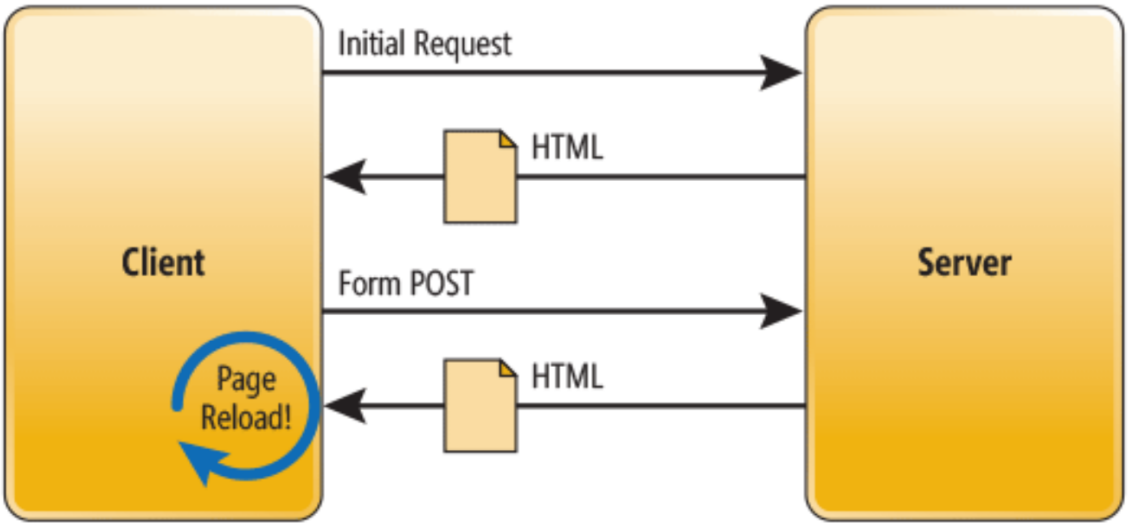

Figura 2.1: ciclo de vida de una MPA [8]

Como el estado de la aplicación se pierde con cada navegación al reemplazarse la página por una diferente, esta arquitectura suele ser preferible cuando los requisitos funcionales en la parte de cliente son simples o de solo lectura [9].

2.1.2. *Single Page Application (SPA)*

En las aplicaciones de una sola página el servidor provisiona al navegador de los recursos necesarios para presentar el contenido, bien en la primera carga de la página o de forma dinámica y bajo demanda. El navegador, que contiene la mayor parte, si no toda, la lógica de la interfaz gráfica, se encarga de reemplazar el contenido de la página con cada navegación.

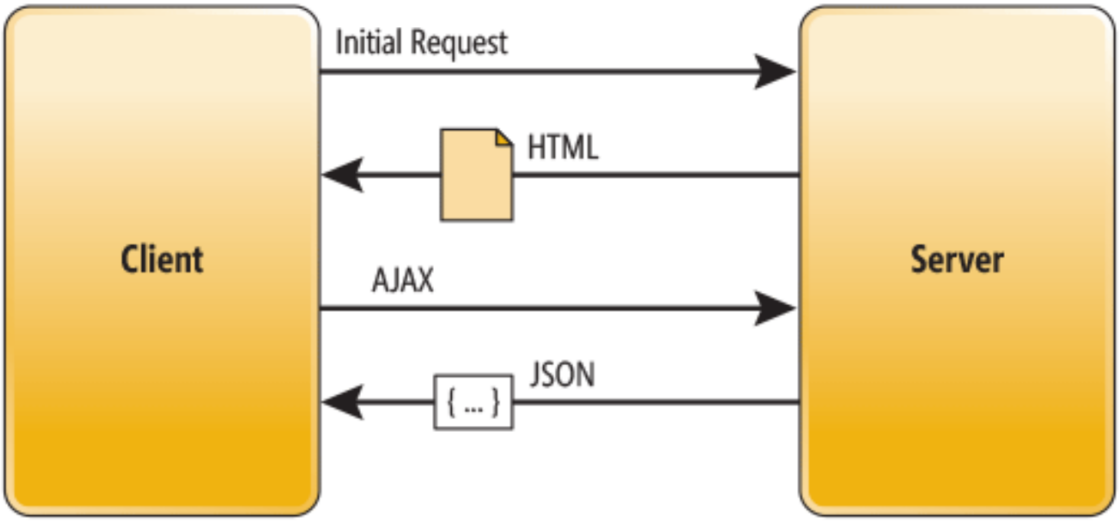

Figura 2.2: ciclo de vida de una SPA [8]

Al no recargarse la página, el estado se mantiene en el cliente. Esto permite ofrecer una mejor experiencia a los usuarios y soportar funcionalidad más compleja [9], haciendo que la página web se asemeje a una aplicación nativa.

Este modelo de aplicaciones trae mejoras sustanciales para la experiencia de usuario, pero con un coste. Mover la lógica de la interfaz del servidor al cliente no elimina la complejidad, simplemente la desplaza, introduciendo a su vez nuevos retos en cuanto a la arquitectura y el despliegue de la aplicación web.

2.2. Tecnología para el desarrollo de aplicaciones web

Con el auge de la popularidad de las aplicaciones web de una página han surgido una gran cantidad de librerías y *frameworks* con el objetivo de facilitar el desarrollo de este tipo de aplicaciones. Si bien cada una de estas herramientas tienen características que las hacen únicas, las más populares y modernas coinciden en emplear un modelo de desarrollo basado en componentes.

2.2.1. Desarrollo orientado a componentes

El desarrollo orientado a componentes propone una metodología para construir aplicaciones de forma modular, basada en la composición de piezas bien definidas e independientes a las que denomina componentes. Estos componentes pueden, a su vez, estar compuestos a partir de otros componentes más pequeños o que cumplen con una función más específica [10]. En la figura 2.3 se muestra el que podría ser el componente principal de una aplicación (*App)*, compuesto a su vez por una serie de componentes encargados de resolver cuestiones como la disposición del contenido, la navegación, el encabezado y el contenido de la página.

```
1   import React from 'react';
2   import {AppLayout, Navigation, Header, Content} from './components';
3
4   export class App extends React.Component {
5       render(){
6           return (
7               <AppLayout>
8                   <Navigation />
9                   <Header />
10                  <Content />
11              </AppLayout>
12          );
13      }
14  }
```

Figura 2.3: ejemplo de componente implementado en React

Desarrollar una aplicación haciendo uso de esta metodología, es decir, separando los distintos bloques funcionales en componentes reutilizables, independientes y diseñados para interoperar entre sí aporta una serie de ventajas notables:

- **Interfaces pequeñas:** al reducir la funcionalidad que implementa cada una de las piezas que forman la aplicación se consigue reducir el tamaño de sus interfaces, lo que hace que sean más fáciles de utilizar.

- **Reusabilidad:** cuando la funcionalidad de la aplicación está separada en módulos bien definidos diseñados para interoperar entre sí se favorece la reutilización de los mismos, incluso desde otras aplicaciones. Esto evita la necesidad de volver a implementar funcionalidad ya existente, lo que reduce tanto el tiempo como el coste del desarrollo, además de la posibilidad de introducir errores.

- **Capacidad de composición:** la posibilidad de crear nuevos componentes a partir de los ya existentes es el principal factor que favorece la reusabilidad. De esta manera, distintos componentes específicos pueden ser integrados en un componente de propósito general que ofrezca una funcionalidad más compleja.

- **Mantenibilidad:** el hecho de que cada componente pueda ser dividido en otros componentes que realicen una función específica favorece el testeo y la documentación de los mismos.

- **Separación de responsabilidades:** si cada componente tiene la responsabilidad de resolver un problema en concreto su implementación y evolución serán más sencillas.

Los puntos anteriores pueden resumirse en la frase: *"El secreto para construir grandes cosas de forma eficiente es, por lo general, evitar construirlas. En su lugar, compón esa gran cosa a partir de piezas más pequeñas y concentradas"* [11].

2.2.2. Tecnologías populares

Actualmente existen tres herramientas que destacan por su gran acogida en la industria. Estas son React, Angular y Vue.

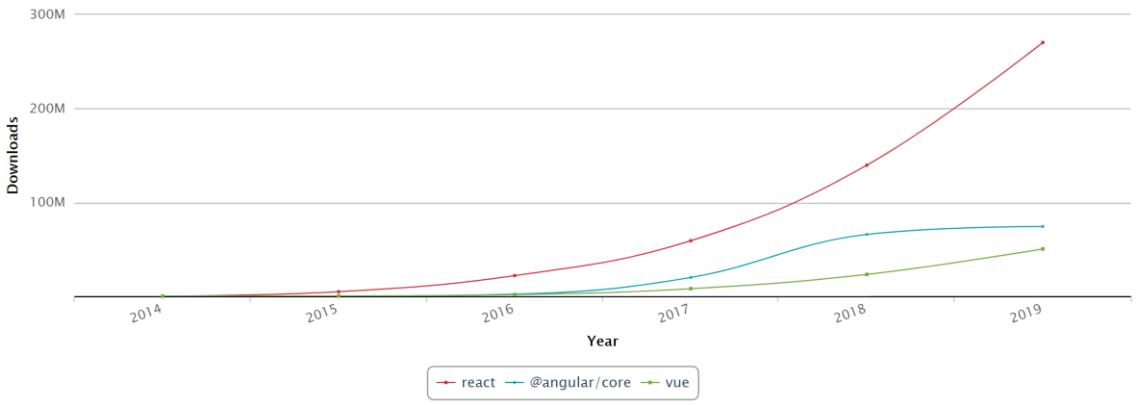

Figura 2.4: descargas anuales por librería durante los últimos cinco años [12]

Las herramientas anteriores tienen grandes ecosistemas y documentación abundante, lo que hace que aprender a utilizarlas sea relativamente sencillo. De la misma forma, cada herramienta cuenta con una serie de ventajas e inconvenientes derivados de las decisiones técnicas o los compromisos tomados por sus respectivos equipos de desarrollo [13] pero las tres presentan un gran problema que resulta especialmente relevante a la hora de desarrollar aplicaciones web de cierta envergadura, y es que, debido a que cada herramienta propone un modelo de componente diferente al del resto, los componentes desarrollados en cada una de las distintas tecnologías **no son interoperables** [14].

