Trabajo Fin de Máster

Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

Universidad Nacional de Educación a Distancia

Arquitecturas Orientadas a Servicios

(31105151)

“SaaS de integración continúa bajo demanda como mejora de la experiencia del desarrollador en MicroFrontends”

Autor: **Pedro Pablo Mena Beltrán**

Directora: **Elena Ruiz Larrocha**

Curso: **2020/21**

Convocatoria: **Septiembre**

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Asignatura: **Arquitecturas Orientadas a Servicios**

Código: **31105151**

Título: **SaaS de integración continúa bajo demanda como mejora de la experiencia del desarrollador en MicroFrontends**

Tipología: **Tipo B**

Estudiante: **Pedro Pablo Mena Beltrán**

Directora: **Elena Ruiz Larrocha**

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha:14/09/2021

Quién suscribe:

Autor(a): Pedro Pablo Mena Beltrán
D.N.I/N.I.E/Pasaporte.: 45709543X

Hace constar que es la autor(a) del trabajo:

SaaS de integración continúa bajo demanda como mejora de la experiencia del desarrollador en MicroFrontends

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Resumen

La arquitectura de Microfrontends ha surgido como solución al aumento de la complejidad de aplicaciones en cliente. Las grandes aplicaciones monolíticas resultan difíciles de desarrollar, mantener y escalar. Este nuevo enfoque propone la división del monolito en fragmentos más manejables y que permiten una mejor escalabilidad. Este punto de vista ha seducido a grandes empresas como DAzn, Ikea, SAP, Spotify y Zalando, entre otras. Sin embargo, a día de hoy no ha sido aceptado de forma masiva, principalmente porque aumenta la complejidad técnica y organizativa, entre otros factores.

El objetivo de este documento se centra en dos partes, en primer lugar, aportar más conocimiento sobre el campo de la arquitectura de Microfrontends, recopilando información general sobre qué opciones existen y cuáles son sus ventajas e inconvenientes. Posteriormente y como objetivo principal, se plantea una solución al aumento de la complejidad organizativa durante la fase de desarrollo de microfrontends, para ello se propone la implementación de un servicio de integración continua bajo demanda, que ayude a mejorar la experiencia del desarrollador de Microfrontends.

La conclusión de esta propuesta es que implantando este sistema se puede alcanzar un entorno de desarrollo local sumamente parecido al de producción, sin los inconvenientes de construir e integrar código de otros equipos en cada máquina. Esto supone un desarrollo más ágil, confortable y seguro para cada trabajador, que consigue implementar sobre un entorno completamente integrado con todos sus componentes, pero abstraído de la complejidad que este proceso conlleva.

Palabras clave: Arquitectura de Microfrontends, Arquitectura cliente, Integración continua, Software como servicio, SaaS, Experiencia Desarrollador, DX.

Índice

Lista de abreviaciones 14

Lista de figuras 16

1. Introducción 18

1.1. Motivaciones 22
1.2. Problemática 24
1.3. Objetivo 26

2. Revisión crítica del estado del arte 28

2.1. Antecedentes 28
2.1.1. Arquitectura JamStack 32
2.1.2. Aplicación renderizada en cliente 32
2.1.3. Aplicación SPA (Single Page Application) 33
2.1.4. Aplicación Isomórfica 34
2.2. MicroFrontends 35
2.2.1. Composición 37
2.2.2. Comunicación 40
2.2.3. Integración y desarrollo 47
2.3. Por qué microfrontends 50
2.3.1. Aumento de la complejidad 52
2.3.2. Cambio hacia nuevas tecnologías es lento y complejo 52
2.3.3. Problemas organizativos 53
2.3.4. Inserción laboral compleja 53
2.3.5. Necesidad de desarrollo escalable 54
2.4. Beneficios de Microfrontends 55
2.4.1. La tecnología avanza rápido. 56
2.4.2. Fragmentos verticales mejor enfocados. Cuanto más pequeño, mejor enfocado. 57
2.4.3. Desarrollo independiente. 57
2.4.4. Reduce el riesgo de fallos a secciones aisladas. 57
2.4.5. Un equipo, una misión. 57
2.5. Problemas en Microfrontends 58
2.5.1. Consistencia en la experiencia de usuario 59
2.5.2. Varias tecnologías, Mayor tamaño 59

2.5.3. Código duplicado 59
2.5.4. Aumenta el nivel de complejidad técnica y organizativa 60
2.5.5. Conocimiento aislado 60
2.5.6. Entornos diferentes 60
2.5.7. Incremento del riesgo durante las actualizaciones 61
2.5.8. Retos de accesibilidad 61
2.6. Integración Continua 62
2.6.1. Jenkins 63
2.6.2. Travis CI 64
2.6.3. Bamboo CI 64
2.7. Quién usa microfrontends 65
2.7.1. Spotify 65
2.7.2. Ikea 67
2.7.3. DAZN 68
2.8. Tecnologías y Frameworks 69
2.8.1. Bit 70
2.8.2. Luigi 71
2.8.3. Piral 71
2.8.4. PuzzleJS 72
3. Solución **74**
3.1. Especificación de requisitos 75
3.2. Actores 77
3.3. Casos de uso 78
3.4. Componentización en servicios 80
3.5. Arquitectura 80
3.2.1. Chef Service 81
3.2.2. Waiter Service 90
3.2.3. Stack tecnológico 91
3.6. Resultado 94
3.3.1. Acceso 94
3.3.2. Registro y configuración de nueva microaplicación 95
3.3.3. Solicitud de construcción de una versión 98
3.3.4. Entorno de desarrollo 99
3.7. Beneficios 99

4. Conclusiones **101**

5. Líneas Futuras **104**

Bibliografía **105**

Lista de abreviaciones

API	Application Programming Interface
CD	Continuous Deployment
CDN	Content Delivery Network
CI	Continuous Integration
CSS	Cascading Style Sheets
DDD	Domain Driven Design
DOM	Document Object Model
DX	Developer Experience
ECMA	European Computer Manufacturers Association
ESI	Edge Side Includes
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JAM	Javascript API Markup
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Model View Controller
ORM	Object-Relational Mapping
OTT	Over The Top
POC	Proof Of Concept
QA	Quality Assurance.
SaaS	Software as a Service
SEO	Search Engine Optimization
SPA	Single Page Application

SSI	Server Side Includes
SSR	Server Side Render
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

Lista de figuras

Fig 1: Radar de Tecnología de la Consultora Thoughtworks (Nov 2016). 19
Fig 2: MicroFrontends. 20
Fig 3: Diferentes Tecnologías por fragmento = Una aplicación 21
Fig 4: Microfrontends despliegue en producción independiente. 23
Fig 5: Guía de conocimiento sobre estilos transversal a la empresa. 24
Fig 6: Comunicación entre microfrontends integrados. 25
Fig 7: Arquitectura Flux para aplicaciones frontend. 31
Fig 8: Microservicios y Frontend Monolítica vs Microservicios y Microfrontends 36
Fig 9: Comunicación Cliente-Servidor con Microfrontends (Punto de vista teórico). 47
Fig 10: Integración y desarrollo: Fragmentos Mock. 48
Fig 11: Integración y desarrollo: Fragmentos aislados. 49
Fig 12: Integración y desarrollo: Fragmentos entorno local. 49
Fig 13: Integración y desarrollo: Fragmentos desde Producción o versiones. 50
Fig 14: Múltiples tecnologías conectadas. 56
Fig 15: Distintas tecnologías, una aplicación 56
Fig 16: Un equipo, una misión. 58
Fig 17: Pipeline de Jenkins. 63
Fig 18: Planes de Bamboo. 65
Fig 19: Logo Spotify 66
Fig 20: Dependencias y coordinación de equipos. 66
Fig 21: Lanzamientos desacoplados. 67
Fig 22: Logo Ikea 67
Fig 23: Logo DaZN 69
Fig 24: Logo Bit 70
Fig 25: Logo Luigi 71
Fig 26: Logo Piral 71
Fig 27: Logo Puzzle 72
Fig 28: Conceptualización de la solución. 74
Fig 29: Chef Service - Autenticación y enrutamiento. 82
Fig 30: Chef Service - Gestión de usuarios. 83

Fig 31: Chef Service - Gestión de MicroAplicaciones. 85
Fig 32: Chef Service - Cola de trabajos de sincronización y construcción de MicroAplicaciones. 86
Fig 33: Chef Service - Notificaciones 87
Fig 34: Chef Service - Sincronización de código desde sistemas de control de versiones. 88
Fig 35: Chef Service - Construcción y alojamiento de versiones construidas. 89
Fig 36: Waiter Service - Despacho de Microaplicación por desarrollador. 90
Fig 37: Chef Service y Waiter Service. 91
Fig 38: Chef Service Login. 94
Fig 39: Chef Service - Listado MicroApps. 95
Fig 40: Chef Service - Registrar MicroApp - Denominación. 96
Fig 41: Chef Service - Registrar MicroApp - Sincronización. 96
Fig 42: Chef Service - Registrar MicroApp - Construcción. 96
Fig 43: Chef Service - Registrar MicroApp - Despacho. 97
Fig 44: Chef Service - Registrar MicroApp. 97
Fig 45: Chef Service - Solicitar construcción MicroApp. 98
Fig 46: Chef Service - MicroApp Construidas para usuario actual. 98

1. Introducción

En las últimas décadas, el sector software no ha parado de evolucionar hacia aplicaciones web cada día más avanzadas, interactivas y amigables. La industria ha reconocido el valor que aporta este tipo de aplicativos y se ha volcado en este apartado, convirtiendo al cliente web en un aspecto decisivo para el éxito de sus servicios.

Las aplicaciones frontend se encuentran en un cambio continuo hacia mejores técnicas de integración con los servicios de backend. Del mismo modo, constantemente aparecen nuevas prácticas de implementación, despliegue y mantenimiento para obtener resultados más óptimos en este tipo de software. Debido a esta tendencia de crecimiento en el sector, cada vez más empresas dedican parte de sus recursos a formar equipos especializados en desarrollo del cliente web, produciendo a su vez aplicaciones más grandes y complejas.

Tal es el aumento de complejidad en frontend, que las aplicaciones comienzan a sufrir problemáticas ya conocidas y experimentadas años antes en el lado servidor. Grandes aplicaciones monolíticas, difíciles de mantener y escalar, hacen que se requiera un cambio de estrategia. Como respuesta a estos problemas, en el lado servidor aparece la arquitectura de Microservicios, *"basada en un Diseño dirigido por el dominio (DDD o Domain Driven Design) y que trata de dividir la aplicación monolítica en un grupo de servicios más pequeños, independientes entre sí, pero con sus propios mecanismos de comunicación"* [6]. Este enfoque, permite un desarrollo independiente y descentralizado, a la vez que mejora la escalabilidad, tanto del software, como de la organización. De hecho, debido a esta independencia en el desarrollo, se consigue que la incorporación de nuevos equipos a la empresa se realice de forma mucho más orgánica y natural.

Tomando la misma idea que se aplicó en el lado servidor con microservicios años antes, en *"Noviembre de 2016 aparece por primera vez el término **Microfrontends** en el radar de tecnología de la consultora Thoughtworks"* (Figura 1) [6]. Certificando el nacimiento de este nuevo enfoque

software pensado para facilitar el desarrollo de aplicaciones complejas en frontend mediante un paradigma escalable horizontalmente a partir del incremento de equipos. Esta jovén arquitectura, a pesar de establecer grandes ventajas, no está exenta de problemas que le impiden conquistar de una manera más contundente a la industria software. Parte de esta problemática es la que trataremos de solventar a lo largo del presente documento.

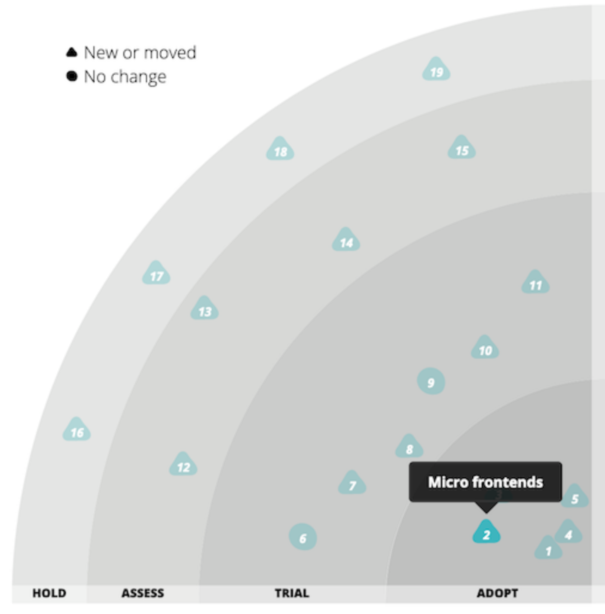

Figura 1: Radar de Tecnología de la Consultora Thoughtworks (Nov 2016).
Fuente: Fowler, Martin; Jason, Cam. (2019). **“Micro Frontends”** [6].
(https://martinfowler.com/articles/micro-frontends.html/).

Microfrontend es una arquitectura software enfocada al lado cliente que se basa en dividir una aplicación en fragmentos siguiendo un diseño dirigido por dominio. Estos fragmentos deben estar aislados unos de otros, permitiendo eso sí, distintos sistemas de comunicación entre ellos. Tal como pasó con los microservicios en servidor, este enfoque ofrece la posibilidad de conseguir un **trabajo independiente**, con **mayor tolerancia a fallos**, mucho **más ágil y descentralizado**, tanto tecnológicamente, como a nivel organizativo, permitiendo **escalabilidad horizontal** mediante la incorporación de nuevos equipos.

Este nuevo paradigma arquitectónico, partiendo de la misma idea base que ya se aplicaba en el lado servidor con microservicios, plantea dividir la aplicación, tradicionalmente monolítica de

cliente, en fragmentos verticales (o microfrontends) que permitan un desarrollo independiente en todos los aspectos [2]. Luca Mezzalira, Vicepresidente de arquitectura de DAzn, Google Developer Expert y uno de los principales expertos en esta arquitectura plantea que *"cada microfrontend debe ser una representación de un subdominio del negocio"* [3].

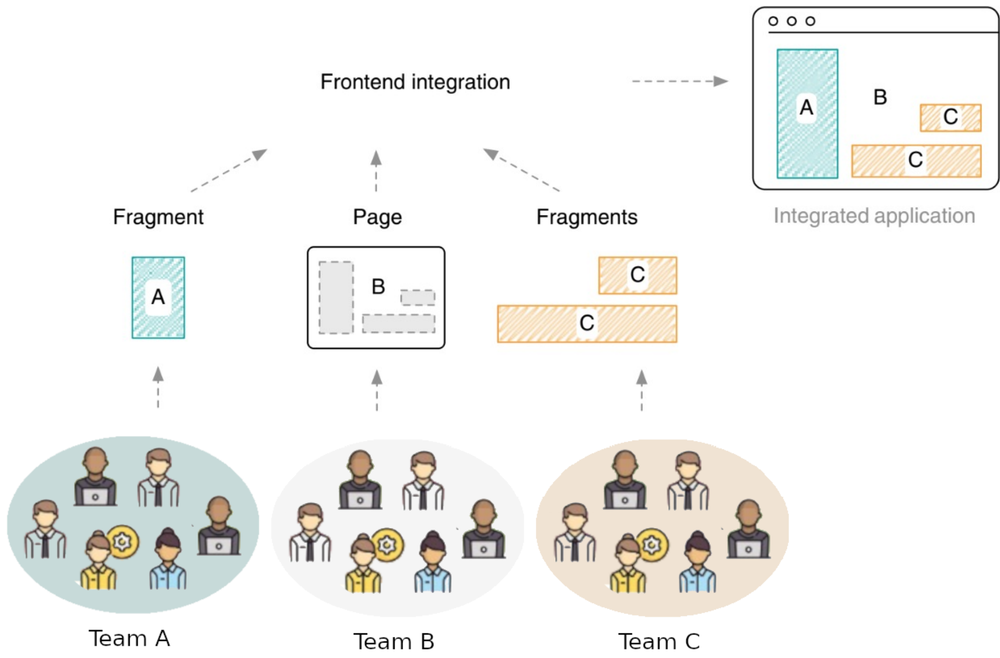

Figura 2: Microfrontends.

Fuente: Adaptación propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

El hecho de implantar microfrontends como arquitectura de nuestra aplicación de cliente supone grandes beneficios para el desarrollo y la organización de la empresa. Por ejemplo, debido a que cada fragmento se encuentra aislado del resto, este paradigma permite el uso de distintas tecnologías en una misma aplicación o la actualización de versiones de forma totalmente independiente entre equipos. Además, al tratarse de fragmentos de aplicación, estos son más pequeños y fáciles de manejar, así el equipo consigue un mejor enfoque de su objetivo. Por otra parte, cada sección está aislada entre sí, por lo tanto, los fallos son contenidos y no afectan a toda la aplicación. (En la sección 2, *Revisión crítica del estado del arte* se describen en profundidad el resto de beneficios, así como las problemas de la implantación de microfrontends que han sido recopilados de acuerdo a los trabajos realizados por distintos autores citados en la bibliografía).

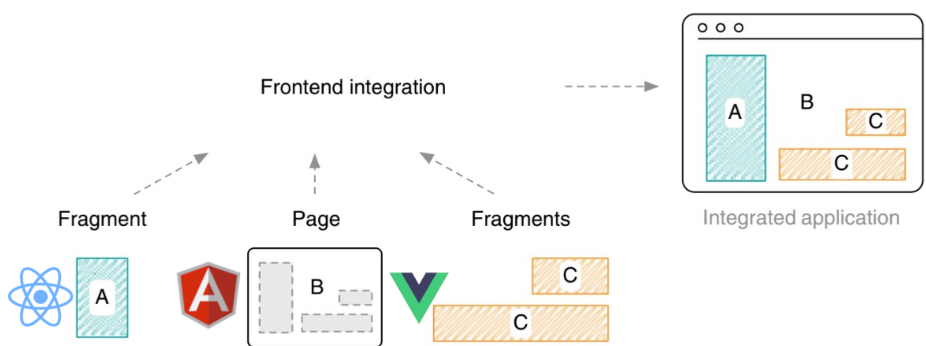

Figura 3: Diferentes Tecnologías por fragmento = Una aplicación.

Fuente: Adaptación propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

En la actualidad, debido a la necesidad de reducir la complejidad de sus aplicaciones y conseguir una mejor escalabilidad, la arquitectura de Microfrontends ha sido adoptada por grandes compañías como DaZN, Ikea, SAP, Spotify o Zalando, entre otras. A pesar de esto, este enfoque todavía no ha convencido de forma general a otras grandes empresas del sector, esto es debido al **incremento de complejidad organizativa** que supone esta arquitectura en otros aspectos.

La decisión de implantar una arquitectura de Microfrontends en una empresa no ha de ser tomada de forma trivial, deben valorarse los pros y contras que tiene este enfoque tecnológico. Resulta vital un entendimiento en profundidad del dominio sobre el que se trabaja y si existen motivos suficientes para afrontar el reto que supone. Además, requiere de una gran experiencia en frontend y de un buen plan de implantación.

Después de un análisis minucioso de la bibliografía seleccionada sobre esta **temática emergente y parcialmente desconocida**, este estudio pretende, en primera instancia, contribuir al sector del software como servicio estableciendo una **base de conocimiento sólida sobre la arquitectura de Microfrontends**, para ello, recopila, resume y ordena información de distintos autores sobre la situación actual de esta arquitectura. Este objetivo busca una documentación apta para guiar a empresas del sector del software como servicio en este tipo de arquitectura, los entresijos y dificultades de su implantación, así como los distintos métodos de integración, comunicación e interacción de los que dispone.

En segundo lugar, aunque en ningún caso menos significativo, se plantea como objetivo aportar una **mejora al aspecto organizativo del desarrollo distribuido de microfrontends** mediante un **proceso que permita disminuir la dificultad técnica de integración de los fragmentos de distintos equipos durante la fase de desarrollo**. De este modo, se espera **mejorar la experiencia del desarrollador (DX) en Microfrontends**, viendo reducida sustancialmente la complejidad de trabajar con múltiples entornos de desarrollo y/o evitar bloqueos o dependencias con otros equipos en la medida de lo posible. Es decir, conseguir que cada desarrollador pueda trabajar en paralelo de la forma más ágil y menos intrusiva posible con el resto de equipos y sus proyectos.

En los siguientes puntos, el lector podrá obtener una mejor comprensión sobre el contexto actual, la problemática y el objetivo que busca la propuesta de mejora de este documento.

1.1. Motivaciones

Este tipo de arquitecturas con equipos y desarrollo distribuido suponen un gran adelanto desde el punto de vista organizativo, así como dan lugar a un desarrollo mucho más ágil, rápido y seguro, pero como se puede intuir, estas mejoras introducen nuevos problemas de orquestación a los que hay que dar respuesta. Por lo tanto, la implantación de este tipo de arquitecturas no ha de tomarse a la ligera, ya que no es simple, ni útil para todas las situaciones.

Teniendo presente el planteamiento anterior, si buscamos un firme **candidato a implantar microfrontends,** se puede decir que un sistema con una **aplicación frontend compleja** o cuya complejidad va en aumento, dentro de una **organización escalable mediante equipos**, sin duda es un gran aspirante. Sin embargo, a pesar de que este enfoque supone una gran mejora a nivel organizativo para este tipo de empresa y reduce la complejidad de la aplicación en fragmentos más manejables, estas mejoras, a alto nivel suponen una mayor complejidad técnica y de orquestación entre los equipos y sus desarrollos que ha de ser valorada y sopesada por la compañía.

Como se ha comentado, algunos de los motivos principales para implantar microfrontends son el incremento de la complejidad de la aplicación. Asimismo, al igual que reduce la complejidad de la aplicación en fragmentos más manejables, se debe tener en cuenta que, de acuerdo a la bibliografía consultada, uno de los beneficios más destacados de su implantación es la mejora organizativa en

cuanto a escalabilidad y gestión, derivada de la división del trabajo en equipos autónomos, así como un desarrollo, despliegue y gestión independiente.

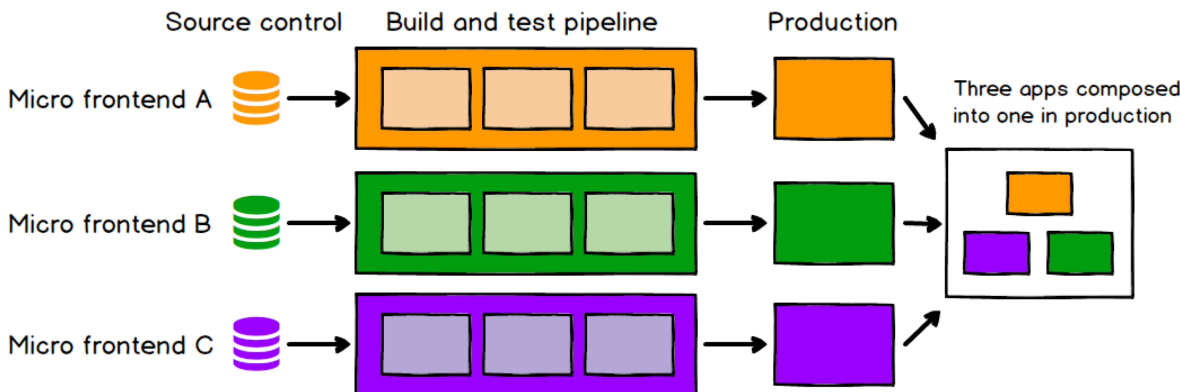

Figura 4: Microfrontends despliegue en producción independiente.
Fuente: Fowler, Martin; Jason, Cam. (2019). **"Micro Frontends"** [7].
(https://martinfowler.com/articles/micro-frontends.html/).

Por contra, este sistema no está exento de problemas. El incremento de la complejidad técnica y de orquestación es debido a que se vuelve complicada la integración de múltiples proyectos en una única aplicación. Cada movimiento en la gestión de los equipos debe ser tomado con sumo cuidado. Esto se debe a que al contrario que en microservicios, en dónde podemos manejar cada servicio de forma independiente y que siga siendo compatible con el resto de microservicios, los microfrontends, son fragmentos separados que deben encajar como una única aplicación que funciona al unísono antes de salir a producción.

Un escenario de múltiples equipos trabajando sobre diferentes microfrontend, con entornos de desarrollo distintos y que deben ser integrados como una única aplicación con un comportamiento homogéneo y consistente, plantea muchas cuestiones. Técnicamente, se vuelve una tarea compleja debido a la dificultad para probar o construir una aplicación completa y actualizada durante la fase de desarrollo.

La motivación de este proyecto ha sido **reducir la complejidad técnica** a la hora de manejar **múltiples microfrontends** de **distintos equipos** con **diferentes entornos de desarrollo en el día a día de un desarrollador**.

1.2. Problemática

En este punto, profundizaremos sobre la problemática concreta que plantea resolver este estudio.

En un entorno ideal, cada microfrontend tiene un comportamiento totalmente autocontenido y autosuficiente. Es decir, para ejecutar toda su funcionalidad no necesita ninguna interacción con ningún otro fragmento. Este planteamiento facilita la orquestación del desarrollo de los componentes en su conjunto y técnicamente no supone apenas problemas. Las preocupaciones técnicas se limitarían a cómo paquetizar los fragmentos para reducir peso de descarga o encontrar la mejor forma de compartir librerías entre proyectos, si fuera necesario.

