Universidad Politécnica de Madrid

Trabajo de Fin de Máster

Marco de trabajo para la definición de tecnologías aplicables a un proyecto software

Alejandro Alcázar

Course of Study:	Máster Universitario en Ingeniería Informática
Examiner:	Óscar Dieste
Supervisor:	Óscar Dieste
Commenced:	January 27, 2020
Completed:	TBD

Resumen

<Short summary of the thesis>

Summary

<Translated summry of the thesis>

Contents

List of Figures

Acronyms

BSON Binary JSON. 22

CMS Content Management System. 15

DOD Document-Oriented Database. 22

e2e End To End. 20

JSON JavaScript Object Notation. 22

ORM Object-Relational Mapping. 22

SEO Search Engine Optimization. 14

SQL Structured Query Language. 21

SSR Server-Side Rendering. 14

1 Introduction

2 Estudio de mercado de las tecnologías disponibles

2.1 Entornos de trabajo del lado de cliente

En esta sección se van a evaluar los entornos de desarrollo más populares en el lado del cliente: React, Svelte, VueJS, Preact y Angular. Como preámbulo, se muestra en la figura Figure 2.1 la satisfacción de los usuarios con respecto a cada entorno. Esto es relevante porque la satisfacción de los usuarios depende de factores como la eficiencia de código y la curva de aprendizaje. Como queremos que nuestro marco de trabajo esté destinado a aplicaciones pequeñas que tienen potencial para crecer, es importante conocer cuál es la opinión media de los usuarios de 2019. Además queremos que el entorno sea accesible al mayor número de usuarios, por lo que la popularidad es el factor más relevante en este estudio.

Parece que los dos más importantes en este sentido son React y VueJS (Svelte lo voy a descartar por el poco tiempo en el mercado). Según la comparación de rendimiento que presenta Schae [Sch19] en su artículo, Vue parece líder en aplicaciones ligeras y rápidas. Por otro lado, en la comparativa que desarrolla Tyagi [Tya19], se puede apreciar que React es utilizado por un 64.8% de los desarrolladores web contra los 28.8% que están con Vue. Además, React cuenta con más soporte y documentación que respaldan sus años de presencia en el mercado, mientras que la para floja de Vue hoy por hoy es su soporte.

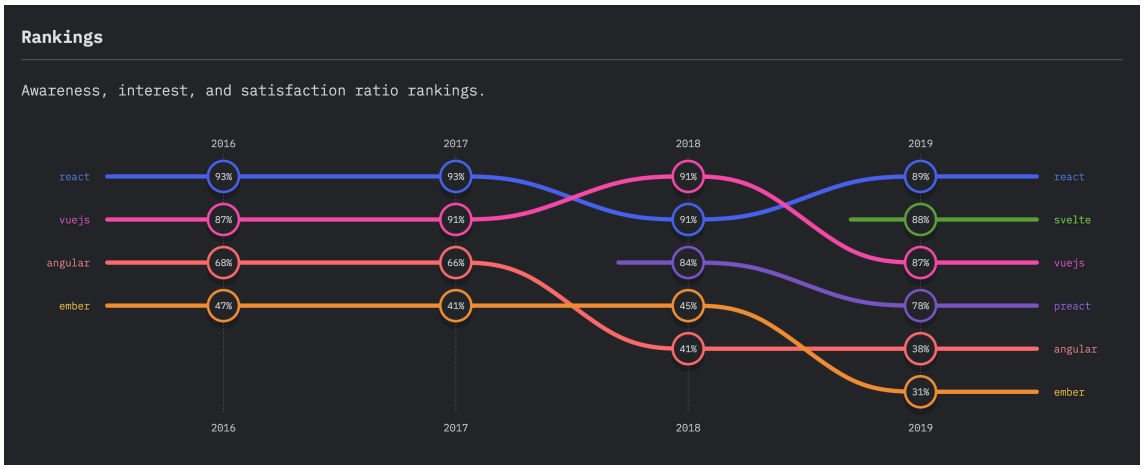

Figure 2.1: 2019 - Opinión popular de los entornos de trabajo front-end

Hay otro análisis a tener en cuenta para realizar esta elección, y es la facilidad para depurar el código. La limpieza, la cantidad de líneas y las herramientas de pruebas disponibles cumplemn un papel clave en este trabajo. Para valorarlo, he tenido en cuenta un artículo de Shah [Sha19] que hace la comparativa desde el punto de vista de la calidad de código.

