UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

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 Letelier Torres

Curso 2019 - 2020

Agradecimientos

Dedico este trabajo a mis padres y, en general, a toda mi familia, por el apoyo, la educación y las oportunidades recibidas.

A mi amigo y mentor, Pedro Hurtado, por introducirme en el mundo del desarrollo web y enseñarme muchas de las cosas que a día de hoy me definen como profesional.

A mi tutor en este trabajo, Patricio Letelier, por la confianza depositada y su trato profesional a la vez que cercano.

Finalmente, quisiera agradecer el esfuerzo de los desarrolladores que han creado y colaboran en el mantenimiento de todas las herramientas de código abierto empleadas durante la realización de este trabajo que de otro modo no habría sido posible.

Resumen

La complejidad del desarrollo web ha ido en aumento durante la última década como consecuencia, principalmente, del incremento en la cantidad de requisitos funcionales a los que se deben hacer frente.

El propósito de este trabajo es estudiar y seleccionar un conjunto de herramientas y técnicas para definir un ecosistema que permita abordar el desarrollo de aplicaciones web de forma escalable, haciendo que la complejidad del desarrollo sea estable a medida que aumenta la cantidad de funcionalidad ofrecida por la aplicación. Para ello se identificarán y estudiarán los aspectos que necesitan ser abordados a la hora de desarrollar una aplicación web y se implementará una herramienta de soporte para ese ecosistema, facilitando el desarrollo modular de aplicaciones web basado en *Micro Frontends*, dividiendo la aplicación principal en distintas aplicaciones autocontenidas pero interoperables.

Para valorar la propuesta se desarrollará un caso de estudio que consistirá en el análisis, diseño, implementación y pruebas de una aplicación de banca electrónica.

Palabras clave: desarrollo web, arquitectura de aplicaciones web, desarrollo modular, *Micro Frontends*.

Resum

La complexitat de el desenvolupament web ha anat en augment durant l'última dècada com a conseqüència, principalment, de l'increment en la quantitat de requisits funcionals als que s'han de fer front.

El propòsit d'aquest treball és estudiar i seleccionar un conjunt d'eines i tècniques per definir un ecosistema que permeti abordar el desenvolupament d'aplicacions web de forma escalable, fent que la complexitat del desenvolupament sigui estable a mesura que augmenta la quantitat de funcionalitat oferta per l'aplicació. Per a això s'identificaran i estudiaran els aspectes que necessiten ser abordats a l'hora de desenvolupar una aplicació web i s'implementarà una eina de suport per a aquest ecosistema, facilitant el desenvolupament modular d'aplicacions web basat en *Micro Frontends*, dividint l'aplicació principal en diferents aplicacions autocontingudes però interoperables.

Per valorar la proposta es desenvoluparà un cas d'estudi que consistirà en l'anàlisi, disseny, implementació i proves d'una aplicació de banca electrònica.

Paraules clau: desenvolupament web, arquitectura d'aplicacions web, desenvolupament modular, *Micro Frontends*.

Abstract

Web development complexity has raised throughout the last decade mainly due to the increase in the amount of functional requirements web applications must face.

The purpose of this work is to study and select a set of tools and techniques to define an ecosystem capable of tackling web applications development in a scalable way, making development complexity stable as the application provided functionality increases. To do this, aspects that need to be addressed when developing web applications will be identified and studied and a support tool for that ecosystem will be developed, easing modular web application development based on Micro Frontends by dividing the main application into different self-contained but interoperable applications.

For evaluating the proposal, a case study consisting in the analysis, design, implementation and testing of an online banking application will be developed.

Keywords: web development, web applications architecture, modular development, Micro Frontends.

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 mejoran, en varios aspectos, la satisfacción 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 340,2% en dispositivos de escritorio y un 1223,8% en dispositivos móviles [2].

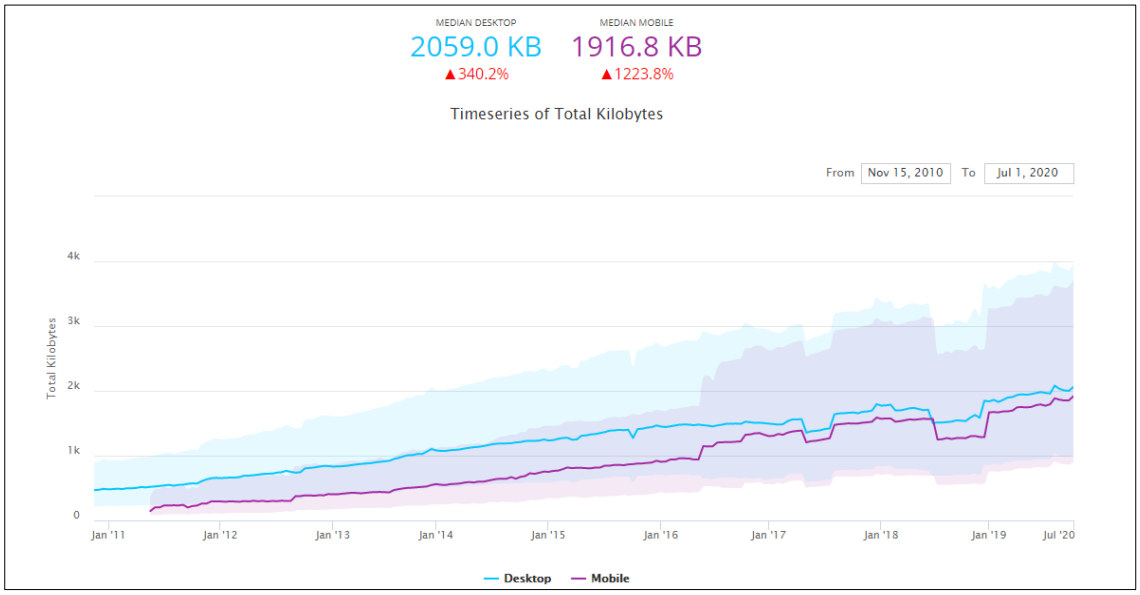

Figura 1.1: serie temporal de la descarga de Kilobytes por página [2]

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 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.

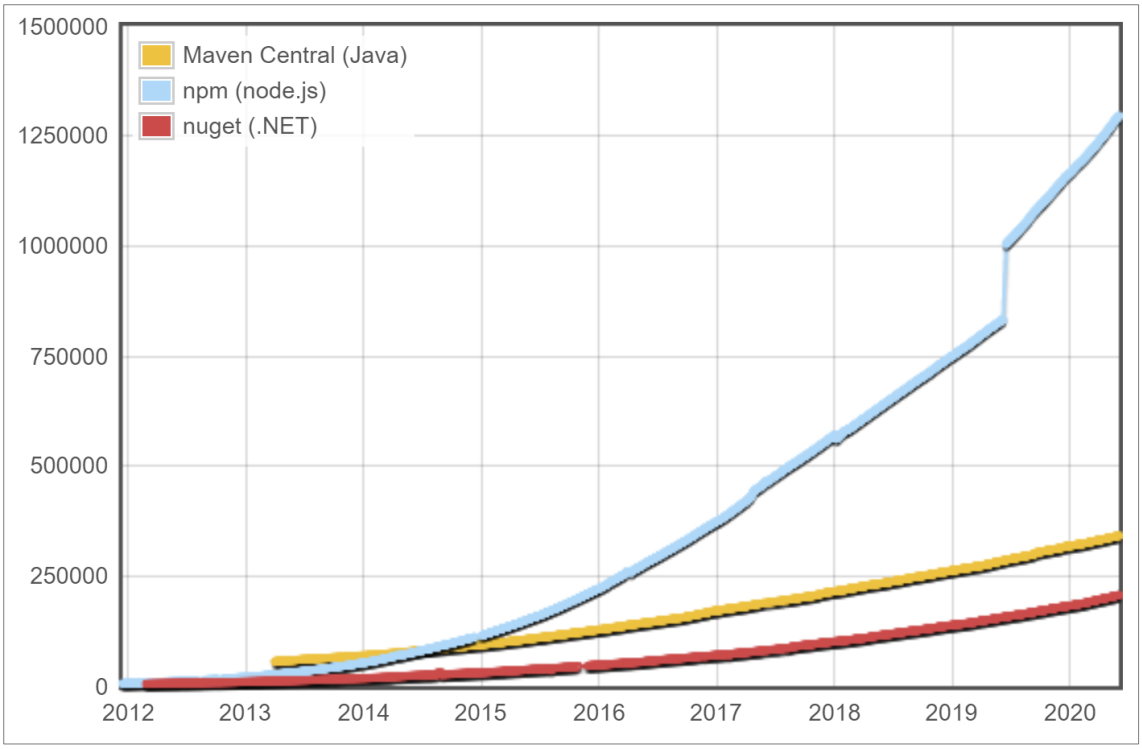

Figura 1.2: número de módulos por año por ecosistema [3]

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 presenta 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 hace una selección de herramientas y técnicas, proponiendo pautas para su empleo, con tal de definir un ecosistema 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. Paralelamente, se implementará una herramienta de apoyo para la aplicación de ese ecosistema.

1.2. Objetivos

El objetivo global de este trabajo de fin de grado (TFG) es la definición de un ecosistema que permita desarrollar una aplicación web de forma sencilla y mantenible, independientemente de la envergadura de la misma y crear una herramienta para facilitar la aplicación de dicho ecosistema. Los objetivos específicos son:

- Definir un ecosistema en el que la complejidad de la aplicación sea estable a medida que aumenta la cantidad de funcionalidad ofrecida por la misma.
- Estudiar diferentes alternativas tecnológicas a la hora de definir el ecosistema e implementar una herramienta que apoye la aplicación del mismo. Esta herramienta debe aportar, además, la funcionalidad ofrecida por otras herramientas de desarrollo más populares en la industria.
- Evaluar los resultados obtenidos desarrollando una aplicación de banca electrónica empleando el ecosistema propuesto y haciendo uso de la herramienta desarrollada para su apoyo.

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

1.3. Estructura

Una vez finalizada la introducción, el resto de la memoria se estructura de la siguiente forma:

En el capítulo 2 se presenta el estado del arte en el desarrollo de aplicaciones web en cuanto a tecnologías disponibles, realizando un resumen detallado de los aspectos a tener en cuenta a la hora de desarrollar aplicaciones web, y en cuanto a al propio proceso de desarrollo, haciendo hincapié en la forma típica de afrontar el desarrollo y presentando los modelos arquitectónicos predominantes. Al final se realiza una crítica al proceso de desarrollo web en la actualidad y se evalúan las distintas alternativas para después proponer una propia.

En el capítulo 3 se analizan todas las consideraciones técnicas que se han tenido en cuenta a la hora de desarrollar el ecosistema propuesto en cuanto a las tecnologías utilizadas y las técnicas aplicadas.

En el capítulo 4 se analizan y especifican los requisitos que la herramienta de apoyo al ecosistema debe cumplir, y se describen el diseño y los detalles de implementación más relevantes de la herramienta desarrollada con el fin de cubrir los requisitos de la solución.

En el capítulo 5 se plantea un caso de estudio a implementar con tal de poner a prueba la solución propuesta, se definen una serie de requisitos funcionales que la aplicación a desarrollar debe cumplir y se describe el proceso, la implementación y las pruebas del caso de estudio para, finalmente, evaluar la efectividad de la solución propuesta y detectar posibilidades de mejora.

Finalmente, en el capítulo 6, se presentan las conclusiones del trabajo respecto a la consecución de los objetivos definidos inicialmente, a la vez que se realiza una valoración personal y se identifican posibles vías de trabajo futuras.

2. Estado del arte en el desarrollo de aplicaciones web

Actualmente son muchos los factores que intervienen en el desarrollo de una aplicación web.

En primer lugar, es necesario decidir dónde se ubicará la lógica de la interfaz de usuario, pudiendo esta residir en la parte del servidor o en la del cliente, para, a continuación, en función de la decisión tomada, hacer frente a múltiples aspectos del desarrollo, desde cómo definir los distintos componentes de la aplicación hasta cómo gestionar el versionado de los mismos. Para ello existen multitud de tecnologías y técnicas que deberán ser evaluadas y, posteriormente, adoptadas.

En segundo lugar, surge la necesidad de definir cómo se desarrollarán y cómo interactuarán las distintas partes de la aplicación, es decir, habrá que definir su arquitectura.

En este capítulo se exponen y analizan las diferentes opciones tecnológicas y arquitectónicas disponibles para el desarrollo de aplicaciones web con la finalidad de evaluar el estado del desarrollo web en la actualidad.

2.1. **Tecnologías para desarrollo 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 que deben tenerse 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.

2.1.1. 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].

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, tal como se muestra en la figura 2.1.

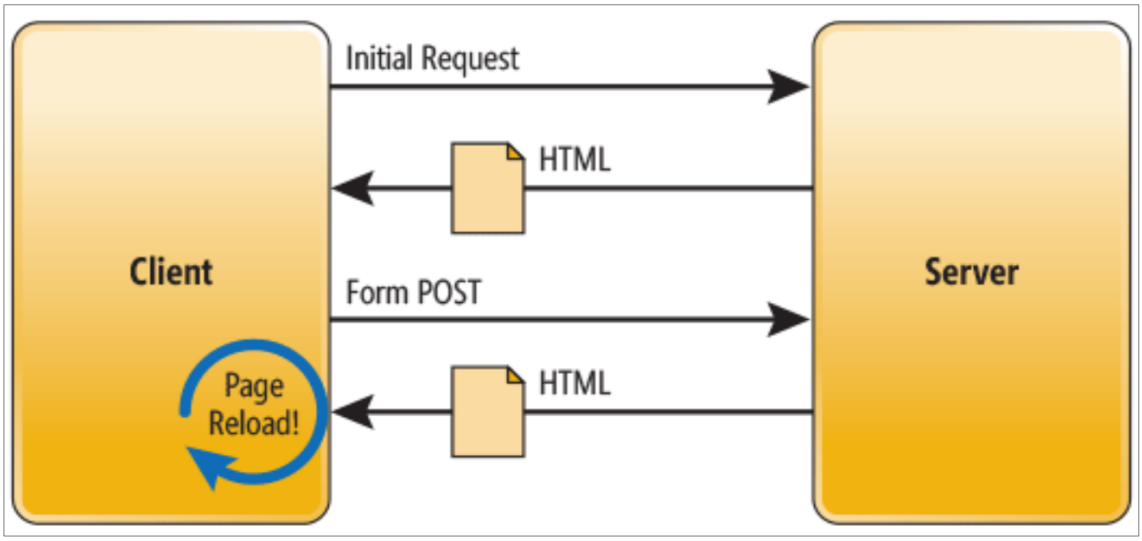

Figura 2.1: ciclo de vida de una MPA [8]

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].

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. Esto queda reflejado en la figura 2.2.

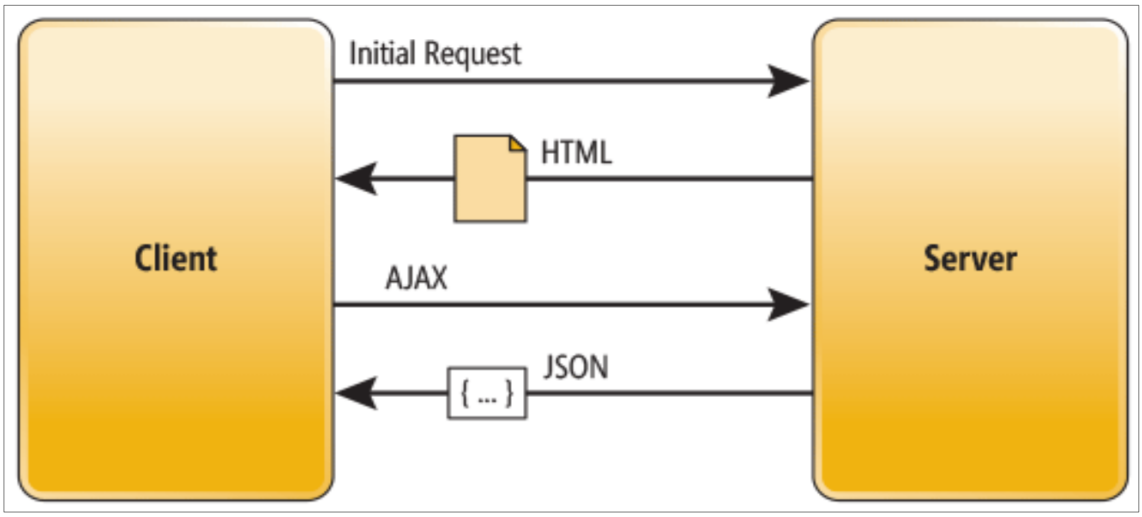

Figura 2.2: ciclo de vida de una SPA [8]

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 el 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.