Sin embargo, en un entorno real, el simple hecho de que varios fragmentos pertenecen a una misma aplicación, implica el desarrollo de una interfaz de usuario homogénea y un comportamiento consistente. De acuerdo, con las buenas prácticas establecidas por Geers, M. en su libro "Microfrontends in action", podríamos plantear una solución sencilla estableciendo una sólida guía de estilos que sea transversal a todos los equipos de la empresa.

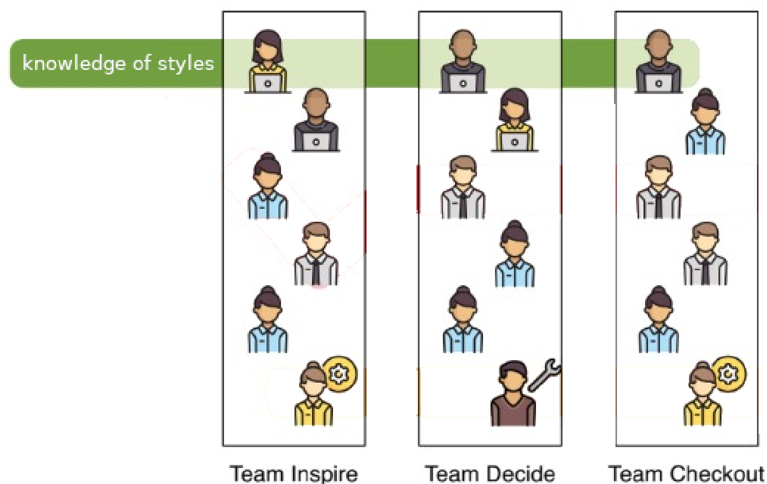

Figura 5: Guía de conocimiento sobre estilos transversal a la empresa.
Fuente: Adaptación Propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

No obstante, este escenario continúa siendo idílico y poco realista. En una aplicación real, normalmente será deseable la interacción entre fragmentos. Esto no tiene por qué implicar

acoplamiento entre los microfrontends, pero sí invalida el planteamiento de un desarrollo completamente independiente por parte de cada equipo. Existen distintos tipos de *Comunicación en cliente* entre microfrontends, tal y como podemos observar en mayor profundidad en el apartado de *Comunicación* de la *Revisión crítica del estado del arte,* cualquiera de ellos hace a los desarrolladores tener que cargar todos los proyectos en su máquina para poder realizar su trabajo de manera consistente. Lo que supone un aumento de la complejidad técnica a la hora de integrar todos los proyectos en un entorno de desarrollo local.

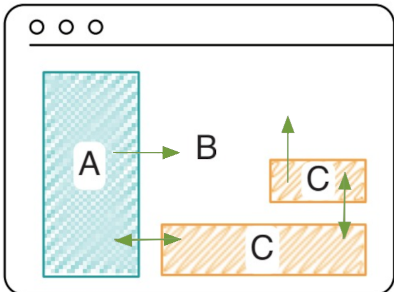

Figura 6: Comunicación entre microfrontends integrados.
Fuente: Adaptación Propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

Uno de los **problemas** que plantea la arquitectura de microfrontends es el mantener ***diferentes entornos de desarrollo*** **en una máquina durante la implementación y depuración.** Normalmente, aunque la dependencia sea leve, se deben probar los distintos fragmentos de los equipos a la vez para saber si la aplicación funciona de forma adecuada. Por ejemplo, si un equipo emite un evento de comunicación con una estructura de mensaje determinada y decide cambiarla o la cambia por error, el equipo que recibe este mensaje podría no ser consciente del fallo hasta las etapas finales de desarrollo o incluso su salida a producción.

A su vez, esto nos lleva a otro de los problemas contemplados en este enfoque, **el incremento del riesgo durante las actualizaciones**. Esto se debe también a la dificultad para construir un entorno de desarrollo lo más cercano posible al entorno real de producción en cliente.

Existen distintas técnicas para disminuir esta complejidad durante el proceso de desarrollo de los equipos y desarrolladores en este tipo de arquitectura. Geers, M. hace una recopilación de esas técnicas, las cuales estudiaremos en profundidad en el apartado de la *Revisión crítica del estado del arte*. Sin embargo, no existe ningún planteamiento que dé una libertad adecuada a los equipos y

desarrolladores sobre los distintos proyectos realizados por el resto de equipos a la hora de probar su código. Esto hace que los equipos no puedan ver el resultado de su producto hasta las etapas finales de su desarrollo o que la complejidad del desarrollo y la experiencia del desarrollador sea mala durante su trabajo. Este trabajo en local se convierte en un reto para equipos distribuidos con un objetivo común en su conjunto.

Cada proyecto debe estar bien integrado con el resto y esto requiere un gran esfuerzo de gestión y de desarrollo, ya que cada microfrontend puede tener su propia tecnología y entorno de desarrollo. Por ejemplo, si un equipo A necesita el código de un equipo B para probar y avanzar en paralelo, cada desarrollador deberá replicar en su máquina el entorno de desarrollo utilizado por el equipo B para poder construir una versión necesaria para probar su trabajo. Si este planteamiento lo extrapolamos a 5 o 10 equipos diferentes, se puede volver una tarea hercúlea (Por ejemplo, montando máquinas virtuales, configurando lenguajes de programación de acuerdo al entorno necesario para cada equipo, etc...).

Por lo tanto, la **problemática** que pretende resolver este estudio es la encontrada a la hora integrar desarrollo de microfrontends con ***"entornos de desarrollo diferentes"*** **y** ***"reducir el riesgo durante la actualización de versión".*** Problemática que se vé reflejada a lo largo de toda la bibliografía como dos de los inconvenientes preocupantes a la hora de implementar este tipo de arquitectura.

1.3. Objetivo

El objetivo a nivel general de este documento será conseguir aliviar esta carga de trabajo a los desarrolladores de todos los equipos y liberar al gobierno de la entidad de la toma de decisiones a bajo nivel, permitiendo un desarrollo más ágil directamente entre equipos independientes. Esta meta la podemos resumir en conseguir **mejorar la experiencia del desarrollador** durante su trabajo.

Concretando, para conseguir alcanzar esta meta, se ha propuesto el diseño de un **Software como Servicio de integración continúa bajo demanda con construcciones independientes por desarrollador**. Este servicio estaría disponible para todos los integrantes de los distintos equipos que trabajan sobre la arquitectura de software basada en microfrontends. Este servicio permitiría a un

desarrollador crear cualquier versión de los distintos fragmentos en remoto e integrarlos directamente con su microfrontend, obteniendo una aplicación completa y más próxima al producto/servicio definitivo.

La gran ventaja es que al delegar la construcción de microfrontends de otros equipos en el servicio de integración continua bajo demanda por usuario, los desarrolladores pueden abstraerse completamente de los entornos de desarrollo definidos por otros equipos para su trabajo. Además, uno de los objetivos es que cada desarrollador sea independiente al resto, por lo que la solución propuesta permitiría a cada desarrollador solicitar las versiones que necesite de cada microaplicación. Esto se traduce en un desarrollo más ágil e independiente.

Una vez presentada la temática, motivación, problemática y objetivos de este proyecto, a continuación se expone cómo se estructura el resto del documento. En el apartado 2, se realiza un *revisión crítica sobre el estado del arte* de la arquitectura objeto de este estudio. En el apartado 3, se expone la solución alcanzada para dar respuesta a la problemática sobre la que se centra este estudio. En el apartado 4, se presentan las conclusiones que han sido alcanzadas una vez ejecutada la solución propuesta. Para finalizar, en el apartado 5, se plantean algunas líneas futuras de desarrollo para profundizar sobre esta misma temática.

2. Revisión crítica del estado del arte

A lo largo de este apartado se presentará el estado actual de la arquitectura de Microfrontends, las ventajas de su implantación y los problemas o inconvenientes que conlleva. En el epígrafe 2.1, se comenzará presentando los antecedentes tecnológicos a este enfoque. Posteriormente, en el punto 2.2 se continuará estudiando cuáles fueron las circunstancias detonantes que dieron lugar a esta arquitectura y en qué consiste este sistema. Posteriormente, se profundizará en los diferentes métodos existentes para su composición y comunicación, aspectos fundamentales de este enfoque. En el punto 2.3, se expondrá por qué resultaría interesante para una empresa la implantación de microfrontends. Se continuará en el epígrafe 2.4, con una explicación sobre las ventajas que ofrece Microfrontends y los problemas actuales que resuelve. En el epígrafe 2.5, se describen los problemas e inconvenientes que se puede encontrar una empresa al implantar Microfrontends. En el epígrafe 2.6, se presentan distintas tecnologías actuales para la integración continua y por qué no solventan la problemática planteada en este trabajo. En el epígrafe 2.7, se enumeran algunas de las grandes empresas que han decidido implantar este enfoque arquitectónico y cuales han sido sus motivos. Por último, en el epígrafe 2.8, se presentan algunas de las tecnologías más relevantes que facilitan la implantación de esta arquitectura en el lado cliente.

2.1. Antecedentes

Antes de profundizar en esta temática, vamos a hacer una breve introducción histórica sobre el concepto de frontend y su evolución. Hace ya más de 30 años que la Web hizo su primera aparición en escena. Tim Berners-Lee, conocido por ser su creador, escribió su propuesta sobre la World Wide Web en 1989 [11], aunque no fue hasta finales de 1990 cuando se formalizó el proyecto llevando a cabo la primera comunicación entre un servidor y un cliente utilizando un protocolo de comunicación proyectado exclusivamente para este fin. Tim Berners-Lee proponía entonces la “web”, *“un entramado de documentos de hipertexto que eran enviados desde un servidor hasta un cliente, lugar donde eran interpretados por un programa llamado “navegador””* [12]. Para ello, tanto él, como su equipo desarrollaron tres tecnologías fundamentales sobre las que se sostiene la World Wide Web.

Estas tecnologías, globalmente conocidas en la actualidad, fueron *"el lenguaje HTML[1] (HyperText Markup Language), el protocolo de comunicación HTTP[2] (HyperText Transfer Protocol) y el sistema de localización de recursos URL[3] (Uniform Resource Locator)"* [18].

Se podría pensar que retroceder tanto en la historia para dar una explicación a qué es un frontend puede estar fuera de lugar, sin embargo, estrictamente hablando, el resultado de la interpretación de los documentos de hipertexto en un navegador hace 30 años es un frontend. Por otra parte, si nos centramos en su acepción más básica, se puede definir el frontend como la parte de un software que el usuario puede ver y contra la que puede interactuar. Desde un punto de vista más técnico, un frontend web se concreta como el conjunto de ficheros HTML, Javascript, hojas de estilos CSS y archivos multimedia enviados al usuario, que son renderizados por un navegador y transformados en una estructura de elementos distribuidos en pantalla con un diseño y comportamiento concreto.

Para continuar, si nos centramos en la evolución de los frontends a lo largo de la historia, nos encontramos con un punto de inflexión claro, este sucede en el año 2015 con la liberación del estándar ECMAScript v6 (ES6 o ES2015) de Javascript [12]. Esta nueva versión supuso el mayor salto evolutivo desde la aparición de este lenguaje en 1995. Otro factor importante para el campo de los frontends actuales tuvo lugar en 1997, cuando la European Computer Manufacturers Association, ECMA[4]. estandariza el DOM[5] (Document Object Model) para así evitar incompatibilidades entre navegadores. Hoy en día, el DOM resulta uno de los ejes fundamentales de los frontends modernos. Este mismo año, la ECMA también se hace cargo de estandarizar javascript y lanza su primera versión estándar, denominada ECMAScript v1.

---

[1] HTML: HyperText Markup Language, Lenguaje de marcado utilizado para definir la estructura de páginas web (https://www.w3.org/TR/html52/).
[2] HTTP: HyperText Transfer Protocol, protocolo para la transmisión de documentos de hipermedia. Fue concebido para la comunicación entre cliente y servidor.
[3] URL: Uniform Resource Location, Dirección basada en una cadena de caracteres que apunta a un recurso concreto en la World Wide Web (https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL).
[4] ECMA: European Computer Manufacturers Association, Asociación industrial fundada en 1961 y dedicada a la estandarización de sistemas de información y comunicaciones. Formada por miembros implicados estrechamente en la industria de este tipo de sistemas, preocupados por conseguir estabilidad y seguridad en el sector a largo plazo. (https://www.ecma-international.org/).
[5] DOM: Document Object Model, API definida para interactuar con documentos HTML o XML. Un modelo del documento se carga en el navegador como un árbol de nodos. Este API permite crear, mover o modificar dichos nodos de forma sencilla (https://developer.mozilla.org/en-US/docs/Glossary/DOM).

Como curiosidad, en 1995, cuando se lanzó javascript, se le denominaba Mocha y pertenecía a Netscape. Su cambio de nombre a Javascript fue debido a que Sun Microsystems, empresa propietaria del lenguaje Java, adquirió Netscape y quiso relanzar el lenguaje con un sello propio. Se debe aclarar para los lectores poco familiarizados con estos lenguajes de programación, que Java y Javascript no tienen absolutamente nada en común desde un punto de vista técnico [12].

En la actualidad, las aplicaciones frontend han pasado a convertirse en un software con una lógica y arquitectura que puede ser tanto o más compleja que cualquier otro tipo de aplicación. Esto se debe a que desde la aparición de la versión ES6, con todas sus novedades, javascript se convirtió en un lenguaje con capacidad de generar aplicaciones similares a las que ofrece un entorno de escritorio, sin necesidad de instalaciones o dependencias. Incluso antes de la aparición de ES6, se mostró un gran interés por el potencial que podría ofrecer el lado cliente y en 2012 Microsoft lanza Typescript [23], un nuevo lenguaje de programación libre, desarrollado y mantenido por la compañía. Aunque realmente, más que un lenguaje en sí mismo, se trata de un super-conjunto de javascript que lo extendía. La característica principal que ofrece a día de hoy, es que introduce tipado.

Todas estas nuevas características de javascript que fueron introducidas con la llegada de ES6 se acogieron con gran aceptación dentro del ámbito del Software como servicio (Software as a Service, SaaS[6]), ya que aumentaba mucho sus posibilidades y alcance. Cualquier usuario con acceso a internet y un navegador podría utilizar una aplicación con la misma experiencia y sencillez que la que ofrecía una aplicación de escritorio, pero con múltiples ventajas, no necesita instalación, no está limitado a un ordenador concreto y todos sus datos están disponibles y salvaguardados en todo momento y puede acceder a ellos desde cualquier lugar.

Dadas todas las ventajas que empiezan a surgir con este nuevo enfoque, muchas grandes empresas comienzan a implantarlo en sus servicios y a desarrollar frameworks y librerías que facilitan y fomentan el desarrollo de aplicaciones frontend. Debido a este auge, la complejidad de las aplicaciones también aumenta y hasta comienzan a aparecer nuevas arquitecturas que se adaptan

[6] SaaS: Software as a Service, Modelo de distribución de software basado en servicios alojados en servidores a los que se accede por internet mediante una aplicación cliente. Tanto la lógica de la aplicación, como los datos del usuario, se gestionan en el servidor remoto.

mejor a este tipo de aplicaciones, aplicaciones que empiezan a convertirse grandes monolitos difíciles de manejar con arquitecturas tipo MVC[7] (Model-View-Controller) [2,9,14]. Cabe destacar el planteamiento de Facebook de la arquitectura Flux[8], pensada para solventar los problemas que comenzaban a tener debido a la complejidad de sus aplicaciones. *"Se trata de una arquitectura basada en el flujo unidireccional de datos y conectada estrechamente con la reacción a eventos"* [24]. A continuación, se muestra un diagrama sobre la estructura propuesta.

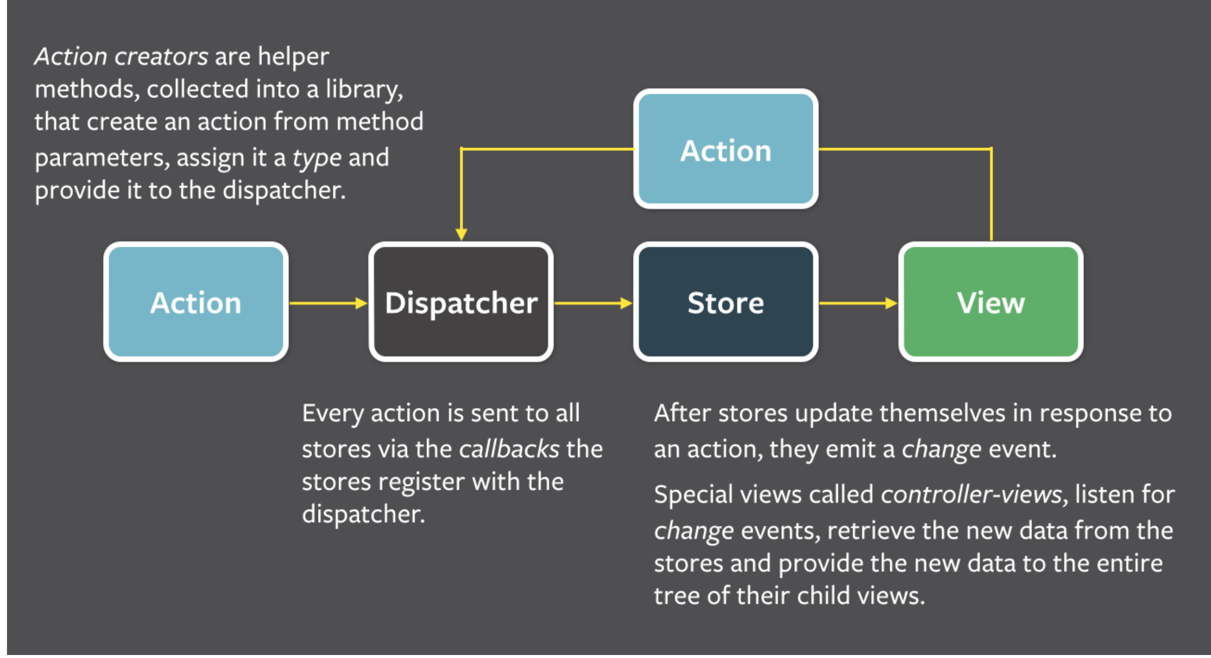

Figura 7: Arquitectura Flux para aplicaciones frontend.
Fuente: Página oficial de Flux (https://facebook.github.io/flux/).

Este tipo de arquitecturas flux suelen estar orientadas a aplicaciones monolíticas enviadas desde el servidor y que son interpretadas por el cliente, pero existen otros muchos distintos enfoques de aplicaciones frontend que son utilizadas de acuerdo a las necesidades de cada proyecto.

[7] MVC: Model-View-Controller, Arquitectura Software basada en los componentes Modelo, Vista y Controlador encargados los primeros de la representación de la información y formateo de la misma, y el último, de la interacción con el usuario.
[8] Flux: Arquitectura software desarrollada inicialmente para el lado cliente y que se encuentra basada en un flujo de datos unidireccional (https://facebook.github.io/flux/).

Con el objetivo de ofrecer al lector una explicación más completa y profunda sobre esta temática y su presente, en los siguientes puntos de este apartado, se presentan enfoques alternativos [1,2,12,13,14] desde un punto de vista arquitectónico para el desarrollo de este tipo de aplicativos.

2.1.1. Arquitectura JamStack

JAMstack es una arquitectura para la construcción de webs modernas que se ha popularizado durante los últimos años debido a que busca la sencillez, la rapidez y la seguridad en el desarrollo web. Se encuentra basada en Javascript, API[9] y Markup [13]. La idea que promueve este enfoque es servir sitios estáticos prerenderizados y con funcionalidad en cliente, gracias a aplicaciones javascript e interacción con un API y funciones Severless[10], si fueran necesarias. Este prerenderizado en ficheros estáticos, permite que las páginas puedan ser almacenadas en servidores de fichero tipo CDN[11], mucho más económicos y rápidos.

Este tipo de arquitectura centra su valor en el desarrollo de frontend y aunque existe una clara división entre backend y frontend, se encuentra basada en aplicaciones monolíticas en javascript.

2.1.2. Aplicación renderizada en cliente

Estas aplicaciones se centran en renderizar en cliente el resultado final que se presenta al usuario. El navegador descarga un archivo HTML junto con archivos javascript que se encargan de manipular el DOM para ofrecer páginas web dinámicas e interactivas, con animaciones y cambios en la interfaz sin necesidad de refresco. Suele estar enfocados a pequeños cambios frente a interacciones del usuario con la página para ofrecer una mejor experiencia y normalmente se encuentra implementado mediante scripts encargados de solventar funcionalidades muy concretas.

[9] API: Application Programming Interface o Interfaz de programación de aplicación, se trata de un conjunto de funciones o métodos utilizado como capa de abstracción para que dos sistemas puedan interactuar entre ellos sin necesidad de conocer su complejidad interna.
[10] Serverless: Contenedor sin estado en el que se ejecuta una aplicación en el lado servidor que se activa debido a eventos, son efímeros, se levantan y se desactivan con cada petición. Una forma de describir serverless es FaaS, funciones como servicio (https://martinfowler.com/articles/serverless.html).
[11] CDN: Content Delivery Network o Red de distribución de contenido, es una red distribuida de servidores con copias exactas y ubicados en distintos puntos de la red con la intención de mejorar la velocidad de acceso a datos de los usuarios. Este paradigma ofrece la posibilidad de que cada usuario acceda a su nodo más cercano en la red, en lugar de a una única ubicación centralizada.

El aumento de complejidad en este tipo de webs y sus scripts derivan en el denominado vulgarmente “spaguetti code”, un código muy difícil de entender y mantener debido a una falta total de orden y coherencia durante su desarrollo. Para evitar esto y conseguir aplicaciones web consistentes, nace el concepto de *Single Page Applications* (SPA), aplicaciones monolíticas encargadas de manejar el DOM siendo capaces de ofrecer una mejor experiencia de usuario y un mejor rendimiento, además de convertirse en un software más maduro y mantenible a largo plazo, en el que ya no sólo prima la parte visual, sino que destacan por la funcionalidad aportada y la descarga de trabajo en servidor.

2.1.3. Aplicación SPA (Single Page Application)

Las SPA son uno de los principales enfoques de aplicaciones frontend en la actualidad. Esto es debido a que a día de hoy, se puede obtener un rendimiento y una experiencia de usuario que nada tiene que envidiar a las aplicaciones de escritorio más sofisticadas. Por este motivo, se trata de uno de los enfoques más explotados por las empresas para ofrecer software como servicio (SaaS).

Este tipo de enfoque se encuentra basado en la carga de una única aplicación monolítica que se encarga de manipular todo el DOM de forma asíncrona y sin necesidad del refresco de las páginas. Consigue muy buenos resultados en la usabilidad y experiencia del usuario al obtener una interacción muy fluida. Actualmente, existen muchos frameworks y/o librerías destinadas a facilitar el desarrollo de este tipo de aplicaciones, las principales y más usadas son Angular[12], React[13] y Vue[14] [25,26,27].

El sistema carga una única paǵina HTML con un punto de referencia en un nodo del DOM, este punto de referencia será utilizado por la aplicación javascript como punto de inicio o raíz para renderizar la app. La aplicación se encarga de encontrar este nodo raíz y construir de forma dinámica la interfaz que se mostrará al usuario. Para mejorar el rendimiento, este tipo de aplicaciones suele manejar un DOM Virtual, que se compara con el DOM real con cada interacción o solicitud que este

[12] Angular: Framework Javascript de código abierto para aplicaciones cliente, desarrollado por Google utilizado para crear SPAs (https://angular.io/).
[13] React: Librería Javascript de código abierto para facilitar el desarrollo de aplicaciones cliente. Se encuentra mantenida por Facebook y una gran comunidad de desarrolladores (https://es.reactjs.org/).
[14] Vue: Framework y Librería Javascript de código abierto para desarrollar o facilitar el desarrollo de aplicaciones cliente (https://es.vuejs.org/).

recibe, pero sólo son re-renderizados los cambios, mejorando de este modo el rendimiento de la aplicación.

Este tipo de aplicaciones, a pesar de ser ampliamente utilizadas por grandes empresas y ser uno de los pilares fundamentales del software como servicio (SaaS), no dejan de tener inconvenientes. Debido a que toda la aplicación se empaqueta en un único fichero, suele ser pesado y reduce la velocidad del primer renderizado al usuario. Además, se trata de una alternativa poco amigable para la optimización de motores de búsqueda. Por otro lado, este enfoque puede crecer rápidamente y convertirse en una aplicación difícil de desarrollar, mantener y escalar, en ese momento, tal vez sea conveniente pensar en los microfrontends como solución.

2.1.4. Aplicación Isomórfica

La idea que persiguen las aplicaciones isomórficas es ser capaces de renderizar una misma web en cliente y en servidor. Esto lo consiguen por medio de aplicaciones javascript enviadas al cliente, pero que a su vez son capaces de renderizar el mismo resultado en forma de archivos HTML en el servidor.