1. Tipado: Sabemos que Javascript no es un lenguaje tipado. Sin embargo, cuando hablamos de pruebas, es importante comprobar el tipo de los datos que fluyen a través de la aplicación. Tanto React como Vue permiten comprobaciones de tipos mediante Javascript tipado: TypeScript es una solución global al problema. Sin embargo, es más sencillo usar TypeScript en React. Además, React dispone de una herramienta oficial, Flow, que permite hacer estas comprobaciones de forma todavía más fácil.
2. Modularidad: Ambos entornos están basados en modularidad de código. Todo depende de lo bien diseñada que esté la aplicación.
3. Curva de aprendizaje: Vue es líder en este aspecto. Uno de los problemas de React es que su curva de aprendizaje es menos suave que el de otros entornos. Vue está hecho para ser dominado en poco tiempo.
4. Pruebas y depuración: React es líder en este aspecto. Hay varias herramientas de pruebas, siendo Jest la recomendada por Facebook. Además, React cuenta con una extensión Chrome para depurar sus componentes de forma rápida y sencilla. Aun así, Vue también tiene herramientas de pruebas.
5. Server-Side Rendering (SSR): Es importante mejorar la velocidad de descarga y potenciar el Search Engine Optimization (SEO). Para eso, se suele utilizar la técnica de SSR. Se puede leer más sobre este tema en el artículo de Singh [Sin19]. Para este aspecto, ambos entornos tienen sus herramientas. React tiene ahora NextJS y para Vue existe Nuxt.js.
6. Mantenibilidad del código: Este es uno de los aspectos más importantes, dado que la idea es que el entorno sirva para proyectos pequeños con potencial de crecimiento. El entorno pretende hacer que un proyecto pequeño sea mantenible de forma sencilla, así que este punto es clave. Para este apartado, Santor [San18] explica en su artículo que, en sus años de experiencia, React es más mantenible y que leer código de React que han escrito otros es más sencillo. Esto no deja de ser un punto opinable, pero dentro de lo opinable que es, la opinión más extendida aboga para React en aplicaciones más grandes.

Así que, pese a que Vue ha estado creciendo y parece el líder en rendimiento, React parece una elección muy sólida y equilibrada que va a llegar a más desarrolladores y realizar aplicaciones más mantenibles durante el 2020. Por tanto, he decidido realizar el marco de trabajo en React. Si el framework tiene éxito, se planteará extenderlo a otros entornos de trabajo.

2.2 Entornos de trabajo del lado de servidor

En esta sección se van a evaluar los entornos de desarrollo más populares en el lado del servidor teniendo en cuenta que se va a utilizar React. Si siguiésemos el mismo criterio que con el apartado del lado de cliente, encontraríamos que numerosas fuentes -por ejemplo, [Bui20] y [Sim20]- están de acuerdo en que PHP es la tecnología más utilizada en el lado de servidor con diferencia. Se

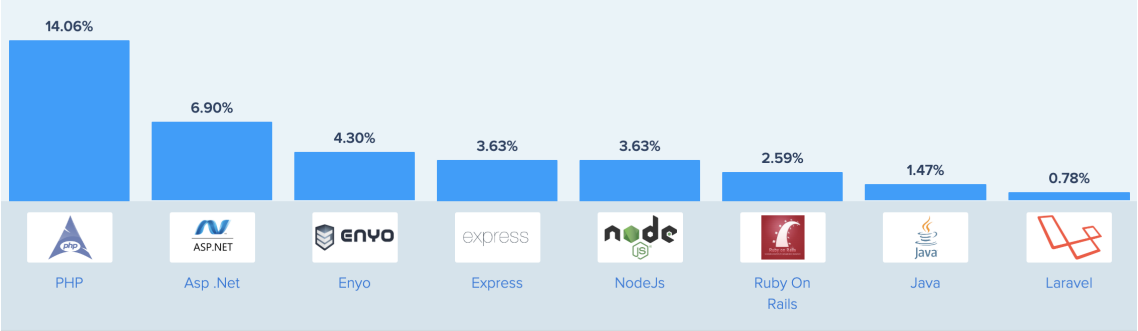

Figure 2.2: Leading Framework technologies share on the web - Top 10K Sites

puede apreciar en la figura Figure 2.2 sacada de [Sim20] en Febrero de 2020. Sin embargo, en este caso particular y por la tecnología elegida en el lado del cliente, se van a valorar positivamente tecnologías que permitan desarrollar al mismo tiempo el cliente y el servidor. PHP es muy utilizado por varias razones:

1. Es el más antiguo de los lenguajes de lado de servidor.
2. Tiene la mayor comunidad que puede tener un lenguaje de servidor y, por tanto, una extensa documentación.
3. Es utilizado por marcos aun más grandes, como Wordpress. Estos son Content Management System (CMS) y están fuera del alcance de este proyecto.