2.1.2. Tecnología para el desarrollo de aplicaciones web

Con el auge de la popularidad de las SPA 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.

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 [11]:

- **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].

Frameworks populares

Actualmente existen tres *frameworks* que destacan por su gran acogida en la industria, como se puede apreciar en la figura 2.4, atendiendo a la evolución del número de descargas anuales. Estos son React[1], Angular[2] y Vue[3].

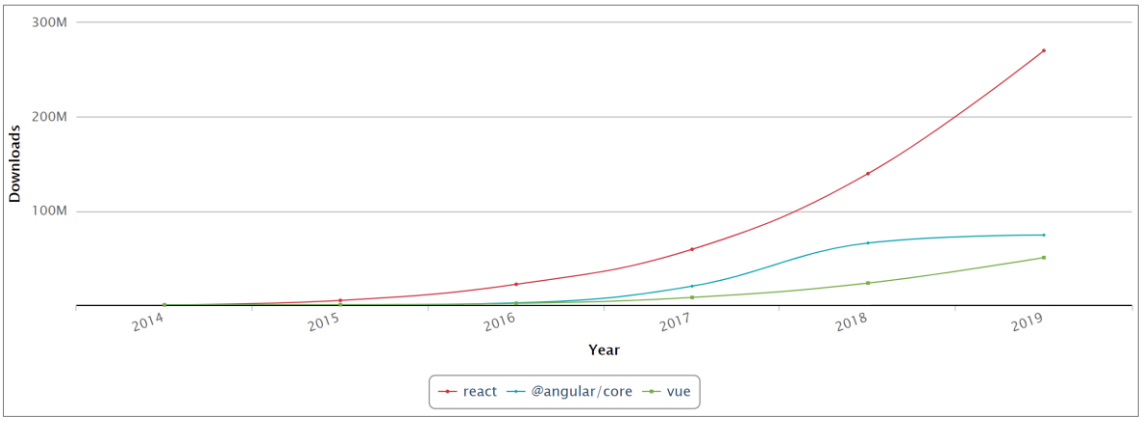

Figura 2.4: descargas anuales por librería durante los últimos cinco años [12]

[1] Página oficial de React: https://reactjs.org/
[2] Página oficial de Angular: https://angular.io/
[3] Página oficial de Vue: https://vuejs.org/

Los *frameworks* anteriores tienen grandes ecosistemas y documentación abundante, lo que hace que aprender a utilizarlos sea relativamente sencillo. De la misma forma, cada *framework* 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 los 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 *framework* propone un modelo de componente diferente al del resto, los componentes desarrollados en cada uno 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.

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, es decir, componentes web, y ofrecen mecanismos para su encapsulación [16].

Los dos grandes bloques que conforman la especificación son *Custom Elements v1*, que definen un modelo de componentes de bajo nivel basado en estándares web, y *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 JS 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 componentes web 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[4], anteriormente conocida como Polymer, como se verá más adelante. Existen otras alternativas como HyperHTML[5] o SetencilJS[6].

```
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.

2.1.3. Módulos en JavaScript

Inicialmente los programas de JS 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 SPA, se da la necesidad de contar con un mecanismo capaz de dividir el programa en módulos separados que puedan interactuar entre sí [21].

[4] Página oficial de LitElement: https://lit-element.polymer-project.org/
[5] Página oficial de HyperHTML: https://viperhtml.js.org/
[6] Página oficial de StencilJS: https://stenciljs.com/

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 JS o empleados en ciertos entornos de ejecución de JS, como Node.js[7].

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 (CJS)[8], el formato de módulos empleado por Node.js o *Asynchronous Module Definition Format* (AMD)[9], 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.

Formato de módulos estandarizado por el lenguaje

Los módulos de ECMAScript (ES), la especificación del lenguaje JS, fueron introducidos en la versión del 2015, también conocida como ECMAScript6, ES6 o ES2015[10]. 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. En la figura 2.7 aparece una columna por cada uno de los navegadores web más populares y en cada fila se muestran en verde las versiones de cada navegador que soportan la carga de módulos en formato estándar y en rojo las que no.

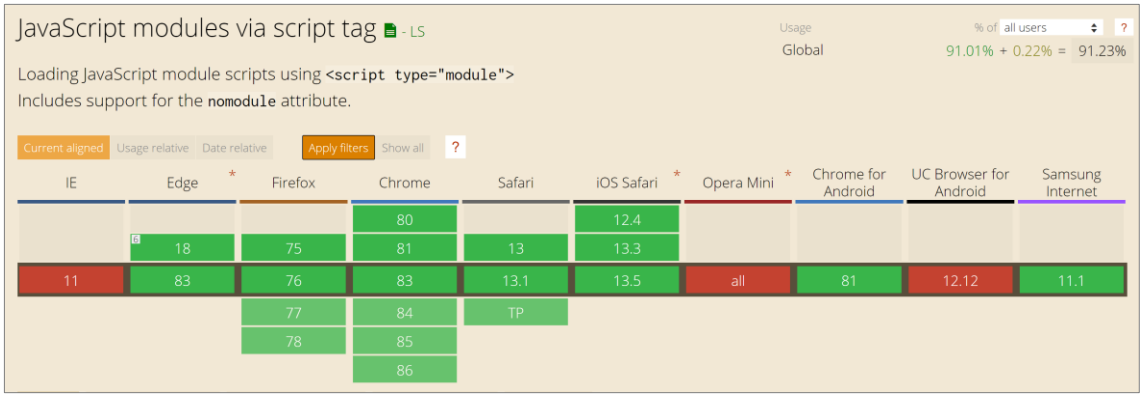

Figura 2.7: tabla de soporte de navegadores para módulos ES [22]

[7] Página oficial de Node.js: https://nodejs.org/
[8] Enciclopedia oficial de CJS: http://wiki.commonjs.org/wiki/CommonJS
[9] Especificación de AMD: https://github.com/amdjs/amdjs-api/blob/master/AMD.md
[10] Especificación de ECMAScript 2015: http://www.ecma-international.org/ecma-262/6.0/

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 [23], 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[11].

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 [24].

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 JS. Un ejemplo de estos primeros empaquetadores es Browserify[12], el cual permitía y permite ejecutar módulos CJS 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.

[11] Borrador de la especificación ECMAScript 2021: https://tc39.es/ecma262/
[12] Página oficial de Browserify: http://browserify.org/

Los dos empaquetadores más populares a día de hoy, según el número de descargas anuales presentado en la figura 2.8, son Webpack[13] y Rollup.js[14].

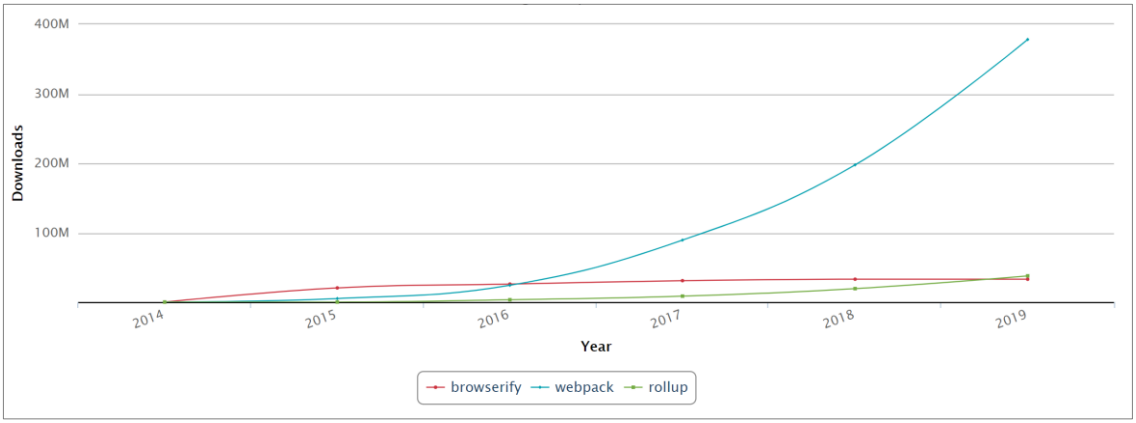

Figura 2.8: descargas anuales por empaquetador en los últimos cinco años [25]

Ambos empaquetadores vienen a aportar la misma funcionalidad, extensible vía el uso de complementos, aunque con unas diferencias notables [26]:

- **Compatibilidad**: los dos empaquetadores permiten consumir módulos definidos en cualquier formato y generar paquetes en el formato de salida deseado. 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.js 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.js 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.js [27]. 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 JS 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.

[13] Página oficial de Webpack: https://webpack.js.org/
[14] Página oficial de Rollup.js: https://rollupjs.org/

- **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 [28].

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.js sea la herramienta preferida a la hora de empaquetar código para ser compartido [29].

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.js, 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 un localizador uniforme de recursos (URL) válido, estos especificadores reciben el nombre de *bare module specifiers.* El especificador empleado en la segunda línea, sin embargo, sí constituye un URL válido y el navegador es capaz de descargar el recurso referenciado 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 [30].

Los *Import maps*[15] 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 [31], es decir, de forma que el cambio en una dependencia no invalide el cache de sus consumidores. En la figura 2.10 se muestra la configuración de un *import map* que asociará el especificador de módulo *some-module* al URL en el que localizar el fichero que contiene su definición y será descargado por el navegador al ser requerido.

[15] Borrador de la especificación *Import maps:* https://wicg.github.io/import-maps/

```
<script type="importmap">
    {
      "imports": {
        "some-module": "/node_modules/some-module/index-a2g3jiu9.js"
      }
    }
</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 [32]. También puede utilizarse a través del entorno de ejecución SystemJS[16], 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.

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 estándar por parte de algunos navegadores.

CSS Modules v1

Uno de los tres pilares de la web, junto al HTML y el JS, son las *Cascading Style Sheets* (CSS) [33], 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* [34] que contienen el código CSS o cargando archivos CSS mediante elementos de tipo *link* [35]. 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 estándar a este problema basadas en el uso de empaquetadores que incluyen el CSS como texto dentro del JS para después crear un elemento de tipo *style* en la carga de hojas de estilo creando un *link* de forma dinámica cuando sea necesario. El complemento de Webpack style-loader[17] es un ejemplo de herramienta que permite aplicar las dos técnicas descritas anteriormente.

Ambas soluciones requieren el uso de herramientas adicionales y no proporcionan un rendimiento óptimo. Por ese motivo surge la propuesta para el estándar *CSS Modules v1*, una extensión de los módulos ES que permitiría la interacción sin dificultades entre estos y módulos de CSS [36] a través de *Constructable Stylesheets*, una interfaz para crear y manejar hojas de estilo desde código JS sin necesidad de elementos de estilo, lo que a su vez permite la reutilización de las mismas [37].

[16] Repositorio oficial de SystemJS: https://github.com/systemjs/systemjs
[17] Documentación oficial de style-loader: https://webpack.js.org/loaders/style-loader/

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.

2.1.4. Gestores de paquetes

Los gestores de paquetes son herramientas que permiten simplificar y automatizar las tareas de mantenimiento de un paquete o proyecto de JS. 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 [38].

Actualmente son tres los gestores de paquetes que predominan en la industria: NPM[18], Yarn[19] y PNPM[20] [39].

- **NPM**: publicado en el año 2010, fue el primer repositorio y gestor de paquetes de Node.js. 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**[21]: 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.

[18] Documentación oficial de NPM: https://docs.npmjs.com/about-npm/
[19] Página oficial de Yarn: https://yarnpkg.com/
[20] Página oficial de PNPM: https://pnpm.js.org/
[21] Página oficial de Lerna: https://lerna.js.org/

- **Rush**[22]: 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. Destaca por ofrecer no sólo una serie de utilidades si no un modelo de trabajo en sí mismo, a costa de la flexibilidad de poder definir un modelo de trabajo personalizado.

2.1.5. Sistemas de control de versión

Los sistemas de control de versión almacenan el histórico de cambios aplicados a los ficheros de un proyecto a lo largo del tiempo. Para ello realizan el seguimiento de los cambios en cada 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, 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 los cambios efectuados en sus respectivos repositorios locales en el repositorio central o remoto, alojado en un servidor [40].

Git

Git[23] es un sistema de control de versión distribuido cuya mayor diferencia con otros sistemas del mismo tipo como Subversion[24] reside en cómo modela los datos. En lugar de almacenar la información como una lista de cambios 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 [41].

Como consecuencia de estas características y otras que cubren casos de uso más avanzados, la popularidad de Git se ha incrementado drásticamente en los últimos años [42].

---

[22] Página oficial de Rush: https://rushjs.io/
[23] Página oficial de Git: https://git-scm.com/
[24] Página oficial de Subversion: https://subversion.apache.org/

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 [43], 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 inspeccionar 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 lentas, esto no debería ser una fuente de preocupación excepto en casos extremos como el mono repositorio de Windows [44] o el que contiene toda la base de código de Google [45].

2.2. Desarrollo de aplicaciones web

Una vez expuestos los distintos aspectos y opciones tecnológicas disponibles para el desarrollo web en la actualidad, en este capítulo se describe cuál sería el procedimiento típico a la hora de desarrollar una aplicación web empleando algunas de las herramientas más populares en la industria para, a continuación, evaluar los distintos modelos arquitectónicos de desarrollo de aplicaciones web.

2.2.1. Procedimiento típico para 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 e incluso entre navegadores. Todas estas herramientas suelen contar, además, con utilidades para la realización de pruebas unitarias y, en ocasiones, de integración, así como con un servidor de archivos estáticos para servir la aplicación durante su implementación.

Una de estas herramientas es Angular CLI[25] diseñada para trabajar con Angular. Se trata de una herramienta de línea de comandos 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[26] o Vue CLI[27], 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.

[25] Página oficial de Angular CLI: https://cli.angular.io/
[26] Página oficial de Create React App: https://create-react-app.dev/
[27] Página oficial de Vue CLI: https://cli.vuejs.org/

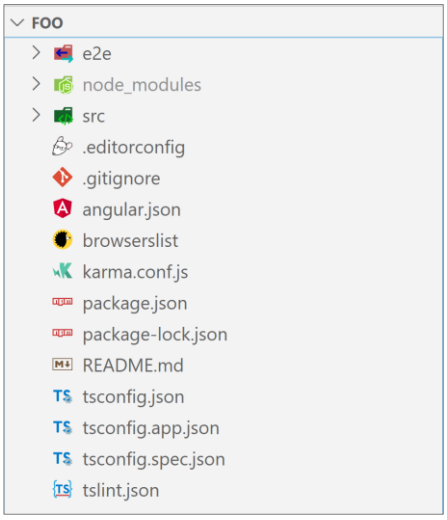

Figura 2.11 resultado de ejecutar el comando "ng new foo"

Una vez generado el proyecto, el desarrollador podrá iniciar el entorno de desarrollo con el comando *npm start*, cuyo resultado será el empaquetado 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, dando como resultado el ilustrado por la figura 2.12.

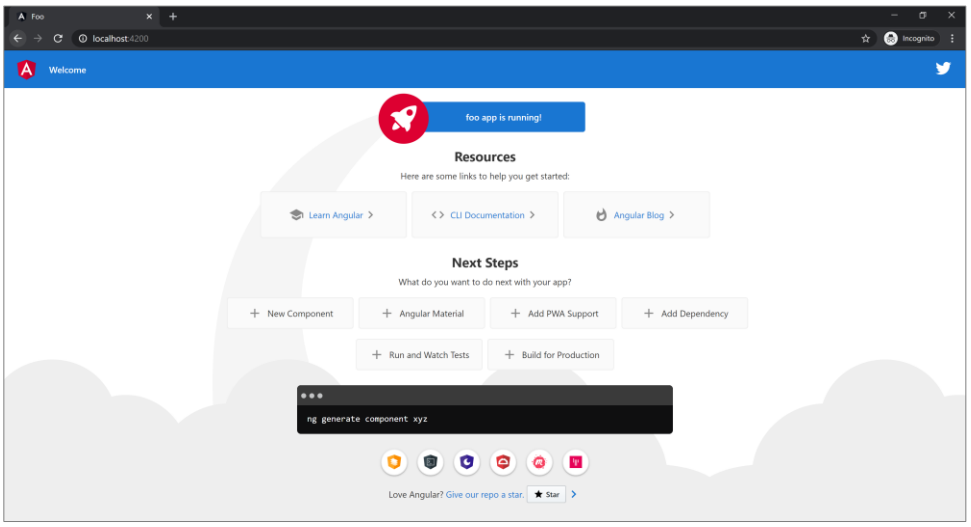

Figura 2.12: resultado de la ejecución del comando npm start

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 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 que más tarde serían subidos al servidor de estáticos del entorno productivo.