Esto implica, por un lado, que migrar un proyecto a otra tecnología, pese a suponer esta una ventaja competitiva, puede no ser viable debido al coste derivado de la necesidad de reescribir cada uno de los componentes y, por otro lado, la imposibilidad de compartir componentes entre distintos proyectos desarrollados en diferentes tecnologías.

2.2.3. *Web Components v1*

La web utiliza documentos escritos en *HyperText Markup Language* (HTML), para definir la estructura y el contenido de la página, también referida como documento. Este lenguaje de marcado proporciona una gran cantidad de etiquetas o elementos que constituyen los componentes básicos a partir de los cuales construir un documento HTML [15].

La especificación *Web Components v1* consiste en una serie de características web que permiten crear elementos HTML personalizados y ofrecen mecanismos para su encapsulación [16].

Los dos grandes bloques que conforman la especificación son los ***Custom Elements v1***, que definen un modelo de componentes de bajo nivel basado en estándares web, y el ***Shadow DOM v1,*** un mecanismo por el cual los componentes pueden encapsular su apariencia y comportamiento. Un elemento que utilice *Shadow DOM* ocultará los detalles de su implementación en un fragmento de documento [17] denominado *ShadowRoot* [18]. Los contenidos de cada fragmento no estarán conectados directamente con el resto de la página, lo cual implica que las reglas de estilo definidas por ese componente no tendrán efecto fuera de él de la misma forma que no se verá afectado por las definidas a nivel global. Además, la única forma de interactuar con estos componentes desde JavaScript es a través de la interfaz de programación (API) definida por su desarrollador.

Gracias a estas dos especificaciones, es posible desarrollar componentes basados en estándares los cuales, por tanto, deberían poder ser utilizados por cualquier herramienta enfocada al desarrollo web, sin hacer distinciones entre estos elementos y cualquier otro elemento HTML definido en el estándar [19].

Es importante tener en cuenta que los *Web Components* no son un *framework* y, por tanto, no cubren todos los aspectos del desarrollo web. Su función es la de proveer los bloques básicos para construir aplicaciones, a partir de los cuales se pueda implementar la funcionalidad que se desee, como el manejo del estado de los componentes o un sistema de renderizado basado en plantillas declarativas [20].

Así pues, han surgido diversas librerías basadas en este estándar para definir su modelo de componentes, incorporando las ventajas que de ello derivan y aportando funcionalidad adicional útil para el desarrollador.

Durante el desarrollo de este proyecto se hará uso de ***LitElement*** [21], anteriormente conocida como ***Polymer,*** como se verá más adelante***.*** Existen otras alternativas como ***HyperHTML*** [22] o ***SetenciIJS*** [23].

```
import { html, LitElement } from 'lit-element';
import './components';
class MyApp extends LitElement {
    render() {
        return html`
            <my-app-layout>
                <my-header></my-header>
                <my-navigation></my-navigation>
                <my-content></my-content>
            </my-app-layout>
        `;
    }
}
customElements.define('my-app', MyApp);
```

Figura 2.5: ejemplo de componente implementado con LitElement

```
<body>
    <my-app></my-app>
    <script src="my-app.js"></script>
</body>
```

Figura 2.6: ejemplo de uso del componente definido en la figura 2.5

La figura 2.5 muestra cómo se definiría el mismo componente de la figura 2.3 utilizando *LitElement* mientras que la figura 2.6 ejemplifica su uso como si de cualquier elemento HTML se tratase.

2.3. Módulos en JavaScript

Inicialmente los programas de JavaScript eran relativamente pequeños, consistían de unas pocas instrucciones para dotar a la página web de cierta interactividad. A medida que los requisitos funcionales fueron aumentando y, principalmente, como consecuencia de la introducción de este lenguaje en entornos de servidor y la proliferación de aplicaciones web basadas en el modelo de *Single Page Application*, se da la necesidad de contar con un mecanismo capaz de dividir el programa en módulos separados que puedan interactuar entre sí [25].

2.3.1. Formatos de módulos estandarizados por la industria

Dada esta nueva necesidad y a falta de una solución estándar implementada en el propio lenguaje, surgen varios formatos de módulos que pueden ser consumidos por librerías de JavaScript o empleados en ciertos entornos de ejecución de JavaScript, como Node.js [26].

Tal fue el grado de adopción de estos formatos de módulos que se convirtieron en estándares de facto, contando, algunos de ellos, incluso con propia especificación. Es el caso de CommonJS [27], el formato de módulos empleado por Node.js o AMD (*Asynchronous Module Definition)* [28], un formato de módulos que ha sido altamente

relevante en el desarrollo de aplicaciones web y que continúa utilizándose a día de hoy.

Esto ha supuesto nuevas oportunidades y ha motivado la especificación de un formato de módulos estándar para el lenguaje, pero también ha introducido una fuente de complejidad adicional en el desarrollo de aplicaciones web: la necesidad de consumir módulos definidos empleando distintos formatos.

2.3.2. Formato de módulos estandarizado por el lenguaje

Los módulos de ECMAScript, la especificación del lenguaje JavaScript, fueron introducidos en la versión del 2015, también conocida como ECMAScript6, ES6 o ES2015 [29]. Es común ver referencias a este formato de módulos por el nombre de *ES Modules* o ESM.

Esta especificación define la sintaxis tanto como para exportar variables declaradas en un módulo como para importar las mismas desde otro, así como las reglas que deben cumplir los distintos motores de JS a la hora de dar soporte a este formato de módulos en sus respectivas plataformas.

A día de hoy este formato es soportado por las últimas versiones de los navegadores web más populares, lo que supone un 91.23% de los usuarios globales de la web como muestra la figura 2.7:

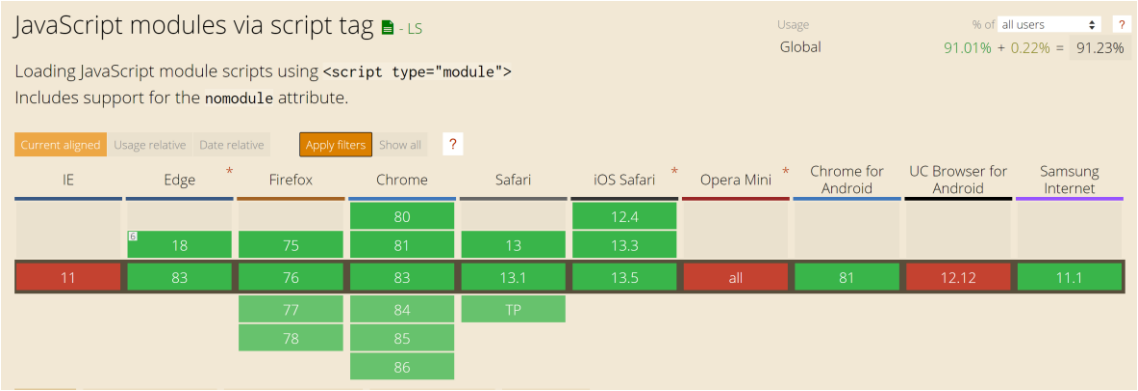

Figura 2.7: tabla de soporte de navegadores para módulos ECMAScript [30]

La especificación inicial contempla un modelo de módulos estático, es decir, cada módulo debe especificar sus dependencias al comienzo del mismo y su ejecución no podrá comenzar hasta que estas hayan sido descargadas y ejecutadas.

En una aplicación web donde el contenido a presentar y, por tanto, los módulos a consumir, dependen de la interacción del usuario, es necesario un sistema de módulos que permita la carga dinámica y bajo demanda de los mismos, con tal de no afectar negativamente al rendimiento de la aplicación empleando recursos en descargar y ejecutar módulos que no van a ser inmediatamente utilizados. Este problema se aborda en la propuesta para añadir la sintaxis *import(specifier)* al lenguaje [31], la cual da la posibilidad de cargar y consumir un módulo de forma dinámica. Esta propuesta ha sido aprobada y se incluirá en la especificación del lenguaje ES2020 [32].

A pesar de que esa versión de la especificación todavía no ha entrado en vigor, actualmente la carga dinámica de módulos ES, está soportada por los navegadores que suponen el 88.77% del uso global de la web [33].

2.3.3. Empaquetadores

Con la introducción de formatos de módulo no estándar surgen herramientas llamadas empaquetadores o *bundlers.* Estas herramientas, permitían, inicialmente, empaquetar los módulos consumidos por una aplicación, independientemente del formato en el que estuviesen definidos, en uno o más archivos denominados *bundles*, los cuales, en función del formato de salida especificado, podrían ser consumidos por otras aplicaciones o entornos de ejecución de JavaScript. Un ejemplo de estos primeros empaquetadores es Browserify [34], el cual permitía y permite ejecutar módulos CommonJS en el navegador.

La necesidad de utilizar estas librerías impone un proceso de construcción en el flujo de trabajo del desarrollo de aplicaciones web. Ya no basta con escribir código en archivos JS e incluirlos en la página. Estos archivos deberán ser analizados por un empaquetador que genere el código necesario para su ejecución. A medida que este proceso de construcción se hace prácticamente obligatorio si se desea tener la capacidad de consumir librerías de terceros, surgen nuevos empaquetadores que no sólo permiten la interoperabilidad entre distintos formatos de módulos, sino que, además, aplican transformaciones en el código fuente con tal de optimizarlo o soportar casos de usos que de otra forma no hubieran sido posibles, como, por ejemplo, incluir hojas de estilo en el paquete final.

Los dos empaquetadores más populares a día de hoy son Webpack [35] y Rollup.js [36].