Sin embargo, PHP está pensado para directamente renderizar HTML, que es exactamente lo que hace React. Pese a que queremos dar soporte a las tecnologías más extendidas, también queremos estar actualizados con pilas tecnológicas que se están utilizando hoy para comenzar nuevos proyectos. Una de estas pilas tecnológicas de las que hablo es MERN (MongoDB, ExpressJS, ReactJS y NodeJS). Esta pila tecnológicas está recomendada por los siguientes motivos:

1. Solo hay que aprender un lenguaje, Javascript, lo cual reduce la curva de aprendizaje.
2. La gestión de datos internos de la aplicación es muy similar a la gestión de base de datos porque MongoDB trabaja con objetos Javascript.
3. Se pueden hacer contenedores Docker que contengan la pila completa de forma muy sencilla, distribuyendo la carga y haciendo la aplicación escalable.

Para esto se van a evaluar dos opciones: Express y Next.js. Se ha considerado utilizar múltiples entornos de servidor, entre ellos los que podemos encontrar en la figura Figure 2.3. Sin embargo, teniendo decidido el lado del cliente, la mayoría de opciones no merecen la pena. React trabaja de forma estupenda con NodeJS y lo más habitual es que vayan de la mano. Además, como se ha comentado anteriormente, así se da soporte a la pila tecnológica que más está de moda. Entre Express y Next.js, la competición reside entre una librería ligera de desarrollo de APIs contra un entorno completo de SSR. Durante el desarrollo del entorno se va a valorar abrazar ambas dos opciones como una configuración extra.

Figure 2.3: 2019 - Opinión popular de los entornos de trabajo back-end

2.3 Control de versiones

El control de versiones consiste en tener todas las versiones de código que se han generado etiquetadas, de forma que se pueda acceder a cualquier versión generada en cualquier momento. Esta es una base imprescindible para un proyecto de calidad, ya que permite trazar errores con mucha facilidad y es una salvaguarda ante fallos. Además, el control de versiones permite tener varias ramas de trabajo, separando el código de la rama de producción, la rama de desarrollo y las distintas ramas de los equipos y características que se están trabjando.

Hay muchas herramientas que permiten realizar este control de versiones y está sujeto a opinión cuál es mejor. Lo que no está sujeto a opinión es cuál se está utilizando más, lo cual se puede ver en gráficas como la figura Figure 2.4, que nos facilita Black Duck [Bla20]. Aparentemente el 70% de los repositorios públicos está controlado mediante git.

Otra ventaja de git es que es considerada la herramienta de alto rendimiento. Esto se puede ver también en la figura Figure 2.5. El origen de esta figura, G2 [G220], nos permite además ir probando distintos parámetros, como el tamaño de la empresa. Y nos muestra que, en cualquier caso, git es la herramienta a elegir para el alto rendimiento.

Así que, con respecto al control de versiones, no hay duda de que git es la herramienta a elegir. Nos permite abarcar la mayoría de desarrollos públicos y es perfecto para el desarrollo de alto rendimiento que necesita un proyecto de calidad. Sin embargo, las decisiones no han acabado aquí. El control de versiones suele subirse a un servidor de repositorios -especialmente si pretende ser código libre-. Este proyecto va a ser código libre, de modo que se puedan proponer cambios y ramificar según distintas necesidades. Para esto, Garbade [Gar18] explica de forma muy completa que, para proyectos de código abierto, Github es el que tiene mayor comunidad, así que va a tener com más facilidad una repercursión real.

Por tanto, han sido seleccionados git y Github para el control de versiones.