No existe un método único para implementar aplicaciones isomórficas, se trata de una arquitectura de desarrollo web que busca optimizar los tiempos en la primera carga de la web, así como tener una mejor optimización para los motores de búsqueda (SEO[15], Search Engine Optimization). Desde Airbnb en 2011, se plantea este enfoque híbrido buscando estas ventajas que se comentan [14]. Estos dos beneficios que aporta este enfoque, son algunas de las carencias de las aplicaciones SPA.

No todo son ventajas con este enfoque isomórfico. Los tiempos de las interacciones con el usuario son mayores debido a que deben utilizar servidor y cliente para generar cada cambio en las vistas. Además, este tipo de aplicaciones tienen problemas de escalabilidad debido a que se deben mantener vistas pre-renderizadas para todos los usuarios que entren en la web, por lo que si entra un número alto de usuarios, puede resultar fatal para el servidor.

[15] SEO: Search Engine Optimization u Opmitización de motores de búsqueda, conjunto de acciones y buenas prácticas en el desarrollo web, así como en la creación de contenido, para mejorar el posicionamiento de un sitio web en la lista de resultados obtenidos desde buscadores en internet.

2.2. MicroFrontends

Los microfrontends *"surgen debido al aumento inagotable de complejidad en las aplicaciones cliente"* [2]. Lejos quedan aquellos años en los que toda la complejidad y capacidad de procesamiento era asumida por el lado servidor y el lado cliente era una mera representación estática de información ya procesada, con una funcionalidad muy limitada.

Hoy en día, este tipo de aplicaciones frontend tienen un papel prioritario para la mayoría de organizaciones, pero conseguir llevar a cabo una buena aplicación cliente no es un asunto menor. La rapidez con la que avanzan las tecnologías relacionadas con estos sistemas y la multitud de frameworks y/o librerías que existen, hacen que diseñar e implementar un buen frontend, sea un trabajo complejo y delicado. En muchos casos, con el paso del tiempo, se acaban convirtiendo en grandes aplicaciones monolíticas difíciles de mejorar, actualizar y mantener.

Estos sistemas no sólo llegan a ser complejos a nivel tecnológico, sino que a nivel organizativo también representan un reto para las empresas. Según crece la funcionalidad de una aplicación, volviéndola cada vez más compleja, parece adecuado aumentar el número de implicados, pero crear equipos muy grandes para trabajar sobre un mismo proyecto implica problemas de coordinación, acoplamiento y dificulta el mantenimiento a largo plazo.

Toda esta problemática que se presenta no es nueva, en backend ya sucedía hace años (y en muchos casos, continúa existiendo), grandes aplicaciones monolíticas, difíciles de escalar, modificar y mantener eran lo habitual. Es entonces, cuando aparece el concepto de microservicios [6], una arquitectura que resuelve estas dificultades, *"dividiendo en servicios toda la funcionalidad altamente cohesionada de un dominio y la aísla del resto sistema, permitiendo comunicación con el exterior mediante un único punto de entrada" [6]*. Con una funcionalidad acotada a un dominio concreto resulta más fácil de implementar, probar, escalar y desplegar en producción un servicio, así como evitar problemas de coordinación o acoplamiento entre equipos. A su vez, permite independencia a la hora de elegir la tecnología adecuada para cada microservicio. Este tipo de enfoque se adapta

perfectamente con el diseño dirigido por dominio (o DDD[16], Domain Driven Design), sin embargo esta idea estaba limitada por cuestiones arquitectónicas al lado servidor.

Los microfrontends, tomando la misma idea de trabajo que ya se aplicaba en el lado servidor con microservicios, plantean dividir la aplicación , tradicionalmente monolítica de cliente, en fragmentos verticales (o microfrontends) que permitan un desarrollo independiente, tanto desde un punto de vista tecnológico, como organizativo [2]. Luca Mezzalira, Vicepresidente de arquitectura de DAzn, Google Developer Expert y uno de los principales expertos en esta arquitectura plantea que *"cada microfrontend debe ser una representación de un subdominio del negocio"* [3].

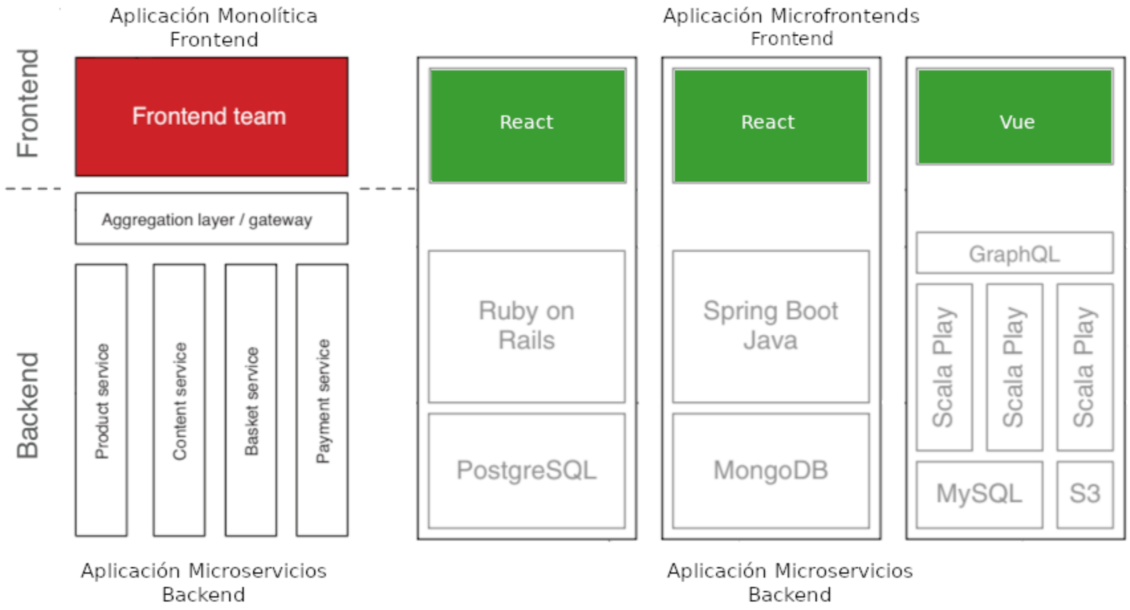

Figura 8: Microservicios y Frontend Monolítica vs Microservicios y Microfrontends.
Fuente: Adaptación Propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

Antes de continuar, debemos aclarar que los microfrontends no son una tecnología en sí mismos, sino que representan una arquitectura software en el lado cliente. Esta arquitectura se encuentra enfocada a mejorar el desarrollo de aplicaciones con una complejidad alta en

[16] DDD: Domain Driven Design o Diseño guiado por dominio, enfoque de desarrollo software centrado en comprender en profundidad los procesos y reglas que establece el dominio de negocio sobre el que se está trabajando. Dirigido especialmente a dominios complejos en los que resulta necesario ordenar la lógica de una aplicación, de acuerdo a sus conceptos de negocio (https://martinfowler.com/bliki/DomainDrivenDesign.html).

organizaciones con un gran número de implicados de distintas áreas divididos en equipos autónomos. Dando así, todo un dominio vertical a un equipo, desde el servidor, hasta el cliente.

Una vez presentado el concepto de microfrontends y sus implicaciones, profundizaremos en tres aspectos claves a la hora de implantar esta arquitectura en cualquier sistema. Se trata de la composición, la comunicación y la integración para su desarrollo [2,4,5,7,8]. En los siguientes puntos explicaremos las distintas técnicas existentes para componer, comunicar y desarrollar los fragmentos que formarán nuestra aplicación.

2.2.1. Composición

La composición de microfrontends es un aspecto crítico para una aplicación basada en esta arquitectura. Esto se debe a que la decisión que tomemos sobre el tipo de composición a implementar definirá el rumbo del desarrollo a lo largo de toda la vida del proyecto.

Uno de los aspectos principales que se deben decidir y que facilitarán la elección del tipo de composición, es si nuestra aplicación necesita un microfrontend por página o si cada página estará compuesta por múltiples microfrontends o si se podrían dar los casos. Por ejemplo, en el caso de una página de login o pasarela de pago, podría interesar un microfrontend por página, mientras que la construcción de un dashboard con múltiples paneles y funcionalidad podría interesar un composición de múltiples frontend por página.

Se puede concluir que existen tres tipos de composición distinta para este tipo de arquitectura, Composición en Cliente (Client-Side), Composición en Servidor (Server-Side), Composición en el Borde (Edge-Side). A lo largo de los siguientes puntos se profundizará sobre ellos.

2.2.1.1. Composición en cliente (Client-side)

La composición de microfrontends en cliente es una técnica a partir de la cual el cliente monta los fragmentos por sí mismo directamente en el navegador en run-time. Existen distintos enfoques a partir de los cuales aplicar este método. Los principales son la composición vía iframes, la composición vía javascript, composición vía routing y la

composición vía Web Components. Estos métodos también son denominados de composición horizontal, salvo el de routing, que se trataría de composición vertical o híbrida.

La composición vía iframes se encarga de controlar el aislamiento y la carga de los distintos fragmentos o microfrontends mediante el uso de iframe añadidos al DOM. Este tipo de composición impide la capacidad de compartir dependencias comunes entre las distintas aplicaciones, como pueden ser hojas de estilo y esto hace que la carga sea más lenta. Además, limita la capacidad de comunicación entre los fragmentos, ya que sólo se puede utilizar el API del iframe concebido para este propósito.

La composición vía javascript carga cada aplicación como un paquete construido de forma independiente y cuya composición es realizada por una aplicación javascript contenedora o se construyen varios nodos de montaje en el DOM para desplegar las distintas aplicaciones cada una en uno de esos puntos. Este tipo de enfoque permite compartir recursos o dependencias, que deben ser transversales a todas las aplicaciones y ofrece mucha más versatilidad en la comunicación entre fragmentos, aunque esto lo estudiaremos en profundidad en la sección 2.2.2.

La composición vía routing [5] se basa en dividir la aplicación en fragmentos que son independientes entre sí por medio de la ruta. Se trata de una técnica bastante simple y eficaz de conseguir una composición de las distintas microaplicaciones completamente aisladas. Se puede conseguir a partir de un enrutador en cliente o directamente desde un servidor web. Este enfoque también se adapta muy bien a modelos híbridos de composición, con rutas con una micro-aplicación única y otras rutas pueden tener a su vez una composición vía javascript, webcomponents u otras. Una de las carencias es que la experiencia de usuario será pobre si se realizan muchos saltos entre rutas.

La composición vía Web Components es posible gracias a la publicación de W3C[17] de un nuevo estándar de HTML. Se basa en la idea de poder construir componentes HTML personalizados que pueden ser utilizados como cualquier otro nodo o tag HTML. Este

[17] W3C: World Wide Web Consortium, comunidad internacional encargada del desarrollo de estándares abiertos y buenas prácticas relacionadas con el crecimiento sólido y estable de la web a largo plazo (https://www.w3.org/).

enfoque permite un aislamiento completo de la implementación de cada fragmento y al igual que la composición vía javascript, permite una mejor comunicación entre componentes. El principal problema de este tipo de composición es que este estándar no se encuentra implementado en todos los navegadores y no está disponible para versiones antiguas de los mismos. Por otro lado, aunque se pueden incluir polyfills[18] para solventar este inconveniente, esto haría que aumentaran los tiempos de descarga.

2.2.1.2. Composición en servidor (Server-side)

La composición de microfrontends en servidor se basa en ensamblar todos los microfrontends en el lado servidor y devolver al usuario una aplicación ya completamente construida a su navegador. Esta construcción puede ser generada en run-time o en tiempo de compilación, dependiendo de la técnica elegida para realizar la composición. Uno de los principales beneficios de este enfoque es la reducción de tiempos en la primera carga de la aplicación, aunque se debe tener en cuenta que si la aplicación es personalizable por usuario, esta forma de composición va a plantear serios problemas de escalabilidad al sistema, que es uno de los beneficios obtenidos mediante *Single Page Apps* [4].

Del mismo modo que con la composición en cliente, la composición en servidor dispone de distintas técnicas, aunque la más utilizada para este propósito es la composición vía Nginx con *Server-Side Includes* (SSI[19]).

La composición vía Nginx con Server-Side Includes se basa en incluir directivas u órdenes SSI en el template que será construido y devuelto por el servidor. Cada directiva debe incluir la url al servidor que construirá ese fragmento de la aplicación. El sistema llamará a cada servicio incluido mediante directivas SSI (Ejemplo: <!--#include virtual="[/url-to-include]" -->) y compondrá la aplicación completa a partir de los fragmentos devueltos.

[18] Polyfills: Fragmento de código que implementa una funcionalidad moderna que no es admitida de forma nativa en navegadores antiguos. De este modo, se puede emular el mismo comportamiento nativo de navegadores modernos (https://developer.mozilla.org/es/docs/Glossary/Polyfill).
[19] SSI: Server-Side Includes, técnica antigua que en la actualidad ha vuelto a tomar relevancia debido a su capacidad de composición de distintos fragmentos código mediante directivas de inclusión. Todo esto realizado en servidor para terminar devolviendo una respuesta estática. Método totalmente robusto y que implica poca sobrecarga en servidor.

2.2.1.3. Composición en el "borde" (Edge-side)

La composición de microfrontends en el borde o Edge-side consiste en unir los fragmentos en un punto intermedio entre el lado servidor y el lado cliente. Es decir, los servidores construyen los microfrontends, pero la labor de ensamblar la aplicación queda delegada a una red de distribución de contenidos (CDN), o a un servidor proxy, como por ejemplo, Varnish.

La capacidad de unir los distintos fragmentos una aplicación en este tipo de infraestructuras se debe a Edge Side Include (ESI[20]), una tecnología basada en XML[21] que ofrecen algunas CDN para componer webs. Este enfoque ofrece una mejor escalabilidad que SSI dado que existen multitud de nodos de CDN repartidos por todo el mundo. Además, resultan un recurso mucho más económico que la mayoría de los servidores tradicionales capaces de manejar lógicas complejas.

Uno de los principales inconvenientes de este tipo de planteamiento es que no todos los CDN implementan del mismo modo el uso de ESI, por lo que esto puede provocar múltiples problemas de integración y despliegue que aumenten la complejidad del uso de microfrontends todavía más.

2.2.2. Comunicación

La comunicación entre microfrontends suele ser necesaria en multitud de ocasiones. Aún así, debe valorarse en profundidad cuando es imprescindible su uso, ya que la comunicación conlleva cierto nivel de dependencia entre fragmentos. Por lo tanto, resulta indispensable no abusar de este

[20] ESI: Edge Side Includes, técnica a partir de la cual mediante un lenguaje de marcado se establece la composición de distintos fragmentos de código en un CDN o en servidores tipo proxy, como Varnish. Respecto a SSI, se cambiaría el lugar de la composición de un servidor Nginx a un servidor Varnish, por ejemplo.
[21] XML eXtensible Markup Language o Lenguaje de marcado extensible, lenguaje de marcado de propósito general. Es necesario definir sus etiquetas, esto permite gran versatilidad. Su uso principal es como lenguaje de estructura de datos utilizado para compartir información entre sistemas diferentes, aunque tiene múltiples usos e implementaciones de lenguajes basados en XML como por ejemplo XHTML, RSS, SVG, etc (https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction).

recurso [3]. Un buen diseño nos proporcionará un uso adecuado para conseguir una aplicación robusta.

En una aplicación con arquitectura de microfrontends nos encontramos dos grandes categorías de comunicación [2]. En la primera parte de este apartado, presentaremos los tipos de comunicación que se pueden dar en cliente entre fragmentos. En la segunda parte, se establece la comunicación necesaria que deben realizar los distintos microfrontends con el servidor y buenas prácticas para desempeñar su objetivo de forma sólida y mantenible.

2.2.2.1. Comunicación en cliente

La comunicación en el lado cliente entre microfrontends permite construir una aplicación interactiva y fluida, lo que da lugar a una muy buena experiencia de usuario. A continuación, se describen los principales métodos de comunicación en cliente que existen actualmente.

2.2.2.1.1. De contenedor a fragmento

Una de las comunicaciones en cliente más básica, es la de contenedor a fragmento o padre a hijo. Se trata de pasar información por medio de atributos desde un componente contenedor a un componente fragmento o hijo.

Este tipo de comunicación viene preestablecida por un manual o contrato de uso del fragmento en cual el equipo propietario establece los atributos, con los tipos y valores soportados por el componente.

2.2.2.1.2. De fragmento a contenedor

La comunicación de fragmento a contenedor o de hijo a padre se realiza por medio de eventos. Para ello, normalmente se hace uso del API de *CustomEvents* nativo del navegador, aunque existen alternativas para la emisión de eventos igualmente válidas. Su uso se basa en que el fragmento llegado un momento o como

consecuencia de una acción, emite un evento que es escuchado por el fragmento padre. El evento en sí mismo puede ser suficiente o se pueden pasar datos extra de acuerdo al método de emisión.

La comunicación mediante eventos tiene las siguientes ventajas:

- El componente emisor no necesita conocer a sus padres.
- Es un tipo de comunicación muy extendida y fácil de comprender.
- La mayor parte de librerías y frameworks soportan el uso de eventos nativos del navegador.
- Se trata de un recurso sencillo de depurar con las herramientas estándar ofrecidas por la mayoría de los navegadores.
- Permite el uso de los eventos nativos utilizados por el propio navegador como un recurso extra.
- Permite generar una nomenclatura propia de acuerdo a convenciones internas de la empresa que faciliten la comprensión de los eventos emitidos por la aplicación. Por ejemplo: “component:item:action”.

2.2.2.1.3. De fragmento a fragmento

En muchas ocasiones resulta interesante poder comunicar fragmentos completamente independientes entre sí. Para ello, técnicamente, existen tres soluciones de comunicación distintas que pueden ser implementadas de forma aislada o combinada.

- **Comunicación directa**: Este tipo de comunicación está basada en la modificación directa del DOM, aunque es técnicamente posible, no es nada aconsejable si se desea implementar MicroFrontends. Este método hace que un microfrontend cambie el contenido de otros fragmentos o microfrontends directamente, lo que puede ocasionar problemas de funcionamiento y resulta muy difícil de mantener, ya que cada fragmento debe ser autocontenido por definición.

- **Orquestación vía contenedor:** Este método se centra en que un componente padre o contenedor coordine la comunicación entre cada uno de sus fragmentos hijo. El mecanismo se basa en que un fragmento hijo emite un evento que escucha el componente padre, y es el componente padre el que organiza y distribuye la información entre los hijos interesados en esta. Para ello, se utilizan las técnicas que hemos visto en los puntos 2.2.1.1 y 2.2.1.2.
- **EventBus:** A través de un bus de eventos se consigue una comunicación global en la aplicación. Cada fragmento puede emitir eventos al bus. Del mismo modo, cualquier fragmento puede suscribirse al canal, escuchar los eventos y reaccionar a estos. Este método reduce al mínimo el acoplamiento entre componentes. Este sistema puede ser implementado mediante el uso del API de CustomEvents, tal y como se ha explicado en el punto 2.2.1.2, o utilizando Broadcast Channel, método que se explicará en el siguiente punto.

2.2.2.1.4. API de Broadcast Channel

El API de broadcast channel ofrece una forma estándar de comunicación. Este sistema aporta una novedad y mejora respecto a los anteriores y es que permite comunicación entre pestañas, ventanas e iframes siempre y cuando se encuentren en el mismo dominio.

El modo de uso en microfrontends se centra en que cada fragmento debe conectarse a un canal central desde el cual puede publicar o suscribirse a notificaciones de todos los microfrontends conectados al canal. El API de Broadcast Channel permite aplicar esta solución de una forma sencilla, solamente son necesarias las siguientes acciones:

- Conectar a un canal: new BroadcastChannel(‘main_channel’)
- Recibir mensajes: channel.onmessage = (message) => {}
- Enviar mensajes: channel.postMessage(message)

Una de las grandes ventajas que ofrece este modelo de comunicación es que permite comunicación entre distintas pestañas, ventanas o iframes, algo que con

CustomEnvents no es posible. Esto podría ser una ventaja si se decide utilizar una composición basada en iframes o si necesitamos varias pestañas que reaccionen entre sí. Además, la creación de distintos canales, unos para uso interno del microfrontend y otros para comunicarse de forma global con el resto de fragmentos, facilita un desarrollo mucho más organizado y desacoplado en la toma de decisiones.

En el momento en el que ha sido realizado este informe, el API de Broadcast Channel se encuentra soportado por las últimas versiones de los navegadores más representativos del mercado, excepto Safari e Internet Explorer. A pesar de esto, si se desea utilizar este sistema de comunicación, existe polyfill para poder utilizar este sistema en navegadores sin soporte.

2.2.2.1.5. Estado Global

La comunicación mediante un estado global parte de la idea que propone como base la librería Redux[22][28] de mantener un estado único para toda la aplicación. Este planteamiento no se ha diseñado pensando en microfrontends, pero la idea es aplicable a esta arquitectura. Existen dos posibles planteamientos, que cada equipo comparta un estado global entre sus fragmentos o que todos los fragmentos y equipos compartan el mismo estado global. Este último planteamiento resulta más difícil de poner en práctica, esto es debido a que un fragmento puede realizar cambios en el estado, que no son esperados por otros componentes, esto puede dar como resultado problemas de estabilidad o errores en la aplicación.

Un inconveniente de este tipo de comunicación respecto a la arquitectura propuesta por microfrontends, es que todos los fragmentos estarían obligados a utilizar esta tecnología para manejar el estado, lo cual limita una de las ventajas principales de los microfrontends y es la de tener libertad en la elección de tecnologías entre fragmentos.

[22] Redux: Librería javascript que facilita la gestión del estado contenedor global en aplicaciones frontend. Está estrechamente relacionada con la arquitectura Flux y es agnóstica a cualquier otra tecnología de renderizado como React, Vue, etc (https://redux.js.org/).

2.2.2.2. Comunicación con servidor

En el apartado anterior hemos podido observar distintos métodos de comunicación entre microfrontends directamente en el lado cliente. Estos podrían ser usados de forma conjunta o aislada de acuerdo a las necesidades de la aplicación. Por contra, en este punto se examinarán las diferentes comunicaciones que los microfrontends deberán establecer con el lado servidor. En este caso, la mayor parte de las aplicaciones cliente serias deberán implementar los tres tipos de comunicación que se describen en los siguientes puntos.

2.2.2.2.1. Contexto global

Toda aplicación con un cierto nivel de complejidad necesita información general sobre el contexto en el cual se despliega. Normalmente, esta información suele ser al menos el idioma del usuario, su franja horaria, el tipo de moneda por defecto, etc. Como se puede deducir, el contexto global es transversal a los distintos microfrontends que forman la aplicación. Por ejemplo, todos los fragmentos deben ser renderizados con el mismo idioma, independientemente de su funcionalidad o del equipo de desarrollo.

Debido a la forma en la que se realiza la composición de aplicaciones basadas en microfrontends, existen dos opciones para manejar la información de contexto. La primera, si existe un fragmento contenedor o plataforma, el equipo que se encargue de su desarrollo será el candidato perfecto para ser responsable de las consultas a servidor y de la gestión de la información de contexto en cliente. La segunda, si no existe un fragmento contenedor, debe ser elegido un equipo para que se encargue de este aspecto. En este segundo caso, debe quedar claro en la empresa que equipo se encarga de esta información para evitar duplicados y peticiones innecesarias al servidor.

El almacenamiento de esta información de contexto es otro punto a tener en cuenta. Para ello, existen distintas posibilidades, como utilizar el API de Global Javascript o el almacenamiento nativo que posee la mayoría de navegadores más

utilizados del mercado. Cualquiera de las tecnologías anteriores permite acceso de lectura desde cualquier componente de la aplicación, por lo tanto, resultan un buen sistema de gestión de datos en cliente.

2.2.2.2.2. Autentificación

La autentificación es una comunicación entre el lado cliente y el lado servidor, en la que el servidor transmite al cliente si tiene permiso de acceso a los recursos de la aplicación. Al igual que la información de contexto global, la aceptación o negación de acceso es transversal a todos los fragmentos de la aplicación. Teniendo en cuenta este factor, el equipo encargado de la gestión de autentificación podría ser el equipo responsable del fragmento contenedor, aunque en este caso se estima una mejor opción dar esta responsabilidad a un equipo especializado en el control de acceso, que sea encargado del fragmento de login.

El equipo designado para esta labor, debe tener en cuenta que la gestión de la autentificación resulta más delicada y debe ser tratada de forma segura, para ello existen distintos mecanismos en los que no profundizaremos, dado que se salen del ámbito de este estudio. De estos métodos, el más utilizado actualmente para controlar el acceso desde recursos distribuidos es JWT (*JSON Web Tokens - RFC-7519*) [20], de este modo se consigue controlar que cada petición a servidor sea validada mediante el uso de un token asociado al usuario autenticado.

2.2.2.2.3. Comunicación cliente-servidor

Como resulta evidente para la mayor parte de los casos de uso, toda aplicación cliente debe establecer comunicación con un servidor. Este servidor es el encargado de gestionar las peticiones del cliente y realizar acciones de persistencia de datos, así como procesar y devolver los recursos solicitados en dichas peticiones.