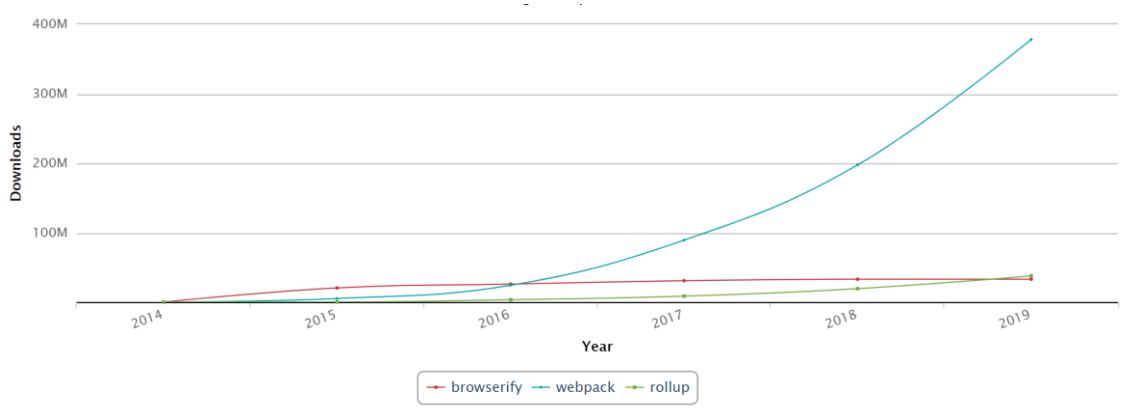

Figura 2.8: descargas anuales por empaquetador durante los últimos cinco años [37]

Ambas herramientas vienen a aportar la misma funcionalidad, extensible vía el uso de complementos, aunque con unas diferencias notables:

- **Compatibilidad**: los dos empaquetadores permiten consumir módulos definidos en cualquier formato y generar paquetes en el formato de nuestra elección. Webpack no soporta la emisión de módulos ES en su versión estable actual.

- ***Code-splitting:*** esta característica permite la división del código en distintos paquetes que podrán ser cargados bajo demanda y en paralelo, mejorando así la experiencia del usuario final de la aplicación. Rollup únicamente permite aplicar esta técnica al generar paquetes en formatos con soporte para la carga dinámica de módulos mientras que Webpack utiliza un entorno de ejecución personalizado en el que carga módulos definidos en su propio formato no estándar.

- ***Tree-shaking*:** este término, acuñado y popularizado por Rollup e introducido posteriormente en Webpack consiste en la eliminación de código muerto, esto es, el código no utilizado por la aplicación no será incluido en el paquete generado.

- **Resolución de dependencias:** debido a la falta de un formato de módulo estándar la mayoría de librerías están distribuidas para ser consumidas empleando el algoritmo de resolución de dependencias de Node [38]. Estas herramientas aplican el mismo algoritmo para localizar y empaquetar librerías de terceros.

- **Inclusión de archivos estáticos:** si bien la finalidad principal de estos sistemas es la de empaquetar módulos JavaScript escritos en distintos formatos, las posibilidades que brinda disponer de un proceso de construcción para la aplicación han hecho que este se utilice para incluir todo tipo de archivos estáticos necesarios para el funcionamiento de una aplicación web en el paquete final, como pueden ser hojas de estilo, fuentes o imágenes.

- **Caché de larga duración:** ambas herramientas son capaces de incluir un identificador en el nombre de los paquetes generados. Este identificador es un hash generado a partir del contenido final del paquete, por lo tanto, cambiará cada vez que se modifique el código fuente de este o alguna de sus dependencias. Esto permite indicar al navegador que estos archivos son inmutables y por tanto deberían ser cacheados, guardados en su memoria, durante un tiempo muy elevado. Si no se producen cambios el usuario recibirá la versión guardada en disco, mejorando el rendimiento y la velocidad de carga de la página. Si, por el contrario, el código ha sido modificado, al haber cambiado el nombre del archivo en función de su contenido, se pedirá y cacheará la versión con el nuevo hash [39].

La gran popularidad de Webpack apreciable en la figura 2.8 y su enorme ecosistema hacen que sea el empaquetador por defecto en la mayoría de aplicaciones web. El hecho de que no soporte la emisión de módulos en formato estándar y que introduzca su propio entorno de ejecución para dar soporte a la carga dinámica de módulos hace que Rollup sea la herramienta preferida a la hora de empaquetar código para ser compartido [40].

2.3.4. *Import maps* y SystemJS

Desde la estandarización y adopción de los módulos ES en la industria la mayoría de las librerías utilizan este formato para su distribución. No obstante, siguen

especificando las dependencias entre módulos de forma que sólo pueden ser resueltas haciendo uso del algoritmo de resolución de Node, el cual se basa en convenciones y en la búsqueda recursiva de módulos en el sistema de archivos. La figura 2.9 ilustra dos formas diferente de especificar de dónde proviene un módulo.

```
1   import someDepenency from 'some-dependency';
2   import anotherDependency from '/node_modules/another-dependency/index.js';
```

Figura 2.9: especificadores de módulo

El especificador empleado en la primera línea, *some-dependency*, no puede ser directamente resuelto al no tratarse de una URL (Uniform Resource Locator) válida, estos especificadores reciben el nombre de *bare module specifiers.* El especificador empleado en la segunda línea, sin embargo, sí constituye una URL válida y el navegador es capaz de descargar el recurso referenciado por ella sin la necesidad de aplicar ningún algoritmo.

El mayor impedimento, por tanto, a la hora de implementar una aplicación web utilizando el formato de módulo ES, es la resolución de *bare module specifiers* desde el navegador de una forma eficiente [41].

Los ***Import maps*** son el mecanismo propuesto para controlar la resolución de especificadores de módulos desde el cliente, creando un mapa de especificadores a URLs. Este mecanismo no sólo soluciona el problema de la resolución de especificadores de módulos, sino que también establece las bases para la implementación de técnicas más avanzadas, como el cacheo de larga duración de archivos estáticos sin invalidación en cascada [42], es decir, de forma que el cambio en una dependencia no invalide el cache de sus consumidores.

```
1   <script type="importmap">
2       {
3         "imports": {
4           "some-module": "/node_modules/some-module/index-a2g3jiu9.js"
5         }
6       }
7   </script>
```

Figura 2.10: ejemplo de Import map

Esta propuesta todavía no ha sido aprobada pero su implementación preliminar se encuentra disponible en navegadores basados en Chromium y puede ser activada por el usuario [43]. También puede utilizarse a través del entorno de ejecución **SystemJS**, el cual permite utilizar todas las características de los módulos ES, incluyendo los *Import maps*, en cualquier navegador, siempre y cuando la aplicación sea empaquetada en formato System [44].

El hecho de emplear un formato de módulo no estándar con el fin de aprovechar las características de los módulos ES puede parecer contradictorio. La diferencia entre este formato y otros es que System no constituye una nueva propuesta, si no que funciona de forma absolutamente conforme al estándar, por lo tanto, su finalidad es la

de proveer una solución temporal a la falta de soporte de algunas características por parte de algunos navegadores.

2.3.5. CSS Modules v1

Uno de los tres pilares de la web, junto al HTML y el JS, es el CSS (*Cascading Style Sheets)* [45], otro lenguaje de marcado que permite definir la apariencia y estructura de un documento creado utilizando HTML.

Las prácticas actuales consisten en incluir estilos en un documento o *shadow root*, bien utilizando elementos de tipo *style* [46] que contienen el código CSS o cargando archivos CSS mediante elementos de tipo *link* [47]. Estos dos métodos están definidos en el estándar web, pero a día de hoy no se cuenta con ningún método estándar para incluir hojas de estilo desde módulos ES, dando la posibilidad a los desarrolladores de importar CSS en el módulo que define el componente que va a hacer uso del mismo.

Existen soluciones no estándares a este problema basadas en el uso de empaquetadores que incluyen el CSS como texto dentro del JavaScript para después crear un elemento de tipo *style* [48] o en la carga de hojas de estilo creando un *link* de forma dinámica cuando sea necesario [49].

Ambas soluciones requieren el uso de herramientas adicionales y no proporcionan un rendimiento óptimo. Por ese motivo surge la propuesta de los CSS Modules v1, una extensión de los módulos ES que permitiría la interacción sin dificultades entre estos y módulos de CSS [50] a través de ***Constructable Stylesheets***, una interfaz para crear y manejar hojas de estilo desde código JavaScript sin necesidad de elementos de estilo, lo que a su vez permite la reutilización de las mismas [51].

Ningún navegador implementa soporte para este tipo de módulos a día de hoy pero su uso es posible a través de SystemJS o emulando su comportamiento mediante complementos para los empaquetadores.

2.4. Gestores de paquetes

Los gestores de paquetes son herramientas que permiten simplificar y automatizar las tareas de mantenimiento de un paquete o proyecto. En el ecosistema de JavaScript se entiende como paquete cualquier directorio que contenga un fichero de nombre *package.json*, el cual define aspectos importantes del proyecto como son el nombre, la versión, las dependencias o el punto de entrada del programa [52].

Actualmente son tres los gestores de paquetes que predominan en la industria: NPM, Yarn y PNPM [53].

- **NPM**: publicado en el año 2010, fue el primer repositorio y gestor de paquetes de Node. Actualmente es el gestor de paquetes empleado en la mayoría de los proyectos y sigue siendo el repositorio donde se publican la mayoría de librerías de código abierto para JavaScript.

- **Yarn**: gestor de paquetes soportado por Facebook que mejora sustancialmente el tiempo de instalación de dependencias comparado con NPM. Incluye características avanzadas interesantes para proyectos de cierta magnitud, como son los *workspaces* o espacios de trabajo, los cuales permiten crear un proyecto a partir de múltiples paquetes locales que son gestionados automáticamente por Yarn.

- **PNPM**: gestor de paquetes que mejora ampliamente el tiempo de instalación necesario y el espacio de disco empleado a la hora de instalar dependencias en comparación con NPM o Yarn, esto es debido a que su enfoque es radicalmente distinto: en lugar de instalar dependencias en una carpeta local dentro de cada proyecto utiliza un caché compartido a nivel del sistema operativo. También cuenta con soporte para espacios de trabajo y con utilidades para agilizar la gestión del mismo.