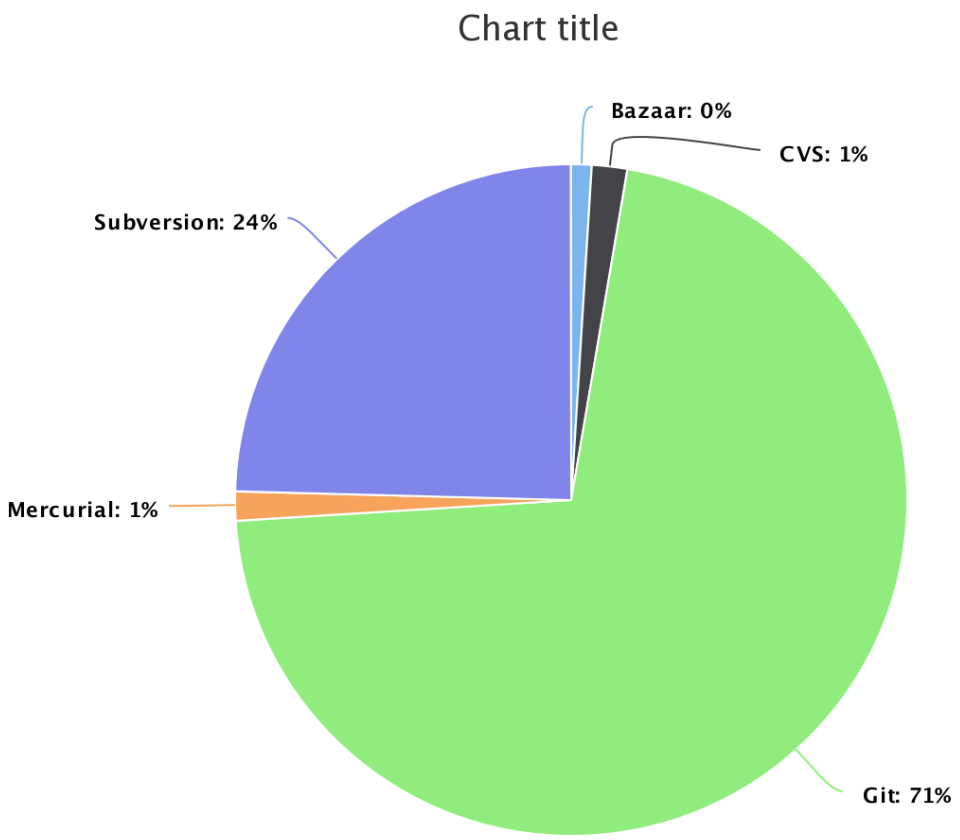

Figure 2.4: 2019 - Compare Repositories

2.4 Gestión de paquetes

La gestión de paquetes consiste en el conjunto de herramientas que permiten descargar y mantener las dependencias que el proyecto va a utilizar. Como se ha decidido utilizar React + Express, el entorno de desarrollo es todo Javascript y, por tanto, en servidor se va a utilizar NodeJS, que tiene su propio gestor de paquetes: NPM.

Aunque NPM es una herramienta muy útil en su campo, no es la única que se ha creado para este propósito. Y hoy por hoy, la herramienta competidora por excelencia es Yarn. Como bien explica Kryukov [Kry19], existe una diferencia de velocidad a favor de Yarn (ver figura Figure 2.6). Sin embargo, Kryukov [Kry19] también nos dice que esa diferencia de velocidad no es grande, así que no debería ser el motivo para elegir un gestor u otro. La funcionalidad de ambos gestores, pese a ser similar, tiene sus diferencias. Además, Yarn sacrifica espacio en disco para ganar ese extra de velocidad que NPM no tiene.

Como en este punto la decisión es tremendamente subjetiva y el coste de implementar ambos es pequeño, para este proyecto me voy a tomar la molestia de contemplar ambas opciones a la vez. Esto quiere decir que, en la configuración inicial del entorno, se presupondrá NPM (que es el gestor por defecto de Node), pero se permitirá al usuario cambiar a Yarn de forma sencilla y rápida.