El enfoque que aporta el uso de microfrontends no cambia en exceso este tipo de comunicación, pero sí establece ciertas directrices para conseguir mantener el menor acoplamiento posible y la mayor cohesión dentro del conjunto de fragmentos

que forman la aplicación. Para ello, si planteamos el uso de microservicios en servidor y una organización de equipos de acuerdo a un Diseño Dirigido por Dominio (Domain Driven Design, DDD) con división vertical, cada equipo debe ser encargado de su/s microservicio/s en el lado servidor y su/s microfrontend/s en el lado cliente. Esto quiere decir que el fragmento de cliente desarrollado por el Equipo B debe comunicarse únicamente con los servicios implementados por el Equipo B y evitar tener comunicación directa con servicios del Equipo A. Este planteamiento es muy difícil de poner en práctica de forma estricta, pero debe ser el objetivo de cualquier proyecto con esta arquitectura.

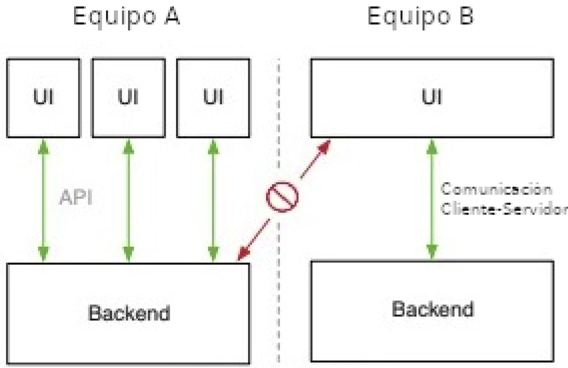

Figura 9: Comunicación Cliente-Servidor con Microfrontends (Punto de vista teórico).
Fuente: Geers, M. (2020). "Microfrontends in Action".

2.2.3. Integración y desarrollo

Existen distintas técnicas para desarrollar microfrontends dentro de un entorno local e intentar disminuir la complejidad técnica que soportan los equipos y desarrolladores en este tipo de arquitectura. Esto es debido a que aunque el desarrollo es principalmente independiente, deben ser tenidos en cuenta el resto de fragmentos para observar si la aplicación en su conjunto cumple con sus objetivos y comportamiento esperado. Geers, M [2]. hace una recopilación de técnicas para conseguir un entorno desarrollo en local para microfrontends. De acuerdo a la complejidad e interacción entre fragmentos, puede ser interesante una propuesta u otra, ya que la complejidad de su implantación también difiere entre sí.

2.2.3.1. Fragmentos Mock o simulados

Esta técnica introduce fragmentos simulados para reemplazar los fragmentos de otros equipos. De este modo, en la aplicación podremos ver nuestros fragmentos y desarrollar sobre ellos, sin embargo, los espacios de los fragmentos de otros equipos aparecerán en blanco.

Esta solución no es válida para todos los casos, plantea problemas cuando existe interacción entre microfrontends de distintos equipos, ya que se vuelve demasiado complicado ejecutar pruebas de funcionamiento/comportamiento.

Figura 10: Integración y desarrollo: Fragmentos Mock.
Fuente: Elaboración Propia - Basada en Gráficos (Geers, M. (2020). “Microfrontends in Action”).

2.2.3.2. Fragmentos aislados:

Este planteamiento se basa en tener un contenedor o página de pruebas en la que se montan únicamente los fragmentos de nuestro equipo.

Este entorno de desarrollo simula las interacciones con otros microfrontends mediante la creación de acciones propias que emiten los resultados esperados desde los fragmentos de otros equipos. Requiere invertir tiempo en simular todas estas interacciones esperadas desde otros fragmentos. Por otro lado, cada equipo de desarrollo no tiene una percepción clara de cómo funciona el sistema en su conjunto o si su comportamiento es consistente y homogéneo hasta el final de la etapa de desarrollo.

Figura 11: Integración y desarrollo: Fragmentos aislados.
Fuente: Elaboración Propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

2.2.3.3. Fragmentos entorno local:

Este enfoque, sugiere crear todo el entorno de desarrollo de cada fragmento en la máquina local del desarrollador. Resulta la técnica más ardua, difícil de mantener y aumenta los tiempos de desarrollo. Ya que implica descargar todos los códigos y construir todas las microaplicaciones en local.

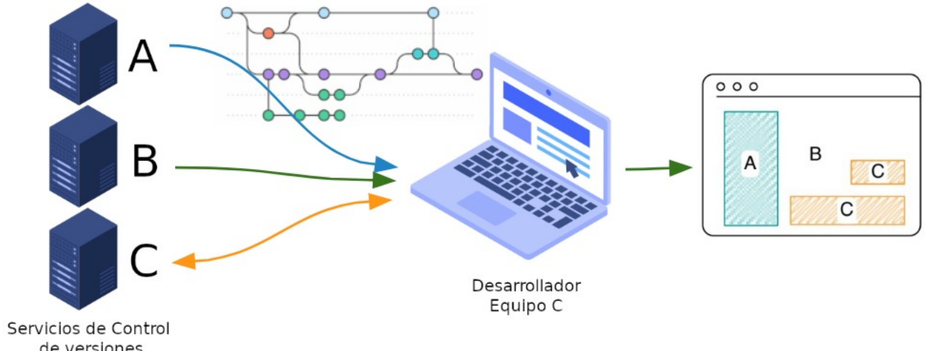

Figura 12: Integración y desarrollo: Fragmentos entorno local.
Fuente: Elaboración Propia - Basada en Gráficos (Geers, M. (2020). "Microfrontends in Action").

2.2.3.4. Fragmentos desde producción o repositorio de versiones:

En muchos casos, con la simulación de fragmentos o con la simulación de sus interacciones no resulta suficiente. Esta técnica plantea montar la aplicación completa, pero evitando descargar el código de otros equipos, ni replicar su entorno de desarrollo y

construcción. Para ello propone descargar las versiones de fragmentos de otros equipos localizadas en producción o desde un repositorio de versiones creado para tal fin.

A pesar de ser una solución más realista, reduce la libertad de los equipos a la hora de avanzar de forma paralela. Ya que dependen siempre de las versiones que se ha decidido construir y publicar su propietario en cada momento, esto puede retrasar los tiempos de desarrollo de otros equipos y los convierte en menos ágiles. Una alternativa podría ser tener un repositorio con versiones construidas para consolidación de código (commit) de cada equipo y cada fragmento, pero esto puede dar lugar a un repositorio de versiones enorme e innecesario en su mayoría, además de una tarea repetitiva y fatigante para el equipo que publica, lo cuál puede derivar en dejadez o aplazamientos. Además, los equipos no serían completamente libres para probar los desarrollos que se están realizando en todo momento y/o sugerir cambios o mejoras que se adapten de forma más adecuada hasta obtener un resultado final. Como comentan distintos autores no existe actualmente ninguna técnica o metodología que dé libertad a los desarrolladores de forma adecuada [1,2,8].

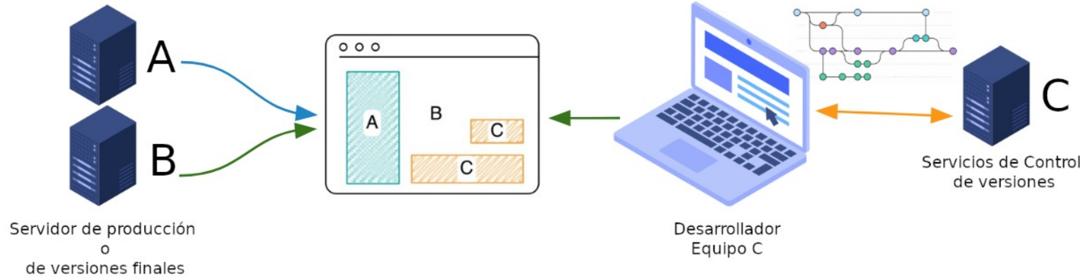

Figura 13: Integración y desarrollo: Fragmentos desde Producción o versiones.
Fuente: Elaboración Propia - Basada en Gráficos (Geers, M. (2020). “Microfrontends in Action”).

2.3. Por qué microfrontends

Tal y como se ha comentado, las aplicaciones de frontend hoy en día están ampliamente extendidas. Cada vez, las empresas dedican más recursos a este apartado, lo que lo acaba convirtiendo en un artefacto complejo. Como ya ha ocurrido en otros ámbitos del software, siempre que se incrementa en exceso la complejidad, se produce la necesidad de plantear mejoras arquitectónicas.

Hasta hace apenas unos años, arquitecturas enfocadas a cliente como Flux o en algunos casos, arquitecturas más comunes en servidor como MVC, habían conseguido sistemas robustos para aplicaciones frontend. Los lenguajes utilizados principalmente para estas aplicaciones son Javascript o Typescript y a día de hoy, sería prácticamente imposible hablar de aplicaciones de cliente sin nombrar tecnologías (librerías y frameworks) que han escrito su nombre con mayúsculas en este ámbito y que ya se han nombrado en este documento, como son Angular, React o Vue [25,26,27]. El desarrollo siguiendo estas ideas y tecnologías, permitía crear aplicaciones monolíticas que cumplían a la perfección con sus funciones. Por lo tanto, si estos sistemas cumplían con su propósito, ¿Por qué son interesantes los microfrontends?.

Los microfrontends aparecen cuando las aplicaciones cliente comienzan a convertirse en grandes artefactos complejos, con los que es arduo trabajar y que son difíciles o imposibles de escalar horizontalmente incrementando el número de equipos. Esta idea no sólo busca una solución tecnológica dividiendo la aplicación en fragmentos más manejables, sino que permite mejorar la estructura organizativa de la empresa, incluyendo la posibilidad de escalar horizontalmente.

Si nos centramos en datos sobre cuáles son los motivos por los que este enfoque debe ser implementado, de acuerdo con el estudio bibliográfico sobre 43 artículos realizado por Peltonen, S. (et al.)[1], la motivación principal que lleva a implantar la arquitectura de microfrontends es principalmente el aumento de complejidad de la capa cliente. Más del 37% de las investigaciones consultadas en ese estudio se refieren a este aspecto como la causa que hace a una entidad afrontar el reto que implica este enfoque, mientras que los problemas de complejidad organizativa se quedan en casi 7% de los estudios (nombramos este aspecto porque más adelante veremos que se convierte en uno de los grandes beneficios obtenidos al aplicar microfrontends).

A continuación, presentamos un pequeño listado descriptivo con los motivos que pueden ser interesantes a la hora de valorar si implantar esta arquitectura en una entidad o valorar otras alternativas [1,2,3,6,7].

2.3.1. Aumento de la complejidad

El aumento de complejidad de las aplicaciones frontend no parece haber llegado a su fin, al contrario, cada día aparecen aplicaciones más y más complejas en nuestro navegador. Aplicaciones que acaban convirtiéndose en gigantes monolitos difíciles de comprender y mantener.

Esta gran complejidad que van adquiriendo este tipo de aplicaciones dada su configuración monolítica hacen que cada vez sea más complicado añadir nueva funcionalidad, depurar la existente o corregir errores en su código. Esta dificultad afecta de forma directa a la velocidad de desarrollo, que se vuelve una tarea pesada y lenta.

Este problema, como hemos comentado a lo largo de este documento, no es nuevo, en el lado servidor pudo ser resuelto de una manera elegante mediante la división de una aplicación en partes menos complejas. Microfrontends plantea reducir la complejidad de este tipo de aplicaciones también dividiendo el sistema en fragmentos altamente cohesionados y con un bajo acoplamiento, esto permitiría una gran mejora en el rendimiento a la hora de desarrollar, mejorar y mantener una aplicación.

2.3.2. Cambio hacia nuevas tecnologías es lento y complejo

Nuevas tecnologías en el ámbito del frontend no paran de aparecer. Del mismo modo, muchas otras quedan obsoletas o en desuso. Esto es debido a que existe un gran aporte de ideas y mejoras continuas para este tipo de desarrollo dado su auge actual.

Una aplicación que termine siendo un gran monolito puede volverse un gigante difícil de adaptar al cambio. La intención de adaptar este tipo de aplicaciones a una nueva tecnología para mejorar el rendimiento, espacio, etc... se vuelve una tarea lenta y compleja, que en muchos casos acaba en una reescritura completa de todo el proyecto utilizando la nueva tecnología elegida, lo cual nos vuelve a llevar al mismo punto de partida.

Estos cambios serían mucho más ágiles con microfrontends, debido a que al tener cada microfrontend una funcionalidad mucho más acotada y reducida, es más fácil de comprender y

modificar. Incluso en el caso de tener que ser reescrito debido a incompatibilidades, sería una reescritura mucho más rápida al tratarse de un escenario mucho más reducido.

2.3.3. Problemas organizativos

Cada vez las empresas de software tienden a formar equipos multidisciplinares sobre un dominio de negocio, en lugar de formar equipos homogéneos desde el punto de vista técnico/tecnológico. Esto permite a todos los miembros del equipo implementar comprendiendo en profundidad las necesidades del negocio y ofreciendo un enfoque global de la situación y su solución.

Además, este paradigma organizativo, permite un desarrollo más ágil, delegando muchas decisiones directamente en los equipos y centrándose sólo en las partes realmente importantes para el conjunto del servicio.

Con grandes aplicaciones monolíticas esto supone grandes problemas, ya que la división en equipos multidisciplinares o la delegación de la toma de decisiones técnicas se vuelve muy complicada. Gracias a la arquitectura de microfrontends, este problema queda resuelto, permitiendo dar la responsabilidad vertical de una funcionalidad o varias funcionalidades bien acotadas a un único equipo, siendo este el encargado de todo lo que acontezca en ese sentido, desde cliente hasta servidor.

2.3.4. Inserción laboral compleja

Cuando la complejidad de una aplicación aumenta, como es el caso que ocurre con los grandes monolitos en cliente, desde un punto de vista organizativo, es tentador contratar a nuevos empleados para acelerar estos procesos de desarrollo y mantenimiento, sin embargo, paradójicamente la propia complejidad de la aplicación hace que no sea fácil esta incorporación de nuevos miembros a un equipo, ya que requerirán de un proceso de adaptación y aprendizaje lento, lo cual acaba retrasando aún más el *roadmap* del servicio o producto.

Del mismo modo, si encierra cierta dificultad incorporar nuevos miembros a un equipo, el planteamiento de extender el desarrollo entre varios equipos requerirá un código excelentemente pensado, modularizado y programado, y aún así, implicaría un punto crítico de acoplamiento el

plasmar el trabajo varios equipos sobre un mismo código, dando lugar a fallos y problemas de integridad.

La división en fragmentos transforma estas tareas organizativas de contratación de personal, debido a que la complejidad de cada microfrontend es reducida y aislada, por lo tanto, fácil de comprender para un externo. Lo cual da lugar a un software mucho más escalable, como veremos a continuación.

2.3.5. Necesidad de desarrollo escalable

El tamaño de un sistema y el número de sus miembros hace que sus procesos de desarrollo, mejora y mantenimiento se vuelvan cada vez más pesados y lentos, por lo tanto, los tiempos de estas tareas no paran de crecer en la misma medida en la que lo hace la aplicación y el grupo de trabajadores.

Este retraso en los tiempos se debe principalmente a una falta de agilidad. Una buena solución practicada en la actualidad por las grandes empresas de software para reducir estos intervalos, deben incorporarse equipos de trabajadores desarrollando en paralelo. El problema se presenta cuando se plantea la incorporación de varios equipos a una aplicación monolítica. Esto requiere grandes esfuerzos de coordinación y se pierde esa ansiada agilidad en el desarrollo.

La división de la aplicación en microfrontends supone un gran avance en este aspecto, ya que permitiría el incluir múltiples equipos de acuerdo a las necesidades de la aplicación, ya que el acoplamiento entre los fragmentos desarrollados por distintos equipos debe ser débil, lo cual lo convierte en un paradigma ideal para conseguir una organización con un alto índice de escalabilidad.

En el siguiente punto, podremos encontrar los beneficios que se pueden obtener con este enfoque arquitectónico, lo cual para muchos puede convertirse en motivaciones para asumir y afrontar el cambio de paradigma.

2.4. Beneficios de Microfrontends

En el apartado anterior, se han presentado los motivos que pueden plantear a una empresa a dar el salto de una arquitectura monolítica en cliente a un sistema basado en microfrontends. Este tipo de sistemas reduce la complejidad de las aplicaciones y resuelve gran parte de los problemas de organización de aplicaciones grandes con difíciles gestión o permite la escalabilidad en equipos de un modo más natural y simple, pero no son los únicos beneficios obtenidos al implantar este enfoque.

De acuerdo al estudio realizado por Peltonen, S. (et al.) [1], en el cual también se realiza un análisis sobre los beneficios obtenidos por las entidades tras implantar microfrontends, podemos observar los aspectos más relevantes en cuanto a beneficios según este autor. Los datos obtenidos reflejan que los dos factores más beneficiosos son la capacidad de utilizar diferentes tecnologías y la mejora sustancial en el aspecto organizativo de la empresa. De acuerdo a este análisis, un 51% de los estudios presenta el utilizar distintas tecnologías como un aspecto beneficio y casi un 77% ellos tratan los beneficios organizativos como el factor más relevante.

De forma paralela a las conclusiones obtenidas por Peltonen, S. (et al.) [1] en su estudio, Geers, M. plantea en su libro "Microfrontends in action", los aspectos organizativos como la mayor ventaja que aportan los microfrontends; *"Los beneficios más importantes que aporta nuestra arquitectura de frontend fragmentado son los organizativos. Permite paralelizar el desarrollo. Factores como el equipo como unidad completamente independiente y la toma de decisiones locales, pueden conducir a innovaciones más rápidas."* [2].

Para explicar un poco más en profundidad las ventajas que pueden ser obtenidas se ha elaborado un listado de los beneficios al usar microfrontends [1,2,3,6,7]. Estos beneficios son extensos y como hemos dicho, no se reduce sólo al ámbito tecnológico, pero hay que dejar claro que esta arquitectura no es una *Silverbullet*[23] [3] que vaya a resolver todos nuestros problemas o que sea ideal para todos los casos que se nos presenten. A continuación, se exponen algunas de las ventajas más significativas que podemos obtener implantando microfrontends.

[23] SilverBullet o bala de plata: Solución sencilla y suficiente para distintos problemas complejos.

2.4.1. La tecnología avanza rápido.

Es conocido globalmente que el software avanza a una velocidad abrumadora. El contexto concreto de las aplicaciones de frontend no es una excepción, al contrario, se mantiene en continuo cambio, apareciendo y desapareciendo librerías y frameworks continuamente.

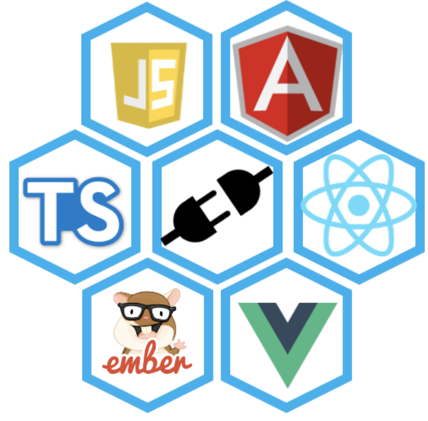

Figura 14: Múltiples tecnologías conectadas.
Fuente: Microfrontends.info (https://microfrontends.info/).

Los microfrontends nos permiten adaptar la tecnología que más nos conviene a cada fragmento independiente de nuestra aplicación, haciendo posible migraciones de tecnología por fragmentos o mantener distintas versiones de una misma tecnología. Esto ofrece una gran versatilidad a la hora de diseñar, desarrollar y mantener aplicaciones, mucho mayor que la obtenida hasta la fecha con aplicaciones monolíticas en las que teníamos que usar una tecnología común o hacer una migración completa si esto fuera necesario, lo que se puede convertir en una tarea hercúlea, sobre todo si nuestro software está en producción.

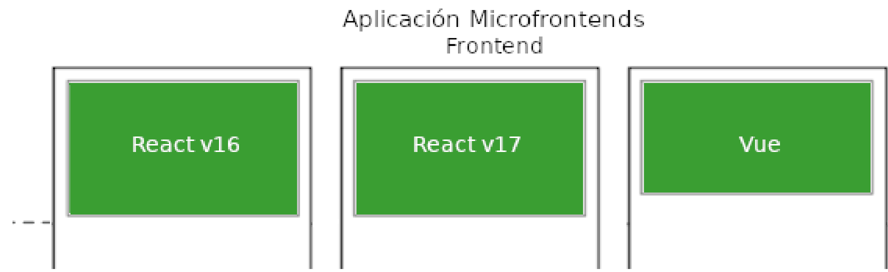

Figura 15: Distintas tecnologías, una aplicación.
Fuente: Adaptación Propia - Basada en Gráficos (Geers, M. (2020). “Microfrontends in Action”).

2.4.2. Fragmentos verticales mejor enfocados. Cuanto más pequeño, mejor enfocado.

El fragmentar la aplicación en artefactos verticales pequeños y centrados en un propósito concreto, evita que se pierda el foco o objetivo de cada fragmento. Esto ayuda a los equipos a no solaparse mientras desarrollan nuevas funcionalidades o amplían el alcance de una funcionalidad ya creada. Además, este enfoque permite asignar el dominio completo a un equipo, que será el responsable, tanto de la aplicación servidor, como de la aplicación cliente, aportándole mucha más independencia en la toma de decisiones.

2.4.3. Desarrollo independiente.

Este tipo de enfoque nos permite que cada equipo tenga un desarrollo independiente del resto. Esto no sólo es una ventaja desde el punto de vista tecnológico, sino que se evitan puntos de bloqueo o espera entre equipos, que son tan indeseados para una empresa, lo que se traduce en un desarrollo mucho más rápido y ágil.

2.4.4. Reduce el riesgo de fallos a secciones aisladas.

La posibilidad de dividir una aplicación en partes aisladas aporta cierto grado de contingencia a errores, ya que un error en un fragmento, no debe comprometer al resto de la aplicación, que podrá seguir funcionando, mientras el sistema se recupera del fallo.

2.4.5. Un equipo, una misión.

Cada equipo de la empresa entiende cuál es su cometido dentro de la aplicación y tiene clara su responsabilidad. En ningún caso se establece que un equipo sea encargado de un único microfrontend, podría ser encargado de varios o incluso, ser encargado de un dominio concreto de la empresa que comprenda parte de servidor y parte de cliente. El avance importante en este aspecto es que al poder dividir la aplicación cliente, permite dejar al claro cuál será la responsabilidad y el alcance de cada equipo, evitando de este modo colisiones o bloqueos durante el desarrollo.

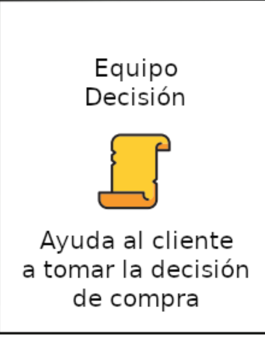

Figura 16: Un equipo, una misión.
Fuente: Geers, M. (2020). "Microfrontends in Action".

2.5. Problemas en Microfrontends

No todo son ventajas a la hora de utilizar microfrontends [1,2,3,6,7,8,10]. Hay que tener en cuenta que este tipo de arquitectura es todavía muy joven y su complejidad no es trivial. Por lo tanto, en ningún caso debe ser tomada como una solución ideal para cualquier proyecto. Cada caso debe ser estudiado con detenimiento y analizando los pros y contras que nos ofrece este enfoque sobre nuestro sistema.

Prosiguiendo con el estudio de Peltonen, S. (et al.) [1], que hemos comentado en los apartados anteriores, los problemas más referenciados en los estudios analizados se dividen en dos grupos, relacionados con la tecnología y relacionados con los trabajadores/equipos. Dando especial relevancia a los problemas de experiencia de usuario y dependencias compartidas a nivel tecnológico con un 39% de presencia (una de las bases sobre las se que se sustenta e intenta mejorar este trabajo), y el incremento de complejidad técnica y de gobierno para la orquestación de los equipos y sus desarrollos en entornos diferentes con un 33% de relevancia.

A continuación, se presenta en forma de listado descriptivo una recopilación general de los problemas más representativos de esta arquitectura de acuerdo a la bibliografía consultada en su conjunto:

2.5.1. Consistencia en la experiencia de usuario

El hecho de establecer equipos autónomos, muy poco o nada dependientes entre sí, convierte en un auténtico reto el conseguir una experiencia de usuario homogénea. Incluso, el diseño de la interfaz puede verse afectado por incongruencias entre los diferentes fragmentos de la aplicación. El separar las hojas de estilos o establecer un marco común de estilos muy bien documentados para toda la empresa puede ayudar a reducir estos problemas. También se podría incluir web components para el uso común de todos los microfrontends. Por otra parte, esto limitaría una de las ventajas de los microfrontends, ya que nos obligaría a utilizar la misma librería o framework como motor de vistas para todos los microfrontends (Por ejemplo: React).

A día de hoy, no hay una respuesta acertada para todos los casos. Cada problema debe ser analizado de forma aislada y plantear la solución que mejor se adapte a nuestro sistema.