Existe otro tipo de herramientas que no son gestores de paquetes como tal, sino utilidades complementarias que agilizan el flujo de trabajo en proyectos constituidos por más de un paquete. En esta categoría destacan:

- **Lerna**: ofrece una serie de comandos para trabajar en espacios de trabajo con total flexibilidad. Implementa soporte para espacios de trabajo en NPM y es compatible con los de Yarn, aunque no con los de PNPM. Entre sus capacidades más notables destaca la posibilidad de gestionar el versionado de los distintos paquetes, ver el grafo de dependencias del espacio de trabajo y consultar información sobre qué paquetes han sido modificados desde un punto determinado en la historia del proyecto [54].

- **Rush**: diseñado por Microsoft con el objetivo de proporcionar un marco de trabajo configurable para trabajar con múltiples paquetes. Consta de los comandos necesarios para trabajar en un entorno de este tipo y añade características interesantes como la posibilidad de crear comandos personalizados para utilizar durante el desarrollo o la implementación de políticas con tal de imponer estándares organizacionales sin necesidad de convenciones [55]. Destaca por ofrecer no sólo una serie de utilidades si no un modelo de trabajo en sí mismo, aunque a costa de la flexibilidad de poder definir un modelo de trabajo personalizado.

2.5. Sistemas de control de versión

Los sistemas de control de versión almacenan el histórico de cambios aplicados a los ficheros de un proyecto durante el tiempo. Para ello realizan el seguimiento de los cambios en cada fichero uno de los ficheros que forman un repositorio o proyecto de forma individual y los almacenan en una base de datos. Esto da la posibilidad de devolver un fichero a un estado anterior en el tiempo, comparar distintas versiones de un mismo archivo o saber quién realizó un cambio determinado.

Uno de los modelos de sistemas de control de versión es el distribuido, en el que cada cliente tiene una copia completa del repositorio. Bajo este modelo varias personas

pueden trabajar a la vez en el mismo proyecto y después integrar sus cambios en el repositorio central alojado en un servidor [56].

2.5.1. Git

Git es un sistema de control de versión distribuido cuya mayor diferencia con otros sistemas del mismo tipo como Subversion [57] reside en cómo modela los datos. En lugar de almacenar la información como una lista de cambios conectados asociados a un fichero, Git almacena una serie de instantáneas que se corresponden con los distintos estados del repositorio, entendiéndose por estado del repositorio el conjunto de estados individuales de cada fichero en el punto en el que el desarrollador decidió persistirlo ejecutando una operación llamada *commit*.

Otra diferencia importante respecto a otros sistemas de control de versión es que prácticamente todas las operaciones de Git se realizan en el repositorio local del usuario, y sólo se integran en el repositorio remoto cuando este lo decide, lo cual implica una mayor velocidad al no depender las operaciones de la latencia de red y la posibilidad de trabajar sin conexión a internet.

Finalmente, perder información en Git resulta realmente difícil, ya que cuenta con un sistema de integridad que imposibilita que ocurra un cambio sin que Git sea consciente de ello. Además, la mayoría de operaciones añaden datos al historial y pueden ser deshechas. Es realmente difícil realizar, de forma inconsciente, una operación destructiva e irreversible [58].

A consecuencia de estar características y otras que cubren casos de uso más avanzados la popularidad de Git se ha incrementado drásticamente en los últimos años [59].

2.5.2. Control de versión para múltiples proyectos

Cuando se plantea la división de una aplicación en múltiples piezas o proyectos encargados de realizar una función en concreto surge la duda de cómo organizar la base de código [60], para la cual existen dos respuestas:

- **Múltiples repositorios:** con este enfoque cada proyecto cuenta con su propio repositorio de Git, esto permite establecer claramente quiénes son los responsables de cada proyecto y al tener cada repositorio menor tamaño las operaciones de Git son más rápidas. Por otro lado, se pierde trazabilidad en las modificaciones que afectan a más de un proyecto al tener los distintos repositorios historias independientes.

- **Mono repositorio:** en este modelo todo el código es almacenado en el mismo repositorio. Esto permite a los desarrolladores tener una visión completa del sistema lo que hace más efectiva la reutilización de código, simplifica la gestión de dependencias y permite la refactorización a gran escala. Además, al contar con un único repositorio, es posible realizar cambios que afecten a distintos proyectos de forma atómica, es decir, en el mismo *commit*, lo cual resulta de mucha utilidad a la hora de comparar o revertir esos cambios. Este tipo de

repositorios presenta un inconveniente: cuando la historia y el tamaño de un repositorio tiene un tamaño considerable, las operaciones de Git pueden ser verdaderamente lentas, esto no debería ser una fuente de preocupación excepto en casos extremos como el mono repositorio de Windows [61] o el que contiene toda la base de código de Google [62].

3. Estado del arte en el desarrollo de aplicaciones web

3.1. **Procedimiento típico en el desarrollo de una aplicación web**

Actualmente los frameworks más populares de desarrollo web cuentan con herramientas que automatizan la mayor parte de las tareas paralelas al desarrollo de modo que el programador pueda dedicarse exclusivamente al código sin necesidad preocuparse excesivamente por aspectos tales como el empaquetado de la aplicación o la compatibilidad entre distintos formatos de módulos o incluso entre navegadores. Todas estas herramientas suelen contar, además, con utilidades para la realización de pruebas unitarias o incluso de integración y un servidor de archivos estáticos para servir la aplicación durante su implementación.

Una de estas herramientas es Angular CLI [63] diseñada para trabajar con Angular. Se trata de una herramienta de línea de comandos (*CLI tool)* que cuenta con todo lo necesario para crear un proyecto, generar código a partir de plantillas, desarrollar la aplicación, testarla y, finalmente, empaquetarla para su uso en entornos productivos. También cuenta con la capacidad de crear espacios de trabajo en los que desarrollar múltiples proyectos independientes utilizando la misma configuración.

A continuación, se ilustra cuál sería la rutina de un desarrollador web utilizando esta herramienta. La elección es casual. El mismo flujo de trabajo podría ser ilustrado con otras herramientas como *Create React App* o *Vue CLI*, pues la funcionalidad ofrecida viene a ser la misma, diferenciándose, principalmente, en el *framework* a utilizar durante el desarrollo.

A la hora de desarrollar un proyecto cualquiera de nombre "*foo"* con Angular CLI, el desarrollador empieza por generar el paquete que definirá el proyecto ejecutando el comando "*ng new foo",* el cual, tras finalizar, resulta en la creación del archivo *package.json* y en la instalación y configuración de todas las librerías necesarias para desarrollar la aplicación, como muestra la figura 2.11.

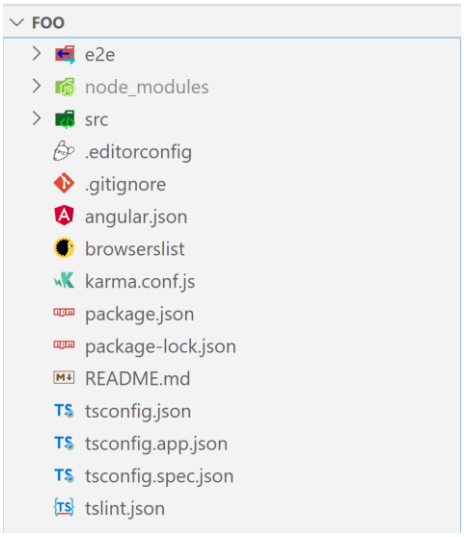

Figura 3.1 resultado de ejecutar el comando "ng new foo"

Una vez generado el proyecto, el desarrollador podrá iniciar el entorno de desarrollo con el comando *npm start*, cuyo resultado será la compilación de la aplicación y sus dependencias y la posterior puesta en marcha de un servidor de archivos estáticos en un puerto local del ordenador a través del cual es posible acceder a la aplicación, tal como ilustra la figura 2.12.

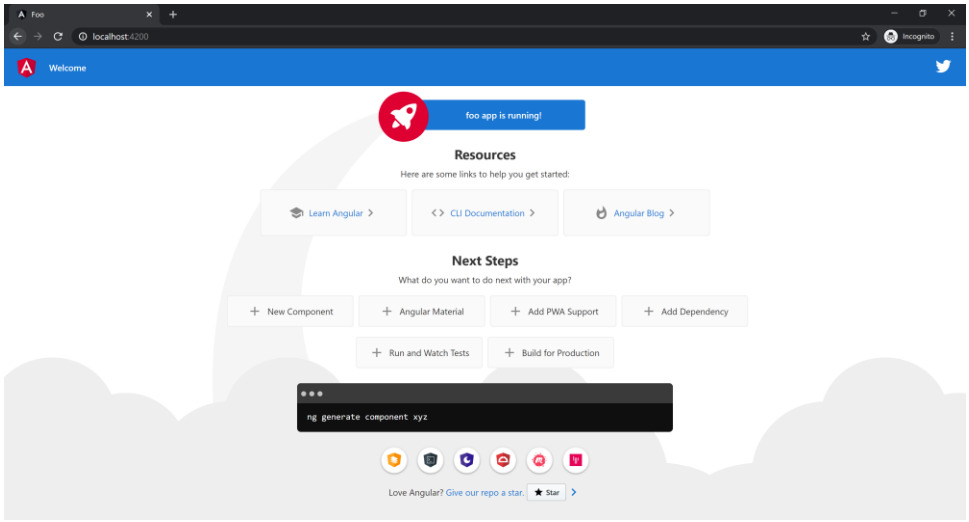

Figura 3.2: resultado de la ejecución del comando "npm start"

A partir de este punto, cualquier cambio efectuado por el desarrollador se verá reflejado en la página de forma automática ofreciendo retroalimentación de forma continuada sobre los cambios que se van produciendo.

Una vez la tarea sea finalizada, el desarrollador puede ejecutar pruebas unitarias con el comando "*npm test*" o de integración con el comando "*npm run e2e*".