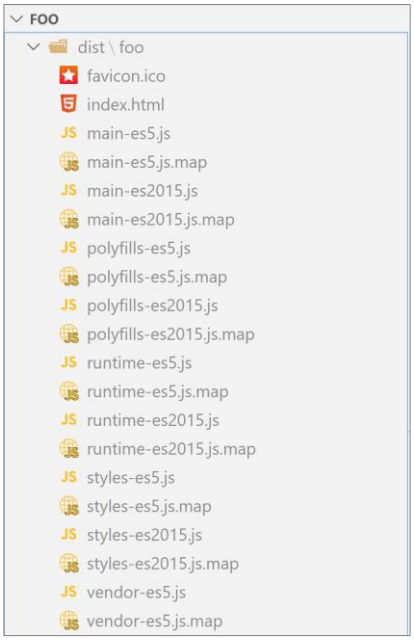

Figura 2.13: empaquetado de una aplicación con npm run build.

2.2.2. Sistemas monolíticos

Las aplicaciones web son comúnmente desarrolladas como aplicaciones monolíticas, un enfoque arquitectónico indirectamente promovido por las herramientas más populares de desarrollo citadas en el apartado anterior, caracterizado por constituir sistemas cerrados y autocontenidos en los que se integra toda la funcionalidad de la aplicación en una única unidad ejecutable [46]. 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” [47]. Así, las barreras autoimpuestas [48] 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 posean el conocimiento del sistema 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 ser 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, de larga duración 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* [49], conmutadores que permiten decidir, en ejecución, qué funcionalidad activar, o herramientas de refactorización a gran escala como *codemod*[28]. 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.

[28] Repositorio oficial de codemod: https://github.com/facebook/codemod

2.2.3. *Micro Frontends*

Qué son los *Micro Frontends*

Las arquitecturas de microservicios [50], 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* [51] 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, referidas como *microfrontends*, con un ciclo de desarrollo y evolución diferente que se componen en una sola aplicación final 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, así como ciertos problemas a los que será necesario hacer frente. Gran parte de la información de esta sección ha sido extraída del artículo “Micro Frontends” escrito por C. Jackson [52].

Entre las principales ventajas resultantes de emplear este tipo de arquitectura destacan:

- **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 lista para su uso en producción independientemente del estado del resto.
- **Evolución independiente:** la posibilidad de desarrollar cada *microfrontend* 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 se requiere de 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 que 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 *microfrontends* 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 por los que 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.

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 2.14 se muestra el proyecto raíz de una aplicación que incluye distintos *microfrontends* en sus dependencias. 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 realizar 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 2.14: 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) [29], 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 2.15 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 2.15: 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* [53], un elemento HTML que permite incrustar una página dentro de otra, como se ilustra en la figura 2.16. 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 2.16: integración mediante iframes

[29] Documentación de Apache sobre SSI: https://httpd.apache.org/docs/2.4/es/howto/ssi.html

- **Integración en el cliente mediante JavaScript:** la técnica de integración más flexible consiste en integrar los distintos *microfrontends* mediante JS 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 2.17, o crear un sistema personalizado como es el caso de la herramienta single-spa[30], 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 2.17: integración mediante Web Components

[30] Página oficial de single-spa: https://single-spa.js.org/

2.3. Conclusiones del estado del arte

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.

2.3.1. 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 2.2.1 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 son, por tanto, aquellos 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 [51] 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 de forma explícita 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 una arquitectura basada en *microfrontends,* 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.

2.3.2. Alternativas

Existen ecosistemas de desarrollo web como MEAN [54], una solución integral para el desarrollo de aplicaciones web, que cubre desde la interfaz de usuario, hasta la base de datos de la aplicación, pasando por su servidor. También se dan otros ejemplos como single-spa, mencionado anteriormente, que define y da soporte a una arquitectura basada en *microfrontends* y cuenta cada vez con un mayor número de integraciones con *frameworks* y herramientas [55], o Nx[31], un ecosistema para desarrollar aplicaciones con Angular o React de forma escalable [56].

No obstante, pese a la gran variedad de opciones disponibles, no existe realmente un ecosistema tecnológicamente agnóstico en su totalidad y que no requiera la implementación de una arquitectura concreta para su aplicación. Estos ecosistemas o bien tienen un enfoque muy amplio como es el caso de MEAN y por tanto no ofrecen soluciones específicas a los aspectos tratados anteriormente, o bien están limitados al uso de ciertas tecnologías, como ocurre con Nx, o, en el caso de single-spa, imponen una forma específica de abordar la arquitectura y la implementación de la aplicación.

Como alternativa a las soluciones habituales de desarrollo web, en este TFG se plantea un ecosistema 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 *microfrontends*, prestando especial interés a las necesidades de infraestructura y dando al desarrollador libertad total en cuanto a cómo implementar la aplicación. Para ello se realizará una selección de herramientas, técnicas y pautas para su empleo, lo que en adelante será referenciado como ecosistema, al mismo tiempo que se desarrolla una herramienta de apoyo para la aplicación del mismo.

[31] Página oficial de Nx: https://nx.dev/

3. Propuesta de ecosistema para desarrollo web

Como solución al problema anteriormente descrito se plantea la definición de un ecosistema para el desarrollo de aplicaciones web con una arquitectura basada en *microfrontends.* A continuación, se describe la serie de recomendaciones arquitectónicas, técnicas y herramientas que conformarán el ecosistema permitiendo abordar los principales aspectos arquitectónicos y de infraestructura a los que este debe hacer frente, como el empaquetado de los distintos *microfrontends* o la gestión de dependencias en un mono repositorio.

Modelo arquitectónico

Como modelo arquitectónico se utilizará el de los *Micro Frontends*, y la integración de los mismos se realizará en el cliente mediante JS 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.* Cabe mencionar que, si bien se da soporte a este modelo de integración, será responsabilidad del desarrollador definir la forma en la que los distintos *microfrontends* interactuarán en tiempo de ejecución.

Control de versiones

El sistema de control de versiones a utilizar será Git, y, dado que resultará 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.

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 [57].

Gestor de mono repositorio

Para gestionar el versionado de las aplicaciones 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 requeriría adaptar el sistema a la herramienta.

Modelo de componente

El modelo de los distintos componentes que formarán parte de la aplicación del caso de estudio 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.

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 ES precisamente por tratarse del formato de módulo estándar de JS. Durante el empaquetado de la aplicación se transformará este formato a System, aportando así compatibilidad con navegadores que no soporten el formato de módulo estándar y dando la posibilidad de utilizar i*mport 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.

Empaquetado

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 *Webpack* hace que no constituya una solución a prueba de futuro.

Para dar soporte a un proceso de empaquetado bajo demanda, es decir, capaz de detectar cambios en un proyecto y, consecuentemente, empaquetarlo, se utilizará Chokidar[32], una herramienta que permite detectar y reaccionar ante cambios en el sistema de archivos.

Con el fin de asegurar que los archivos empaquetados siempre se corresponden con la versión actual del código fuente se utilizará Husky[33], una herramienta que permite definir *Git Hooks* [58], procesos que ejecutan una determinada acción antes distintos eventos ocurridos en el repositorio local de Git. Por ejemplo, sería posible, detectar la introducción de cambios en el repositorio y empaquetar las aplicaciones modificadas.

Carga de hojas de estilo

Se empleará la propuesta de estándar de módulos CSS de modo que puedan importarse hojas de estilo desde JS, 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.

[32] Repositorio oficial de Chokidar: https://github.com/paulmillr/chokidar
[33] Repositorio oficial de Husky: https://github.com/typicode/husky

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[34], una librería que permite crear, de forma sencilla y altamente configurable, este tipo de herramientas empleando JS.

Servidor de estáticos

El servidor de archivos estáticos a emplear durante el desarrollo se implementará con Browsersync[35], debido al gran abanico de posibilidades que ofrece, 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.

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 formato. En este sistema se empleará ESLint[36] para analizar el código JS, Stylelint[37] para el CSS y, finalmente, Prettier[38] para formatear el código de forma automática, sea este escrito en JS, CSS o HTML. 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.

Pruebas unitarias

El soporte para la realización de pruebas unitarias se dará a través de Karma[39]. Aunque existen alternativas más populares como Jest[40], 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 *Web Components*.

[34] Página oficial de Yargs: http://yargs.js.org/
[35] Página oficial de Browsersync: https://www.browsersync.io/
[36] Página oficial de ESLint: https://eslint.org/
[37] Página oficial de Stylelint: https://stylelint.io/
[38] Página oficial de Prettier: https://prettier.io/
[39] Página oficial de Karma: https://karma-runner.github.io/
[40] Página oficial de Jest: https://jestjs.io/

Pruebas de integración

La posibilidad de realizar pruebas de integración será ofrecida a través de CodeceptJS[41]. CodeceptJS presenta una interfaz para definir pruebas de integración y cuenta con conectores para ejecutarlas empleando una gran variedad de herramientas. Por ejemplo, puede optarse por emplear el conector de WebDriverIO[42] con Selenium[43] para lanzar las pruebas en distintos navegadores o el de Puppeteer[44] 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.

[41] Página oficial de CodeceptJS: https://codecept.io/
[42] Página oficial de WebDriverIO: https://webdriver.io/
[43] Página oficial de Selenium: https://www.selenium.dev/
[44] Página oficial de Puppeteer: https://pptr.dev/

4. Herramienta de apoyo para el ecosistema

Una vez definido el ecosistema para desarrollo de aplicaciones web, se implementará una herramienta que facilite su aplicación, permitiendo automatizar las tareas asociadas al desarrollo tal y como hacen otras herramientas populares en la industria, de modo que la adopción de este ecosistema no suponga un esfuerzo adicional por parte de los desarrolladores, que, de otra forma, deberían hacer frente a la complejidad derivada de la solución.

Esta herramienta será desarrollada en JS y se ejecutará en el entorno de ejecución Node.js. Contará con una interfaz de línea de comandos que el desarrollador podrá iniciar desde la terminal y ofrecerá distintos comandos y opciones mediante los cuales hacer uso a la funcionalidad ofrecida. También exportará una serie de utilidades, como configuraciones base para ciertas utilidades del ecosistema, para ser consumidas desde JS.

4.1. **Análisis**

La herramienta a desarrollar ha de hacer frente a muchos aspectos del desarrollo de aplicaciones web. La funcionalidad necesaria se especificará mediante una serie de casos de uso y los aspectos a tener en cuanta durante la implementación de cada uno de esos casos de uso serán definidos por una serie de requisitos técnicos.

4.1.1. Casos de uso

Los casos de uso a los que la herramienta debe hacer frente se especifican en el diagrama de la figura 4.1 y se describen a continuación.

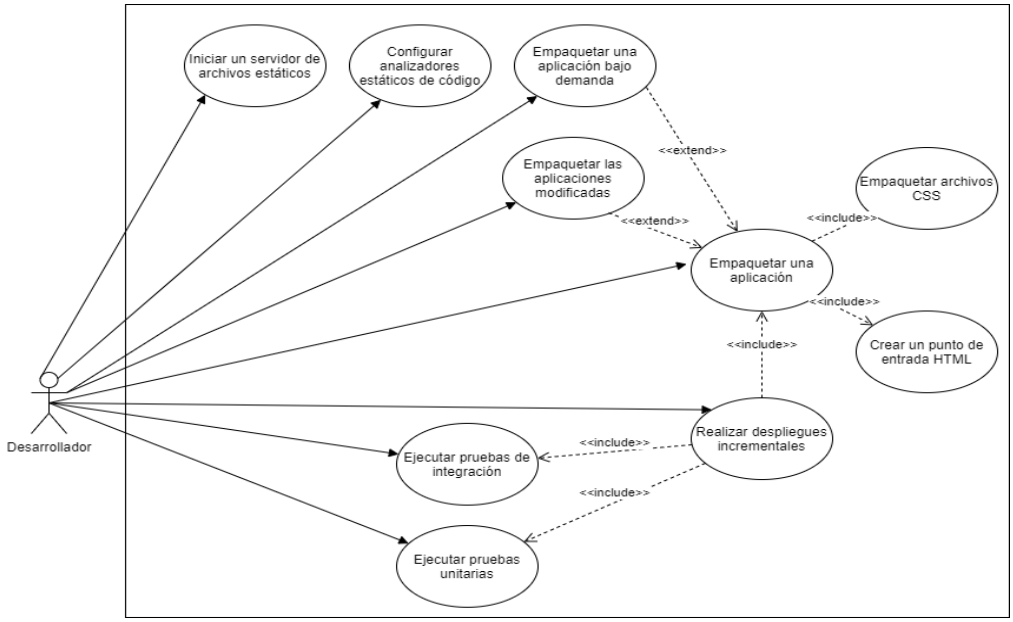

Figura 4.1: diagrama de casos de uso de la herramienta

Caso de uso	Descripción
Ejecutar pruebas unitarias	La herramienta debe contar con un comando capaz de ejecutar las pruebas unitarias pertenecientes a una aplicación específica.
Ejecutar pruebas de integración	La herramienta debe contar con un comando capaz de ejecutar las pruebas de integración pertenecientes a una aplicación específica.
Iniciar un servidor de archivos estáticos	La herramienta debe contar con un comando para ejecutar un servidor de archivos estáticos que sirva la aplicación. El servidor debe refrescar la página en el navegador ante cualquier cambio en los artefactos empaquetados.
Configurar analizadores estáticos de código	La herramienta debe ofrecer configuraciones base para los analizadores estáticos de código ESLint, Stylelint y Prettier.
Empaquetar una aplicación	La herramienta debe ofrecer un comando a través del cual empaquetar la aplicación. El comando debe contar con opciones para elegir qué aplicaciones empaquetar y especificar si se desea o no empaquetar las dependencias de terceros. También debe disponer de un alias para empaquetar todas las aplicaciones y dependencias en un mismo proceso.
Empaquetar una aplicación bajo demanda	El comando de empaquetado debe contar con una opción para que no se empaquete ninguna aplicación de forma inmediata. Al emplear esa opción ha de ponerse en marcha un sistema de detección de cambios capaz de detectar modificaciones en cualquier proyecto y empaquetarlo en ese momento. De este modo será posible iniciar el proceso de desarrollo de forma casi instantánea.
Empaquetar las aplicaciones modificadas	Cuando se integren cambios en el repositorio local, las aplicaciones afectadas deberán ser automáticamente empaquetadas con tal de que el código empaquetado sea consistente con el código fuente.
Empaquetar archivos CSS	El empaquetador debe ser capaz de incluir hojas de estilo desde código JS.
Crear un punto de entrada HTML	El proceso de empaquetado debe contar con la capacidad de generar un documento HTML a partir de una plantilla que sirva como punto de entrada de la aplicación y cargue todas las dependencias necesarias para su ejecución.

Realizar despliegues incrementales	La herramienta dará soporte a la realización de despliegues incrementales, es decir, despliegues únicamente de las aplicaciones modificadas. Para ello se dará soporte, en los comandos de pruebas unitarias, pruebas de integración y empaquetado, a una opción que permita procesar todos los proyectos modificados desde un punto de la historia del repositorio, así como los proyectos que dependan directa o indirectamente de estos.

4.1.2. Requisitos técnicos

La serie de requisitos técnicos a tener en cuenta a la hora de dar soporte a los casos de uso descritos anteriormente son los siguientes:

Requisito	Descripción
Soporte para mono repositorios con Lerna y Yarn.	La herramienta debe ser capaz de analizar las dependencias entre distintos proyectos que formen parte de un mono repositorio que haga uso de Lerna y espacios de trabajo de Yarn.
Ejecución de pruebas unitarias en el navegador	Las pruebas unitarias deben poder ser ejecutadas en el navegador. La elección del navegador debe poder ser configurable.
Ejecución de pruebas de integración en cualquier navegador	Las pruebas de integración deben poder ser ejecutadas en cualquier navegador a elección del usuario.
Import maps	Cada *microfrontend* debe generar un *import map* con tal de especificar dónde se ubican sus puntos de entrada consumibles desde otras aplicaciones. Al finalizar el proceso de empaquetado todos los *import maps* deben combinarse en un único *import map* que será utilizado en tiempo de ejecución.
Caché de larga duración	Tras empaquetar cualquiera de las aplicaciones, el nombre de los archivos generados debe reflejar su contenido. De esta manera será posible implementar estrategias de caché de larga duración.
Evitar la invalidación de caché en cascada	Modificar el comportamiento interno de un módulo no debe invalidar la caché de módulos dependientes que no hayan sido modificados.
Formato System	El formato de salida de los distintos paquetes deberá ser System con tal de dar soporte al uso de *import maps* mientras estos no sean estandarizados.