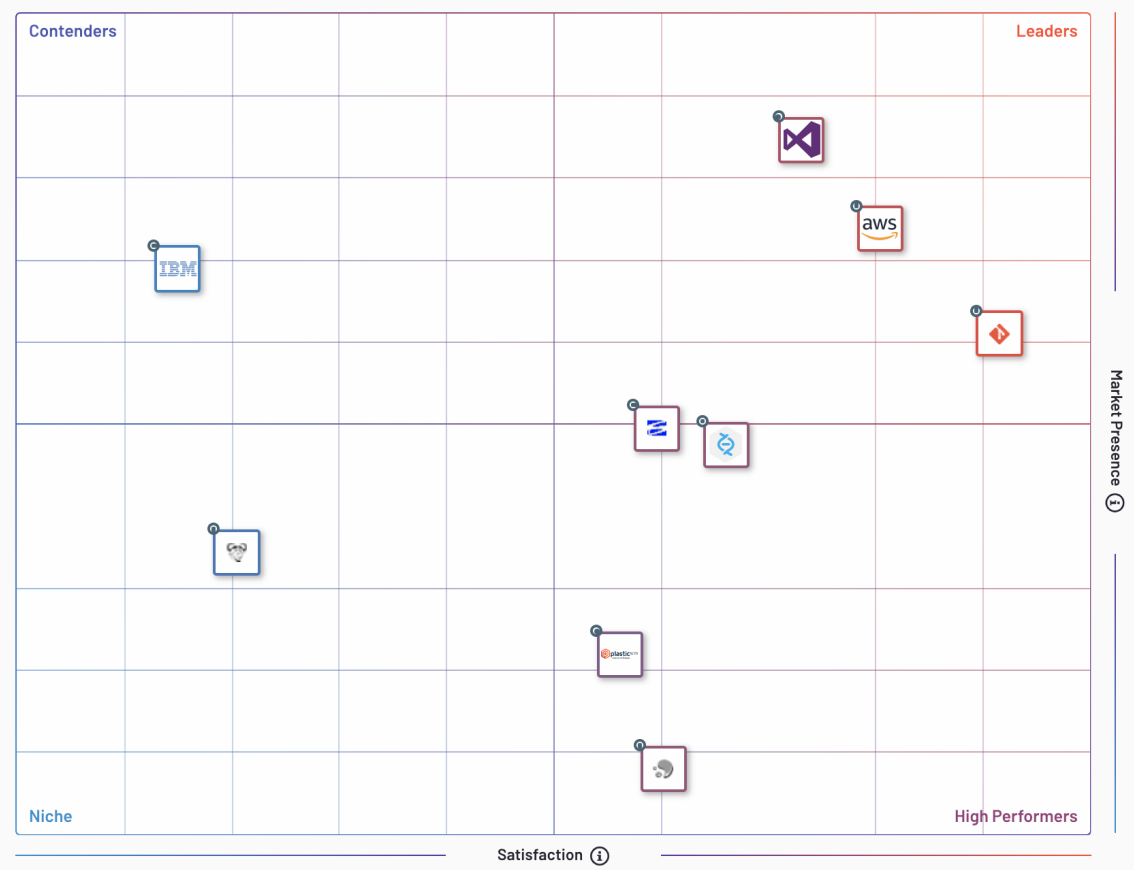

Figure 2.5: 2019 - Best Version Control Systems

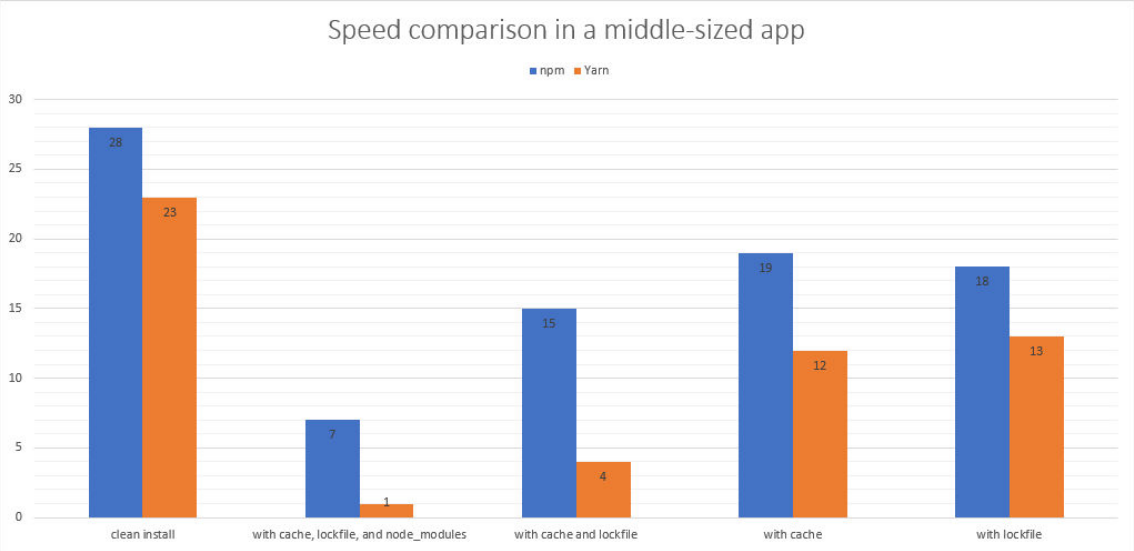

Figure 2.6: NPM vs Yarn. Speed comparison in a middle-sized app

2.5 Pruebas unitarias

Las pruebas son una pieza esencial del código de calidad. Sirven para tener la certeza de que al usuario le llega siempre una versión de la aplicación que funciona como se espera. Hay tres tipos de pruebas: las unitarias, las de integración y las funcionales.

En concreto, las pruebas unitarias se encargan de comprobar que un componente de la aplicación funciona como se espera, independientemente del resto de componentes. Si se desea conocer más sobre las pruebas unitarias, recomiendo conocer la historia de Dascalu [Das18].