En caso de que el resultado de las pruebas sea satisfactorio, el desarrollador querrá, presumiblemente, integrar sus cambios en el repositorio y, posteriormente, empaquetar la aplicación para su despliegue en producción, para lo cual emplearía el

comando “*npm run build”* generando como resultado los archivos que se muestran en la figura 2.13.

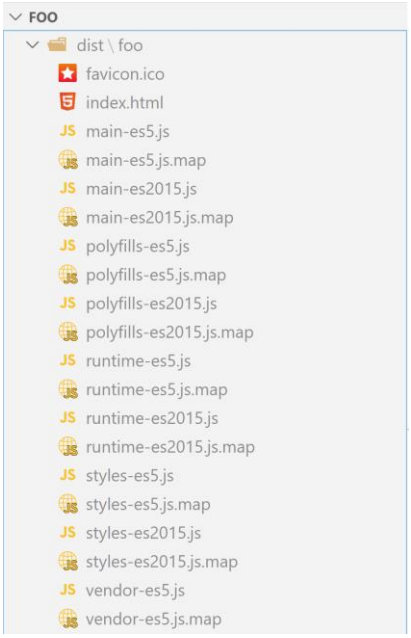

Figura 3.3: empaquetado de una aplicación con "npm run build".

3.2. **Sistemas monolíticos**

Las aplicaciones web son comúnmente desarrolladas como aplicaciones monolíticas, pues las herramientas más populares de desarrollo citadas en el apartado anterior promueven este enfoque.

Estas aplicaciones se caracterizan por constituir sistemas cerrados y autocontenidos en los que se integra toda la funcionalidad de la aplicación en una única unidad ejecutable [64]. Es por esto que la arquitectura de estos sistemas a menudo resulta simple (aunque no por ello deficiente) y su despliegue no suele suponer mayor dificultad.

Esta sencillez resulta positiva en el desarrollo de aplicaciones de complejidad reducida, pero a medida que estas aplicaciones crecen y evolucionan, la falta de flexibilidad que supone la imposibilidad de gestionar cada parte de la aplicación de forma independiente implica la aparición o acentuación de los siguientes problemas:

- **Degradación de la arquitectura:** a medida que una aplicación evoluciona su arquitectura tiende a degradarse, por distintos factores, hasta convertirse en una “gran bola de lodo” [65]. Así, las barreras autoimpuestas [66] entre las distintas partes de la aplicación acaban cediendo, dejando estas de tener una separación lógica, y la información importante acaba dispersa entre todos los módulos de la aplicación, haciendo necesario y a la vez, casi imposible, que los desarrolladores tengan un conocimiento del sistema prácticamente en su totalidad.

- **Despliegues completos:** al integrar las aplicaciones monolíticas toda su funcionalidad en una única unidad, cada vez que se desee desplegar un cambio efectuado, la aplicación en su totalidad deberá ser empaquetada y desplegada. Esto supone la posibilidad de bloqueos, pues se hace necesario esperar a que toda la funcionalidad que se desee desplegar esté finalizada, aunque sea independiente. Otro inconveniente derivado de desplegar a la vez funcionalidad no relacionada es que, debido a la mayor superficie del despliegue, aumenta el riesgo de introducir errores y la dificultad de identificar qué cambio produjo qué error. La necesitad de construir la aplicación en su totalidad ante cada cambio también introduce tiempos de construcción mayores a medida que el tamaño de esta aumenta.

- **Dependencias:** debido a que la aplicación se construye como un todo, no es posible tener distintas versiones de una misma dependencia para emplear desde distintos módulos de la aplicación. Además, las interdependencias entre distintas partes del proyecto hacen que la introducción de cambios en la interfaz de un módulo pueda afectar a otros de forma negativa, introduciendo errores, ya que evaluar el alcance de un cambio es difícil y propenso a errores cuando no se cuenta con información detallada sobre el grafo de dependencias de la aplicación.

- **Duplicación de código:** con el aumento del tamaño de la aplicación y la degradación de su arquitectura, la falta de visibilidad del sistema y sus componentes aumenta la posibilidad de duplicar funcionalidad ya existente en otra parte de la aplicación.

- **Estancamiento tecnológico:** el hecho de no poder emplear versiones de distintas librerías en distintas partes de la aplicación hace que migrar una tecnología a una nueva versión o sustituirla por otra diferente deba hacerse en la totalidad de la aplicación al mismo tiempo y no de forma progresiva. Además, el riesgo puede resultar alto al no tener una visión clara de todos los consumidores de una librería que deben ser actualizados para emplear una nueva versión de la misma. Por estos motivos, es frecuente que la introducción de mejoras tecnológicas sea un proceso costoso, duradero y propenso a producir errores.

Esto no quiere decir que todos los sistemas monolíticos deban, necesariamente, seguir esta evolución. Los problemas citados anteriormente pueden ser prevenidos o mitigados prestando un especial interés a la arquitectura de la aplicación y empleando técnicas como el uso de *feature flags* [67], conmutadores que permiten decidir, en ejecución, qué funcionalidad activar, o herramientas de refactorización a gran escala como *codemod* [68]. No obstante, se requiere de esfuerzo y dedicación constante por parte del equipo de desarrollo a fin de corregir la tendencia de este tipo de sistemas a la decadencia.

3.3. *Micro Frontends*

3.3.1. Qué son los *Micro Frontends*

Las arquitecturas de microservicios [69], basadas en componer una aplicación a partir de una colección de servicios desarrollados independientemente, han demostrado ser eficaces a la hora de resolver muchos de los problemas asociados con los sistemas monolíticos en la parte del servidor. Esta arquitectura goza de gran acogida en la industria, pero la parte de cliente de las aplicaciones web sigue siendo, mayoritariamente, desarrollada de forma monolítica.

El concepto *Micro Frontends* hace referencia a un estilo arquitectónico similar al de los microservicios, pero haciendo enfoque en la parte de cliente, de forma que la aplicación web esté formada por varias aplicaciones o módulos con un ciclo de desarrollo y evolución diferente que se componen en una sola aplicación final [70], preferiblemente en tiempo de ejecución.

El hecho de desarrollar distintas partes de una aplicación como si se tratase de aplicaciones independientes presenta una serie de beneficios altamente relevantes en el proceso de desarrollo:

- **Simplicidad:** cada uno de los *microfrontends* que forme parte de la aplicación ofrece una funcionalidad específica y, por tanto, el tamaño de su código fuente va a ser reducido. Trabajar con bases de código de menor tamaño hace que el proceso de desarrollo sea más simple y sencillo. Los desarrolladores no necesitarán conocer el sistema en su totalidad sino únicamente la parte a desarrollar. La separación lógica entre distintas partes de la aplicación será más fácil de definir y más difícil de romper, pues el alcance de cada *microfrontend* estará acotado por la funcionalidad que proporcione. Todo esto mitigará, en gran medida, la tendencia natural de la arquitectura a la degradación.

- **Despliegues independientes:** la capacidad de cada *microfrontend* de ser desplegado como una sola aplicación, independiente del resto, reduce la superficie de cada uno de los despliegues, reduciendo así el riesgo asociado. Por el mismo motivo se evita la aparición de bloqueos entre distintas partes de la aplicación. Cada unidad funcional debería poder desplegarse en el momento en que esté listo para entrar en producción independientemente del estado del resto.

- **Evolución independiente:** la posibilidad de desarrollar cada aplicación de forma independiente hace que se puedan tomar decisiones en cuanto a la evolución de esa parte sin afectar al resto, esto permite emplear diferentes versiones de una misma dependencia entre distintos *microfrontends* e incluso migrar una parte de la aplicación a otra tecnología sin afectar al resto, por lo cual, se mitiga el peligro de actualizar la aplicación en su totalidad y el requiere menos tiempo y esfuerzo.

- **Trazabilidad:** al constituir cada parte funcional de la aplicación una aplicación en sí misma, es posible analizar las dependencias entre distintos *microfrontends,* ya estas han de configurarse de forma explícita. Esto permite conocer el alcance de los cambios realizados en módulos compartidos de la aplicación reduciendo así la posibilidad de introducir modificaciones que

podrían, potencialmente, introducir errores en otras partes de la aplicación de forma no intencionada.

Adoptar una arquitectura basada en *Micro Frontends* supone un coste. El hecho de necesitar integrar cada aplicación de forma independiente en tiempo de ejecución introduce una fuente de complejidad en la arquitectura global del sistema, como ocurre en cualquier sistema distribuido. Será necesario aportar soluciones técnicas para:

- Establecer mecanismos para **compartir funcionalidad**, evitando la duplicación de código y dependencias en tiempo de ejecución.
- Definir un **modelo de comunicación** de modo que las distintas aplicaciones puedan interoperar.
- Contar con la **infraestructura** adecuada para ubicar y consumir los distintos *microfrontends* en tiempo de ejecución.

Finalmente, como resultado de separar el sistema en partes independientes, habrá de hacer frente a la nueva complejidad organizativa consecuente de la falta de un control centralizado.

3.3.2. Soluciones existentes

Existe una gran variedad de técnicas a la hora de implementar arquitecturas basadas en *Microfrontends* atendiendo al punto en el que se combinan las distintas aplicaciones dando la apariencia de un único sistema y en cómo lo hacen:

- **Integración durante el empaquetado:** una de las técnicas más sencillas consiste en integrar los distintos *microfrontends* en el propio proceso de construcción de la aplicación. Para ello se crea una aplicación o proyecto que define y consume el resto como dependencias, las cuales son empaquetadas durante el proceso de despliegue. En la figura 3.4 se muestra el proyecto raíz de una aplicación que incluye como dependencias distintas *microfrontends*. Este enfoque es el que requiere de menor infraestructura y coordinación, pero, a pesar de permitir el desarrollo independiente de cada *microfrontend,* resulta subóptimo debido a que introduce la necesidad de desplegar la aplicación al completo después de un cambio en cualquiera de sus componentes, perdiendo así la posibilidad de hacer despliegues independientes.

```
{
  "name": "@app/root",
  "dependencies": {
    "@app/microfrontend-1": "0.0.0",
    "@app/microfrontend-2": "0.0.0",
    "@app/microfrontend-3": "0.0.0"
  }
}
```

Figura 3.4: integración durante el empaquetado.