Evitar dependencias implícitas	La herramienta debe contar con la rigidez suficiente para evitar la aparición de dependencias implícitas, es decir, aquellas que un proyecto no declara como tal, pero se encuentran disponibles a través de las dependencias configuradas en otro proyecto.
Evitar duplicidad de dependencias	Las dependencias compartidas por distintos *microfrontends* deben empaquetarse de forma que no se dupliquen en el cliente al ser descargadas por varios *microfrontends*. Dicho de otro modo, sólo debe existir una copia de la misma dependencia en tiempo de ejecución.
Independencia entre aplicaciones	Cada aplicación debe poder empaquetarse y ser desplegada de forma independiente.
CSS Modules	La inclusión de hojas de estilo desde JavaScript debe hacerse a través de módulos CSS.
Configuración de empaquetado por aplicación	Cada uno de los *microfrontends* debe contar con la posibilidad de establecer su propia configuración para el proceso de empaquetado, partiendo de una configuración central compartida. De este modo se contará con la flexibilidad necesaria para que cada *microfrontend* pueda hacer frente a casuísticas específicas sin afectar al resto.
Convenciones explícitas	Las distintas convenciones empleadas por la herramienta, como el nombre de los archivos que contienen los *import maps,* deben ser explícitamente configuradas.
Agnosticismo tecnológico	La herramienta debe servir para desarrollar cualquier aplicación de una sola página independientemente de la tecnología empleada en el cliente.

4.2. Diseño

Con el fin de cubrir los requisitos anteriormente descritos, la herramienta constará de seis grandes bloques o módulos encargados de agrupar la funcionalidad ofrecida por la herramienta. Estos bloques, a su vez, serán divididos en varios proyectos que se integrarán con tal de ofrecer la funcionalidad requerida.

Los bloques funcionales que forman la herramienta, a alto nivel, son:

- ***Commands:*** herramienta de línea de comandos a través de la cual el usuario podrá acceder a la funcionalidad ofrecida por los otros módulos de la herramienta.
- ***Server:*** funcionalidad relacionada con el servidor de archivos estáticos a emplear durante el desarrollo.
- ***Testing:*** bloque encargado de aportar las utilidades necesarias para la ejecución de pruebas unitarias y de integración.
- ***Builder:*** conjunto de módulos encargados de toda la funcionalidad relacionada con el empaquetado de las aplicaciones durante el proceso de desarrollo y para su puesta en producción. También gestionará la creación de los *import maps* necesarios para ubicar cada uno de los *microfrontends.*
- ***Utils:*** utilidades a emplear internamente por la propia herramienta, como un gestor del repositorio que permita consultar información sobre los distintos proyectos que lo integran.
- ***Config:*** módulos donde exportar las distintas configuraciones a ser empleadas, como las de los analizadores de código estáticos o la propia configuración de la herramienta.

Las relaciones entre estos bloques, así como las relaciones internas entre sus proyectos, se ilustra en el diagrama de componentes que aparece en la figura 4.2, donde aparecen en verde aquellos proyectos o componentes que constituyen la interfaz pública de la herramienta.

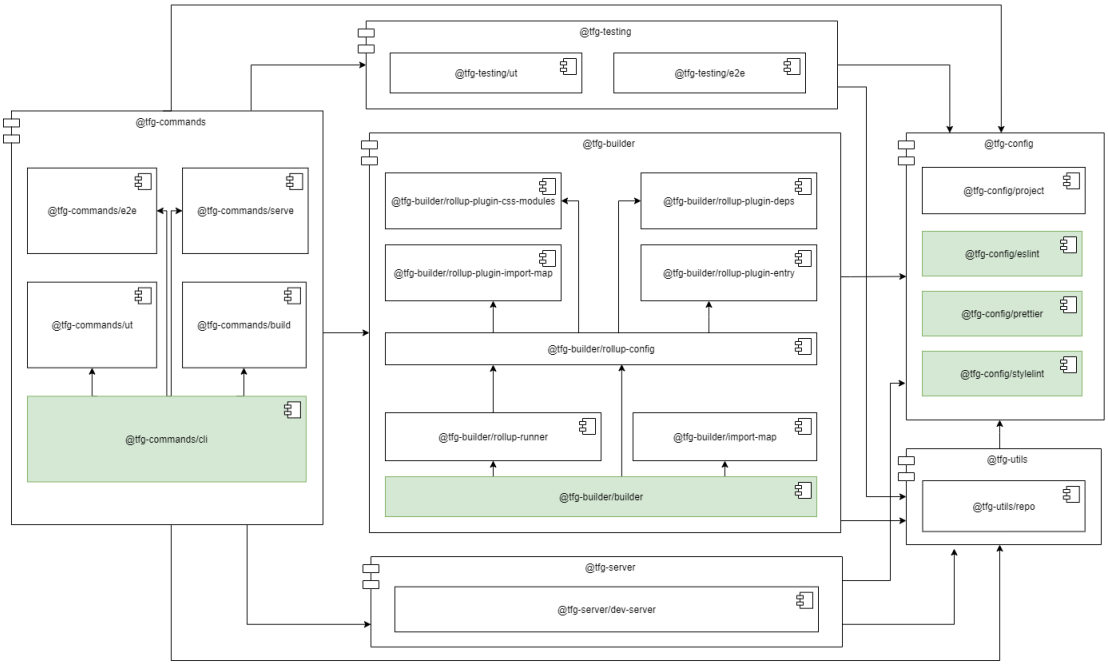

Figura 4.2: diagrama de componentes de la herramienta

4.3. **Implementación**

La herramienta ha sido desarrollada en un mono repositorio y, siguiendo la convención de nombres de los paquetes de NPM [59], cada uno de los proyectos que formen parte de un bloque estará contenido dentro del espacio de nombres *@tfg-[bloque]/[proyecto].* Es decir, el nombre de un proyecto de nombre *foo* perteneciente al bloque *bar* sería *@tfg-bar/foo.* A continuación, se describen los proyectos integrantes de cada bloque atendiendo a su propósito y a los detalles más relevantes sobre su implementación.

4.3.1. @tfg-config

Config es el nombre que recibe el bloque de configuración. El bloque constará de cuatro proyectos, uno de ellos contiene la configuración de la herramienta para uso interno mientras el resto podrán ser empleados por el usuario como configuración base y extensible para ciertas herramientas de terceros a emplear, opcionalmente, durante el desarrollo. Así pues, los proyectos que conforman este bloque son:

- **@tfg-config/project*:*** contiene todos los parámetros de configuración utilizadas por el resto de proyectos que constituyen la herramienta. En la figura 4.3 aparecen parámetros tales como la carpeta raíz del proyecto, la versión de Node.js soportada por la herramienta o la ruta de distribución, es decir, el directorio donde extraer las aplicaciones empaquetadas tras el proceso de construcción.

```
3  module.exports = {
4    get ROOT() {
5      return resolve(__dirname, '..', '..', '..', '..');
6    },
7    get LOCK_FILE() {
8      return resolve(this.ROOT, 'yarn.lock');
9    },
10   get NODE_VERSION() {
11     const package = require(resolve(this.ROOT, 'package.json'));
12     return package.engines.node;
13   },
14   get DIST() {
15     return resolve(this.ROOT, 'dist');
16   },
```

Figura 4.3: parámetros de configuración de la herramienta

Al centralizar la configuración en un punto y referenciarla en el resto, es más fácil realizar cambios y se podría dar al usuario la posibilidad de sobrescribir valores.

- **@tfg-config/prettier:** define una configuración base para Prettier, una herramienta capaz de formatear automáticamente el código de modo que el estilo en el que está escrito el código sea homogéneo independientemente de las preferencias personales del programador.
- **@tfg-config/eslint:** exporta varias configuraciones para Eslint, un analizador de código estático de JS que permite detectar y, en ocasiones, solucionar errores o malas prácticas. Para ello define una serie de reglas e incluye ciertos complementos en cada configuración que resultarán útiles a la hora de analizar archivos destinados a ser ejecutados en el navegador o bien en Node.js, así pues, se exporta una configuración para usar en la parte del cliente y otra para utilizar en la parte de infraestructura. Destaca, en esta última configuración, el uso de un complemento que permite prohibir el uso de características del lenguaje no disponibles en la versión mínima de Node.js que se desee soportar como ilustra la figura 4.4, donde se asigna el código 2, que Eslint utiliza para representar errores, al uso de características de ES no soportadas por la versión de Node.js configurada.

```
17     'node/no-unsupported-features/es-syntax': [
18       2,
19       {
20         version: PROJECT.NODE_VERSION,
21       },
22     ],
```

Figura 4.4: regla para características no soportadas por el entorno

- **@tfg-config/stylelint:** de forma similar al proyecto anterior, exporta una configuración para Stylelint, un analizador estático de código CSS. Incorpora la configuración estándar de la herramienta y reglas relacionadas con el orden de las propiedades CSS, lo cual resulta especialmente útil para prevenir la aparición de conflictos al modificar más de un desarrollador la misma hoja de estilo.

4.3.2. @tfg-utils

Este bloque está destinado a ofrecer la funcionalidad transversal a ser empleada por el resto de proyectos que constituyen la herramienta:

- **@tfg-utils/repo:** proyecto destinado a manejar la información del mono repositorio y los distintos proyectos que lo forman. Provee de utilidades tales como identificar las librerías de terceros empleadas por los distintos proyectos o la posibilidad de obtener una lista de los proyectos modificados desde un punto en la historia del repositorio, para lo cual ejecuta Lerna en un proceso independiente y recoge la información emitida por la salida estándar.

4.3.3. @tfg-builder

Contiene toda la funcionalidad relacionada con el empaquetado de la aplicación y la generación de los *import maps* que permitirán consumir cada uno de los *microfrontends* en tiempo de ejecución. Consta de una serie de complementos para Rollup.js que permitirán hacer frente a distintos requisitos, utilidades para configurar y ejecutar el proceso de empaquetado, una herramienta para combinar distintos *import maps* y, finalmente, un módulo que exporta toda la funcionalidad a ser utilizada por el resto de herramientas:

- **@tfg-builder/rollup-plugin-css-modules:** complemento de Rollup.js que permite importar archivos CSS convirtiéndolos en módulos CSS según son definidos en la propuesta de estándar.

  Los complementos de Rollup.js cuentan con el método *transform,* que recibe como argumentos el código y el identificador de cada uno de los módulos importados por la aplicación y permite devolver código diferente para ser empaquetado. Al detectar que el identificador, que hace referencia a la ruta del fichero, tiene la extensión *.css*, en lugar de devolver el código original se crea y exporta una instancia de *CSSStyleSheet,* el objeto que representa las hojas de estilo en el navegador, con el código CSS original como contenido de la hoja. De esta forma cada archivo CSS importado por la aplicación se transforma, en tiempo de construcción, en un módulo CSS, tal y como lo haría en el navegador una vez el estándar sea aprobado.

- **@tfg-builder/rollup-plugin-deps:** este complemento de Rollup.js resuelve los aspectos relacionados con el empaquetado de las dependencias. Por un lado, hace que el proceso de empaquetado emita un error al detectar que el proyecto importa un paquete no declarado en sus dependencias, evitando así la aparición de dependencias implícitas. Por otro lado, permite especificar una dependencia como externa, de modo que no pase a formar parte del código empaquetado, lo que resulta imprescindible para evitar dependencias duplicadas. Para ello recibe como argumentos las dependencias definidas en el archivo de configuración del proyecto a empaquetar y una lista de dependencias que deben ser consideradas externas. Utilizando el método *resolveId* ofrecido por Rollup.js, intercepta las importaciones de cada módulo durante el proceso de empaquetado y comprueba que el módulo en cuestión pertenezca al proyecto, se encuentre en su lista de dependencias o esté definido como dependencia externa. Si el módulo no pertenece al proyecto ni está declarado como dependencia se emitirá un error. Si el módulo aparece en la lista de dependencias del proyecto y además se trata de una dependencia externa, se marcará como tal, evitando que acabe siendo empaquetada.

- **@tfg-builder/rollup-plugin-entry:** complemento de Rollup.js que, al ser empleado durante el empaquetado de una aplicación, genera un documento HTML partiendo de una plantilla que incluye los elementos de tipo *script* necesarios para cargar los módulos de entrada de la aplicación generados por Rollup.js. Además, cuenta con soporte para realizar optimizaciones precargando esos módulos de forma asíncrona utilizando elementos *link* de tipo *preload* [60] y reduciendo el tamaño del documento generado utilizando un *minificador* de HTML, una herramienta que permite reducir el tamaño del código eliminando caracteres o elementos innecesarios.

  La información sobre qué módulos cargar se obtiene a través del método *generateBundle* de los complementos de Rollup.js, el cual recibe, al ser invocado, toda la información referente al paquete generado. Concretamente, se indica, para cada fichero que forma parte del paquete final, si constituye un punto de entrada de la aplicación o no.

- **@tfg-builder/rollup-plugin-import-map:** este complemento es una de las piezas más importantes de la herramienta. Permite crear un *import map* con la información del paquete generado y, paralelamente, utiliza ese *import map* para dar soporte a estrategias de caché de larga duración a la vez que evita la invalidación de caché en cascada. Su funcionamiento es complejo, pero gracias a la facilidad de definir complementos de Rollup.js su implementación no lo es tanto.

  En primer lugar, se itera por cada uno de los ficheros generados durante el empaquetado y, para cada fichero, se calcula el hash de su contenido, es decir, se aplica una función criptográfica que genera un código alfanumérico de longitud fija partiendo del contenido del fichero, y modifica su nombre de modo que este quede reflejado. De esta forma, cualquier modificación en los contenidos del fichero generará un nuevo hash, cambiando el nombre del archivo y haciendo, consecuentemente, que el navegador siempre descargue la última versión del mismo.

  En segundo lugar, para cada fichero generado, crea un alias entre el nombre original y el nombre modificado para incluir el hash. Con esto se consigue que los distintos módulos siempre referencien el fichero original, pero se descargue la versión del fichero especificada en el *import map*, donde es almacenado este alias. Al no reflejar el nombre real del fichero, si no el original, cambios en el contenido de un módulo y, por tanto, de su hash, no requieren modificar el contenido de otros módulos que lo importen, evitando así la invalidación de caché en cascada.

  Finalmente, cada módulo definido como punto de entrada en la configuración de Rollup.js, es incluido en el *import map* asociando el alias especificado por el usuario con el nombre del fichero emitido por Rollup.js para ese módulo, el cual, previamente, ha sido modificado para incluir el hash. Una vez finalizado el proceso, se creará un archivo con el *import map* generado en el directorio especificado por el usuario.

Para ilustrar el funcionamiento de este complemento es conveniente emplear un ejemplo: suponiendo que se desea empaquetar una aplicación con los puntos de entrada *foo* y *bar*, que hacen referencia, respectivamente, a los ficheros *src/foo.js* y *src/bar.js*, Rollup.js generará dos ficheros, *foo.js* y *bar.js*, uno para cada punto de entrada. El complemento modificará el nombre de cada fichero incluyendo el hash de su contenido y almacenará esa información creando dos entradas en el *import map,* de modo que los ficheros puedan ser referenciados dentro del propio paquete. Finalmente, incluirá dos entradas más, asociando el nombre dado a los puntos de entrada con los ficheros a los que referencia, permitiendo su consumo desde otras aplicaciones sin conocer el nombre real del fichero ni el URL donde este resulta accesible. El resultado final del *import map* puede consultarse en la figura 4.5.

```
// configuración de Rollup:
/*
  input: {
    foo: 'src/foo.js',
    bar: 'src/bar.js'
  }
*/

// Resultado:
{
  "imports":{
    "/foo.js":"/foo.d8d7953d.js",
    "foo":"/foo.d8d7953d.js",
    "/bar.js":"/bar.356c6ba1.js",
    "bar":"/bar.356c6ba1.js"
  }
}
```

Figura 4.5: import map generado por el complemento

Importar cualquiera de los módulos anteriores, utilizando el alias definido por el usuario, o el URL del fichero empleada internamente por Rollup.js, resultará en la carga, por parte del navegador, del fichero real cuyo nombre refleja el contenido mediante el hash.