2.5.2. Varias tecnologías, Mayor tamaño

Las aplicaciones basadas en microfrontend deben mantener los fragmentos que comparten escenario lo más aislados posible unos de otros. Cada micro aplicación debe tener su propia tecnología (framework o librerías, con sus respectivas versiones), el evitar compartir código permite un desarrollo mucho más independiente y consigue un sistema más fácil de construir y mantener. Esta ventaja tecnológica y organizativa no es gratuita, tiene asociado un problema directo, puede implicar duplicidad de librerías o incluso provocar la carga de distintas librerías o frameworks para un propósito similar, lo que incrementa el tamaño de la aplicación.

2.5.3. Código duplicado

Del mismo modo que ocurre con las librerías o frameworks de terceros, también nos podemos encontrar con código propio duplicado en distintos microfrontends. Tal y como se ha demostrado anteriormente en microservicios, el hecho de duplicar código entre micro aplicaciones, en ciertos casos es beneficioso a la hora de conseguir una mejor legibilidad y mantenimiento del software. Hay que tener en cuenta que en el caso de los microfrontends, la aplicación debe ser enviada al navegador, por lo que el tamaño de la misma debe ser una cuestión tratada con cuidado, por lo tanto, la duplicidad de código también.

2.5.4. Aumenta el nivel de complejidad técnica y organizativa

Se debe tener en cuenta que debido a la naturaleza de este tipo de arquitectura, su implementación conlleva inevitablemente un incremento de la complejidad del sistema en su conjunto. Cada uno de los microfrontends se desarrollan de forma independiente, por lo que deben ser orquestados e integrados de forma adecuada para que la aplicación funcione correctamente. Actividades que con aplicaciones monolíticas no eran necesarias. Por lo tanto, no se debe tomar a la ligera el adoptar este enfoque, se tiene que realizar un análisis exhaustivo de la aplicación y entender en profundidad las necesidades para hacerlo, tanto técnicas, como organizativas.

2.5.5. Conocimiento aislado

Este enfoque es ideal para añadir equipos y escalar con facilidad, sin embargo, al estar cada equipo trabajando sobre códigos diferentes, cada equipo ha de encontrar soluciones a problemas iguales o similares. Esto se da debido a un aislamiento técnico de los equipos.

Este problema se podría solucionar fomentando reuniones internas, dentro de un ambiente distendido, en la que los miembros de los distintos equipos puedan compartir sus preocupaciones, dudas y soluciones. O de una manera más formal, presentando cada cierto tiempo charlas sobre la solución que ha encontrado un equipo X sobre un problema Y.

2.5.6. Entornos diferentes

El desarrollar aplicaciones en entornos distintos al entorno de producción siempre implica un cierto riesgo de fallo. Cuando hablamos de microfrontends, este riesgo aumenta debido a que no sólo se desarrolla en un entorno distinto al de producción, sino que además, se debe integrar con otros microfrontends, con los que en muchos casos deben comunicarse. Si el resultado esperado es un desarrollo lo más desacoplado entre equipos, esta comunicación entre los fragmentos puede ser complicada de mantener.

En el concepto ideal de microfrontends, no debería existir comunicación entre los mismos, pero en un entorno real, esta premisa se vuelve demasiado difícil de cumplir. Suele resultar interesante, o necesario, compartir/comunicar entre las distintas micro aplicaciones atributos del

estado de cada una de ellas, esto hace que cada equipo deba estar al tanto de posibles cambios en las estructuras de datos que compartan, lo que complica la idea de un desarrollo independiente y desacoplado.

El intentar solventar esta problemática es uno de los aspectos que han sido considerados fundamentales para este trabajo.

2.5.7. Incremento del riesgo durante las actualizaciones

El despliegue e integración de este tipo de aplicaciones resulta un punto crítico en muchos casos. Desde un punto de vista teórico e idealista, cada microfrontend debe ser completamente independiente y autosuficiente, pero en una aplicación real, como comentamos en el punto anterior, suele ser necesaria la interacción entre microfrontends, por eso existen los distintos sistemas de comunicación que vimos en el apartado anterior. Debido a esto los cambios o errores pueden aparecer directamente en la última etapa del desarrollo o incluso en producción, retrasando todo el proceso porque los equipos tienen dificultades a la hora de probar la aplicación completa.

Este trabajo además, trata de aliviar las consecuencias de este tipo de riesgos, preparando un entorno más realista al de producción para cada desarrollador.

2.5.8. Retos de accesibilidad

Estos problemas de accesibilidad no resultan endémicos de la arquitectura de microfrontends, sino es que suelen estar presente en la mayoría de aplicaciones que realizan su renderizado en el lado cliente.

Si este punto fuera crítico para nuestro proyecto, podríamos optar por una composición en servidor o una composición en el ”borde”, tal como vimos en el apartado de composición del epígrafe Microforntends.

2.6. Integración Continua

Un punto que debemos valorar en este apartado de estado del arte son las soluciones de integración continua que existen en el mercado y cuál es su funcionamiento, así como porque no se adaptan adecuadamente a mejorar la experiencia de un desarrollador de Microfrontend.

Las herramientas de integración continua (CI) son sistemas encargados de realizar las tareas necesarias para la puesta en producción de una aplicación, pero que resultan repetitivas y no aportan ningún valor. Estas tareas, suelen ser las siguientes, la sincronización de código desde sistemas de control de versiones, la construcción, el lanzamiento de tests y en el caso de despliegue continuo, también el despliegue en el servidor destino. Todas estas operaciones pueden ser realizadas cada vez que se realice una consolidación de código o simplemente bajo demanda.

Uno de los métodos o estrategias aplicados junto con los sistemas de integración continua es el control de la transición entre entornos. Esta estrategia se centra en desplegar la aplicación en distintos entornos de acuerdo al estado de maduración que hayan alcanzado las fases de desarrollo, pruebas y aseguramiento de la calidad (QA, *Quality Assurance*). Normalmente, esta estrategia se centra en tres entornos, en primer lugar y el menos maduro, es el entorno de desarrollo, suele estar relacionado con el estado de pruebas más temprano, cuando la funcionalidad aún se encuentra en desarrollo. En segundo lugar, cuando una funcionalidad ya ha sido completamente desarrollada, debe ser desplegada en un entorno de pruebas y aseguramiento de la calidad o también denominado preproducción. En este segundo escenario, se suelen encontrar errores o fallos que han pasado desapercibidos para los desarrolladores, por lo que se vuelve a depurar esos problemas. Por último, una vez la funcionalidad ha sido comprobada por la sección de calidad y no se notifican errores, se realiza la integración y despliegue en producción.

En sintonía con la estrategia comentada, existen distintos flujos de trabajo de git que se pueden utilizar, sin embargo, la base para cumplir con el sistema más básico, es que deben tenerse tantos branchs en nuestro repositorio, como entornos se necesiten. En el caso propuesto en el párrafo anterior, necesitaríamos tres branch, uno de desarrollo, uno de preproducción o QA y un tercero de producción. Cada uno de estos branchs serán los que alimenten las integraciones y despliegues continuos llevados a cabo por nuestros sistemas de CI.

2.6.1. Jenkins

Jenkins (https://www.jenkins.io) [30] es un sistema de integración continua basado en java y distribuido como *open source.* Se trata de uno de los sistemas CI más extendidos en el mundo. En 2017, contaba con más de 155.000 instalaciones activas y más de un millón y medio de usuarios, y sus números no han parado de crecer [21].

Este entorno se distribuye para ser montado en un servidor tomcat y permite la ejecución de tareas repetitivas para la puesta en producción de aplicaciones. Permite configurar la sincronización de código con múltiples servicios de control de versiones, construir aplicaciones (con soporte para numerosos lenguajes), lanzar tests y desplegar las aplicaciones en servidores remotos.

Para conseguir su objetivo, el sistema utiliza *Pipelines*, basada en el patrón de software con misma denominación, a partir del cual el proceso va transformando el resultado de acuerdo a su paso por las distintas fases contiene, por ejemplo, sincronización de código, construcción, pruebas y despliegue.

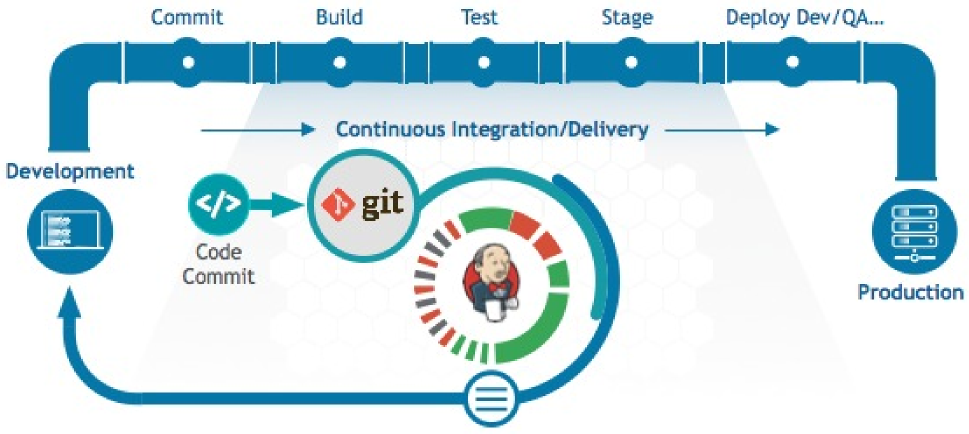

Figura 17: Pipeline de Jenkins.

Fuente: RedHat (https://www.redhat.com/en/blog/continuous-delivery-jboss-eap-and-openshift-cloudbees-jenkins-platform).

2.6.2. Travis CI

Travis CI (https://www.travis-ci.com) [31] es, junto con Jenkins, otro de los sistemas de integración continua más conocidos y usados del mundo. Se encuentra desarrollado con Ruby y aunque se trataba de un proyecto *open source,* en 2019 fue adquirido por Idera, Inc. y en la actualidad se distribuye como software como servicio, eso sí, dispone de un plan gratuito para proyectos *open source* [22].

Es un servicio pensado para la sincronización de código desde sistemas de control de versiones, construcción de aplicaciones, tests y despliegue. Como podemos observar posee las mismas características que Jenkins, como diferencia principal podemos destacar que su configuración es más simple, ya que se realiza mediante la carga de un archivo de configuración básico y nos abstrae de mantenimiento y seguridad debido a que es ofrecido como servicio (Se puede encontrar en distintos proveedores de hosting y servicios a Jenkins en este mismo formato). Por contra, actualmente resulta un servicio de pago para todos los proyectos que no sean de código abierto.

2.6.3. Bamboo CI

Bamboo (https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html) [32] es un sistema de integración continua de la propiedad de la compañía Atlassian. Se trata de un software comercial con licencia privativa. Al igual que sus competidores, Bamboo sincroniza código, construye, testea y despliega aplicaciones de forma automática.

La forma de trabajar de Bamboo se basa en un árbol de definición de procesos compuestos por escenarios, trabajos y tareas. Resulta el más complicado de configurar de los sistemas de integración continua expuestos en este documento, aunque no implica una dificultad excesiva. Además, está bien documentado y su interfaz es más amigable que el resto de sistemas comentados.

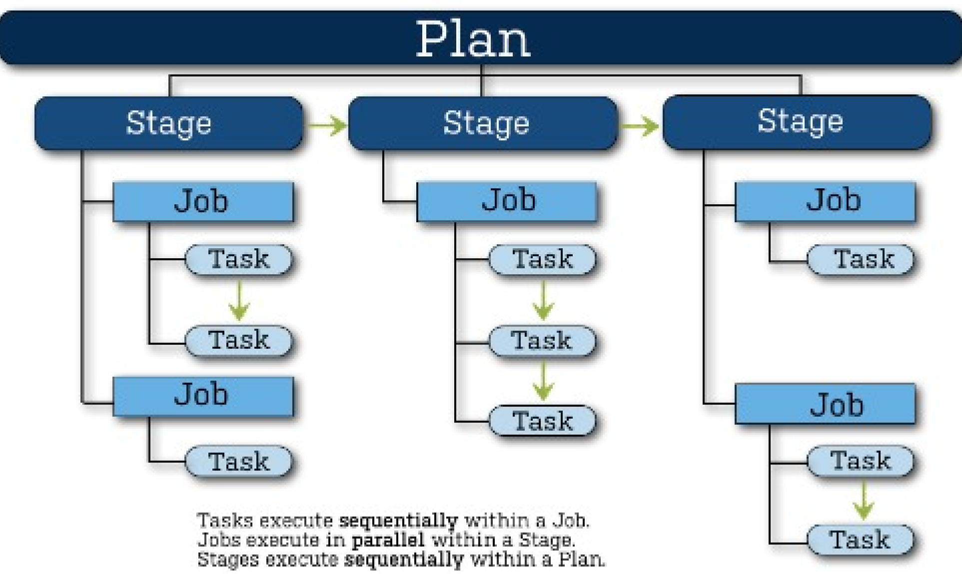

Figura 18: Planes de Bamboo.

Fuente: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

Si nos fijamos, ninguno de estos sistemas ofrece al desarrollador una libertad sobre las distintas versiones de código de distintas microaplicaciones [2,8]. Es decir, no le aporta independencia suficiente para probar las versiones de código que desee como si estuviera en su propia máquina. Sólo podría ejecutar las versiones se haya decidido sean desplegadas en cada en momento por sus propietarios. En resumen, no se aproxima a una experiencia similar a la de su desarrollo en local y sigue resultando dependiente de las decisiones de otros equipos.

2.7. Quién usa microfrontends

A pesar de que la arquitectura de microfrontends es un enfoque muy novedoso, muchas grandes empresas han decidido apostar por él debido a las mejoras que suponen para sus servicios y su compañía. Algunas de las más relevantes son:

2.7.1. Spotify

Spotify fue fundada en el año 2006 por Martin Lorentzon y Daniel Ek, en Estocolmo, Suecia. La aplicación de Spotify es un fenómeno de masas y es casi imprescindible para cualquier persona y/o

negocio en todo el mundo. En la actualidad, cuenta con 100 millones de suscriptores de pago, 217 millones de usuarios activos mensualmente, más de 50 millones de canciones y está disponible en 79 países.

Figura 19: Logo Spotify.
Fuente: Spotify (https://www.spotify.com/).

Para Spotify uno de los principales factores de su éxito ha sido la capacidad de construir una "cultura de ingeniería ágil" en su empresa. Esta estrategia está basada en formar equipos multidisciplinares (Squads) y autónomos en la toma de decisiones, que se encuentren levemente acoplados dentro del propósito general [15,16]. Cada equipo es responsable de todo el ciclo de vida de su misión, diseño, despliegue, mantenimiento, fechas de entrega, etc… El equipo decide qué hacer y cómo hacerlo dentro del alcance establecido para su misión y los plazos de entrega de su resultado. Esto ofrece un entorno más ágil con despliegues más rápidos y sin las complicaciones que conllevan las dependencias y coordinación entre equipos.

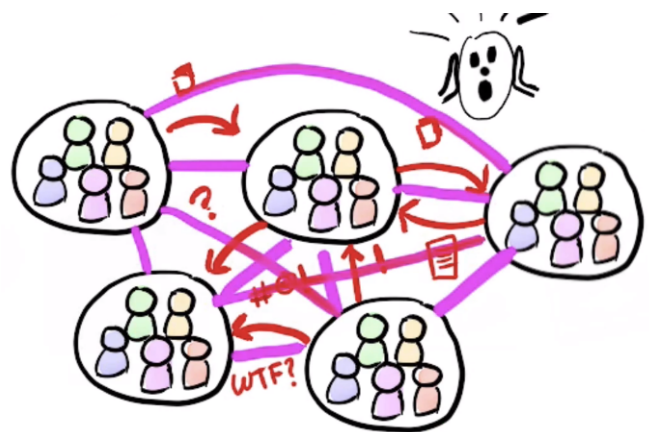

Figura 20: Dependencias y coordinación de equipos.
Fuente: Spotify Engineering Culture (https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1/).

Este enfoque ha sido todo un éxito para Spotify y se adapta perfectamente a un Diseño Dirigido por Dominio (DDD) y a una arquitectura de microservicios en servidor, pero en el lado cliente seguir esta "cultura de ingeniería ágil" era complicado, debido al uso prioritario de aplicaciones monolíticas. Gracias a los microfrontends, han conseguido extender esta idea también a la aplicación cliente, en la que cada equipo es responsable de un dominio concreto y se divide la aplicación verticalmente, siendo encargado cada equipo de su parte concreta.

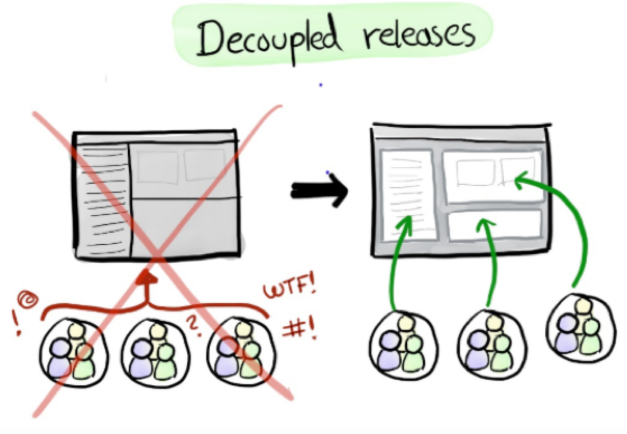

Figura 21: Lanzamientos desacoplados.

Fuente: Spotify Engineering Culture (https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1/).

2.7.2. Ikea

Ikea, la mundialmente conocida tienda de muebles, cuenta con una sección de desarrollo software con la que hacer frente al reto que supone vender a nivel global con un sistema distribuido y adaptado a cada zona y con más de 2500 millones de visitas anuales a su tienda online.

Figura 22: Logo Ikea.

Fuente: Ikea (https://www.ikea.com/).

Ikea ha decidido adoptar este tipo de arquitectura dentro de su apartado online. Después de modificar su arquitectura de servidor y migrar su gran aplicación monolítica a microservicios, observaron que el resultado fue una gran mejora en cuanto a rendimiento y escalabilidad, no sólo a nivel tecnológico, sino también organizacional. Fue en este momento cuando, Gustaf Nilsson Kotte (Senior Software Engenier en Ikea) planteo *"Los microservicios son una arquitectura muy diversa y heterogénea, que permite diferentes lenguajes, creo que esto debería ser aplicado también en frontend"* [17].

Por esto, Ikea comenzó a implementar su aplicación cliente basándose en una arquitectura de microfrontends, esperando obtener las ventajas que consiguieron previamente en su sistema de backend con microservicios. Ellos además plantean de este modo, paliar la velocidad de cambio de las tecnologías de frontend, pudiendo realizar migraciones parciales e incrementales en periodos de tiempo mucho más cortos que si tuvieran que migrar una gran aplicación monolítica, en cuyo caso, estimaba Gustaf Nilson Kotte, *"podría llevarles de 2 a 4 años de ciclos de reescritura, algo que sería fatal para el negocio"* [17].

Además, la llegada de los microfrontends, les ha permitido dividir todo su sistema verticalmente en dominios, por lo tanto, sus equipos no se dividen de acuerdo a tecnología de servidor y cliente, sino de acuerdo a propósito de negocio. Por ejemplo, si ellos tuvieran tres equipos de 8 personas, no destinan dos a servidor y uno a cliente, sino que cada uno de esos equipos se encargará del lado servidor y cliente del propósito que les haya sido asignado, por ejemplo, equipo de ventas, equipo de inspiración y equipo de producto.

2.7.3. DAZN

DAzn es un servicio de OTT (over-the-top) especializado en la emisión de deportes que transmite sus contenidos en vivo y bajo demanda. La empresa es propiedad de Perform Group y fue lanzada por primera vez en 2016 en Austria, Alemania, Japón y Suiza. Hoy en día se encuentra disponible en más de 200 países y cuenta con más de 2600 empleados.

Figura 23: Logo DAzn.
Fuente: DAzn (https://www.dazn.com/).

Al igual que otras empresas de similares características, después de un primer lanzamiento con una aplicación monolítica en cliente, la empresa se dió cuenta que debían repensar su plataforma si querían ser capaces de escalar, no sólo tecnológicamente, sino también desde un punto de vista empresarial. Aunque se comenzó a escalar mediante métodos más tradicionales incorporando nuevo personal, en el lado cliente estos métodos no resultaron una buena solución, al contrario, complicaba mucho más la gestión del programa.

Con esta problemática presente, el arquitecto software de DAzn, Luca Mezzalira (Es actualmente uno de los principales precursores de este tipo de arquitectura), se plantea arriesgar con un nuevo concepto de arquitectura enfocada a cliente, llamada microfrontends. Esta arquitectura de cliente se adaptó perfectamente a su arquitectura distribuida de servidor enfocada a servicios y les permitía descentralizar el desarrollo y la toma de decisiones sobre la aplicación cliente [3,9].

2.8. Tecnologías y Frameworks

Microfrontend es una arquitectura software para cliente, tal y como hemos explicado a lo largo del presente documento, es por ello, que la parte de vital importancia es la conceptual. Teniendo esta premisa presente, a continuación, se presentan algunas tecnologías que pueden facilitarnos la implementación de este enfoque, pero que en ningún caso son indispensables para poder desarrollar un sistema basado en microfrontends.

2.8.1. Bit

Figura 24: Logo Bit.
Fuente: Bit (https:/bit.dev/).

Bit (https://bit.dev/) puede ser una de las plataformas/tecnologías más extendidas dentro de la gestión de frontends. Funciona como un servicio con planes de pago y gratuitos de acuerdo a la funcionalidad ofrecida. No sólo está enfocada a la construcción de aplicaciones basadas en microfrontends, sino que abarca mucho más, desde construcción de aplicaciones con componentes resuables, servicio para component cloud o integraciones con otras plataformas, entre otros muchos servicios.

Una de sus características principales es que realiza la composición de microfrontends en "tiempo de construcción" (o build-time) y sirve el artefacto generado como una aplicación monolítica, en lugar de ejecutar esta composición en "tiempo de ejecución" (o run-time), que es más habitual.

Cabe destacar que dentro de su plataforma, Bit ofrece un servicio de Integración y despliegue continuo (CI/CD) que permite automatizar y comprobar que tanto la construcción, como el despliegue, se realicen de forma correcta y sin incidencias. Una herramienta que puede resultar muy útil, aunque existen alternativas *open source* con este mismo propósito, como Jenkins (https://www.jenkins.io/). Sin embargo, este sistema no se adapta a cada desarrollador, al igual que el resto de sistemas de integración continua y además se vuelve la aplicación dependiente de su servicio.

2.8.2. Luigi

Figura 25: Logo Luigi.
Fuente: Luigi (https://luigi-project.io/).

Luigi (https://luigi-project.io/) es un farmework para facilitar el desarrollo de aplicaciones con arquitecturas basadas en microfrontends. Es una herramienta open source apoyada por la multinacional SAP (https://www.sap.com), una de las compañías de software ERP y CRM más importantes del mundo.

La idea principal en la que se basa Luigi es en generar una aplicación cliente contenedora que es agnóstica a cualquier otra tecnología. Esta aplicación se encargará de contener y comunicar los microfrontends, además del enrutamiento del sistema.

Luigi se encuentra formada por una librería core y librerías clientes de acuerdo a las necesidades y objetivos de cada proyecto.

2.8.3. Piral

Figura 26: Logo Piral.
Fuente: Piral (https://github.com/smapiot/piral/).

Piral (https://github.com/smapiot/piral) es un framework a partir del cual construir aplicaciones basadas en microfrontends. Al igual que Luigi, se trata de una herramienta *open source*. A pesar de ser una herramienta relativamente joven, dada su facilidad de uso ha tenido buena acogida por la comunidad.

Permite crear aplicaciones desacopladas y construidas en tiempo de ejecución mediante llamadas a módulos denominados pilets. Cada pillet se puede desarrollar de forma completamente independiente al resto. Se basa en una aplicación central o core que es la encargada de construir mediante los Pilets, la aplicación final.

2.8.4. PuzzleJS

Figura 27: Logo PuzzleJS.
Fuente: PuzzleJS (https://github.com/puzz le-js/puzzle-js).

Puzzlejs (https://github.com/puzzle-js/puzzle-js) se trata una herramienta para construir aplicaciones basadas en microfrontends y que se encuentra inspirada en el concepto de Big-Pipes[24] de facebook.

Una de las características principales que ofrece este framework es la capacidad de establecer gateway o puertas de enlaces, que nos permiten conectar y comunicar los microfrontends entre sí,

[24] Big Pipe: (https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/)

todo esto a partir de un archivo de configuración. Su modo de construcción está contemplado en tiempo de compilación.

Permite preparar un renderizado previo en servidor, lo que lo convierte en una aplicación amigable con los motores de búsqueda. Además, es capaz de garantizar que aunque alguno de los microfrontends se haya roto, el resto de fragmentos de la aplicación puede continuar en funcionamiento.

3. Solución

El objetivo de este proyecto es plantear una forma de facilitar la integración entre distintos microfrontends, con distintos entornos de desarrollo y distintas versiones de la forma más ágil posible. De este modo, se delegaría con mayor facilidad las labores de integración y pruebas directamente entre los equipos. Esto mejora la experiencia del desarrollador y facilita la integración entre equipos y en consecuencia, la gestión a alto nivel.