- **Integración en el servidor:** otra técnica de desarrollo de *microfrontends* consiste en emplear un servidor capaz de generar el contenido de la página requerida por el cliente combinando distintos *microfrontends*. Para llevarlo a cabo suelen emplearse tecnologías como *Server Side Includes (SSI)* [71], disponible en la mayoría de servidores de aplicaciones web, que permiten declarar en la propia página fragmentos que deben ser incluidos desde el mismo u otro servidor. La figura 3.5 ilustra el uso de la directiva *include* de *SSI* que permite insertar el contenido de un documento en otro. Esta técnica sólo es aplicable en aplicaciones multi página pues es el servidor el encargado de construir el contenido ante cada petición, además, tener que incluir distintos fragmentos desde posiblemente distintos orígenes antes de presentar la página puede introducir latencia empeorando así el rendimiento de la aplicación.

```
<body>
  <!--#include virtual="https://my-app.com/microfrontend-1.html" -->
  <!--#include virtual="https://my-app.com/microfrontend-2.html" -->
  <!--#include virtual="https://my-app.com/microfrontend-3.html" -->
</body>
```

Figura 3.5: integración mediante SSI

- **Integración en el cliente mediante *iframes*:** para integrar distintos *microfrontends* en tiempo de ejecución desde el cliente es posible cargar cada una de las aplicaciones en un *iframe,* un elemento HTML que permite incrustar una página dentro de otra, como se ilustra en la figura 3.6. El uso de *iframes* incorpora nuevas fuentes de complejidad en el desarrollo, pues al tratarse de contextos aislados dentro una página la comunicación es más difícil de solucionar y además requieren adoptar las correctas medidas de seguridad para no hacer la página vulnerable a ataques relacionados con el uso de estos elementos.

```
<body>
  <iframe src="https://my-app.com/microfrontend-1.html"></iframe>
  <iframe src="https://my-app.com/microfrontend-2.html"></iframe>
  <iframe src="https://my-app.com/microfrontend-3.html"></iframe>
</body>
```

Figura 3.6: integración mediante iframes

- **Integración en el cliente mediante JavaScript:** la técnica de integración más flexible consiste en integrar los distintos *microfrontends* mediante JavaScript en el cliente web. Para ello se ha de definir e implementar la lógica necesaria para cargar y renderizar cada una de las aplicaciones desde el cliente. Al tratarse de un sistema personalizado, da la flexibilidad para realizar la integración como se desee, sería posible, por tanto, utilizar un modelo de integración basado en *Web Components,* en el que cada *microfrontend* esté definido como un componente web como se muestra en la figura 3.7, o crear un sistema personalizado como es el caso de la herramienta *single-spa* [72], que permite

integrar en tiempo de ejecución distintas aplicaciones implementadas en diferentes tecnologías a través de un sistema de *microfrontends* definido por los autores de la librería.

```
<body>
  <app-microfrontend-1></app-microfrontend-1>
  <app-microfrontend-2></app-microfrontend-2>
  <app-microfrontend-3></app-microfrontend-3>
  <script src="https://my-app.com/microfrontend-1.js"></script>
  <script src="https://my-app.com/microfrontend-2.js"></script>
  <script src="https://my-app.com/microfrontend-3.js"></script>
</body>
```

Figura 3.7: integración mediante Web Components

3.4. Análisis del problema

Analizados los aspectos que intervienen en el desarrollo web en la actualidad y las distintas opciones disponibles en cuanto a tecnología y modelos arquitectónicos, es posible identificar qué aspectos resultan positivos y enriquecedores para el desarrollador y cuáles son mejorables o, directamente, constituyen una fuente de problemas que, en muchas ocasiones, son inevitables y deberán ser adecuadamente resueltos.

3.4.1. Crítica al desarrollo web en la actualidad

Uno de los aspectos más positivos del desarrollo web en la actualidad es la comodidad. La posibilidad de crear y configurar una aplicación que será funcional desde un primer momento ejecutando una simple instrucción desde la línea de comandos es un ejemplo de ello.

Las herramientas descritas en el capítulo 3.1 cuentan con una serie de utilidades que facilitan, en gran medida, el trabajo del desarrollador. Haciendo uso de estas herramientas, se evita, en la mayoría de los casos, tener que prestar atención a aspectos tales como la interoperabilidad entre distintos formatos de módulos, el empaquetado de la aplicación o incluso la implementación de una estrategia de caché eficiente.

Los aspectos negativos o mejorables se ubican, por tanto, en los aspectos no abordados, o abordados parcialmente, por esas herramientas:

- **Cantidad de opciones disponibles y variabilidad del entorno:** el gran número de tecnologías disponibles y la rápida evolución del entorno hacen que la elección de una herramienta en concreto resulte complicada, especialmente si se desea optar por una solución que siga siendo relevante en el futuro.

- **Interoperabilidad entre tecnologías:** las herramientas citadas anteriormente han sido concebidas con la intención de facilitar el desarrollo de aplicaciones con una tecnología determinada. Puede darse la situación de que una

organización desee compartir componentes entre distintos proyectos implementados utilizando modelos diferentes. También es posible que una organización determinada desee migrar parte de su aplicación a otra tecnología de forma progresiva. En estos casos la interoperabilidad es crucial y las herramientas actuales no cuentan con una solución a este problema.

- **Arquitectura predominantemente monolítica:** el concepto de *Micro Frontends* es reciente. El término apareció en los medios de divulgación a finales de 2016 [73] y ha ido adquiriendo popularidad desde entonces, no obstante, son pocas las herramientas que facilitan desarrollar una arquitectura de estas características, especialmente en el caso de SPAs.

  El enfoque promovido por las herramientas más populares de la industria, indirectamente, al no dar soporte directo a otros enfoques, es el monolítico. Cabe mencionar que Angular permite crear espacios de trabajo en un mono repositorio y realizar la integración de las distintas aplicaciones en tiempo de empaquetado, no obstante, esta no es su finalidad principal y por lo tanto el resultado es subóptimo. Si se desea adoptar un sistema diferente, que cuente con todas las ventajas de los *Micro Frontends,* es necesario adoptar los modelos propuestos por las pocas alternativas disponibles o desarrollar uno propio, lo cual resulta costoso e implica la resolución de no pocos problemas con tal de poder competir con las herramientas predominantes en el ecosistema.

3.4.2. Alternativa

Como alternativa a las soluciones habituales del desarrollo web, se plantea el desarrollo de un sistema tecnológicamente agnóstico, que cuente con las ventajas y la facilidad de uso que ofrecen las herramientas comúnmente utilizadas en el desarrollo de SPAs, aportando además la flexibilidad y las utilidades necesarias para aplicar un enfoque arquitectónico basado en *Micro Frontends*, prestando especial interés a las necesidades de infraestructura.

4. Consideraciones técnicas

Durante el desarrollo de la solución se harán uso de distintas técnicas y tecnologías relacionadas con distintos aspectos del sistema.

4.1. Aspectos base

A continuación de describen las consideraciones tomadas a la hora de definir los aspectos principales del sistema.

4.1.1. Modelo arquitectónico

Como modelo arquitectónico se utilizará el de los *Micro Frontends*, y la integración de los mismos se realizará en el cliente mediante JavaScript y *Web Components*. Este modelo de integración permite beneficiarse de todas las ventajas de los *Micro Frontends* en contraposición a, por ejemplo, la integración en el empaquetado, sin introducir la complejidad derivada del uso de *iframes* y sin penalizar el rendimiento de la aplicación siempre y cuando se resuelva el problema de la duplicidad de dependencias entre los distintos *microfrontends.*

4.1.2. Control de versiones

El sistema de control de versiones a utilizar será Git, y, dado que será necesario gestionar varios proyectos debido a la arquitectura elegida, se opta por tener un único mono repositorio en lugar de crear un repositorio para cada proyecto. El uso de un mono repositorio facilita en gran medida la gestión de dependencias entre distintos proyectos y, lo que es más importante, permite realizar cambios en varios proyectos de forma atómica.

4.1.3. Gestor de paquetes

A la hora de elegir un gestor de paquetes NPM queda automáticamente descartado por su falta de soporte para espacios de trabajo. Se hará uso de Yarn en lugar de PNPM debido a que el último presenta incompatibilidades con algunas herramientas debido a su enfoque característico a la hora de instalar dependencias [74].

4.1.4. Gestor de mono repositorio

Para gestionar el versionado y extraer información de los distintos paquetes del mono repositorio se hará uso de Lerna debido a que cuenta con las herramientas y la flexibilidad adecuada para ser integrado en el sistema, a diferencia de Rush, que implicaría adaptar el sistema a la herramienta.

4.1.5. Modelo de componente

El modelo de los distintos componentes que formarán parte de la aplicación final será definido mediante *Web Components*. La elección de esta tecnología está motivada en primer lugar por tratarse de una solución estándar y, por tanto, interoperable y a

prueba de futuro. En segundo lugar, los mecanismos de encapsulación que ofrece el *Shadow DOM* resultan ideales a la hora de desarrollar una arquitectura modular. En cuanto a la librería con la que desarrollar estos componentes, se hará uso de LitElement, en primer lugar por tratarse de un complemento sencillo y eficaz que cubre los casos de uso no abordados por el estándar sin introducir cambios sustanciales a la hora de desarrollar y, en segundo lugar, por contar con el respaldo de Google [24].

4.1.6. Formato de módulos

Como formato de módulos cabe distinguir entre el formato a emplear para escribir el código fuente y el que será emitido tras el proceso de empaquetado. Durante el desarrollo se utilizará el formato de módulo definido en EcmaScript precisamente por tratarse del formato de módulo estándar de JavaScript. Durante el empaquetado de la aplicación se transformará este formato a SystemJS, aportando así compatibilidad con navegadores que no soporten el formato de módulo estándar y dando la posibilidad de utilizar *Import maps* para ubicar los distintos *microfrontends.* Cuando los *import maps* sean estandarizados y el soporte de los distintos navegadores aceptable, podría dejar de utilizarse este formato de módulo y distribuir la aplicación empleando módulos estándar, sin realizar ninguna transformación pues sería el formato en el que está escrita la aplicación.