- **@tfg-builder/rollup-config:** este proyecto define y exporta diversas configuraciones de Rollup.js para ser empleadas por distintas partes de la herramienta o por el usuario final, es decir, el desarrollador. Estas configuraciones definen aspectos como el formato de salida del paquete, que en este caso es System, los complementos de Rollup.js a emplear y la configuración de los mismos.

  El proyecto cuenta con una configuración base que cualquier *microfrontend* de la aplicación puede extender o modificar para definir su propio proceso de empaquetado específico. De esta forma, se logra que cada aplicación pueda ser empaquetada utilizando la configuración base sin ninguna dificultad, únicamente especificando los puntos de entrada de la aplicación, a la vez que se da soporte a casos de uso más complejos dando la posibilidad de modificar aspectos específicos de la configuración en aquellos proyectos en los que sea necesario. En la figura 4.6 se define una configuración en la que no sólo se especifica la entrada del programa, sino que también se sobrescriben las opciones de salida a la vez que se hace uso de complementos adicionales.

```
5   module.exports = rollupConfig.project({
6     input: {
7       bootstrap: 'index.js',
8     },
9     output: {
10      format: 'iife',
11      name: 'tfg',
12      dir: resolve(PROJECT.DIST, 'core', 'bootstrap'),
13      entryFileNames: '[name].[hash].js',
14    },
15    plugins: {
16      entry: {
17        input: resolve(__dirname, 'public', 'index.html'),
18        dest: PROJECT.TEMPLATE_HTML,
19        basePath: PROJECT.DIST,
```

Figura 4.6: ejemplo de configuración específica

Las otras configuraciones disponibles consisten en una configuración para empaquetar las librerías comunes a todos los *microfrontends* de la aplicación y otra que será empleada para empaquetar los archivos que definen las pruebas unitarias de las aplicaciones. Estas dos configuraciones no son extensibles pues al estar disponibles únicamente para uso interno de la herramienta cubren los casos de uso específicos para los que han sido diseñadas.

- **@tfg-builder/rollup-runner:** paralelamente al proyecto de configuraciones, este proyecto proporciona los métodos necesarios para ejecutar Rollup.js e iniciar el proceso de empaquetado usando la configuración adecuada para cada caso, así pues, cuenta con métodos para empaquetar proyectos, librerías de terceros o pruebas unitarias. También incluye una utilidad específica para construir proyectos en modo de detección de cambios, esta función es similar a la que sirve para empaquetar proyectos, pero además almacena en memoria información de construcciones anteriores que luego aprovechará para acelerar el proceso de construcciones futuras del mismo proyecto realizadas por le mismo proceso.
- **@tfg-builder/import-map:** este proyecto tiene una única función, leer los distintos *import maps* generados en diferentes procesos de empaquetado a través del complemento de Rollup.js @tfg-builder/rollup-plugin-import-map y combinarlos en un solo archivo que después será insertado en el documento HTML que constituya el punto de entrada de la aplicación, según haya sido definido en el módulo de configuración de la herramienta. Es necesario realizar este proceso cada vez que se genera un nuevo *import map* pues, de momento, el estándar no provee de un mecanismo para combinar varios *import maps* en tiempo de ejecución.

- **@tfg-builder/builder:** este proyecto carece de funcionalidad propia, simplemente reexporta la funcionalidad ofrecida por el resto de proyectos de este bloque que debe ser empleada por el usuario final o por otros bloques funcionales de la herramienta. Es decir, define la interfaz pública del bloque @tfg-builder.

  Al tener que declarar todos los *microfrontends* este proyecto como dependencia para poder ser empaquetados, todos los *microfrontends* serán dependientes, indirectamente, de todos los proyectos que intervienen en el proceso de empaquetado, lo que implica que cualquier cambio en uno de estos proyectos hará que se detecten cambios en todos los *microfrontends* y, en consecuencia, estos sean empaquetados de nuevo.

4.3.4. @tfg-server

Este bloque implementa el soporte para servir archivos estáticos durante el desarrollo de la aplicación. Actualmente cuenta con un único proyecto, pero se define como un bloque aparte en caso de que se desee ampliar su funcionalidad en el futuro.

- **@tfg-server/dev-server:** proyecto encargado de ejecutar un servidor de archivos estáticos capaz de servir la aplicación. Para implementar el servidor se utiliza Browsersync, una herramienta que ya cuenta con todo lo necesario para ese fin y, además, es totalmente configurable. El proyecto ofrece dos configuraciones o perfiles que pueden ser elegidos por el usuario, uno para ser empleado durante el desarrollo y otro para la ejecución de las pruebas de integración. La diferencia principal entre estos dos perfiles reside en que el perfil de desarrollo incluye un proceso que detectará cambios en el directorio donde se emiten los paquetes generados por el proceso de construcción y recargará el navegador de forma automática cada vez que un archivo sea añadido o modificado.

4.3.5. @tfg-testing

Este bloque contiene los proyectos necesarios para la realización de pruebas unitarias y de integración:

- **@tfg-testing/ut:** proyecto que permite la ejecución de pruebas unitarias definidas en los ficheros especificados por el usuario. Para ello utiliza Karma, una herramienta que permite lanzar las pruebas en cualquier navegador siempre y cuando se instalen los conectores correspondientes, por defecto, al tratarse de pruebas unitarias que, por definición, no deberían tener en cuenta el entorno, se emplea Google Chrome en modo *headless*, esto es, como un proceso en segundo plano, sin interfaz gráfica, lo que hace que la ejecución sea más rápida y consuma menos recursos que al emplear otras opciones.

  Por defecto, Karma permite cargar JS empleando el formato de módulos estándar. Para dar soporte a System ha sido necesario modificar el contexto de ejecución de las pruebas, o lo que es lo mismo, el código empleado por Karma en el navegador para sincronizar la ejecución de las mismas. Esto es posible gracias a que Karma cuenta con una opción precisamente para indicar un contexto de ejecución personalizado. En este contexto, se carga el *import map* de la aplicación y a continuación la librería SystemJS. Posteriormente, se emplea SystemJS para cargar cada uno de los ficheros donde se definen las pruebas, de uno en uno, empleando una cola de tareas asíncronas. Una vez todos los módulos necesarios han sido registrados se notifica a Karma de que el entorno está listo para comenzar la ejecución de las pruebas.

Podría haberse optado por empaquetar las pruebas empleando el formato de módulo estándar, pero emplear System tiene una ventaja: al haber empaquetado previamente el resto de módulos de la aplicación y disponer del *import map* en el que se especifica la ubicación de cada uno de ellos, es posible empaquetar únicamente los ficheros de pruebas y reutilizar los paquetes previamente generados, consiguiendo así reducir el tiempo necesario para ejecutar las pruebas unitarias al no tener que empaquetar la aplicación entera de nuevo.

- **@tfg-testing/e2e:** proyecto empleado para ejecutar las pruebas de integración mediante CodeceptJS. Para ello configura la herramienta y expone una función a través de la cual lanzar las pruebas pertenecientes al proyecto especificado. A diferencia del proyecto anterior, no ha habido ningún reto en la implementación de la solución, pues las pruebas de integración no requieren de un entorno de ejecución distinto al de la aplicación.

4.3.6. @tfg-commands

Contiene el proyecto que implementa la herramienta de línea de comandos a partir de la cual el usuario final podrá hacer uso de los proyectos creados anteriormente. Para cada comando disponible existe un proyecto que lo define e implementa.

La herramienta de línea de comandos ha sido desarrollada empleando Yargs, una librería de Node.js que sirve para eso mismo. Consecuentemente, los comandos definidos por cada uno de los proyectos son compatibles con el formato que consume esta librería, es decir, cada comando cuenta con un nombre, una descripción, una serie de opciones y, finalmente, una función que recibe esas opciones y ejecuta la lógica necesaria para llevar a cabo la tarea especificada por el usuario.

Los proyectos que forman este bloque son:

- **@tfg-commands/cli:** herramienta de línea de comandos que define la interfaz entre el usuario y el resto de herramientas. Empleando Yargs, registra cada uno de los comandos que se describen a continuación y permite su uso a través del ejecutable *tfg.* Las opciones disponibles pueden consultarse con el comando *tfg --help*, el cual muestra la lista de comandos con su descripción, de la forma que aparece en la figura 4.7.

```
tfg [command]

Commands:
  tfg build  build the given list of projects
  tfg e2e    run e2e tests for the given projects
  tfg serve  starts the development server
  tfg ut     run given project unit tests
```

Figura 4.7: resultado del comando tfg --help

- **@tfg-commands/build:** comando que hace uso del proyecto @tfg-builder/builder con tal de empaquetar la aplicación. Permite especificar los proyectos a empaquetar mediante una lista o proporcionando una referencia a un punto en la historia del repositorio a partir del cual se desea empaquetar todos los proyectos modificados. De forma complementaria, permite especificar si se desean empaquetar las librerías de terceros empleadas por cualquier proyecto de la aplicación.

También cuenta con una opción para empaquetar todos los proyectos y librerías, lo que resulta especialmente útil para preparar el entorno de desarrollo a la hora de ejecutar la aplicación por primera vez. Finalmente, incluye una opción con la que iniciar el proceso de empaquetado bajo demanda, empleando Chokidar para detectar cambios en los proyectos a medida que estos sean modificados durante el desarrollo y empaquetándolos en consecuencia, lo que evita la necesidad de empaquetar la aplicación entera cada vez que se desea empezar a desarrollar, ahorrando una cantidad de tiempo cada vez mayor a medida que evoluciona la aplicación. La lista completa de opciones puede consultarse en la figura 4.8.

```
tfg build

build the given list of projects

Options:
  --version       Show version number                                  [boolean]
  --help          Show help                                            [boolean]
  --projects, -p  projects to build                                     [array]
  --libs, -l      build project libs                 [boolean] [default: false]
  --all, -a       build all projects and libraries   [boolean] [default: false]
  --since, -s     build changed projects since ref                     [string]
  --watch, -w     start the watcher to build projects on changes
                                                     [boolean] [default: false]
```

Figura 4.8: resultado del comando tfg build –help

- **@tfg-commands/serve:** el comando *serve* se utiliza para iniciar el servidor de estáticos de la aplicación durante el desarrollo. Cuenta con una opción para elegir el perfil a emplear por el proyecto @tfg-server/dev-server, esta opción acepta el valor *dev* para cargar el perfil de desarrollo con detección de cambios y *e2e* para seleccionar el perfil necesario para ejecutar las pruebas de integración.
- **@tfg-commands/ut:** comando empleado para ejecutar pruebas unitarias mediante Karma, haciendo uso del proyecto @tfg-testing/ut. Como opciones cuenta con un parámetro para indicar la lista de proyectos cuyas pruebas unitarias se desean ejecutar y otro parámetro a emplear en caso de querer ejecutar pruebas sobre los proyectos modificados desde un punto en la historia del repositorio.

  Al ejecutar el comando se empaquetan todos los archivos con extensión *.spec.js,* habitualmente utilizada para definir pruebas unitarias en herramientas de JS, y se actualiza la página a emplear como contexto de Karma con el *import map* generado, finalmente, inicia el proceso para ejecutar las pruebas. En próximas versiones, podría incrementarse la lista de opciones para dar soporte al mismo número de casos de uso que Karma contempla.

- **@tfg-commands/e2e:** este comando sirve para ejecutar las pruebas de integración asociadas a un proyecto específico utilizando CodeceptJS a través del proyecto @tfg-testing/e2e. Se ejecutarán las pruebas definidas por cada archivo con extensión *.feat.js,* comúnmente empleada para definir pruebas de integración, ubicadas en la carpeta de nombre *e2e* contenida en el directorio del proyecto especificado. También cuenta con una opción para lanzar las pruebas asociadas a cualquier proyecto modificado desde un punto en la historia del repositorio.

  En un futuro se podría dar soporte al resto de opciones de CodeceptJS y a la posibilidad de definir una configuración para la herramienta por cada proyecto, tal como se hace en el proceso de empaquetado.

5. Caso de estudio

Con el tal de evaluar los efectos de un ecosistema de desarrollo de aplicaciones web basado en *microfrontends* y poner a prueba la herramienta desarrollada que debe dar soporte al mismo, se implementará una aplicación de banca en línea.

5.1. **Requisitos**

La funcionalidad específica a ofrecer por esta aplicación no es relevante ya que su propósito es el de ilustrar un modelo de desarrollo de aplicaciones web haciendo uso de una herramienta concreta y con un estilo arquitectónico específico. Por tanto, se definirán los requisitos funcionales necesarios para ejemplificar los aspectos del desarrollo que, más tarde, serán evaluados en el capítulo 5.5.

5.1.1. Requisitos funcionales

Requisito	**Descripción**
Menú de navegación	Menú lateral a través del cual navegar a las distintas rutas de la aplicación.
Barra de aplicación	Encabezado que contiene el nombre de la aplicación y un botón mediante el cual es posible desplegar el menú de navegación.
Página de ruta no encontrada	Cuando el usuario navega a una página inexistente, se ha de mostrar una página informando sobre la situación con un botón para volver a la página de inicio.
Lista de cuentas	Se debe mostrar la lista de cuentas bancarias del usuario, mostrando el número de cuenta y el balance. Esta página debe ser accesible desde el menú de navegación.
Detalles de cuenta	Al pulsar sobre una cuenta en la lista de cuentas se navega a una página donde se muestran los detalles de la misma además de sus últimos movimientos y sus tarjetas asociadas.
Lista de tarjetas	Se debe mostrar una lista con las tarjetas contratadas por el usuario. Esta página debe ser accesible desde el menú de navegación.

5.2. **Diseño**

La aplicación ha sido diseñada como un mono repositorio que cuenta con un proyecto principal, @tfg-root, en el cual definir las *scripts* a ser empleadas por el desarrollador durante su día a día y configurar aspectos generales de la aplicación como las utilidades de análisis estático. Este proyecto raíz, tiene, por tanto, carácter meramente organizativo. Su función será englobar el resto de proyectos y dar acceso al usuario a la funcionalidad ofrecida por la herramienta de desarrollo.

La funcionalidad será dividida en tres grandes capas o bloques funcionales a saber:

- **@tfg-apps:** contiene cada uno de los *microfrontends* o aplicaciones que forman la aplicación final. En este bloque se definirán proyectos para hacer frente a las distintas entidades de negocio de la aplicación, es decir, cuentas y tarjetas. También se definirá un proyecto, @tfg-apps/app, que sirva como contenedor en el cual integrar el resto de aplicaciones.
- **@tfg-core:** bloque en el que implementar toda la funcionalidad transversal a las distintas aplicaciones. En este bloque se encontrarán proyectos tales como un cliente de HTTP para realizar peticiones al servidor de la aplicación, o una librería a través de la cual gestionar el enrutamiento y la navegación. Contará, además, con un proyecto especial, @tfg-core/bootstrap, mediante el cual cargar el entorno de ejecución de SystemJS y el contenedor para el resto de *microfrontends*, @tfg-apps/app.
- **@tfg-style:** este bloque agrupará aquellos proyectos relacionados con el estilo global de la aplicación, como, por ejemplo, proyectos mediante los cuales implementar una guía de estilos en la que se definan los tamaños de fuente o el esquema de colores a emplear por la aplicación o proyectos que aporten utilidades para la disposición del contenido.