La conceptualización de esta idea es ofrecer al desarrollador de Microfrontends la experiencia de que tiene todo el control sobre el código de la aplicación. Como si estuviera trabajando de forma completamente independiente. Por ejemplo, si la última consolidación de código de un equipo A falla, un desarrollador del equipo B podría construir la anterior (como si hiciera un *revert* en *git*) y continuar con su trabajo hasta que el equipo A solucione el fallo. Otro ejemplo, puede ser, si un equipo A está desarrollando sobre un *branch* de *funcionalidad* (según el paradigma de GitFlow) un fragmento y el equipo B necesita integrarse con esa nueva funcionalidad, pueden ir probando en paralelo los avances sin comprometer, ni ser intrusivos entre equipos.

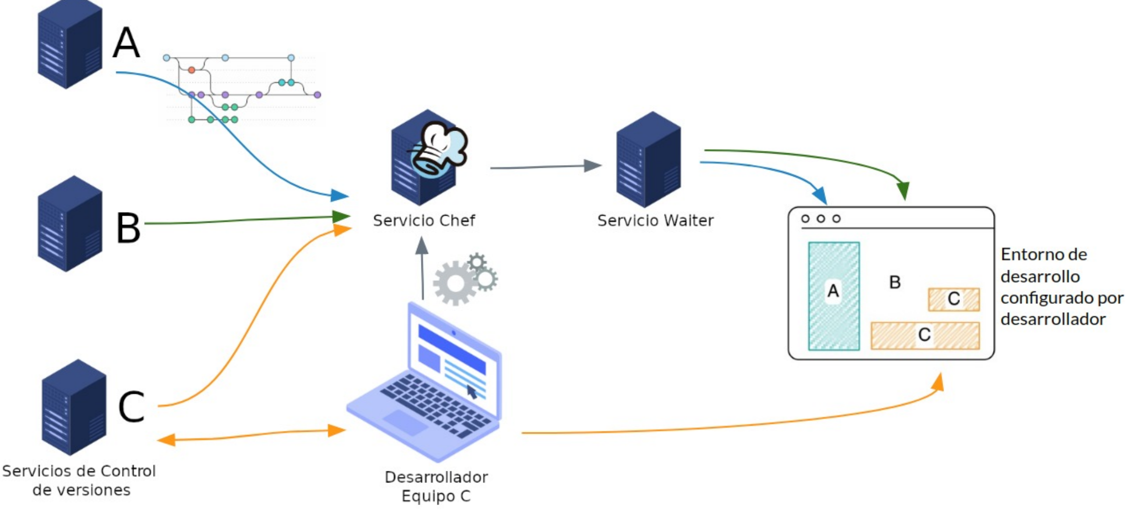

Figura 28: Conceptualización de la solución.
Fuente: Elaboración Propia (Gráfico "Pantalla" - Geers, M. (2020). "Microfrontends in Action").

Por lo tanto, se busca la capacidad de tener un sistema de control de versiones y la construcción de las mismas a elección del desarrollador, pero en un servicio remoto, abstrayendo al trabajador de toda la complejidad de la sincronización y construcción de múltiples aplicaciones, pero obteniendo todas las ventajas de tener todas las líneas de código en su ordenador.

Para conseguir este objetivo, se propone diseñar un Software como Servicio (SaaS) de integración continua bajo demanda disponible para todos los integrantes de los distintos equipos que trabajan sobre software basado en la arquitectura de microfrontends. La idea que busca este servicio es la construcción de versiones de microfrontends bajo demanda. De este modo, se abstrae a los desarrolladores de los entornos de desarrollo de otros equipos, centrando sus esfuerzos en su/s fragmento/s, pero con la capacidad de operar sobre una aplicación en su conjunto, evitando simulaciones del resultado final.

3.1. Especificación de requisitos

Para afrontar la finalidad de este proyecto, comenzaremos por desglosar cuáles serían los requisitos funcionales necesarios para satisfacer las necesidades contempladas. Describimos los requisitos funcionales como los encargados de definir el comportamiento interno que debe tener la aplicación. Los requisitos no funcionales, por su parte, exponen criterios y/o directrices a tener en cuenta a la hora de comprobar el comportamiento general de un sistema. Ahora, nos centraremos en los requisitos funcionales que debe cumplir este proyecto, ya que los requisitos no funcionales serán principalmente conseguir que el sistema sea lo más sencillo, ágil e intuitivo a la hora de conseguir la integración de fragmentos en un entorno local:

3.1.1.1. Restringir acceso a usuarios registrados

El acceso al servicio de cualquier usuario debe estar controlado mediante un sistema de autentificación basado en usuario y contraseña. Esta autenticación permitirá al usuario acceder al panel de control del servicio de integración continua bajo demanda.

3.1.1.2. Asignar cada fragmento construido con su solicitante

La aplicación principal sobre la que se esté desarrollando en cada momento, debe solicitar las microaplicaciónes ya construidas mediante un token único para cada usuario. De este modo, el servicio podrá saber que microaplicación debe ser servida en cada petición.

1. **Registrar nuevos usuarios**

El sistema debe permitir la creación de nuevos usuarios. En este caso sólo el superadministrador será el responsable de esta labor. Al registrar a un usuario, éste recibirá un email con una contraseña anónima y aleatoria.

* Este sistema de registro de contraseña se ha planteado para simplificar las partes no relevantes del proyecto. Se podría implementar un sistema de autentificación mediante un sistema de verificación y creación de contraseña o cualquier otro paradigma de registro/autenticación que se desee.

2. **Registrar nuevas microaplicaciones**

Cada usuario del sistema debe poder registrar nuevas microaplicaciones. Sólo el usuario que haya registrado una microaplicación podrá modificarla. Cada uno de estos fragmentos tendrá un identificador externo a modo de nombre que lo identifica a nivel organizativo en la empresa. Este identificador será utilizado para solicitar las aplicaciones ya construidas (Por ejemplo: ‘cart’).

3. **Configurar sinc/construcción de microaplicaciones**

En cada microaplicación registrada en el sistema debe ser posible configurar el repositorio desde dónde se sincronizará el código, así como los credenciales de acceso para su servicio, establecer su entorno de desarrollo y modelar cómo debe ser construida por medio del listado de comandos necesarios para este fin.

4. **Solicitar construcción de una microaplicación**

Cada usuario debe poder solicitar la construcción de cualquier aplicación registrada en el sistema de acuerdo a la consolidación de código (commit) que estime oportuno o necesite para poder avanzar en su trabajo.

Esta construcción estará disponible sólo para el usuario que la ha solicitado, pudiendo cada desarrollador montar una composición de microaplicaciones completamente a medida y de acuerdo a las necesidades que tenga en cada momento.

5. **Notificar finalización de una construcción**

Las construcciones de microaplicaciones pueden tardar incluso varios minutos en completarse. Por lo tanto, el usuario debe ser notificado de algún modo de que la microaplicación ya está disponible para ser utilizada.

Esta notificación puede ser el envío de un email o una notificación vía websocket desde el servicio notificando la finalización de la construcción y disponibilidad de la microaplicación para ser servida.

6. **Servir microaplicación construida**

Las microaplicaciones construidas deben ser entregadas a la aplicación de cada desarrollador que haya solicitado una construcción. Para ello, el usuario debe montar su aplicación y añadir a las solicitudes de microaplicaciones al servicio de integración continua bajo demanda un token único que los identifique.

3.2. Actores

El servicio ha sido planteado desde un punto de vista simple, dando importancia a la idea de construir y servir versiones de microaplicaciones bajo la demanda de cada desarrollador. Es por esto que se ha decidido mantener el tipo de usuarios implicados al mínimo, lo que también da una mayor

versatilidad y horizontalidad al proyecto, ya que todos los desarrolladores pueden registrar nuevas microaplicaciones y/o solicitar la construcciones de cualquiera de los microfrontends registrados en el mismo.

1. **Superadministrador**

Debe ser configurado al instalar el servicio. Este usuario recibirá un email con su contraseña y será el único usuario con permiso para registrar nuevos usuarios.

2. **Desarrollador**

El usuario desarrollador debe ser capaz de registrar nuevas microaplicaciones y modificar las que sean de su propiedad. A su vez, debe poder solicitar construcciones específicas de cualquier microaplicación registrada en el sistema.

3.3. Casos de uso

En este punto se describen los distintos casos de uso que debe cubrir el comportamiento del servicio de integración continua bajo demanda que se ha propuesto como mejora para la implantación de arquitecturas basadas en microfrontends. Se describen casos de uso básicos como el de identificación y registro de usuarios, que no resultan especialmente relevantes para el servicio que nos atañe, pero que forman parte del diseño y nos ayudarán a implementarlo.

1. **Identificar usuario**

Objetivo: Obtener permiso para acceder a la aplicación.

Descripción: El servicio debe ofrecer un sistema de autenticación a los usuarios para que puedan acceder al servicio.

2. **Registrar a un nuevo usuario**

Objetivo: Añadir un nuevo usuario desarrollador.

Descripción: El servicio debe gestionar el registro de nuevos usuarios al sistema. Sólo un superadministrador tendrá permisos para el registro de nuevos usuarios.

3. Registrar una nueva microaplicación

Objetivo: Añadir una nueva microaplicación en el sistema.

Descripción: El servicio debe dar la capacidad de registrar nuevas microaplicaciones. Este debe gestionar la información necesaria para poder sincronizar código desde su respectivo repositorio, así como las instrucciones necesarias para ejecutar la construcción de la microaplicación y poder configurar su entorno de desarrollo.

4. Construir una microaplicación

Objetivo: Construir una versión específica de microaplicación.

Descripción: El servicio debe ser capaz, de una vez elegida una microaplicación y una consolidación de código concreta (commit), sincronizar dicho código, cargar el entorno de desarrollo necesario y construir la microaplicación específica, de acuerdo a las instrucciones registradas previamente. Esta microaplicación construida estará disponible sólo para el usuario que la solicitó, de este modo, cada usuario puede tener distintas versiones de cada aplicación de forma paralela.

5. Solicitar microaplicación construida

Objetivo: Obtener microaplicaciones construidas para montar una aplicación completa basada en microfrontends.

Descripción: El servicio debe poder despachar las microaplicaciones construidas. Cada usuario debe obtener las versiones de microfrontends de las que ha solicitado su construcción.

3.4. Componentización en servicios

La componentización de servicios [6] se encuentra estrechamente relacionada con el diseño de microservicios. Debido a que la complejidad relativa de la propuesta no es muy alta desde un punto de vista técnico, no se estima necesario el uso de microservicios, al menos inicialmente. Aún así, este planteamiento de diseño nos ofrece la capacidad de plantear la funcionalidad requerida de una manera modular y con un bajo acoplamiento.

Existen distintos distintos procedimientos para componentización en servicios. Para este proyecto se ha decidido realizar de acuerdo a las capacidades del dominio. Inicialmente, se han identificado las siguientes capacidades:

1. Gestión de usuarios
2. Gestión de microaplicaciones
3. Despachador de versiones construidas de microaplicaciones

En la actualidad, los servicios más simples están compuestos por múltiples modelos que ayudan a gestionar mejor la información del sistema. Debido a que no se desea hacer perder el foco de atención en modelos relacionados con gestión interna del servicio, autenticación, etc, no los hemos desglosado en este acercamiento, aunque los presentaremos a lo largo de la definición de la arquitectura propuesta.

3.5. Arquitectura

De acuerdo a los resultados obtenidos en el epígrafe anterior de la propuesta, se plantea una arquitectura basada en dos servicios principales. El primero de estos servicios será un gestor y constructor de aplicaciones. El segundo, será un servicio encargado de la entrega de microaplicaciones por versión acorde al usuario solicitante. A modo de simplificar las referencias a estos dos servicios, de ahora en adelante, nos referiremos al primero de ellos como “Chef Service” y al segundo, como “Waiter Service”, haciendo referencia a las distintas labores de un cocinero y un camarero, respectivamente.

De acuerdo con el paradigma de un servicio moderno y escalable, la arquitectura cliente-servidor que ofrecen ambos servicios se encuentra basada en *API Restful*, como tal, el sistema es completamente *stateless* o sin sesión. Por lo tanto, no se mantiene ningún tipo de información del usuario o sesión entre peticiones. Este tipo de enfoque, facilita la integración desde otros servicios de terceros, por lo que es ideal para integrar con otras herramientas/servicios de desarrollo ya creadas.

A lo largo de este apartado, se nombrarán tecnologías específicas que serán definidas en epígrafes posteriores, en los que se ofrecerá al lector una breve descripción y enlaces a la documentación de cada una de ellas.

3.5.1. Chef Service

Tal y como se ha introducido, este servicio *Chef* será el encargado de gestionar y elaborar las construcción de microaplicaciones solicitadas por sus usuarios. Para ello, el servicio debe ser capaz de controlar el acceso y distribuir las peticiones al componente software indicado, así como gestionar el almacenaje de resultados. La propuesta de este servicio se encuentra dividida en los siguientes componentes software.

Autenticación y enrutamiento

El servicio necesita una puerta de entrada para las solicitudes realizadas al servidor desde cualquier cliente. Este componente software debe incorporar seguridad a las solicitudes a partir de un mecanismo de comprobación de los requisitos de autenticación a los recursos del sistema. Si dicha comprobación no es satisfecha, la solicitud debe ser rechazada.

Como hemos visto, al tratarse de un sistema *stateless*, el componente comprobará mediante un sistema de autentificación basado en tokens de acceso la acreditación de un usuario para obtener recursos del sistema. La generación y comprobación de este tipo de autenticación puede ser implementada completamente dentro del componente o utilizar algún proveedor externo de autenticación.

Para el primer caso, planteamos el control de acceso de usuarios mediante el estándar de *JWT, Json Web Token* (RFC-7519) [20] implementado directamente en el servicio. JWT representa un

método de comunicación de información, seguro y autocontenido entre dos partes. Ya que dicho método propone un sistema de firma, chequeo y confirmación.

En el segundo caso, mediante un proveedor externo de autenticación, se encuentra ampliamente extendido el uso del estándar *OAuth 2.0* (RFC-6749) [19], que es implementado por la mayor parte de proveedores de servicios. Este estándar propone ofrecer una autenticación basada en un token de acceso limitado a un servicio de terceros basado en HTTP, mediante la autorización de un usuario del primer servicio.

Existen multitud de librerías que pueden ayudar a implementar cualquiera de los métodos comentados. Cabe destacar el *middleware* de autenticación *Passportjs* debido a que se encuentra ampliamente extendido. Aunque como comentamos no resulta indispensable para el desarrollo de este apartado.

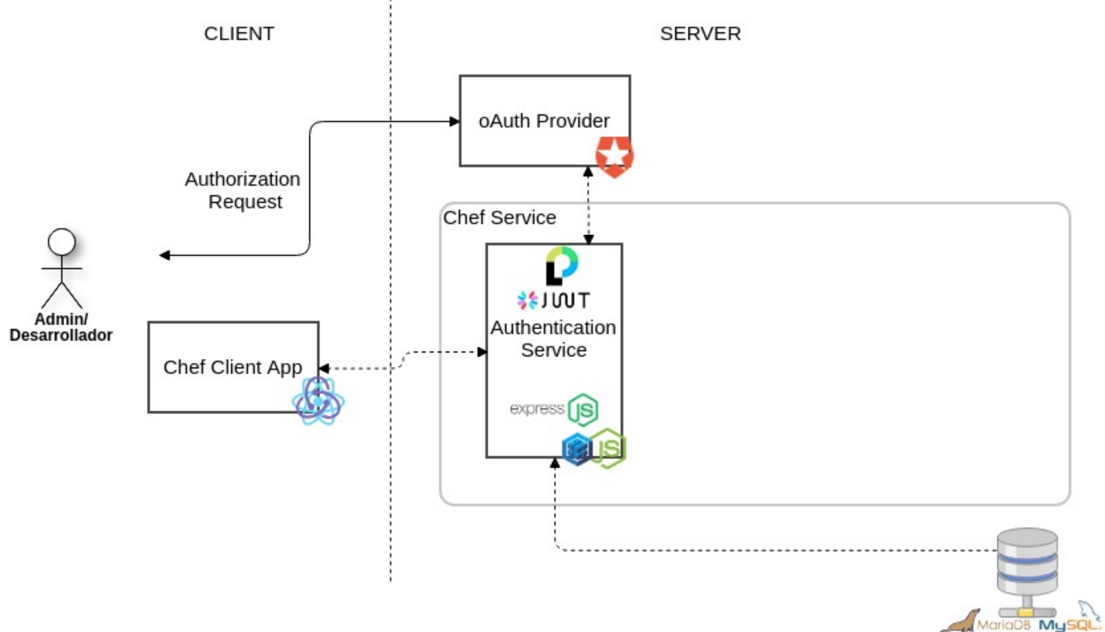

Figura 29: Chef Service - Autenticación y enrutamiento.

Fuente: Elaboración propia.

El enrutamiento de las solicitudes al API también se encontraría en este componente. Se encarga de redireccionar todas las peticiones solicitadas a los componentes o servicios adecuados, así como de devolver los resultados de dichas solicitudes. Estos métodos deben recuperar toda la información enviada en cada petición y pasarla al servicio, así como adjuntar el usuario recuperado

desde el middleware de autenticación. Existen también múltiples tecnologías para este propósito, cabe señalar *Express* si este desarrollo se decidiera realizar en *Nodejs*. Aún así se trata de una lógica ampliamente extendida para la que cualquier lenguaje moderno puede encontrar soluciones sencillas.

Comentar que este componente sería un punto ideal si nos interesa monitorizar las peticiones al sistema, realizar tests o pruebas de estrés. Dado que la idea de este servicio es para uso interno y la implementación de sus componentes no es muy compleja, no se estima necesario el uso de este tipo de estrategias de monitorización y pruebas, al menos de acuerdo al planteamiento actual.

Gestión de usuarios

Dados los requisitos de la propuesta actual, desde el punto de vista arquitectónico, la gestión de usuarios no implica mucha dificultad. Este componente simplemente se encargará de ejecutar las solicitudes de registro y modificación del modelo de datos de usuario.

De acuerdo con el carácter de uso interno que propone este servicio, no se estima necesario un sistema de registro externo. Ya que este sistema sólo debería estar disponible para los equipos de desarrollo que se encuentren trabajando sobre un mismo proyecto.

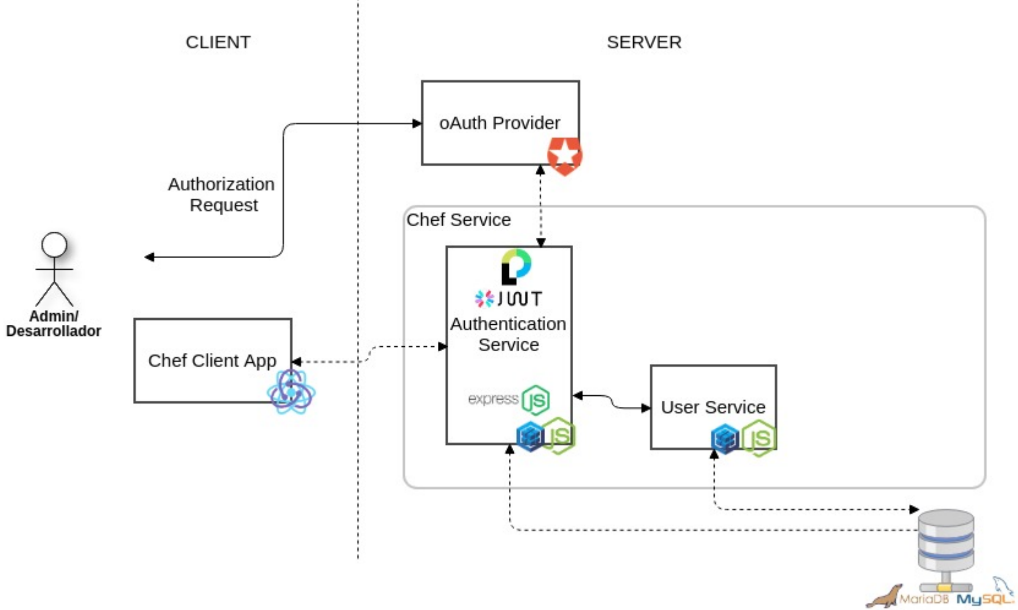

Figura 30: Chef Service - Gestión de usuarios.

Fuente: Elaboración propia.

Podríamos profundizar en la lógica de registro mediante el uso de un sistema de verificación de cuentas, si fuese necesario, o tomar los datos de la cuenta un usuario desde un proveedor de autenticación basado en el estándar *OAuth 2.0* (RFC-6749) [19], pero dado que el grueso de la propuesta no se basa en el sistema de gestión de usuarios, dejamos estas decisiones dependiendo de las necesidades de cada implementación.

Gestión de microapps

El componente para gestión de microaplicaciones se trata del grueso del servicio Chef. Este será el componente encargado del registro y gestión de microaplicaciones, así como de la sincronización de código desde el sistema de control de versiones y las construcciones necesarias por versión y usuario solicitante. Dicho esto, podemos establecer que este se puede subdividir en dos apartados, la gestión de microaplicaciones y la sincronización y construcción de Microaplicaciones por versión.

- **Gestión de microapps**

  Del mismo modo que ocurría con la gestión de usuarios, la gestión de microaplicaciones se basa en una simple capacidad de registro y modificación del modelo de datos, en este caso, el de la microaplicación. Esta información será usada por el componente constructor para obtener el código solicitado desde un servidor remoto de control de versiones, configurar el entorno de construcción y construir la microaplicación para el usuario solicitante.

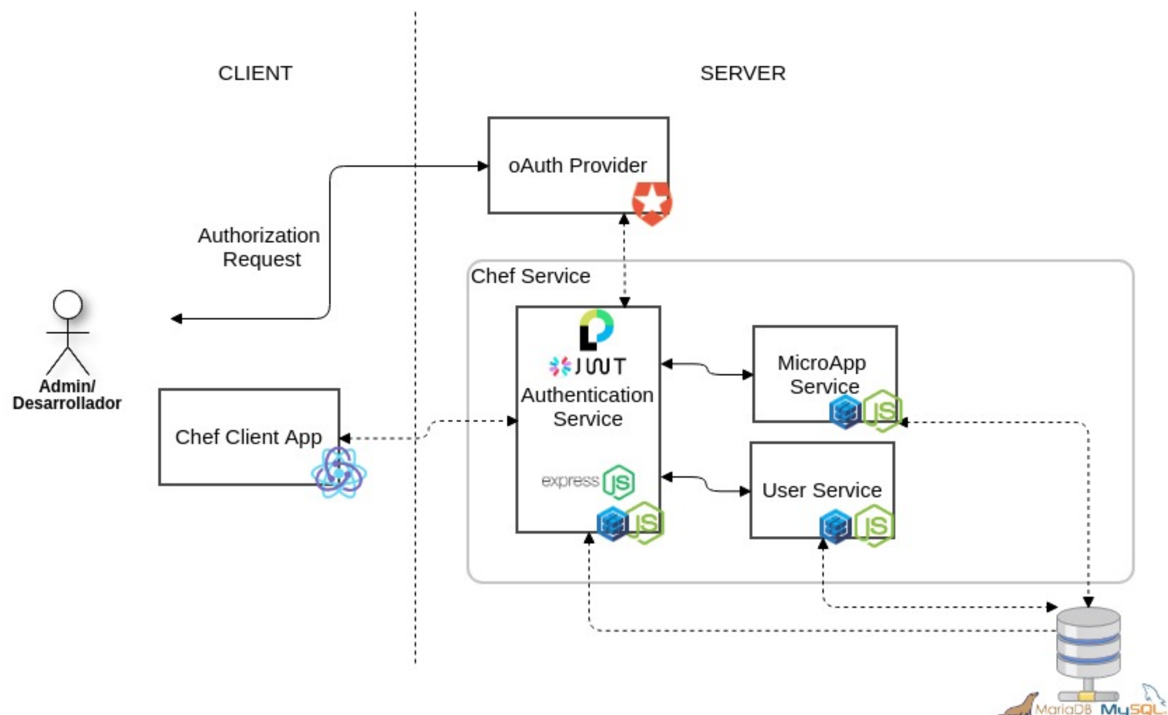

Figura 31: Chef Service - Gestión de MicroAplicaciones.
Fuente: Elaboración propia.

- **Sincronizador y constructor de microapps por versión**

Este constructor o gestor de construcciones representa el núcleo del servicio propuesto. Debe ser capaz de recuperar código de una versión consolidada concreta desde un repositorio, configurar el entorno de construcción específico para esa microaplicación, construir la microaplicación y manejar el paquete generado en un repositorio de versiones.

Proceso en *background:* Como se puede intuir, este proceso no será rápido, debido al número de servicios implicados, sincronización de código y la propia construcción, por lo que debe ejecutarse en *background* (de fondo), convirtiéndose de este modo en un proceso asíncrono. Cada lenguaje tiene métodos propios para la generación de procesos de fondo, por lo tanto, no profundizaremos en esta parte.