En cualquier caso, cualquier aplicación con potencial de crecimiento necesita pruebas automáticas. En el mercado hay una infinidad de herramientas que nos resuelven este problema. Aunque en el caso de React, solo hay unas pocas que estén recomendadas oficialmente por el equipo de React. Como se especifica en la documentación ([Rea18]), se recomienda utilizar Jest junto con una de las siguientes librerías: React Testing Library o Enzyme. Jest parece la opción clara, dado que es la que utiliza Facebook (creadores de React), es la que más se utiliza en el mercado y es la que más satisfacción genera, como se puede comprobar en la figura Figure 2.7.

El dilema surge entre Enzyme y React Testing Library. ¿Qué diferencia hay entre ellos? Como primera respuesta, recomiendo leer la que ha dado Polvara [Pol19] en Stack Overflow. Básicamente, Enzyme nos permite acceder a los métodos de nuestros componentes y React Testing Library siempre simula el componente completo. Además, Enzyme permite algo llamado Shallow Render. Esto significa dibujar únicamente el componente que se desea probar, simulando sus hijos. Enzyme es más fácil de utilizar, pero más peligroso y, a la larga, menos mantenible. El hecho de que React Testing Library nos obligue a utilizar los componentes tal y como se renderizan en un ambiente de producción implica tener más seguridad en nuestro código, tal y como explica Dodds [Dod18] en su artículo y, posteriormente, revisa Tucker [Tuc19] en 2019. La filosofía es sencilla: es mejor hacer pruebas que den la confianza de que el código funciona y no emularlo.

Por estos motivos, las pruebas unitarias van a ser mediante Jest y React Testing Library, que son las herramientas recomendadas oficialmente. Durante el transcurso de este trabajo, se valorará utilizar Enzyme haciendo comprobaciones sobre la curva de aprendizaje y la velocidad en proyectos más pequeños, pudiendo ser implementado como opción. Sin embargo, como este entorno pretende ser útil en proyectos con potencial de crecimiento, se preferirá el uso de React Testing Library por su mejor mantenibilidad.

2.6 Pruebas de integración

Las pruebas de integración son aquellas pruebas que permiten comprobar de forma automática que un componente de la aplicación está funcionando bien cuando se utiliza junto con otro componente. Es posible que dos componentes funcionen correctamente cuando están aislados, pero en el momento en el que se unen ocurre funcionalidad inesperada. Las pruebas de integración permiten detectar este fenómeno para poder alertar al desarrollador. Si se desea conocer más sobre las pruebas de integración, recomiendo visitar DZone [DZo19].

Figure 2.7: 2019 - Opinión popular de las herramientas de pruebas unitarias

Durante las pruebas unitarias, hemos hablado de dos librerías de pruebas en React: Enzyme y React Testing Library. De hecho, gran parte de la discusión se ha centrado en un tema que, pese a que no tocaba en ese momento, era indivisible del resto de la comparación. El motivo por el que se ha elegido React Testing Library sobre Enzyme es porque es más preciso en las pruebas de integración. De hecho, Enzyme no realiza pruebas de integración dado que simula la generación de los hijos de un componente.

Así que en React las pruebas de integración y las de unidad están íntimamente relacionadas. Las unitarias se encargarían de comprobar la funcionalidad dentro de un componente React y las de integración, ver cómo se comportan los componentes hijos dentro de distintos componentes padre. En cualquier caso, tal y como nos dice Russom [Rus18], las librerías de pruebas unitarias y de integración son las mismas: Jest y React Testing Library.

2.7 Pruebas funcionales

Las pruebas funcionales o End To End (e2e), son aquellas pruebas que son totalmente agnósticas de la tecnología utilizada para desarrollar la aplicación. Se encargan de interactuar con ella como si fuesen el usuario final. En el caso de las aplicaciones web, se suelen utilizar navegadores de tipo Headless para realizar estas pruebas de forma automática. Estas pruebas son totalmente independientes de React y NodeJS, aunque lo ideal es desarrollarlas en el mismo lenguaje que el resto de pruebas que se realizan en la aplicación.

Para decidir qué herramienta de pruebas e2e se va a utilizar, se han considerado las citadas en el artículo [TFT19]. Dentro de estas, se han descartado todas ellas que no tienen compatibilidad con NodeJS para el desarrollo de los casos de prueba o aquellas que no tiene una versión gratuita. Así que se han tenido en cuenta únicamente Selenium y Crypress.