La relación entre los distintos bloques y proyectos queda reflejada en el diagrama de componentes de la figura 5.1:

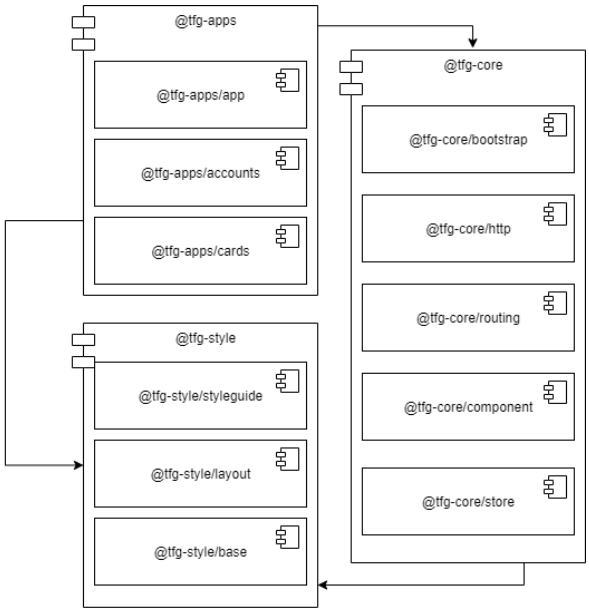

Figura 5.1: diagrama de componentes del caso de estudio

Cada una de las aplicaciones o *microfrontends* cuya finalidad sea crear contenido, deberá exportar un componente web con tal de poder integrarse con el resto de aplicaciones y este será su único requisito para la integración. El componente web podrá ser desarrollado haciendo uso de la tecnología que se desee o incluso empleando, internamente, otro modelo de componente como los propuestos por Angular, Vue, React o cualquier otra tecnología.

En cuanto a la funcionalidad transversal, aquellas utilidades a emplear por el resto de aplicaciones, deberán ser desarrolladas en JS nativo o empleando librerías cuyo uso no sea específico a una tecnología concreta con el fin de permitir su consumo desde cualquier *microfrontend.*

Finalmente, cada proyecto deberá incluir los siguientes archivos o directorios:

- ***package.json:*** archivo en el que se configuran los aspectos del proyecto tales como el nombre, la versión o sus dependencias. Será empleado por Yarn y Lerna a la hora de gestionar el mono repositorio.
- ***build-config.js:*** archivo en el cual se empleará el proyecto @tfg-builder/builder para exportar una configuración de empaquetado específica a este proyecto. Se deberán indicar, al menos, los puntos de entrada al proyecto.
- ***src:*** carpeta en la que almacenar el código fuente del proyecto.

5.3. Implementación

A la hora de implementar el caso de estudio, se ha hecho uso del mismo mono repositorio en el que se aloja la herramienta de desarrollo. De este modo, ha sido posible realizar cambios en esta última de forma rápida tras detectar cualquier problema o carencia durante su puesta a prueba.

5.3.1. Tecnologías empleadas

Durante la implementación del caso de estudio se han empleado diversas tecnologías con el fin de cubrir los aspectos relacionados con la aplicación que la herramienta desarrollada no cubre, es decir, aquellas a ser empleadas en el cliente y no durante el desarrollo y aquellas que no tienen relación directa con el alcance de este trabajo.

A diferencia de las herramientas mencionadas en el capítulo 3, estas no tendrán mayor impacto a la hora de definir la arquitectura de la aplicación, y, por tanto, podrían ser perfectamente reemplazadas, la elección de unas u otras queda en manos del desarrollador.

Las herramientas empleadas en este caso han sido:

- **JSON server[45]:** librería que permite implementar un servidor HTTP totalmente funcional a través del cual consumir datos de prueba mientras se desarrolla la aplicación.
- **SuperAgent[46]:** librería para realizar peticiones HTTP centrada en la flexibilidad y la sencillez.

[45] Repositorio oficial de JSON server: https://github.com/typicode/json-server
[46] Página oficial de SuperAgent: https://visionmedia.github.io/superagent/

- **Navaid[47]:** librería para gestionar el enrutamiento en el navegador. Es común que este tipo de librerías no permitan instanciar más de un enrutador al mismo tiempo, se ha elegido esta precisamente por la posibilidad de crear múltiples instancias del enrutador a la vez, lo que hará que distintos *microfrontends* puedan crear el suyo propio si lo necesitan.
- **LitElement:** librería para definir e implementar componentes web. Se hará uso de LitElement, por tratarse de un complemento sencillo y eficaz a la vez que extensible, que cubre los casos de uso no abordados por el estándar sin introducir cambios sustanciales a la hora de desarrollar.
- **Material Web Components [48] :** librería de componentes basada en componentes web que ofrece una gran colección de componentes, como botones o campos de texto, implementados según la especificación de Material Design[49].

5.3.2. Configuración del proyecto

Antes de empezar el proceso de desarrollo es necesario configurar el proyecto raíz donde se indican las dependencias de la aplicación necesarias para su desarrollo. Para ello se configura en el archivo *package.json* del directorio raíz del proyecto las propiedades *workspaces*, indicando en qué directorios se encuentran el resto de proyectos de la aplicación, y *devDependencies*, especificando la lista de dependencias de desarrollo del proyecto y su versión.

A continuación, ha de instalarse Lerna e indicar en su propio archivo de configuración que debe emplear Yarn como gestor de paquetes y, una vez, más, cuáles son los proyectos pertenecientes al mono repositorio.

Una vez configuradas las herramientas que serán empleadas para la gestión del mono repositorio, es posible instalar y configurar el resto de herramientas de desarrollo: @tfg-commands/cli, @tfg-config/eslint, @tfg-config/stylelint, @tfg-config/prettier, Husky y JSON server.

Finalmente, se definen en el *package.json* las *scripts* que el desarrollador tendrá a su disposición para interactuar con esas herramientas. Como se aprecia en la figura 5.2, Se definen *scripts* para iniciar el proceso de desarrollo, empaquetar la aplicación con distintas opciones, lanzar pruebas unitarias o de integración y ejecutar los destinos analizadores estáticos de código:

[47] Repositorio oficial de Navaid: https://github.com/lukeed/navaid
[48] Repositorio oficial de Material Web Components: https://github.com/material-components/material-components-web-components
[49] Página oficial de Material Desing: https://material.io/

```
16   "scripts": {
17     "start": "run-p build:dev serve:dev mock",
18     "test": "tfg ut",
19     "test:changed": "tfg ut -s HEAD@{1}",
20     "e2e": "tfg e2e",
21     "lint": "run-p lint:*",
22     "lint:js": "eslint --fix **/*.js",
23     "lint:css": "stylelint --fix **/*.css",
24     "build": "tfg build",
25     "build:changed": "tfg build -s HEAD@{1}",
26     "build:dev": "tfg build -w",
27     "serve": "tfg serve",
28     "serve:dev": "tfg serve -p dev",
29     "serve:e2e": "tfg serve -p e2e",
30     "mock": "json-server -q -w db.json",
31     "prepare:e2e": "run-p mock serve:e2e"
32   },
```

Figura 5.2: scripts del proyecto

Cualquiera de estas *scripts* podrá ser ejecutada a través del comando *yarn* seguido del nombre de la *script.*

Una vez finalizada la configuración del proyecto, es posible empezar el proceso de desarrollo ejecutando *yarn start*, cuyo resultado se muestra en la figura 5.3.

```
info WATCH Watcher started
[Browsersync] Access URLs:
 ---------------------------------------
       Local: http://localhost:8080
    External: http://192.168.42.65:8080
```

Figura 5.3: resultado del comando yarn start

5.3.3. Configuración de las herramientas

Después de instalar las herramientas a emplear durante el desarrollo y definir la forma en que pueden ser empleadas, es necesario configurar algunas de ellas.

Para configurar los analizadores estáticos de código, basta con crear un fichero con un nombre específico, que, convencionalmente, suele ser un punto, seguido del nombre de la herramienta, finalizando en *rc.js,* que exporta la configuración definida por los proyectos del espacio de nombres @tfg-config. La implementación de los archivos de configuración para ESLint, Stylelint y Prettier puede observarse en la figura 5.4.

```
1  // .prettierrc.js
2  module.exports = require('@tfg-config/prettier');
3
4  // .stylelintrc.js
5  module.exports = require('@tfg-config/stylelint');
6
7  // .eslintrc.js
8  module.exports = require('@tfg-config/eslint').client;
```

Figura 5.4: configuraciones de los analizadores de código

Al emplear JSON Server es necesario crear un fichero en formato JSON donde insertar los datos que después podrán ser consumidos y modificados mediante peticiones HTTP, esto se hace en el fichero *db.json*.

Por último, se configura Husky en el fichero *.huskyrc.js* de la forma que aparece en la figura 5.5, para definir los eventos de Git ante los cuales será necesario reconstruir los proyectos que hayan sido modificados, haciendo uso de la *script* de nombre *build:changed.* Estos eventos son, después de integrar cambios, *merge,* y después de cambiar los archivos del directorio de trabajo, *checkout.*

```
module.exports = {
  hooks: {
    'post-checkout': 'yarn build:changed',
    'post-merge': 'yarn build:changed',
  },
};
```

Figura 5.5: configuración de Husky

La configuración de Husky no se exporta desde la herramienta para dar la posibilidad al usuario de definir comandos adicionales a ejecutar ante estos eventos, aunque podría exportarse la configuración anterior como la propuesta para su uso por defecto.

5.3.4. Estilos compartidos

Cada proyecto de la aplicación cuenta con la posibilidad de definir y emplear sus propias hojas de estilo, no obstante, con el fin de reutilizar ciertas reglas de estilo y asegurar un diseño consistente, se elaboran varios proyectos cada uno con una función específica.

El primero, @tfg-style/base, define una serie de reglas de estilo globales que tendrán efecto en la página, como el tamaño de fuente o la familia de fuentes a emplear, el segundo, @tfg-style/layout, cuenta con una serie de utilidades para la disposición del contenido de las que los distintos componentes podrán hacer uso y, finalmente, el proyecto @tfg-style/styleguide define una serie de variables CSS que deberán ser empleadas por el resto de hojas de estilo de la aplicación, como, por ejemplo, el esquema de colores que aparece en la figura 5.6.

```
:root {
  --tfg-color-background: #fff;
  --tfg-color-primary: #6200ee;
  --tfg-color-primary--light: #9e47ff;
  --tfg-color-primary--dark: #0400ba;
  --tfg-color-secondary: #018786;
  --tfg-color-secondary--light: #4fb7b6;
  --tfg-color-secondary--dark: #005959;
  --tfg-color-on-primary: #fff;
  --tfg-color-on-secondary: #fff;
  --tfg-color-error: #b00020;
  --tfg-color-info: #1565c0;
  --tfg-color-success: #8bc34a;
  --tfg-color-warning: #ffa726;
}
```

Figura 5.6: esquema de colores de la aplicación

De esta forma es posible asegurar un diseño consistente entre componentes de distintos *microfrontends* y, al centralizar la configuración en un punto, realizar cambios que tengan efecto en la totalidad de la aplicación resulta inmediato.

5.3.5. Modelo de componente base

Si bien tener la posibilidad de emplear modelos de componente diferentes en distintas partes de la aplicación puede resultar provechoso a la hora de compartir código, evaluar nuevas tecnologías o permitir que desarrolladores con distintos conocimientos colaboren en un mismo proyecto, este no debería ser el estándar a la hora de desarrollar una aplicación.

Adoptar un modelo de componente específico aporta consistencia y reduce la cantidad de código que ha de ser descargada para ejecutar la aplicación, por tanto, en esta aplicación se define un modelo de componente base en el proyecto @tfg-core/component mediante una subclase de LitElement. El motivo por el que se extiende de LitElement y no se emplea directamente es que, de esta forma, es posible hacer uso de todas sus características a la vez que se deja la puerta abierta para introducir modificaciones o mejoras si fuese necesario, sin necesidad de reescribir el código del resto de aplicaciones.

5.3.6. Enrutamiento y navegación

Uno de los aspectos a abordar en el desarrollo de toda aplicación web es el sistema de rutas. En este caso se requiere de una solución que, ante una navegación por parte del usuario, permita descargar y, posteriormente, insertar, un componente web en un elemento HTML específico. Además, debe contar con la posibilidad de instanciar varios enrutadores, pues se pretende que cada *microfrontend* pueda contar con su propio sistema de rutas, evitando así una configuración centralizada que con el tiempo sería poco mantenible.

Para cubrir con los requisitos descritos anteriormente, se emplea Navaid, una librería muy ligera que permite asociar una ruta a una función, la cual será ejecutada cuando se navegue a esa ruta. Navaid no funciona con componentes web que hagan uso de *Shadow DOM* [61], pero, afortunadamente, al tratarse de un proyecto de código abierto, ha sido posible copiar y modificar su código fuente permitiendo así su uso en esta aplicación.

Ya que Navaid ofrece una solución de bajo nivel, se desarrolla una utilidad que hace uso de la misma y permite crear enrutadores definiendo las rutas en un objeto de configuración. Para cada ruta es posible especificar:

- ***path***: segmento del URL al que se asocia el objeto de configuración de la ruta. La configuración de este atributo es obligatoria.
- ***tag:*** etiqueta HTML que se desea insertar en el documento cuando la ruta sea activada
- ***redirect:*** ruta a la que se desea redirigir cuando esta se active. El parámetro tag tiene preferencia sobre esta opción.
- ***load:*** función que el enrutador ejecutará antes de realizar la navegación. Resulta especialmente útil para descargar el código empleado por una página de forma dinámica.

- ***hasChildren:*** parámetro booleano que debe configurarse como booleano en caso de que la ruta cargue un componente que instancie su propio enrutador. Al detectar esta propiedad, se añadirá al *path* especificado el segmento */:__CHILD_ROUTE?*, un patrón empleado por Navaid para especificar parámetros opcionales. Si se toma como ejemplo la ruta configurada en la figura 5.7, navegaciones tanto al segmento de URL principal, */accounts*, como a segmentos secundarios, como */accounts/list*, resultarán en la activación de esta ruta.

```
11  {
12    path: '/accounts',
13    tag: 'tfg-accounts',
14    load: () => import('@tfg-apps/accounts'),
15    hasChildren: true,
16  },
```

Figura 5.7: ejemplo de configuración de ruta con rutas secundarias

Una vez configurado el enrutador, es posible iniciarlo indicando en qué elemento HTML ha de insertar el componente web asociado a la ruta activa haciendo uso del método *start.*

5.3.7. Gestión de estado y comunicación

Las arquitecturas basadas en *microfrontends* plantean una dificultad a la hora de gestionar el estado global de la aplicación y compartir datos entre las distintas sub aplicaciones.

Como solución a este problema, se ha optado por un método de gestión de estado centralizado. Para ello se ha desarrollado el proyecto @tfg-core/store inspirado en Vuex[50], una librería de Vue que proporciona un contendor en el que almacenar el estado de la aplicación y una forma característica de modificarlo. No ha sido posible emplear Vuex pues su funcionamiento está estrechamente ligado a Vue y depende en gran cantidad de los detalles de implementación de sus componentes.

Así pues, se crea un contendor para almacenar el estado de la aplicación que consta de las siguientes propiedades:

- ***state:*** objeto de JavaScript en el que almacenar el estado de la aplicación. Podrá ser leído, pero no modificado, por los distintos módulos de la aplicación con tal de acceder a sus valores.
- ***actions:*** funciones que pueden invocar otras acciones o modificar el estado del contenedor.
- ***namespace:*** como medida organizativa, se obliga a que cada contenedor esté registrado bajo un espacio de nombres. De esta forma es posible definir barreras entre el estado perteneciente a distintas partes de la aplicación. Las acciones registradas en un espacio de nombres determinado no tendrán acceso al estado definido en otros.

[50] Página oficial de Vuex: https://vuex.vuejs.org/

Además, el contenedor implementa el patrón observador [62], haciendo posible observar los cambios que se efectúan en una parte del estado con el fin de dotar al desarrollador de un mecanismo automático de detección de cambios. Cada vez que una acción modifique una parte del estado todos los observadores asociados a esa parte serán notificados.

Finalmente, se implementa una utilidad, *ConnectStore* que permite definir mapeos entre el estado y las acciones del contendor a las propiedades de un componente, ofreciendo una gran facilidad de uso. En la figura 5.8 se muestra como un componente es capaz de mapear la propiedad *menuItems* definida bajo el espacio de nombres *app* a una propiedad de nombre *items*, cuyo valor se actualizará ante cualquier modificación de esa parte del estado.

```
 9  class TFGAppMenu extends ConnectStore(Component) {
10    static mapState = {
11      app: {
12        menuItems: 'items',
13      },
14    };
```

Figura 5.8: ejemplo de uso de ConnectStore

Empleando este mecanismo para gestionar el estado de la aplicación, cualquier parte es capaz de consultar el estado y ejecutar las acciones de otra, pero es cada una de las partes la que define, internamente, cómo se modifica su estado. De esta forma, es posible definir las interfaces de entrada (acciones) y salida (estado) de cada *microfrontend,* de una forma trazable y, por tanto, mantenible. Y, si se acepta la premisa de que cada acción que realiza el usuario tiene como objetivo modificar una parte del estado de la aplicación, este será el único modelo de comunicación necesario entre aplicaciones.