Cola de trabajos: El servicio podría recibir varias solicitudes de construcción y tal como se ha comentado en el párrafo anterior, el proceso de construcción no es inmediato, por lo que podría colapsar el servicio. Por lo tanto, se debe implementar un sistema de *cola de*

trabajos para poder dar un soporte sólido a las construcciones realizadas en este servicio sin que el sistema tenga un mal rendimiento o simplemente colapse. Con este propósito existen multitud de tecnologías que dependen del lenguaje con el que se decida implementar el servicio, si se trabaja sobre nodejs, podemos destacar *Bull* debido a su uso amigable y compatibilidad con Redis (Sistema de base datos basado en tablas de Hashes o clave/valor), además de integrarse perfectamente con la gestión de notificaciones.

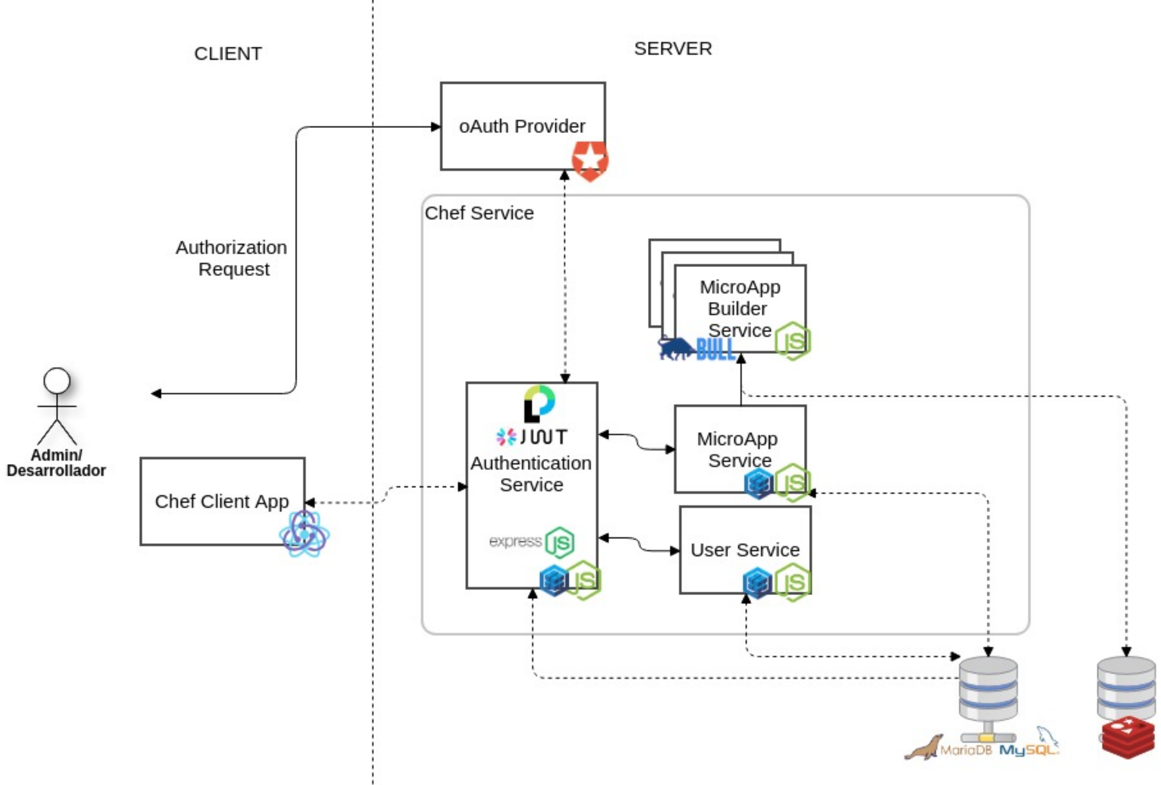

Figura 32: Chef Service - Cola de trabajos de sincronización y construcción de MicroAplicaciones.
Fuente: Elaboración propia.

Avisos y notificaciones: Debido a este tiempo de ejecución, el usuario no puede obtener un feedback inmediato, sólo puede saber de forma síncrona si el proceso se ha iniciado satisfactoriamente o no. Por lo tanto, se debe encontrar una forma de informar al usuario sobre cuándo dispondrá de su aplicación lista para usar. Se propone la implementación de dos métodos de notificación con el propósito de avisar sobre la finalización del proceso (con éxito o fracaso). El primero, un servicio de mail que notifique al usuario enviando un correo electrónico cuándo su pedido esté listo. El segundo, un servicio de

websocket para la comunicación con un cliente web sobre el estado del pedido de construcción en vivo.

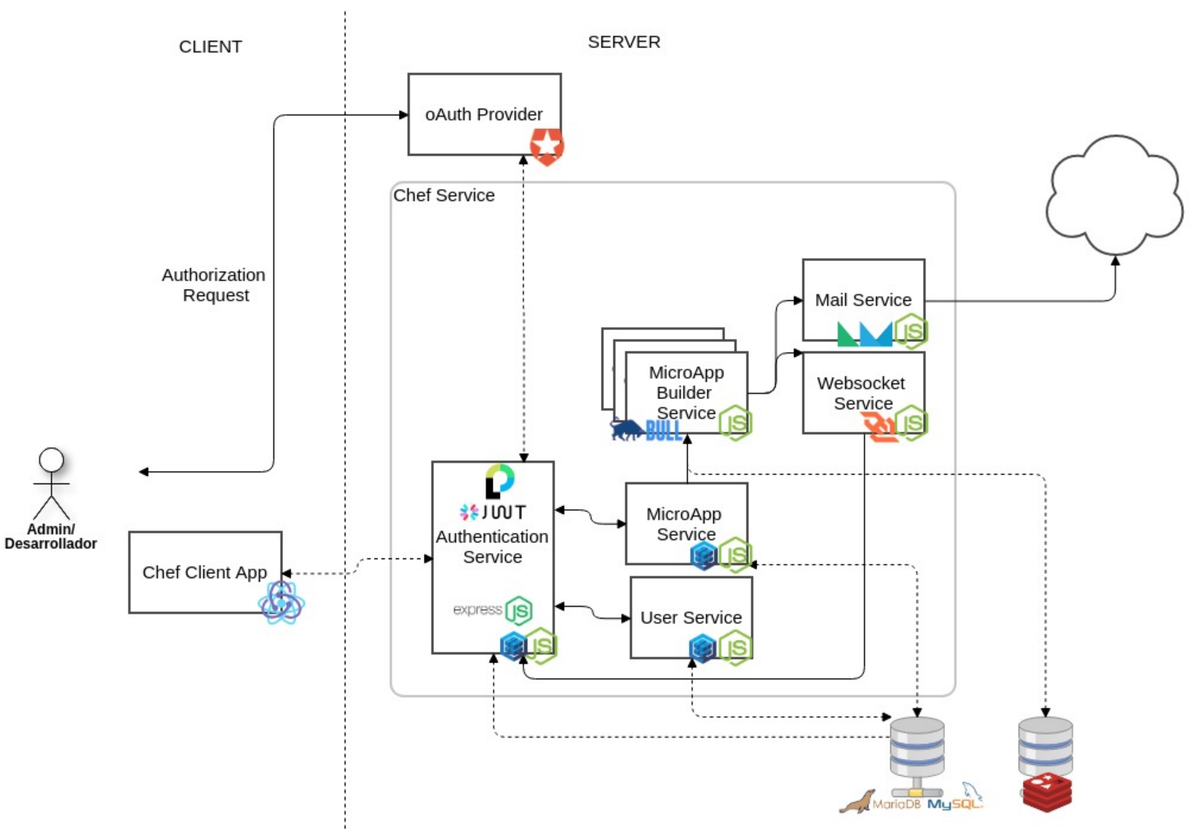

Figura 33: Chef Service - Notificaciones

Fuente: Elaboración propia.

Un desarrollador, una versión: Uno de los principios fundamentales de esta propuesta es que cada desarrollador pueda sincronizar una versión de código a su elección de cada microfrontend para obtener su construcción a medida. Está versión construida se asociará al usuario de acuerdo a un modelo de datos propio de microaplicaciones construidas.

Sincronización con sistemas de control de versiones: Cómo se establece en el punto anterior, cada desarrollador podrá elegir una versión de código a construir. Por lo tanto, el servicio necesita un componente capaz de sincronizar el código desde un sistema de control de versiones remoto. Dicho componente debe encargarse de conectarse al servicio de control de versiones externo (Github, Bitbucket, etc..) y sincronizar la versión de código solicitada

para construir. La información necesaria para conocer qué servicio de control de versiones, credenciales y ubicación del código debe estar definida en el modelo de datos de la microaplicación. La mayor parte de servicios de control de versiones remotos ofrecen un API externa para realizar este tipo de consultas o podemos crear un servicio que utilice un cliente *git* para conectar con los distintos repositorios.

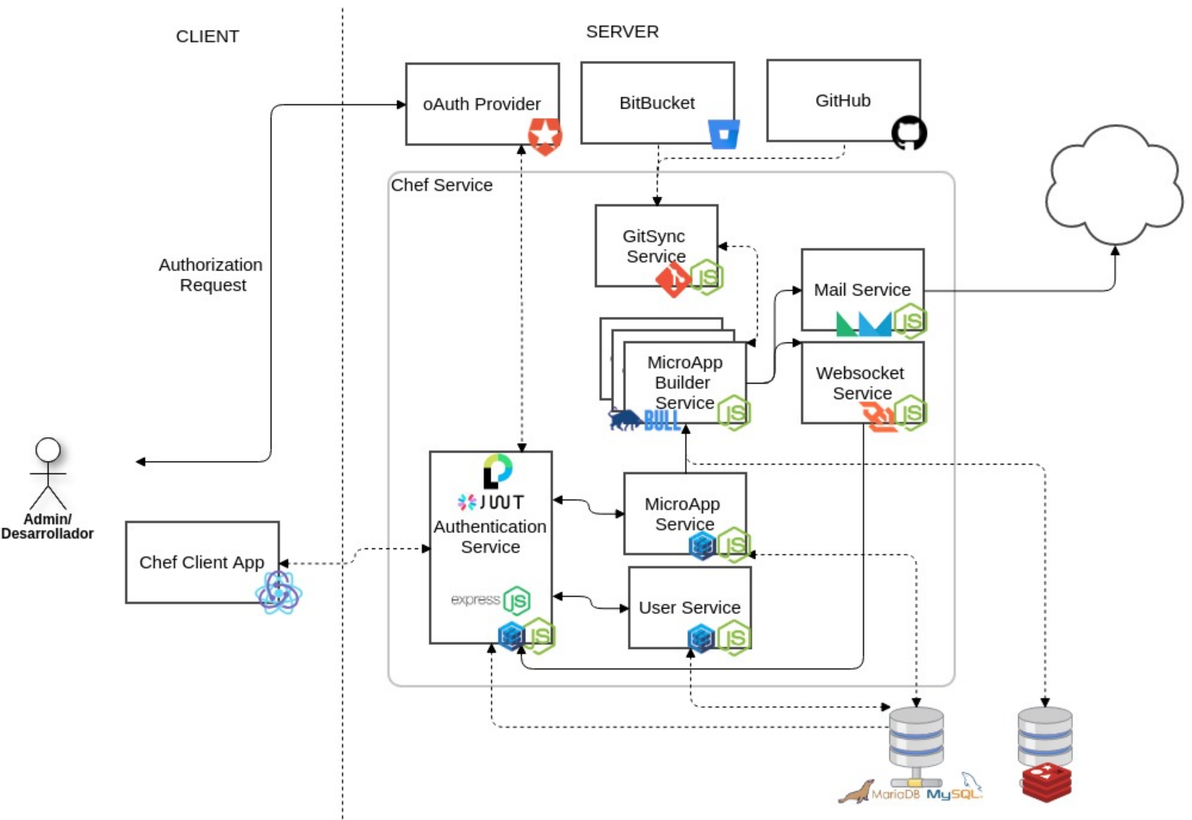

Figura 34: Chef Service - Sincronización de código desde sistemas de control de versiones.

Fuente: Elaboración propia.

Configuración de entorno de desarrollo: Debido a que cada aplicación puede estar desarrollada en un entorno distinto, el servicio debe dar soporte a esta condición y configurar el entorno de acuerdo a las necesidades preestablecidas por el equipo dueño de la microaplicación. Al tratarse de aplicaciones de frontend el entorno de desarrollo será sobre nodejs, aunque cada equipo podrá tener una versión distinta y sus propias necesidades en cuanto a librerías. El servicio deberá configurar la versión de nodejs adecuada y la instalación del entorno necesario. En este caso, se puede optar por un gestor de versiones de nodejs,

como puede ser *nvm* o montar dockers con la versión de nodejs adecuada para cada construcción.

Construcción y almacenaje: Una vez el servicio ha sincronizado el código solicitado del microfrontend y configurado su entorno de desarrollo, se debe ejecutar el proceso de construcción. El equipo de desarrollo propietario del microfrontend debe establecer en el modelo el script necesario para dicha construcción. El sistema lanzará este script, previo análisis, y una vez terminada la construcción, alojará su contenido en una ubicación que llamaremos repositorio de microaplicaciones. El contenido será alojado dentro de un directorio con la concatenación del hash de la consolidación de código (commit) y la identificación uuid de la microaplicación solicitada, quedando identificado inequívocamente para el servicio Waiter.

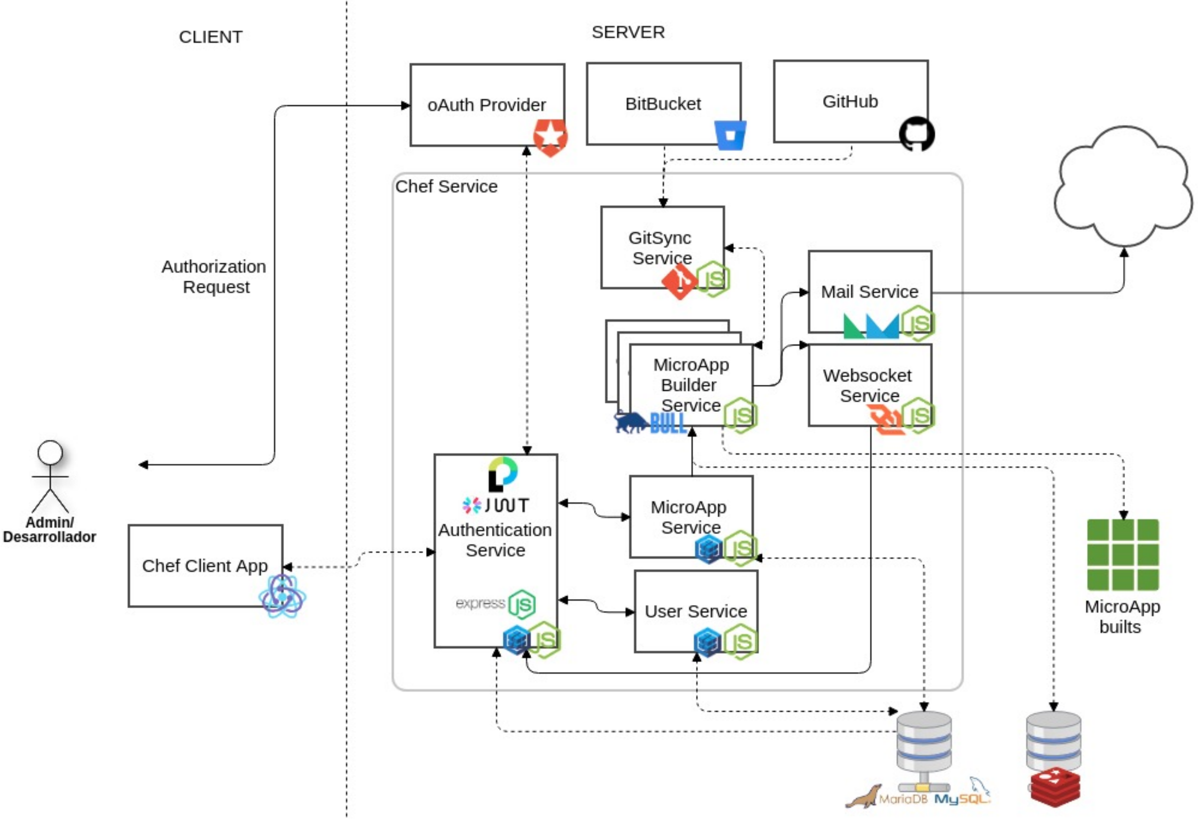

Figura 35: Chef Service - Construcción y alojamiento de versiones construidas.
Fuente: Elaboración propia.

Notificación de finalización: El servicio una vez finalizada la construcción, notificará al usuario mediante un email, haciéndole saber que la aplicación ya ha sido construida su petición y estará disponible desde el servicio Waiter que describiremos a continuación.

3.5.2. Waiter Service

El servicio Waiter será el encargado de despachar todos los microfrontends construidos que han solicitado los desarrolladores registrados en este servicio de integración contínua bajo demanda. Para ello, el servicio debe identificar las peticiones de acuerdo a una url con el identificador único de la microaplicación y un token que será pasado como un parámetro en la *query* de la petición, a partir del cuál se identifica al usuario con el microfrontend construido para él.

Por ejemplo:

https://waiter-service.com/microfrontends/cart?token={uuid}

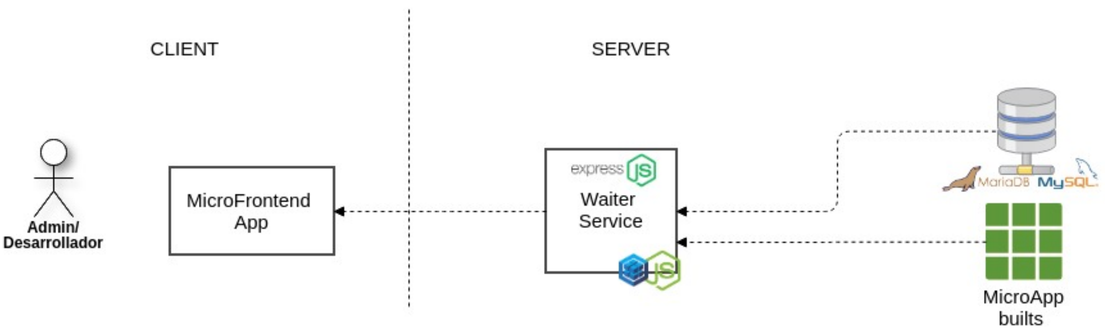

Figura 36: Waiter Service - Despacho de Microaplicación por desarrollador.
Fuente: Elaboración propia.

Esta solicitud al servicio waiter devolverá una versión construida del microfrontend *"cart"* para el desarrollador al que le haya sido otorgado el token "*{uuid}*". De este modo, el desarrollador sólo tendría que tener añadido esta consulta en su proyecto y siempre se le devolverá la última versión del microfrontend que haya solicitado.

```
<script
        type="text/javascript"
        src="https://waiter-service.com/microfrontends/cart?token={uuid}"/>
```

3.5.3. Stack tecnológico

En este apartado se presentan las tecnologías que se proponen para la implementación de esta arquitectura, aunque en ningún caso suponen una restricción o limitación para ser implementados mediante otro tipode lenguajes o tecnologías.

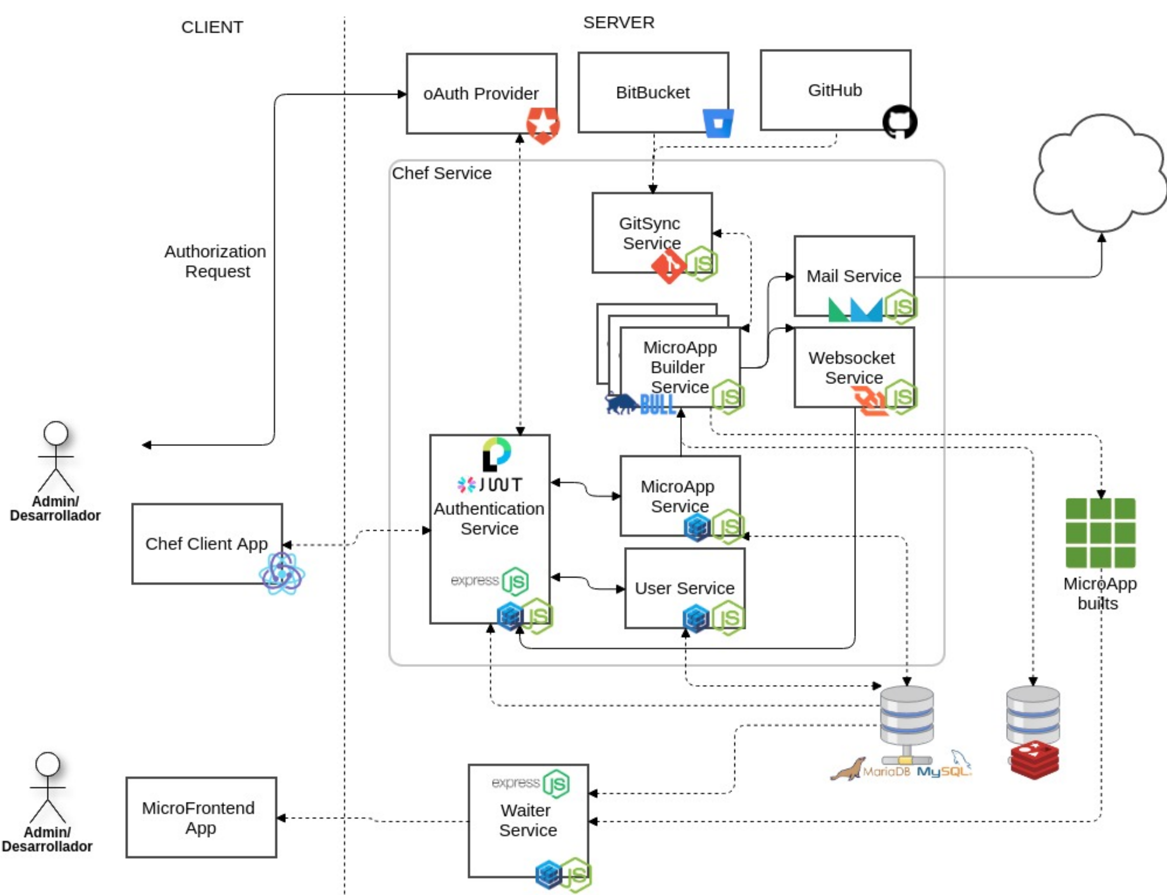

Figura 37: Chef Service y Waiter Service
Fuente: Elaboración propia.

3.5.3.1. Nodejs

Nodejs (https://nodejs.org/) [33] se trata de un entorno de ejecución en servidor basado en Javascript. Posee una arquitectura orientada a eventos y es un

lenguaje asíncrono. Fue diseñado pensado en la creación de aplicaciones altamente escalables y dispone de una baja latencia. Se encuentra basado en el motor javascript de Google V8 y es *open source*.

3.5.3.2. ExpressJs

Expressjs (https://expressjs.com/) [34] es una librería de nodejs dentro del marco de backend para construir infraestructuras para servicio de aplicaciones web. Se caracteriza por simplificar las tareas típicas de para la generación de servicios web y permite crear APIs sólidas de forma rápida y flexible. Es propiedad de OpenJS Foundation y se distribuye como *open source.*

3.5.3.3. Passport

Passport (http://www.passportjs.org/) [35] es un *middleware* de autentificación para nodejs. Permite múltiples métodos, tanto de autentificación, como de autorización de acceso a recursos mediante la definición de estrategias.

3.5.3.4. Sequelize

Sequelize (https://sequelize.org/) [36] Se trata de un ORM (*Object-Relational mapping*) desarrollado para nodejs que permite el manejo de múltiples bases de datos relacionales como Maria DB o MySQL. Simplifica el uso de modelos, maneja transaccionalidad, relaciones, así como cargas *lazy* o bajo demanda, entre otras muchas facilidades que añade esta capa.

3.5.3.5. Bull

Bull (https://optimalbits.github.io/bull/) [37] es una librería para nodejs enfocada a la gestión de colas de trabajos. Se trata de un sistema robusto que se encuentra apoyado por el uso de Redis para la gestión de colas, pero que libera al desarrollador de tareas a bajo nivel sobre este paradigma.

3.5.3.6. MariaDB

MariaDB (https://mariadb.org/) [38] es un sistema de gestión de base de datos relacionales derivado MySQL. Tanto MariaDB, como MySQL son dos de los sistemas de bases de datos más populares del mundo. Gran parte de esta popularidad es porque suele ser utilizado para muy frecuentemente en el desarrollo de aplicaciones web debido a su alta velocidad de lecturas en su modo “no transaccional”. Se trata de un software con licencia GPL (General Public License) y es mantenido de forma conjunta por la *MariaDB Foundation* y la comunidad *open source*.

3.5.3.7. Redis

Redis (https://redis.io/) [39] es un almacén de estructura de datos en memoria. Normalmente, es utilizada como sistema de caché, intercambio de mensajes o como base de datos.

3.5.3.8. Nvm

Nvm (https://github.com/nvm-sh/nvm) [40] se trata de un gestor de control de versiones para nodejs. Permite cambiar la versión nodejs dentro de un entorno de desarrollo, desde la consola y por usuario. Del mismo modo que Expressjs, también es propiedad de OpenJS Foundation y se distribuye como *open source.*

3.5.3.9. Git

Sistema (https://git-scm.com/) [41] de control de versiones basado en git. Se encuentra distribuido bajo una licencia *open source.* Actualmente es el sistema de control de versiones de uso más extendido a nivel mundial y es soportado por la mayoría de servicios de control de versiones, como Bitbucket o GitHub.