4.1.7. Empaquetador

Para empaquetar la aplicación se hará uso de Rollup.js debido a la facilidad de implementar complementos que serán necesarios para resolver ciertos aspectos relacionados con la infraestructura y el despliegue de la aplicación como la creación de un *import map*. Además, la imposibilidad de emitir módulos en formato estándar desde otros empaquetadores como *Webpack* hacen que no constituyan una solución a prueba de futuro.

4.2. Paridad funcional

Con tal de contar con la misma funcionalidad ofrecida por otras herramientas de la industria se hará uso de distintas utilidades que permitan cubrir esos aspectos.

4.2.1. Carga de hojas de estilo

Se empleará la propuesta de estándar de módulos CSS de modo que puedan importarse hojas de estilo desde JavaScript, se dará soporte a esta propuesta para el estándar desarrollando un complemento de Rollup.js, el cual dejaría de ser necesario una vez finalizado el proceso de estandarización.

4.2.2. Herramienta de línea de comandos

Para dotar al desarrollador de una herramienta de línea de comandos con la que ejecutar los distintos procesos necesarios para el desarrollo se empleará Yargs [75], una librería que permite crear de forma sencilla y altamente configurable, este tipo de herramientas empleando JavaScript.

4.2.3. Servidor de estáticos

El servidor de archivos estáticos a emplear durante el desarrollo se implementará con Browsersync, debido al gran abanico de posibilidades que ofrece [76], entre las que destaca una configuración total y la posibilidad de recargar la página de forma automática tras detectar cambios en los ficheros que componen la aplicación.

4.2.4. Calidad de código

La mayoría de herramientas incluyen analizadores estáticos de código que permiten detectar errores durante el desarrollo y hacer cumplir estándares de calidad y de formato del código. En este sistema se empleará Eslint [77] para analizar el código JavaScript, Stylelint [78] para el CSS y, finalmente, Prettier [79] para formatear el código de forma automática, independientemente del lenguaje en el que esté escrito. La elección de estas herramientas se basa únicamente en que constituyen el estándar de facto en la industria, cuentan con un gran ecosistema y con complementos para la integración con la mayoría de editores de código.

4.2.5. Pruebas unitarias

El soporte para la realización de pruebas unitarias se dará a través de Karma [80]. Aunque existen alternativas más populares como Jest [81], que simulan el entorno del navegador desde Node.js y por tanto son más rápidas, Karma lanza los test en un navegador web real, elegido por el usuario, reduciendo las diferencias entre el entorno de pruebas y el de producción. Ejecutar las pruebas en un entorno real da más fiabilidad a los resultados y evita problemas derivados de emular un entorno tan complejo como es el del navegador, especialmente a la hora de usar estándares recientemente aprobados como el de los *Web Components*.

4.2.6. Pruebas de integración

La posibilidad de realizar pruebas de integración será ofrecida a través de CodeceptJS [82]. CodeceptJS presenta una interfaz para definir las pruebas y cuenta con conectores para ejecutarlas empleando una gran variedad de herramientas. Por ejemplo, puede optarse por emplear el conector de WebDriverIO [83] con Selenium [84]para lanzar las pruebas en distintos navegadores o el de Puppeteer [85] para lanzar las pruebas directamente contra Google Chrome de forma más rápida. En cualquier caso, al contar todos los conectores con la misma interfaz, sería posible migrar de uno a otro simplemente cambiando la configuración, siempre y cuando el nuevo conector soporte todas las características necesarias para realizar las pruebas.

5. Solución propuesta

5.1. **Análisis**

Descripción de las distintas piezas y la relación entre ellas.

5.2. **Diseño**

Descripción detallada de cada una de las piezas que pertenecen a cada pieza arquitectónica explicando su responsabilidad.

5.3. **Implementación**

5.4. **Pruebas**

6. Caso de estudio

6.1. **Implementación**

6.2. **Evaluación**

7. Conclusiones

Conclusiones obtenidas tras la realización del trabajo y comparación entre el resultado obtenido en el apartado anterior, y los objetivos propuestos en resultados anteriores.

7.1. **Relación del trabajo desarrollado con los estudios cursados**

Identificar con cuáles de las asignaturas relación el trabajo y por qué.

8. Trabajos futuros

Posibles mejoras no abordadas y otras vías de investigación.

9. Referencias

[1] V. Collins, «The Decline Of The Native App And The Rise Of The Web App,» Forbes, 5 Abril 2019. [En línea]. Available: https://www.forbes.com/sites/victoriacollins/2019/04/05/why-you-dont-need-to-make-an-app-a-guide-for-startups-who-want-to-make-an-app/. [Último acceso: 31 Mayo 2020].

[2] HTTP Archive, «HTTP Archive,» [En línea]. Available: https://httparchive.org/reports/page-weight?start=earliest&end=latest&view=list. [Último acceso: 31 Mayo 2020].

[3] «Module Counts,» [En línea]. Available: http://www.modulecounts.com/. [Último acceso: 31 Mayo 2020].

[4] K. Ball, «The increasing nature of frontend complexity,» 30 Enero 2019. [En línea]. Available: https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae/. [Último acceso: 31 Mayo 2020].

[5] G. Kapllani, I. Khomyakov, R. Mirgalimova y A. Sillitti, «An Empirical Analysis of the Maintainability Evolution of Open Source Systems,» *IFIP Advances in Information and Communication Technology,* vol. 582, 2020.

[6] J. Wijnmaalen, C. Chen, D. Bijlsma y A. M. Oprescu, «The Relation between Software Maintainability and Issue Resolution Time: A Replication Study,» *Seminar Series on Advanced Techniques & Tools for Software Evolution,* vol. 2510, 2019.

[7] J. Posnick, «Beyond SPAs: alternative architectures for your PWA,» 14 Enero 2019. [En línea]. Available: https://developers.google.com/web/updates/2018/05/beyond-spa. [Último acceso: 6 Junio 2020].

[8] M. Wasson, «ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET,» *MSDN Magazine Issues,* vol. 28, nº 11, 2013.

[9] S. Smith, G. Warren, P. Marcano y M. Wenzel, «Choose Between Traditional Web Apps and Single Page Apps (SPAs),» 12 abril 2019. [En línea]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-

page-apps. [Último acceso: 6 Junio 2020].

[10] J. Saring, «A Guide to Component Driven Development (CDD),» 16 Junio 2019. [En línea]. Available: https://itnext.io/a-guide-to-component-driven-development-cdd-1516f65d8b55. [Último acceso: 6 Junio 2020].

[11] A. Osmani, «Components Should Be Focused, Independent, Reusable, Small & Testable (FIRST),» [En línea]. Available: https://addyosmani.com/first/. [Último acceso: 06 Junio 2020].

[12] P. Vorbach, «npm-stat,» [En línea]. Available: https://npm-stat.com/charts.html?package=react&package=vue&package=%40angular%2Fcore&from=2014-12-12&to=2019-12-19. [Último acceso: 6 Junio 2020].

[13] J. Hannah, «The Ultimate Guide to JavaScript Frameworks,» 16 Enero 2018. [En línea]. Available: https://jsreport.io/the-ultimate-guide-to-javascript-frameworks/. [Último acceso: 6 Junio 2020].

[14] S. Contreras, «Web Components: Seamlessly interoperable,» 15 Abril 2019. [En línea]. Available: https://medium.com/@sergicontre/web-components-seamlessly-interoperable-82efd6989ca4. [Último acceso: 6 Junio 2020].

[15] MDN Contributors, «HTML: Hypertext Markup Language,» MDN, 28 Mayo 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. [Último acceso: 7 Junio 2020].

[16] Google Developers, «Building Components,» Web Fundamentals, 12 Febrero 2019. [En línea]. Available: https://developers.google.com/web/fundamentals/web-components. [Último acceso: 6 Junio 2020].

[17] MDN Contributors, «DocumentFragment,» MDN, 13 Enero 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/API/DocumentFragment. [Último acceso: 7 Junio 2020].

[18] MDN Contributors, «ShadowRoot,» MDN, 23 Abril 2019. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ShadowRoot. [Último acceso: 7 Junio 2020].

[19] R. Dodson, «Custom Elements Everywhere,» [En línea]. Available: https://custom-elements-everywhere.com/. [Último acceso: 06 Junio 2020].

[20] C. Haynes, «Web Components aren't a framework replacement - they're better than that,» 18 Enero 2020. [En línea]. Available: https://lamplightdev.com/blog/2020/01/18/web-components-arent-a-framework-replacement-theyre-better-than-that/. [Último acceso: 6 Junio 2020].

[21] The Polymer Project, «Lit Element,» [En línea]. Available: https://lit-element.polymer-project.org/.

[22] A. Giammarchi, «hyper(HTML),» [En línea]. Available: https://viperhtml.js.org/hyper.html.

[23] Stencil, «Stencil,» [En línea]. Available: https://stenciljs.com/.

[24] MDN Contributors, «JavaScript modules,» MDN, 22 Abril 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules. [Último acceso: 6 Junio 2020].

[25] NodeJS Contributors, «About Node.js,» [En línea]. Available: https://nodejs.org/es/about/. [Último acceso: 6 Junio 2020].

[26] CommonJS, «CommonJS,» 9 Octubre 2011. [En línea]. Available: http://wiki.commonjs.org/wiki/CommonJS. [Último acceso: 6 Junio 2020].

[27] «AMD,» [En línea]. Available: https://requirejs.org/docs/whyamd.html#amd. [Último acceso: 6 Junio 2020].

[28] «ECMAScript® 2015 Language Specification,» Ecma International, Junio 2015. [En línea]. Available: http://www.ecma-international.org/ecma-262/6.0/#sec-modules. [Último acceso: 6 Junio 2020].