Tal y como explica Kinsbruner [Kin19], Selenium es una herramienta madura y estable, con mucha funcionalidad y que tiene configuración algo más pesada. Sin embargo, Selenium permite grabar las acciones que se realizan sobre el navegador y generar los casos de prueba automáticamente. Y esto es exactamente lo que interesa al marco de trabajo que se propone: ser de configuración rápida y fácil uso para una aplicación pequeña. De la configuración se encargará el propio marco, así que la capacidad de grabar el navegador es una gran ventaja con respecto a Cypress. Además, Selenium está preparado para trabajar con Jest.

Por otro lado, Cypress está específicamente diseñado para entornos de NodeJS y está creciendo a gran velocidad. Como sugiere Kinsbruner [Kin19], Cypress debería considerarse como una alternativa a futuro, que según vaya creciendo se puede ir incorporando a este proyecto. Hoy por hoy, Selenium es líder en herramientas de pruebas e2e y, con todo el apoyo que da a entornos de Javascript y Node, no hay ninguna otra que sea apropiada para la tarea.

2.8 Gestor de Base de Datos

Esta decisión es probablemente la más difícil de todas las que se han tomado en este proyecto. Por un lado, la gestión de bases de datos depende mucho de los conocimientos técnicos del desarrollador. Por otro, la base de datos puede acelerar mucho el tiempo de desarrollo en función de la afinidad que haya entre los datos y el gestor elegido. Además, hay gestores que son más escalables o tienen distintas prestaciones en función del resto de tecnologías elegidas.

Para tomar esta decisión, se han abordado los puntos con sumo cuidado, comenzando por un análisis de popularidad, tal y como se ha hecho en las anteriores secciones. Según el análisis de Chand [Cha19] (resumido en la figura Figure 2.8), los gestores de bases de datos más populares son:

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL
5. MongoDB

Las cuatro primeras opciones son todo bases de datos relacionales. En quinta posición se encuentra MongoDB, que es la única opción no relacional dentro del top de popularidad. Como uno de los objetivos de este marco es llegar al mayor número de desarrolladores posible, se ha intentado mantener el marco dentro de estos gestores de bases de datos. Además, otro factor a tener en consideración es que no se va a incluir ninguna herramienta que no sea gratuita, así que Oracle y Microsoft SQL Server quedan descartadas. Esto reduce las posibilidades a tres gestores de base de datos: MySQL, PostgreSQL y MongoDB.

MySQL y PostgreSQL son, como comenté, bases de datos relacionales que utilizan un lenguaje común: Structured Query Language (SQL). Además, hay otros muchos gestores que utilizan este lenguaje de consultas y cada usuario puede preferir uno u otro. Hay una discusión abierta sobre cuál es mejor y hay defensores y detractores de cada uno. Se puede leer un poco más de este tema en la reflexión de Hristozov [Hri19].

MongoDB es un gestor no relacional (en concreto, se le conoce como lenguaje No SQL). Esto quiere decir que no se basa en un modelo estricto. Específicamente, MongoDB es un Document-Oriented Database (DOD). Se puede leer más sobre este tipo de gestores en la propia documentación de MongoDB ([Mon16]). Lo que hace a MongoDB tan buen gestor de base de datos para entornos NodeJS es que esos documentos que gestiona se guardan en formato Binary JSON (BSON), que es una forma de guardar en disco de forma eficiente documentos JavaScript Object Notation (JSON). La forma de comunicarse con este gestor es enviando precisamente objetos con exactamente la misma sintaxis que los objetos en JavaScript. Así que es un lenguaje estupendo para trabajar desde entornos con JavaScript.

Por otro lado, hay que tener en cuenta los Object-Relational Mapping (ORM), que son herramientas que hacen de capa intermedia entre la aplicación y la base de datos. Permiten modelizar los datos como si fuesen objetos y se encarga de mantener la coherencia entre la base de datos y la estructura proporcionada. Realizar operaciones básicas es muy directo utilizando un ORM, pero aumentan la dificultad y reducen la eficiencia de operaciones complejas. Por tanto, utilizar o no un ORM depende mucho de las necesidades de la aplicación. Se puede leer más sobre los criterios de utilización de un ORM en los artículos de Rogers [Rog19] o Sasidharan [Sas16]. Sin embargo, los ORM permiten utilizar la misma sintaxis para comunicarse con bases de datos diferentes, lo cual es algo que beneficia mucho a un marco de lanzamiento rápido como se está desarrollando. Si buscásemos a un ORM adecuado, TypeORM podría ser un ORM adecuado porque tiene soporte a MySQL, PostgreSQL y MongoDB a la vez.