5.3.8. Gestión de peticiones HTTP

El último aspecto que falta por abordar en cuanto al desarrollo de una aplicación web es la gestión de peticiones HTTP. Para ello existen multitud de librerías de entre las cuales se ha elegido SuperAgent simplemente por su facilidad de uso.

Empleando esta librería, se ha escrito una clase con métodos preconfigurados para cubrir las peticiones HTTP más comunes, es decir, las de lectura, escritura, modificación y borrado. Además, con el objetivo de reutilizar código, se ha definido un sistema por el cual extender esta clase base configurando parámetros que serán empleados por defecto en todas las peticiones a no ser que sean sobrescritos.

En la figura 5.9 se muestra cómo es posible combinar distintas subclases de la clase base configurando los distintos parámetros de las peticiones.

```
2   import {HTTPService} from '@tfg-core/http';
3
4   class APIService extends HTTPService {
5     get config() {
6       return this.extendedConfig(super.config, {
7         baseURL: 'http://localhost:3000/api',
8         headers: {
9           Authorization: `Bearer ${token}`,
10        },
11      });
12    }
13  }
14  class AccountsService extends APIService {
15    get config() {
16      return this.extendedConfig(super.config, {
17        baseURL: 'accounts',
18      });
19    }
20  }
```

Figura 5.9: ejemplo de composición usando HTTPService

Las peticiones realizadas desde instancias de la clase *APIService* apuntarán al URL *http://localhost:3000/api* e incluirán la cabecera de autorización con el valor configurado. Las realizadas desde *AccountsService* extenderán la configuración de *APIService* de modo que emplearán como URL *http://localhost:3000/api/accounts* e incluirán las mismas cabeceras. De esta forma se consigue centralizar la configuración de modo que si se requieren cambios estos solo necesiten aplicarse en un punto, y favorecer la reutilización de código.

5.3.9. Aplicación de arranque

Con tal de iniciar la aplicación es necesario, por un lado, incluir el entorno de ejecución de SystemJS y, por otro, insertar el código de arranque en el documento HTML de la aplicación.

Para ello se crea el proyecto @tfg-core/bootstrap que importa SystemJS, y lo utiliza para cargar el contenedor del resto de aplicaciones además de los proyectos que definen la guía de estilos de la aplicación y sus estilos base. Al cargar la estructura básica de la aplicación, pero sin contenido, la carga inicial es más rápida. Será el contendor el que elija qué *microfrontend* cargar en función de la ruta a la que haya accedido el usuario.

Para generar el documento HTML de la aplicación se define una plantilla que incluye enlaces a archivos de fuente e iconos que serán empleados más tarde y renderiza el componente *tfg-app*, el componente raíz de la aplicación.

Al empaquetar este proyecto, se hace uso del complemento de Rollup.js @tfg-builder/rollup-plugin-entry para generar el documento HTML a partir de la plantilla mencionada anteriormente, haciendo que incluya el código del proyecto y en la que, posteriormente, se insertará el *import map* generado por el resto de procesos de empaquetado. En la figura 5.10 se puede observar el fichero HTML generado y cómo incluye, en la línea 17, el paquete de la aplicación de arranque mediante un elemento de tipo *script*, incluyendo en el nombre el código resultante de calcular el hash del contenido del paquete.

```html
<!doctype html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link href="https://fonts.googleapis.com/css?family=Roboto:300,400,500" rel="stylesheet">
  <link href="https://fonts.googleapis.com/css?family=Material+Icons&display=block" rel="stylesheet">
  <link rel="preload" as="script" href="/core/bootstrap/bootstrap.246d16d0.js">
  <title>TFG Bank</title>
</head>

<body>
  <tfg-app></tfg-app>
  <script type="systemjs-importmap">${importMap}</script>
  <script src="/core/bootstrap/bootstrap.246d16d0.js"></script>
</body>

</html>
```

Figura 5.10: documento HTML de la aplicación

5.3.10. Componente raíz

El componente raíz de la aplicación define la estructura de la página y actúa como un contenedor en el que cargar el resto de aplicaciones. Se define en el proyecto @tfg-apps/app con la etiqueta *tfg-app.*

Cuando se realice cualquier navegación a una página, el enrutador de esta aplicación cargará el *microfrontend* asociado a esa ruta e insertará el componente web especificado en el lugar adecuado. Si, por el contrario, se navega a una ruta inexistente, el contenedor cargará una página avisando de la situación y dando la posibilidad de navegar a la página de inicio.

Para cumplir con los propósitos anteriores, define el componente *tfg-app-layout* que incluye toda una serie de componentes secundarios como el menú de navegación o la cabecera de la aplicación y, dentro de este componente, inserta el elemento HTML *main,* el cual sirve para especificar el contenido principal de la aplicación, y que el enrutador empleará para insertar los componentes de cada una de las rutas a medida que sean activadas.

En la figura 5.11 se muestra parte del código empleado en la definición del componente *tfg-app*. En el método *render,* definido al comienzo de la línea 22, se especifica el HTML que debe renderizar este componente al ser insertado en el documento.

```
8  class TFGApp extends Component {
9    static styles = appCSS;
10
11 >  async connectedCallback() {···
16   }
17 >  disconnectedCallback() {···
21   }
22   render() {
23     return this.html`
24       <tfg-app-layout>
25         <main id="outlet" class="app"></main>
26       </tfg-app-layout>
27     `;
28   }
29 }
30
31 customElements.define('tfg-app', TFGApp);
```

Figura 5.11: definición del componente tfg-app

El resultado de la carga del componente anterior por la aplicación de arranque resulta en la página observable en la figura 5.12, en la que se muestra la cabecera y el botón para desplegar el menú de navegación lateral seguida de un espacio en blanco donde se mostrará el contenido definido por los otros *microfrontends* cuando sean implementados y registrados en el enrutador de esta aplicación.

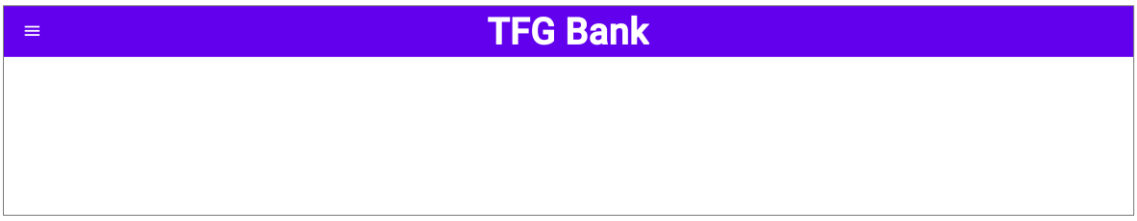

Figura 5.12: componente raíz de la aplicación

5.3.11. Aplicación de tarjetas

La aplicación de tarjetas cuenta con una página en la que se muestra la lista de tarjetas contratadas por el usuario.

Para implementar esa página se define un componente web, *tfg-cards,* encargado de actuar como contenedor para la aplicación de tarjetas. Este componente hace uso de las utilidades expuestas por los proyectos transversales de la aplicación para definir su propio sistema de rutas, un cliente para realizar las peticiones HTTP relacionadas con las tarjetas y el contenedor en el que gestionar su estado interno, es decir, la lista de tarjetas. En general, cada *microfrontend* empleará estas tres utilidades pues es común que se deban afrontar estos aspectos, enrutamiento, peticiones HTTP y gestión de estado, en cualquier parte de la aplicación.

En el enrutador de esta aplicación se define una ruta para cargar el componente *tfg-cards-list* el cual obtiene y presenta cada una de las tarjetas mediante el componente *tfg-cards-summary,* que será empleado además en la aplicación de cuentas al acceder a la página de detalles de una cuenta específica.

Compartir funcionalidad entre aplicaciones es posible gracias a que la herramienta @tfg-builder/builder permite definir varios puntos de entrada para una aplicación. En la figura 5.13 se muestra la configuración de empaquetado de este proyecto definiendo el componente principal del mismo y el componente que será más tarde reutilizado.

```
3   module.exports = rollupConfig.project({
4     input: {
5       '@tfg-apps/cards': 'index.js',
6       '@tfg-apps/cards/summary': 'src/components/summary.js',
7     },
8   });
```

Figura 5.13: configuración de empaquetado de la aplicación de tarjetas

De esta forma el enrutador del contenedor de aplicaciones podrá importar el paquete @tfg-apps/cards que define el componente web principal de la aplicación de tarjetas y, cualquier otra aplicación que necesite emplear el componente *tfg-cards-summary* podrá hacerlo importando el paquete @tfg-apps/cards/summary.

5.3.12. Aplicación de cuentas

La aplicación de cuentas es similar a la de tarjetas en cuanto a que define un componente principal que registra el contenedor de estado y el enrutador para esta aplicación, además de configurar un cliente HTTP para interactuar con el servicio de cuentas.

En esta aplicación se crean dos páginas, una para mostrar la lista de cuentas, definida por el componente *tfg-accounts-list* y otra para mostrar los detalles específicos de cada cuenta, definida por el componente *tfg-accounts-detail.* A diferencia de la aplicación anterior, exporta un solo paquete @tfg-apps/accounts, que será empleado por el enrutador del contenedor de aplicaciones cuando se navegue a la ruta definida para esta aplicación.

5.4. Pruebas

Con el objetivo de ilustrar cómo sería el proceso de pruebas bajo este marco de trabajo, se ha desarrollado una prueba de integración y otra unitaria.

5.4.1. Pruebas de integración

La prueba de integración hace uso de CodeceptJS y WebDriverIO para verificar que el título de la aplicación sea el correcto, para ello se define una *característica* de prueba que engloba una serie de escenarios que deben ser validados y, para cada escenario, se especifican la lista de acciones y comprobaciones que han de ser ejecutadas. La implementación de este caso de prueba puede observarse en la figura 5.14, donde se define la característica *home,* que hace referencia a la página de inicio, y el escenario *Enter home page*, es decir, qué debe ocurrir cuando se navega a la página principal. Dentro de este escenario se dan las instrucciones para llegar a esa situación y ejecutar las aserciones correspondientes.

```
1   Feature('home');
2
3   Scenario('Enter home page', (I) => {
4     I.amOnPage('/');
5     I.see('TFG Bank', {
6       shadow: ['tfg-app', 'tfg-app-layout', 'tgf-app-header', 'h1'],
7     });
8   });
```

Figura 5.14: prueba de integración de la aplicación

Como el título es responsabilidad del proyecto @tfg-apps/app, ya que este es el que define la cabecera de la página, es posible ejecutar esta prueba con el comando *yarn e2e -p @tfg-apps/app.*

5.4.2. Pruebas unitarias

La prueba unitaria desarrollada comprueba que el componente principal del contenedor de aplicaciones defina un componente web con la etiqueta *tfg-app*. Para ello se emplea un mecanismo de aserciones basado en describir una parte del sistema mediante la especificación de su comportamiento deseado, así, en la figura 5.15 se describe el comportamiento de la aplicación especificando que debería definir el componente web *tfg-app.*

```
1  import TFGApp from './app';
2  describe('App', function () {
3    it('should define tfg-app', () => {
4      expect(customElements.get('tfg-app')).toBe(TFGApp);
5    });
6  });
```

Figura 5.15: prueba unitaria de la aplicación

Esta prueba puede ser ejecutada con el comando *yarn test -p @tfg-apps/app*, el cual ejecutará todas las pruebas unitarias asociadas al proyecto especificado.

5.5. **Evaluación**

La implementación del caso de estudio ha permitido evaluar distintos aspectos relacionados con el ecosistema propuesto y la herramienta desarrollada para dar soporte al mismo.

Con tal de realizar la evaluación, se describirá el procedimiento seguido para implementar la página que muestra la lista de tarjetas del usuario, de modo que pueda obtenerse una visión general del funcionamiento del ecosistema.

Una vez hecho esto, se evaluará la solución propuesta con el fin de especificar qué aspectos suponen una mejora respecto a otros ecosistemas de desarrollo, cuáles resulta negativos, y en qué puntos la implementación de la solución tiene posibilidad de mejora.

5.5.1. Procedimiento de desarrollo

Una vez definido y configurado el proyecto de la manera descrita en el capítulo 5.3, para implementar un nuevo *microfrontend* ha sido necesario crear un directorio donde definir el proyecto. Para ello se crea el archivo *package.json* que lo define y se instala @tfg-builder/builder como dependencia de desarrollo, empleada para definir la configuración de empaquetado.

A continuación, se crea la carpeta *src*, que contendrá el código fuente del proyecto, y se escribe el archivo que actuará como punto de entrada de esta aplicación. Una vez hecho esto, se crea un archivo de nombre *build-config.js* en el que se define el proyecto anterior como un punto de entrada para el empaquetado, y se le asigna un alias mediante el cual podrá ser consumido por otras aplicaciones.

En este momento el proyecto ha sido configurado y es posible comenzar su desarrollo ejecutando el comando *yarn start* que detectará la existencia de cambios en los archivos del repositorio, en este caso, los del proyecto de tarjetas, y empaquetará la aplicación. Además, iniciará el proceso de detección de cambios que volverá a empaquetarla tras cualquier modificación en su código fuente y el servidor de estáticos que recargará el navegador tras detectar modificaciones en los paquetes.

Una vez hecho esto, se implementan tres módulos encargados de resolver la funcionalidad transversal del proyecto de tarjetas: un módulo que emplea @tfg-core/routing con tal de definir el enrutador de la aplicación, otro módulo que hace uso de @tfg-core/store para implementar el contenedor de estado de la aplicación y, finalmente un módulo que, sirviéndose de @tfg-core/http, exporta un cliente HTTP preconfigurado para comunicarse con el servicio de tarjetas. A medida que se van empleando otros módulos de la aplicación, estos son configurados como dependencias del proyecto actual.

Ahora que ya están listos todos los módulos necesarios para implementar la funcionalidad de la aplicación, se implementa el componente principal de la misma, *tfg-card*, que define el elemento donde el enrutador de tarjetas debe insertar los componentes asociados a la ruta activa y registra el contenedor de estado en el módulo @tfg-core/store, de modo que sea accesible desde el resto de componentes. Además, se modifica el enrutador del componente raíz, *tfg-app*, añadiendo una ruta para cargar este *microfrontend.* A partir de este punto ya sería posible navegar a la ruta en la que se registró esta aplicación para evaluar su desarrollo.

Finalmente, se crean dos componentes *tfg-cards-list* y *tfg-cards-summary*, el primero empleará el contenedor de estado y el cliente HTTP para recuperar la lista de tarjetas y hará uso del segundo para pintar los detalles de cada una de ellas. Además, se definen hojas de estilo y se emplean los estilos definidos en los proyectos del espacio de nombres @tfg-styles, para personalizar la apariencia de estos componentes.

Cabe mencionar que, durante todo el proceso, los analizadores estáticos de código previamente configurados han estado formateando el código según las reglas especificadas y avisando al desarrollador de posibles errores.

Una vez finalizada la tarea, o paralelamente al desarrollo, sería posible definir pruebas tanto unitarias como de integración que verifiquen su correcta implementación.

Cuando se integran estos cambios en el repositorio, la herramienta Husky ejecuta el comando *yarn build:changed,* que construirá los proyectos modificados desde el punto anterior en el que se encontraba el repositorio, es decir, @tfg-apps/cards y @tfg-apps/app. Una vez empaquetados, es posible lanzar las pruebas asociadas a los proyectos modificados empleando los comandos *yarn test:changed* y *yarn e2e:changed.*

El uso de estos comandos no sólo tendrá efecto en los proyectos modificados si no también en sus dependientes, como el proyecto de arranque de la aplicación, @tfg-core/bootstrap, depende internamente de @tfg-apps/app, sus pruebas asociadas, si las hubiese, también serían ejecutadas.

Si la ejecución de las pruebas es satisfactoria, puede ejecutarse el comando *yarn release*, que hará uso de Lerna para, de manera automática, modificar la versión de todos los proyectos modificados a la especificada, actualizando en el proceso todos los paquetes dependientes e integrando los cambios en el repositorio remoto.

En el momento en que se desee desplegar la aplicación en producción, bastará con subir los archivos generados durante el empaquetado al servidor de estáticos deseado. Los archivos cuyo contenido haya sido modificado verán su nombre alterado para reflejar esos cambios con un nuevo hash, y este cambio de nombre quedará reflejado en el *import map* de la aplicación, haciendo que el navegador consuma la última versión de los archivos, pero pueda seguir empleando la versión cacheada de aquellos que no presenten modificaciones.