[29] Can I Use, «JavaScript modules via script tag,» Can I Use, [En línea]. Available: https://caniuse.com/#feat=es6-module. [Último acceso: 6 Junio 2020].

[30] tc39, «proposal-dynamic-import,» 20 Junio 2019. [En línea]. Available: https://github.com/tc39/proposal-dynamic-import. [Último acceso: 6 Junio 2020].

[31] B. Couriol, «ES2020's Feature Set Finalized,» Infoq, 4 Abril 2020. [En línea]. Available: https://www.infoq.com/news/2020/04/es2020-features/. [Último acceso: 6 Junio 2020].

[32] Can I Use, «JavaScript modules: dynamic import(),» Can I Use, [En línea]. Available: https://caniuse.com/#feat=es6-module-dynamic-import. [Último acceso: 6 Junio 2020].

[33] Browserify, Browserify, [En línea]. Available: http://browserify.org/. [Último acceso: 7 Junio 2020].

[34] Webpack, «Webpack,» [En línea]. Available: https://webpack.js.org/. [Último acceso: 6 Junio 2020].

[35] Rollup.js, «Rollup.js,» [En línea]. Available: https://rollupjs.org/guide/en/. [Último acceso: 7 Junio 2020].

[36] P. Vorbach, «npm-stat,» [En línea]. Available: https://npm-stat.com/charts.html?package=rollup&package=webpack&package=browserify&from=2014-12-12&to=2019-12-19. [Último acceso: 7 Junio 2020].

[37] NodeJS, «Modules,» [En línea]. Available: https://nodejs.org/api/modules.html#modules_modules. [Último acceso: 7 Junio 2020].

[38] I. Akulov, «Make use of long-term caching,» Web Fundamentals, 2 Septiembre 2019. [En línea]. Available: https://developers.google.com/web/fundamentals/performance/webpack/use-long-term-caching. [Último acceso: 7 Junio 2020].

[39] hoangbkit, «JavaScript Module Bundlers,» 20 Febrero 2020. [En línea]. Available: https://advancedweb.dev/javascript-module-bundlers. [Último acceso: 7 Junio 2020].

[40] D. Denicola, h.-g. J. Fagnani, M. Borins y K. Ueno, «Import maps,» WICG, 19 Mayo 2020. [En línea]. Available: https://github.com/WICG/import-maps#background. [Último acceso: 7 Junio 2020].

[41] P. Walton, «Cascading Cache Invalidation,» 9 Octubre 2019. [En línea]. Available: https://philipwalton.com/articles/cascading-cache-invalidation/. [Último acceso: 8 Junio 2020].

[42] «Import Maps,» Chrome Platform Status, 2 Junio 2020. [En línea]. Available: https://www.chromestatus.com/feature/5315286962012160. [Último acceso: 7 Junio 2020].

[43] G. Bedford, «SystemJS,» [En línea]. Available: https://github.com/systemjs/systemjs. [Último

acceso: 7 Junio 2020].

[44] MDN Contributors, «CSS: Cascading Style Sheets,» MDN, 6 Junio 2020. [En línea]. Available: https://developer.mozilla.org/es/docs/Web/CSS. [Último acceso: 7 Junio 2020].

[45] WHATWG , «The style element,» 5 Junio 2020. [En línea]. Available: https://html.spec.whatwg.org/multipage/semantics.html#htmlstyleelement. [Último acceso: 7 Junio 2020].

[46] WHATWG , «The link elemenet,» 5 Junio 2020. [En línea]. Available: https://html.spec.whatwg.org/multipage/semantics.html#the-link-element. [Último acceso: 7 Junio 2020].

[47] Webpack contributors, «Style Loader,» Webpack, [En línea]. Available: https://webpack.js.org/loaders/style-loader/. [Último acceso: 7 Junio 2020].

[48] S. Jehl y Z. Leatherman, «loadCSS,» Filament Group, [En línea]. Available: https://github.com/filamentgroup/loadCSS. [Último acceso: 7 Junio 2020].

[49] D. Clark, «CSS Modules V1 Explainer,» W3C, [En línea]. Available: https://w3c.github.io/webcomponents/proposals/css-modules-v1-explainer. [Último acceso: 7 Junio 2020].

[50] T. Atkins, E. Willigers y R. Zata Amni, «Constructable Stylesheet Objects,» WICG, 3 Marzo 2020. [En línea]. Available: https://wicg.github.io/construct-stylesheets/. [Último acceso: 7 Junio 2020].

[51] «npm-package.json,» npm, [En línea]. Available: https://docs.npmjs.com/files/package.json. [Último acceso: 7 Junio 2020].

[52] M. Goldwater, «An abbreviated history of JavaScript package managers,» 28 Diciembre 2019. [En línea]. Available: https://medium.com/javascript-in-plain-english/an-abbreviated-history-of-javascript-package-managers-f9797be7cf0e. [Último acceso: 13 Junio 2020].

[53] «Lerna,» [En línea]. Available: https://lerna.js.org/. [Último acceso: 13 Junio 2020].

[54] «Welcome to Rush!,» [En línea]. Available: https://rushjs.io/pages/intro/welcome/. [Último

acceso: 13 Junio 2020].

[55] S. Chacon y B. Straub, «Getting Started - About Version Control,» de *Pro Git*, Apress, 2014.

[56] Apache Software Fundation, «Apache Subversion,» [En línea]. Available: https://subversion.apache.org/. [Último acceso: 8 Junio 2020].

[57] S. Chacon y B. Straub, «Getting Started - What is Git?,» de *Pro Git*, Apress, 2014.

[58] Hugo, «StackExchange,» 21 Febrero 2012. [En línea]. Available: https://softwareengineering.stackexchange.com/a/136207. [Último acceso: 8 Junio 2020].

[59] P. Belagatti, «Microservices and Difference Between Mono Repo and Multiple Repositories,» 29 Octubre 2019. [En línea]. Available: https://dzone.com/articles/microservices-difference-between-mono-repo-and-mul. [Último acceso: 8 Junio 2020].

[60] S. Noursalehi, «Git at Scale,» 21 Febrero 2018. [En línea]. Available: https://docs.microsoft.com/en-us/azure/devops/learn/git/git-at-scale#extra-large-repos. [Último acceso: 8 Junio 2020].

[61] R. Potvin y J. Levenberg, «Why Google Stores Billions of Lines of Code in a Single Repository,» *Communications of the ACM,* vol. 59, nº 7, pp. 78-87, 2016.

[62] [En línea]. Available: https://cli.angular.io/. [Último acceso: 15 Junio 2020].

[63] R. Annett, «What is a Monolith?,» 19 Noviembre 2014. [En línea]. Available: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html. [Último acceso: 30 Junio 2020].

[64] B. Foote y J. Yoder, «Big Ball of Mud,» *Technical Report #WUCS-97-34,* vol. 29, 1999.

[65] M. Fowler, «ApplicationBoundary,» 11 Septiembre 2003. [En línea]. Available: https://martinfowler.com/bliki/ApplicationBoundary.html. [Último acceso: 2 Julio 2020].

[66] P. Hodgson, «Feature Toggles (aka Feature Flags),» 9 Octubre 2017. [En línea]. Available: https://www.martinfowler.com/articles/feature-toggles.html. [Último acceso: 2 Julio 2020].

[67] Facebook, «codemod,» [En línea]. Available: https://github.com/facebook/codemod. [Último acceso: 2 Julio 2020].

[68] C. Richardson, «What are microservices?,» [En línea]. Available: https://microservices.io/. [Último acceso: 6 Julio 2020].

[69] C. Jackson, «Micro Frontends,» 19 Junio 2019. [En línea]. Available: https://martinfowler.com/articles/micro-frontends.html. [Último acceso: 6 Julio 2020].

[70] Nginx, «Module ngx_http_ssi_module,» [En línea]. Available: http://nginx.org/en/docs/http/ngx_http_ssi_module.html. [Último acceso: 7 Julio 2020].

[71] Canopy, «single-spa,» [En línea]. Available: https://single-spa.js.org/. [Último acceso: 7 Julio 2020].

[72] M. Geers, «Micro Frontends,» [En línea]. Available: https://micro-frontends.org/. [Último acceso: 8 Julio 2020].

[73] PNPM, «pnpm does not work with <YOUR-PROJECT-HERE>?,» [En línea]. Available: https://pnpm.js.org/en/faq#pnpm-does-not-work-with-your-project-here. [Último acceso: 8 Julio 2020].

[74] J. Fagnani, «Lightning-fast templates & Web Components: lit-html & LitElement,» Google Developers, 12 Febrero 2019. [En línea]. Available: https://developers.google.com/web/updates/2019/02/lit-element-and-lit-html. [Último acceso: 6 Junio 2020].

[75] «yargs,» [En línea]. Available: http://yargs.js.org/. [Último acceso: 8 Julio 2020].

[76] «Browsersync,» [En línea]. Available: https://www.browsersync.io/. [Último acceso: 8 Julio 2020].

[77] «Eslint,» [En línea]. Available: https://eslint.org/. [Último acceso: 9 Julio 2020].

[78] «Stylelint,» [En línea]. Available: https://stylelint.io/. [Último acceso: 9 Julio 2020].

[79] «Prettier,» [En línea]. Available: https://prettier.io/. [Último acceso: 9 Julio 2020].

[80] «Karma,» [En línea]. Available: https://karma-runner.github.io/. [Último acceso: 9 Julio 2020].

[81] «Jest,» [En línea]. Available: https://jestjs.io/. [Último acceso: 9 Julio 2020].

[82] «CodeceptJS,» [En línea]. Available: https://codecept.io/. [Último acceso: 9 Julio 2020].

[83] «WebDriverIO,» [En línea]. Available: https://webdriver.io/. [Último acceso: 9 Julio 2020].

[84] «Selenium,» [En línea]. Available: https://www.selenium.dev/. [Último acceso: 9 Julio 2020].

[85] «Puppeteer,» [En línea]. Available: https://pptr.dev/. [Último acceso: 9 Julio 2020].

10. Abreviaciones

11. Anexos

Anexos si se necesita alguno.

12. Glosario