3.6. Resultado

En este apartado se mostrará, a modo de demostración práctica mediante una prueba de concepto (PoC), el resultado de una implementación basada en la arquitectura de la solución propuesta en este documento. Esta presentación se basa en un caso práctico en el cual un desarrollador inicia el registro y configuración de una de sus aplicaciones y más tarde, crea una nueva construcción de una versión de una aplicación de otro equipo.

3.6.1. Acceso

Debido a que este servicio es de uso interno, cada usuario debe iniciar una sesión para acceder al panel de control de microaplicaciones. Es en este espacio donde el usuario podrá registrar y configurar los procesos para crear nuevas microaplicaciones, así cómo solicitar sincronizaciones de código y construcciones exclusivamente para él.

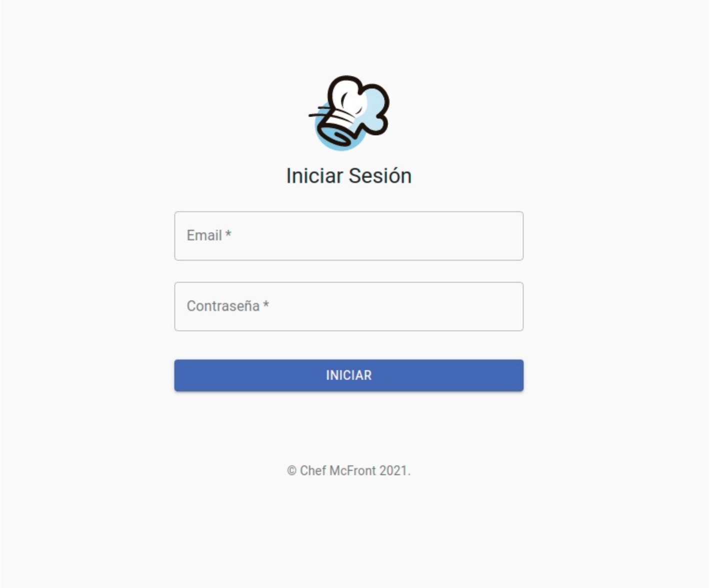

Figura 38: Chef Service - Login
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

Como cualquier sistema de autentificación basado en usuario/contraseña, la aplicación solicitará estos datos para permitir el acceso a la plataforma.

3.6.2. Registro y configuración de nueva microaplicación

Este apartado de registro de la aplicación sería el equivalente a la configuración de un sistema de integración continua tradicional. Concretamente, y tal como vimos en el apartado de *Sistemas de Integración Continua* del apartado de *Estado del Arte,* sería similar a la configuración de *Pipelines* en Jenkins, la configuración de *Planes* en *Bamboo* o la configuración mediante archivo de construcciones en *Travis*. La información contenida en este apartado permite al sistema conectar remotamente con servicios remotos de control de versiones, así como configurar el entorno de construcción, construir y localizar los paquetes resultantes.

En el listado, el desarrollador podrá observar las microaplicaciones disponibles en el sistema en el apartado "MicroApps". Estos datos sólo pueden ser editados por el desarrollador propietario de la microaplicación. Además, el nombre por microaplicación será único, ya que establece el identificador usado por el sistema para servir la aplicación, este aspecto debe ser establecido por convenio dentro de la organización, por ejemplo, el nombre de la microaplicación *"carrito de la compra"* será *cart* y la petición para obtenerla será https://waiter-service.com/microfrontends/cart?token={uuid}.

Figura 39: Chef Service - Listado MicroApps
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

Cada vez que un usuario añade una nueva aplicación debe configurar una información básica que podemos dividir en cuatro apartados. En primer lugar, la información de la aplicación a nivel corporativo y de coordinación, se trata de un nombre que será utilizado para solicitar más tarde la microaplicación construida.

Figura 40: Chef Service - Registrar MicroApp - Denominación
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

En segundo lugar, se debe añadir la configuración para la sincronización del código desde el sistema de control de versiones, para ello el usuario debe configurar el tipo de repositorio (Sólo disponible Bitbucket para esta prueba de concepto), añadir la dirección del repositorio, el usuario y token, así como los branchs desde los que se pueden hacer construcciones

Figura 41: Chef Service - Registrar MicroApp - Sincronización
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

En tercer lugar, se debe configurar cuál va a ser el entorno y los comandos de configuración y construcción de la microaplicación. Para ello, se solicita la versión de nodejs sobre la que está construida la microaplicación y el listado de comandos necesarios para construir la aplicac

Figura 42: Chef Service - Registrar MicroApp - Construcción
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

Por último, el sistema necesita información sobre dónde se encuentra el paquete generado. Debe añadirse la dirección en donde la aplicación aloja sus construcciones.

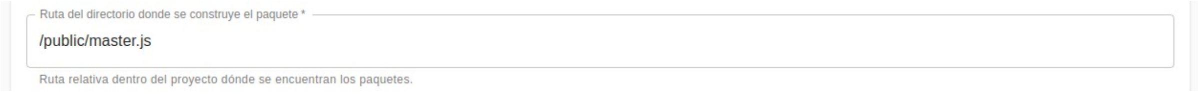

Figura 43: Chef Service - Registrar MicroApp - Despacho
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

Toda esta información se rellena en un único formulario que define la sincronización, construcción y despacho de cada microaplicación.

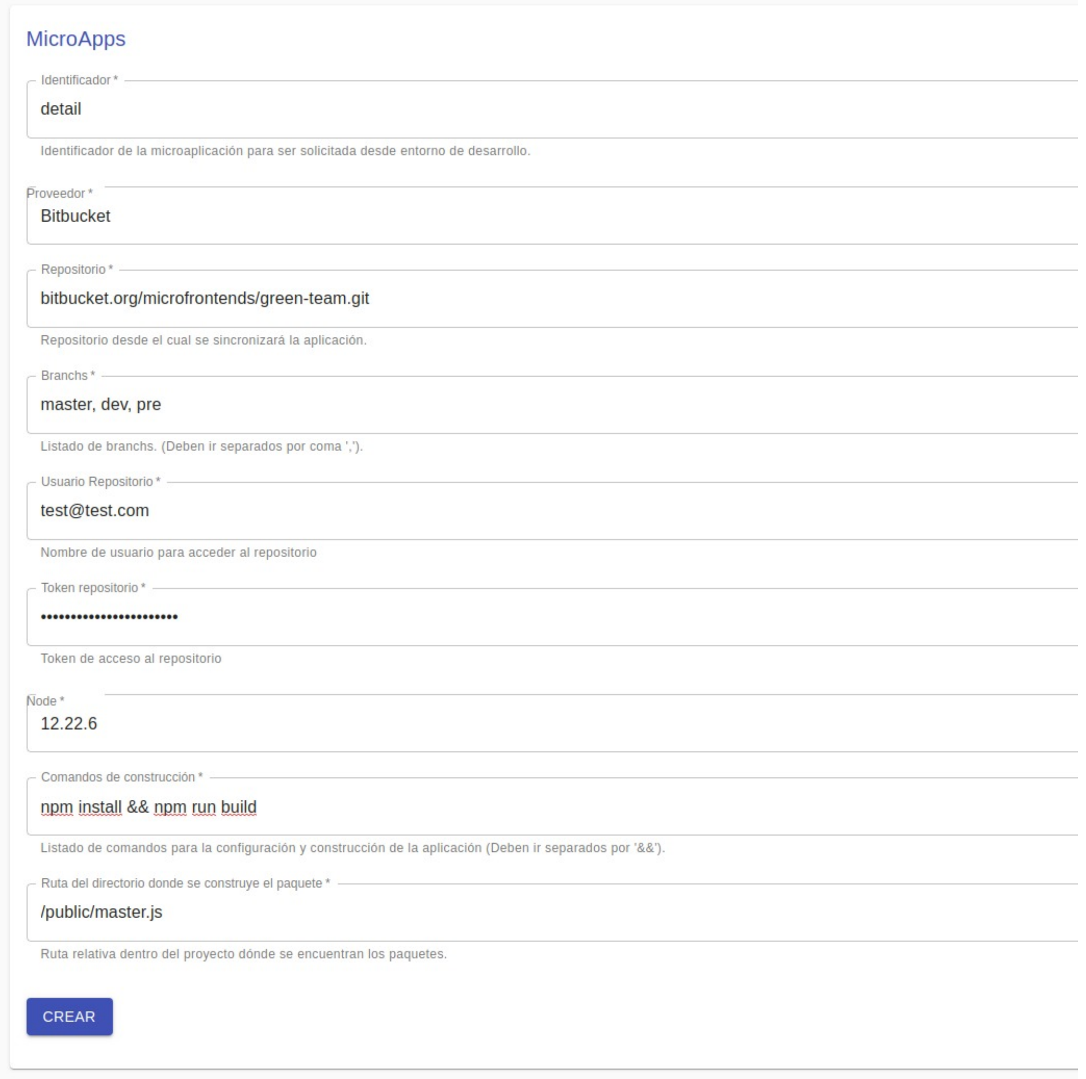

Figura 44: Chef Service - Registrar MicroApp
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

3.6.3. Solicitud de construcción de una versión

Una vez configurada la sincronización, construcción y despacho de las microaplicaciones, cada desarrollador puede solicitar las construcciones que necesite, como si estuviera trabajando desde su máquina. El usuario en este apartado solicita construir una microaplicación concreta de acuerdo a un branch, en cuyo caso se tomará su *head* (última consolidación de código) o desde una consolidación de código concreta especificada por el desarrollador. De este modo, se utiliza esta información para sincronizar y construir una versión específica para él.

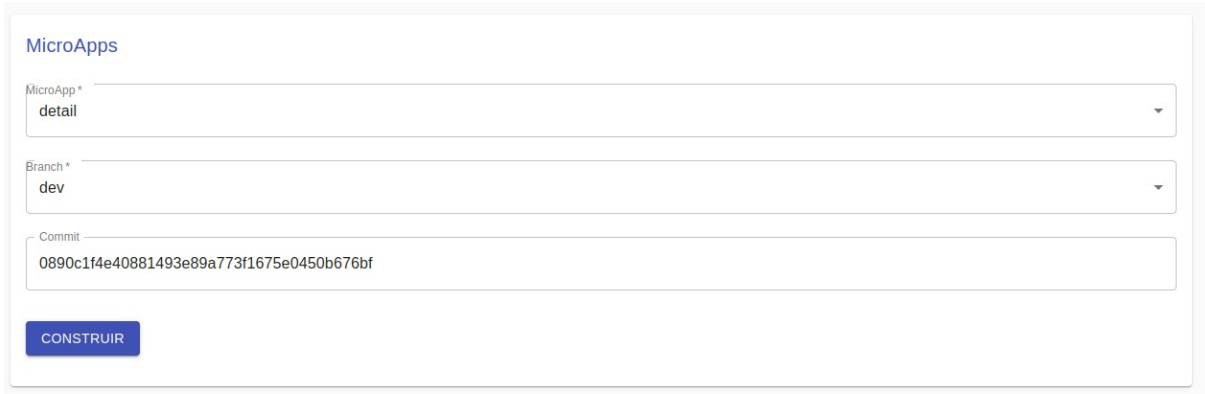

Figura 45: Chef Service - Solicitar construcción MicroApp
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

En el listado de “Mis MicroApps” el desarrollador podrá ver las microApps que ha solicitado para él y el estado en el que se encuentran en ese momento.

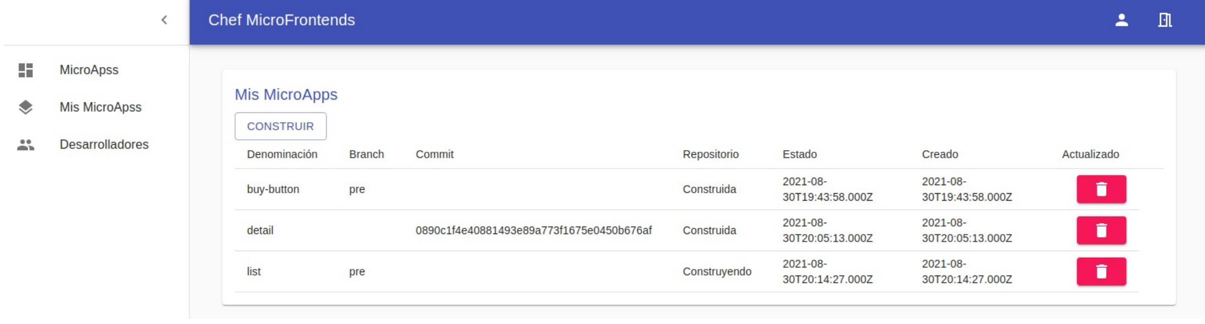

Figura 46: Chef Service - MicroApp Construidas para usuario actual.
Fuente: Elaboración propia - Captura Servicio Chef (PoC).

3.6.4. Entorno de desarrollo

Una vez configuradas y construidas las versiones solicitadas por un desarrollador, este puede cargarlas en el entorno de desarrolllo de su máquina mediante una llamada al servicio waiter con su token de despacho de microaplicaciones. Este servicio, tomará la versión creada por este desarrollador y se la entregará sin la necesidad de sincronizar código, cambiar su entorno de desarrollo, ni construir nada, simplemente debe añadir su token como parámetro del query en la petición.

```
<script
       type="text/javascript"
       src="https://waiter-service.com/microfrontends/cart?token={uuid}"/>
```

3.7. Beneficios

Los beneficios alcanzados con la puesta en marcha de esta propuesta se pueden sintetizar en que se ha conseguido dar agilidad a los equipos y sus desarrolladores con un entorno de trabajo más realista para cada uno de ellos. Desde un punto de vista más detallado los siguientes puntos describen los beneficios que aportaría el software como servicio planteado en el presente documento.

- **Construir aplicación completa en un entorno de desarrollo local:**
  Permite construir una aplicación completa basada en microfrontend para desarrollo en local, sin la necesidad de réplicas de los entornos de desarrollo de todos los equipos que han participado en el proceso de creación.

- **Integración ágil y unilateral de cualquier versión de código de la aplicación:**
  Permite generar una aplicación de una forma rápida y unilateral, basada en cualquier combinación de versión de código consolidado (commit) que exista en los repositorios de otros equipos.

- **Aplicación personalizada para cada desarrollador**:
  Cada desarrollador puede generar su propia combinación de microaplicaciones que necesite en cada momento y trabajar de forma completamente independiente.

- **No necesita publicar versiones internas para desarrollo**:
  Evita petición de liberación de versiones a otros equipos, menos intrusiva.

- **Mayor abstracción entre el desarrollo de backend y frontend**:
  Permite a los desarrolladores de backend abstraerse de los entornos de desarrollo y construcción de frontend. Incluso dentro de su propio equipo.

4. Conclusiones

Este trabajo se ha centrado inicialmente en la recopilación de información sobre la arquitectura de MicroFrontends, presentando así, una base de conocimiento sólida y lo suficientemente completa como para guiar al lector sobre los distintos conceptos y fundamentos de esta temática.

Se ha podido observar durante el estudio, que la mayor motivación para adoptar la arquitectura de Microfrontends dentro de una empresa son las altas cotas de complejidad que están alcanzando las aplicaciones de frontend, junto con la capacidad para escalado horizontal que este nuevo enfoque permite. Por otra parte, MicroFrontends ha conseguido demostrar de forma empírica que mejora sustancialmente la organización de la empresa y la arquitectura de la aplicación monolítica tradicional en frontend, al transformarla en fragmentos más fáciles de gestionar. Sin embargo, el día a día de los desarrolladores se vuelve más complicado.

Desde el punto de vista del autor, uno de los aspectos clave para que el avance de esta arquitectura destaque definitivamente en el sector y en el resto de grandes compañías, es conseguir mejorar la experiencia del desarrollador al enfrentarse a la implementación de una aplicación basada MicroFrontends.

De acuerdo a la premisa anterior, el autor ha concluido que una manera de alcanzar una experiencia de desarrollo más confortable y segura es facilitar a los desarrolladores una herramienta que les permita trabajar en un entorno sumamente parecido al entorno de producción. Es decir, facilitar la implementación de su trabajo de forma aislada al resto de microaplicaciones, pero integrado con estas. Esto permite observar el comportamiento general de la aplicación y un desarrollo completamente en paralelo con el trabajo del resto de equipos. De este modo, se logra evitar la alta incertidumbre de trabajar en un entorno burbuja, sin conocimiento sobre el resultado de su trabajo hasta el último momento, reduciendo a su vez la aparición de resultados inesperados.

La herramienta propuesta se presenta en forma de software como servicio de integración continua bajo demanda. Basada actualmente en las etapas de sincronización de código y construcción de microaplicaciones, este servicio permite a cada usuario solicitar una construcción a medida de acuerdo a sus necesidades en cada momento. En la misma se observan los siguientes comportamientos que suponen una mejora de la experiencia de desarrollo y resuelven o minimizan las problemáticas de trabajar con múltiples entornos de desarrollo y el riesgo en las actualizaciones de microaplicaciones:

1. Permite construir aplicaciones distribuidas con gestión de versionado de código multi-repo o poly-repo, para ser servidas e integradas en un entorno local de desarrollo de una manera ágil y no intrusiva.

2. Permite construcciones independientes por desarrollador. Cada desarrollador puede integrar su trabajo con las versiones de código de otras microaplicaciones que estime oportunas sin interferir o afectar al resto de trabajadores.

3. Permite un desarrollo completamente en paralelo entre equipos. Evita bloqueos y reduce el riesgo durante las actualizaciones en producción.

4. Permite una mayor abstracción entre backend y frontend. Permite a los desarrolladores de backend trabajar y probar sus desarrollos contra distintas versiones de frontend sin necesidad de construir múltiples entornos de desarrollo distintos en sus máquinas.

Debido a lo novedoso de esta arquitectura, en el momento en el cual se ha realizado el estudio no existen herramientas o servicios enfocados a conseguir un entorno de desarrollo ideal para MicroFrontends, sobre todo si hablamos de aplicaciones con composición horizontal en cliente, es decir, varios fragmentos que actúan como una única SPA (*Single Page Application*).

Finalmente, dado que este estudio ha estado enfocado en mejorar la experiencia del desarrollador sobre arquitecturas Microfrontends desde un punto de vista fundado en las deficiencias encontradas por distintos autores y empresas en este ámbito, sería interesante conocer el feedback real de desarrolladores inmersos en este tipo de arquitecturas ante la implantación de este servicio en una

empresa con múltiples equipos distribuidos. Esto nos permitiría contrastar si esta mejora cualitativa se vé también reflejada desde una perspectiva cualitativa.

5. Líneas Futuras

En último lugar, a lo largo del desarrollo de este trabajo han ido apareciendo nuevas ideas que pueden ayudar a mejorar y ampliar el alcance de este proyecto, habilitando la posibilidad de desarrollar nuevas líneas de investigación y análisis.

En este apartado, se presentan las líneas que han sido consideradas de mayor interés o relevancia, de acuerdo con los argumentos expuestos en este documento. Estas nuevas ideas o líneas de trabajo son orientativas y susceptibles a cambiar de acuerdo a nuevos análisis o reinterpretaciones de las mismas. Conforme los estudios realizados se considera principalmente el siguiente punto como una gran carencia en este tipo de sistemas:

1. **Tests de Integración para Microfrontends**

Debido al desarrollo y despliegue distribuido con integración en cliente que caracteriza a esta arquitectura, no existe actualmente ningún mecanismo automático o estratégia de test de integración sólida para microfrontends. Esto sucede especialmente en aplicaciones basadas en microfrontends con composición horizontal en cliente, donde la integración de distintos fragmentos es llevada a cabo en el navegador del usuario .

Dado que no existe ningún mecanismo para este propósito, una vez compuesta la aplicación, la comprobación de la correcta comunicación entre los distintos microfrontends debe ser realizada a mano, con los problemas de mantenimiento y control que eso conlleva a largo plazo.

Sería interesante para mejorar la solidez de aplicaciones basadas en microfrontends alcanzar un sistema automático de testeo de la integración de microfrontends antes de ser lanzados a producción o cualquier otro estado de control de calidad manual.

Bibliografía

[1] Peltonen, S.; Mezzalira, L.; Taibia, D. (2020). **“Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review”.** Tampere University, Tampere, Finland; DAZN, London, United Kingdom.

[2] Geers, M. (2020). **“Microfrontends in Action”.** Manning Publications.

[3] Mezzalira, L. (2020). **“Lessons from DAZN: Scaling Your Project with Micro-Frontends”**. Recuperado de https://www.infoq.com/presentations/dazn-microfrontend/

[4] Pavlenko, A.; Askarbekuly, N.; Megha, S.; Mazzara, M. (2020). **“Micro-frontends: application of microservices to web front-ends”**. Innopolis University, Innopolis. Russia.

[5] Yang, C.; Liu, C.; Su, Z. (2019). **“Research and Application of Micro Frontends”**. Beijing University of Posts and Telecommunications, Beijing, China.

[6] Fowler, Martin; Lewis, James. (2014). **“Microservices, a definition of this new architectural term”.** Recuperado de https://martinfowler.com/articles/microservices.html

[7] Fowler, Martin; Jason, Cam. (2019). **“Micro Frontends”.** Recuperado de https://martinfowler.com/articles/micro-frontends.html

[8] Mezzalira, L. (2019). **“Building Micro-Frontends. Chapter 4. Build and Deploy Micro-Frontends”**. Recuperado de
https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/ch04.html

[9] Mezzalira, L. (2019). **“Micro Frontend Architecture - DAZN”.** Recuperado de https://www.youtube.com/watch?v=BuRB3djraeM

[10] Ball, K. (2019). **“Microfrontends: the good, the bad, and the ugly”**. Recuperado de https://zendev.com/2019/06/17/microfrontends-good-bad-ugly.html

[11] Berners Lee, T. (2017). **“Longer Bio for Tim Berners-Lee”**. W3. Recuperado de ”https://www.w3.org/People/Berners-Lee/Longer.html”

[12] Azaustre, C. (2018). **“Aprende ECMAScript 6, el nuevo estándar de JavaScript”**. Recuperado de https://carlosazaustre.es/ecmascript6

[13] FruhLinger, J. (2020). **“Jamstack: The static website revolution upending web development”.** InfoWorld. Recuperado de https://www.infoworld.com/article/3563829/jamstack-the-static-website-revolution-upending-web-development.html

[14] Berhm, S. (2013). **“Isomorphic Javascript: The Future of Web Apps”**. Airbnb. Recuperado de https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc

[15] Kniberg, H (2014). **“Spotify engineering culture (part 1)”**. Recuperado de https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1

[16] Kniberg, H (2014). **“Spotify engineering culture (part 2)”**. Recuperado de https://engineering.atspotify.com/2014/09/20/spotify-engineering-culture-part-2

[17] Stenberg, J. (2018). **“Experiences Using Micro Frontends at IKEA”**. Recuperado de https://www.infoq.com/news/2018/08/experiences-micro-frontends

[18] CERN (2018). **“A short history of the Web”**.
Recuperado de https://home.cern/science/computing/birth-web/short-history-web

[19] (2012) **“The OAuth 2.0 Authorization Framework”.** Recuperado de https://datatracker.ietf.org/doc/html/rfc6749

[20] (2015) **“JSON Web Token (JWT)”**. Recuperado de https://datatracker.ietf.org/doc/rfc7519

[21] Heller, M. (2020). **“What is Jenkins? The CI server explained”.** Recuperado de https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html

[22] Toledo, F. (2018). **“Travis-CI para integración continua”.** Recuperado de https://www.federico-toledo.com/travis-ci-para-integracion-continua/

[23] Documentación **Typescript**. Recuperado de https://www.typescriptlang.org/

[24] Documentación **Arquitectura Flux**. Recuperado de https://facebook.github.io/flux/

[25] Documentación **Angular**. Recuperado de https://angular.io

[26] Documentación **Reactjs**. Recuperado de https://reactjs.org

[27] Documentación **Vuejs**. Recuperado de https://vuejs.org

[28] Documentación **Redux**. Recuperado de https://redux.js.org

[29] Documentación **Redux**. Recuperado de https://redux.js.org

[30] Documentación **Jenkins**. Recuperado de https://www.jenkins.io

[31] Documentación **Travis CI**. Recuperado de https://www.travis-ci.com

[32] Documentación **Bamboo**. Recuperado de https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

[33] Documentación **Nodejs**. Recuperado de https://nodejs.org/

[34] Documentación **Express**. Recuperado de https://expressjs.com/

[35] Documentación **Passportjs**. Recuperado de http://www.passportjs.org/

[36] Documentación **Sequelize**. Recuperado de https://sequelize.org/

[37] Documentación **Bull**. Recuperado de https://optimalbits.github.io/bull/

[38] Documentación **MariaDB**. Recuperado de https://mariadb.org/

[39] Documentación **Redis**. Recuperado de https://redis.io/

[40] Documentación **Nvm**. Recuperado de https://github.com/nvm-sh/nvm

[41] Documentación **Git**. Recuperado de https://git-scm.com/