5.5.2. Aspectos positivos

Tras evaluar el proceso de implementación del caso de estudio se aprecian una serie de aspectos positivos, entre los cuales podemos diferenciar dos grupos, aquellos aportados por el ecosistema y la herramienta y aquellos frutos de las decisiones técnicas tomadas.

En cuanto al uso de la herramienta y la implantación del ecosistema, destacan los siguientes aspectos:

- Configurar los analizadores estáticos de código es inmediato, al igual que ocurre en el resto de herramientas de desarrollo de la industria.
- El hecho de construir los paquetes de forma automática ante cambios en el repositorio o cuando se detectan cambios locales al iniciar el proceso de desarrollo hace que el contenido de los paquetes esté siempre actualizado, correspondiéndose con el código fuente de las aplicaciones, lo que hace que el proceso de desarrollo sea más rápido, pues no es necesario empaquetar todas las aplicaciones cada vez que se desea desarrollar. No obstante, como se explica a continuación en el capítulo 5.5.4, este proceso es mejorable.
- Generar el documento HTML de la aplicación, insertando los archivos necesarios para su ejecución de forma automática reduce el trabajo necesario por el desarrollador y, por tanto, la posibilidad de introducir errores.
- Emplear una función de hash para reflejar el contenido de cada archivo en el nombre del mismo habilita el uso de estrategias de caché eficientes. Sumando esto al uso de *import maps*, es posible invalidar un archivo en caché cambiando su nombre sin alterar el resto de archivos que hacen uso del mismo, evitando así la invalidación de caché en cascada.
- La flexibilidad total a la hora de configurar el proceso de empaquetado hace que se puedan tomar decisiones concretas en proyectos específicos, como el de arranque de la aplicación, cubriendo así cualquier caso de uso.
- La utilización de módulos CSS para importar hojas de estilo desde código JS permite reutilizar reglas de estilo de forma óptima.
- El ecosistema proporciona un mecanismo de integración de *microfrontends* genérico que soporta cualquier caso de uso. Su funcionamiento se basa en compartir código JS evitando la carga de dependencias duplicadas y resolviendo la localización de cada uno de ellos. Por tanto, es posible realizar la integración mediante módulos de JS, componentes web o a través del sistema de enrutamiento. El desarrollador tendrá la posibilidad de definir su arquitectura basada en *microfrontends* como desee.
- La posibilidad de conocer qué proyectos han sido modificados además de qué proyectos dependen de estos hace que el impacto de un cambio sea trazable, permitiendo ejecutar las pruebas asociadas a todos los proyectos cuyo comportamiento es susceptible de haber sido alterado, reduciendo por tanto el riesgo de introducir errores.

En cuanto a las decisiones técnicas, es posible identificar tres factores que contribuyen de forma positiva al proceso de desarrollo.

En primer lugar, la adopción de una arquitectura basada en *microfrontends* permite que cada área funcional de la aplicación pueda ser abordada como una aplicación en sí misma, que tendrá dependencias únicamente con los módulos encargados de resolver aspectos transversales del desarrollo de aplicaciones web, como la gestión de peticiones HTTP, el sistema de rutas o el estado de la aplicación. Gracias a este modelo, las barreras entre aplicaciones pueden definirse de forma clara y por tanto serán más difíciles de romper. El hecho de que cada aplicación tenga como objetivo implementar una funcionalidad concreta hará que su complejidad sea constante respecto al tamaño y la evolución del sistema, pues no dependerá del resto de aplicaciones. La complejidad del sistema, por tanto, será repartida de forma lógica entre las distintas aplicaciones, según la funcionalidad que aporte cada una.

En segundo lugar, la elección como sistema de control de versiones de un mono repositorio en el que cohabitan los distintos proyectos que forman la aplicación permite realizar cambios transversales de forma atómica y gestionar el versionado y las interdependencias entre proyectos de forma automática.

Finalmente, las decisiones tomadas en cuanto a aspectos concretos del desarrollo suponen una serie de características beneficiosas. A saber:

- Definir una guía de estilos mediante código, empleando variables de CSS, aporta uniformidad al diseño y la posibilidad de realizar modificaciones de forma efectiva.
- En cuanto al sistema de rutas, dar la posibilidad a cada *microfrontend* de definir su propio enrutador hace que cada aplicación pueda actuar como contenedor de otras aplicaciones, lo que permite una total independencia al no necesitar registrar cada una de las rutas en el enrutador principal de la aplicación.
- El uso de una aplicación que actúe como contenedor y defina la estructura de la página permite una primera carga rápida mientras que el contenido se obtiene de forma progresiva.
- El uso de componentes web como modelo de componente base permitiría, si se desease, compartir componentes de esta aplicación con otras y, dado que el sistema de rutas funciona con componentes web, sería posible emplear un componente o una aplicación entera programada empleando otra tecnología de desarrollo de componentes siempre y cuando esta fuera envuelta en un componente web.
- Emplear un sistema de estado centralizado permite conocer los datos disponibles y los medios por los que modificarlos, así como hacer uso de estos desde cualquier parte de la aplicación. Además, al obligar a que cada contenedor de estado sea registrado empleando un espacio de nombres se facilita la separación de responsabilidades, evitando que partes de la aplicación modifiquen estado que no les pertenece.

5.5.3. Aspectos negativos

La adopción de esta solución conlleva una gran inversión en tecnología e infraestructura, son muchas las tecnologías que intervienen en el proceso de desarrollo y la curva de aprendizaje inicial puedan resultar elevada. En proyectos de menor envergadura, el balance entre los beneficios aportados y la complejidad inicial de incorporar esta solución podría resultar negativo, haciendo su uso contraproducente.

5.5.4. Posibilidades de mejora

Durante la implementación de la solución se han detectado las siguientes posibilidades de mejora, ordenadas de menor a mayor importancia:

- La configuración inicial de un proyecto resulta tediosa, la herramienta podría contar con un comando que permita crear un proyecto empleando una configuración por defecto.
- Actualmente la herramienta no permite la carga de distintas versiones de una misma librería en tiempo de ejecución. Si bien esta práctica resulta poco recomendable, pueden darse casos en los que sea necesario. Este caso de uso podría ser soportado mediante la implementación de un complemento de Rollup.js y empleando el *import map* de la aplicación.
- Al empaquetar todas las dependencias de terceros de la aplicación de forma independiente al código que las utiliza, Rollup.js no puede optimizar el código empleando *tree-shaking*, pues es incapaz de detectar qué partes del código van a ser empleadas, y cuáles no, por la aplicación, resultando en un tamaño de empaquetado ligeramente superior. Esto podría resolverse, seguramente, implementando un complemento de Rollup.js y empleando un analizador sintáctico con el fin de detectar qué módulos exportados por las distintas librerías acaban siendo empleados por la aplicación.
- Hacer uso de las herramientas de ejecución de pruebas desde una herramienta de comandos hace necesario, o, al menos, deseable, dar soporte a todas las opciones que las herramientas soportan de por sí. Esto no se realiza actualmente y por tanto la funcionalidad de las herramientas es limitada.
- Es posible mejorar el comando de empaquetado al emplear la opción que construye las aplicaciones modificadas desde un punto de la historia del repositorio. Los proyectos del cliente no deberían ser construidos ante cambios en sus dependencias en tiempo de ejecución, pues estas se empaquetan por separado. Los proyectos únicamente deberían ser empaquetados al modificarse estos o sus dependencias de desarrollo que intervienen en el resultado del paquete final.

- A pesar de contar con un proceso que asegura que los paquetes siempre están actualizados, este proceso no es del todo óptimo. Realmente no se detectan los proyectos que necesitan ser construidos, sino los modificados. Es posible, por tanto, que un proyecto presente cambios en la historia del repositorio mientras que su última versión ya ha sido construida. Para solucionar este problema, que mejoraría en gran medida el tiempo empleado empaquetando la aplicación, sería necesario asociar los paquetes generados al sistema de control de versiones en lugar de al código fuente, almacenando una referencia al punto en el que fueron empaquetados. Existe una librería en el ecosistema de Rush.js, @rushstack/package-deps-hash[51], que sirve precisamente para esto.
- Actualmente no se cuenta con un sistema que permita detectar cambios en las dependencias de terceros y sus versiones instaladas. Es por esto que cada proceso de empaquetado vuelve a empaquetar, por defecto, todas las librerías. Este proceso consume una gran cantidad de tiempo en comparación con el empleado en construir únicamente los proyectos modificados. Una posible vía de investigación para dar con una solución a este problema y empaquetar las dependencias de terceros únicamente cuando sea necesario sería mediante el análisis de un fichero generado por Yarn, *yarn.lock,* que contiene la información sobre todas las dependencias del espacio de trabajo y sus versiones.

[51] Repositorio oficial de @rushstack/package-deps-hash: https://github.com/microsoft/rushstack/tree/master/libraries/package-deps-hash

6. Conclusiones

Respecto al objetivo principal de este trabajo, ha sido posible constatar mediante la implementación del caso de estudio que, efectivamente, el ecosistema definido y la herramienta desarrollada con tal de facilitar su aplicación permiten abordar el desarrollo de aplicaciones web de forma sostenible, es decir, evitando el incremento de la complejidad del sistema a medida que este evoluciona.

En cuanto a los objetivos específicos de esta memoria, es posible concluir que:

- Un ecosistema basado en el uso de una arquitectura modular basada en *microfrontends* permite la evolución de una aplicación de forma sostenible, pues, aunque la aplicación evoluciona en su conjunto, la evolución experimentada por cada *microfrontend* de forma individual depende de la funcionalidad ofrecida, y no del resto del sistema. Esto queda reflejado en el hecho de que la complejidad de la aplicación de tarjetas no se ha visto incrementada tras el desarrollo de la aplicación de cuentas.
- Existen multitud de aspectos a tener en cuenta a la hora de definir un ecosistema e implementar una herramienta con la finalidad de ser empleados para el desarrollo de aplicaciones web, especialmente, si se desea dar soporte a una arquitectura basada en *microfrontends*. De la misma manera, existe una gran variedad de tecnologías y técnicas que permiten abordar cada uno de estos aspectos. En esta memoria se han descrito estos aspectos y se han justificado las decisiones técnicas tomadas a la hora de abordar cada uno de ellos.
- Los requisitos funcionales del caso de estudio han servido para poner a prueba la solución propuesta y evaluar tanto el funcionamiento de la herramienta de apoyo al ecosistema como los aspectos positivos y negativos del propio ecosistema. Aunque la evaluación ha sido realizada de forma cualitativa, los resultados han sido, mayormente, positivos.

Como reflexión personal, he de remarcar, en primer lugar, que la solución propuesta no debe ser la elegida por defecto a la hora de desarrollar cualquier aplicación web, su uso sólo debería ser planteado en aquellas aplicaciones en las que por tamaño o complejidad realmente vaya a ser posible obtener un beneficio superior al coste de implantar la solución.

En segundo lugar, creo que, pese a la cada vez mayor aparición de defensores o promotores de este modelo arquitectónico, todavía queda un largo camino por recorrer, pues las herramientas disponibles actualmente en la industria empleadas en el desarrollo de aplicaciones bajo el modelo monolítico tradicional son de gran calidad y cuentan con un gran ecosistema. Las herramientas que hagan frente al desarrollo orientado a *microfrontends* deben estar a la altura a la vez que solucionan los problemas derivados de este modelo arquitectónico.

En tercer lugar, me gustaría hacer hincapié en la importancia de los estándares web a la hora de definir el futuro de la plataforma. El desarrollo de este proyecto no habría sido posible, o la solución propuesta habría sido mucho menos efectiva, de no existir estándares como el de *Web Components*. Así mismo, las propuestas para el estándar empleadas, *Import maps* y *CSS Modules*, han demostrado ser de gran utilidad tras haber podido experimentar con ellas y estoy seguro de que supondrán una gran mejora en el desarrollo de aplicaciones web una vez hayan sido estandarizadas e implementadas por un número representativo de navegadores.

Este trabajo constituye un reflejo de mis conocimientos adquiridos en la universidad y, principalmente, durante mi carreara laboral. Por una parte, he empleado mis conocimientos obtenidos en la universidad sobre el proceso, diseño y mantenimiento de *software* en las asignaturas Proceso, Diseño y Mantenimiento y evolución de *software,* respectivamente, así como aquellos de JavaScript y Node.js adquiridos en la asignatura Tecnología de sistemas de información en la red. Estas asignaturas han servido como base a la hora de adquirir los conocimientos específicos que me han permitido desarrollar este proyecto.

Por otra parte, he tenido la oportunidad de trabajar en múltiples empresas desarrollando aplicaciones web de distinta envergadura. Fue durante el desarrollo de uno de estos proyectos, en el que intervenían, después de más de cinco años de desarrollo, decenas sino cientos de desarrolladores, donde experimenté por primera vez el desarrollo web a gran escala y comencé a buscar soluciones sostenibles para el desarrollo de aplicaciones web. Podría decirse que ese proyecto ha sido la inspiración de esta memoria.

Finalmente, tras este proyecto surgen varias vías de trabajo. En primer lugar, podrían aplicarse las mejoras a la herramienta de apoyo al ecosistema propuestas en el capítulo 5.5.4. En segundo lugar, la herramienta desarrollada no cuenta con un sistema de pruebas automatizado que asegure su correcto funcionamiento y estas deberían ser desarrolladas. En tercer lugar, la definición de métricas con las que evaluar los resultados obtenidos en el caso de estudio permitirían conocer en profundidad el impacto de la solución propuesta en términos cuantitativos. Finalmente, el caso de estudio podría ser enriquecido implementando parte de la aplicación en una tecnología diferente y haciendo uso de un sistema de integración continua para poder apreciar cómo sería el ciclo completo de desarrollo en un entorno más similar al de una aplicación real de cierta envergadura.

7. 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, «Report: Page Weight,» [En línea]. Available: https://httparchive.org/reports/page-weight?start=earliest&end=latest&view=list. [Último acceso: 7 Agosto 2020].

[3] E. DeBill, «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] 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].

[22] 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].

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

[24] 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].

[25] 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].

[26] web.dev, «Tooling.Report,» [En línea]. Available: https://bundlers.tooling.report/. [Último acceso: 5 Agosto 2020].

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

[28] 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].

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

[30] 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].

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

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

[33] 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].

[34] MDN Contributors, «<style>: The Style Information element,» 12 Abril 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style. [Último acceso: 7 Junio 2020].

[35] MDN Contributors, «<link>: The External Resource Link element,» 3 Julio 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link. [Último acceso: 7 Agosto 2020].

[36] 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].

[37] 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].

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

[39] 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].

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

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

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

[43] 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].

[44] 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].

[45] 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.

[46] 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].

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

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

[49] 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].

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

[51] M. Geers, «Micro Frontends,» [En línea]. Available: https://micro-frontends.org/. [Último acceso: 5 Agosto 2020].

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

[53] MDN Contributors, «<iframe>: The Inline Frame element,» 4 Julio 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. [Último acceso: 7 Agosto 2020].

[54] IBM Cloud Education, «MEAN Stack,» 9 Mayo 2019. [En línea]. Available: https://www.ibm.com/cloud/learn/mean-stack-explained. [Último acceso: 6 Agosto 2020].

[55] Canopy, «The single-spa ecosystem,» [En línea]. Available: https://single-spa.js.org/docs/ecosystem. [Último acceso: 6 Agosto 2020].

[56] Nrwl, «Why Nx?,» [En línea]. Available: https://nx.dev/react/getting-started/why-nx. [Último acceso: 6 Agosto 2020].

[57] 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].

[58] S. Chacon y B. Straub, «Customizing Git - Git Hooks,» de *Pro Git*, Apress, 2014.

[59] NPM, «Scoped Packages,» [En línea]. Available: https://docs.npmjs.com/using-npm/scope.html. [Último acceso: 13 Julio 2020].

[60] MDN Contributors, «Preloading content with rel="preload",» 6 Julio 2020. [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Preloading_content. [Último acceso: 14 Julio 2020].

[61] I. Torregrosa, «Support for shadow dom,» 26 Abril 2020. [En línea]. Available: https://github.com/lukeed/navaid/issues/30. [Último acceso: 19 Julio 2020].

[62] A. Osmani, «The Observer Pattern,» de *Learning JavaScript Design Patterns*, O'REILLY, 2017.