Como se puede comprobar, cada tecnología en este apartado tiene sus ventajas e inconvenientes. Modelos relacionales permiten una estructura consistente en base de datos, mientras que MongoDB permite introducir en la base de datos objetos que tiene guardados en su memoria interna sin ninguna transformación. Por otro lado, el modelo puede requerir otros tipos de gestores. Los datos podrían requerir otros modelos (ver artículo de Panwar [Pan20]) o los desarrolladores podrían estar más familiarizados con algún otro en particular. Es importante recalcar que no debería ser tarea del marco elegir la tecnología que debería utilizar el usuario para la permanencia de información. Sin embargo, sí que es tarea del marco reducir al mínimo el tiempo que emplea un usuario en la configuración inicial del entorno. El marco está principalmente orientado a la calidad de código. Esto puede conseguirse independientemente del gestor de base de datos elegido.

Por todo esto, se ha decidido que el marco va a dar soporte a una configuración rápida para un entorno con MySQL, PostgreSQL, MongoDB o TypeORM. Será una opción, se gestionará localmente en la máquina que está preparada para alojar el proyecto y siempre será un elemento de libre configuración. De este modo, si la aplicación no necesita permanencia de datos o necesita una más especializada, el marco no pondrá límites en este aspecto.

2.9 Capa de datos

La persistencia de datos es muy importante de cara a mantener una aplicación web. Sin embargo, en los marcos de trabajo actuales, es habitual descargar mucha carga de datos en el cliente y que este lo vaya administrando. Esta gestión puede ser muy tediosa si no se utilizan unas tecnologías llamadas capas de datos, que permiten mantener la coherencia de los datos en toda la aplicación. Las herramientas más populares se muestran en la Figura Figure 2.9.

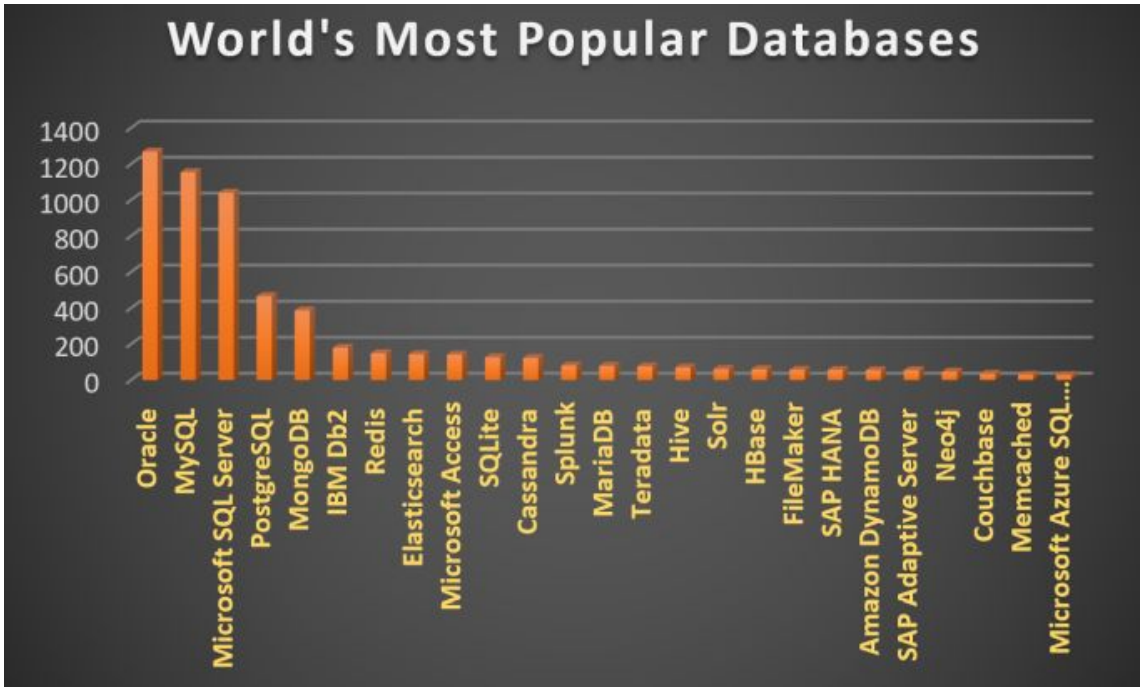

Figure 2.8: 2019 - Most Popular Databases In The World

Se puede apreciar que GraphQL es líder en este sector, y no por pocos motivos. Esta herramienta fue creada por Facebook (al igual que React y su principal competidora, Redux) y ha nacido a raíz de algunos problemas que surgían al utilizar React. Sin embargo, no está diseñada exclusivamente para React. GraphQL se puede utilizar junto con cualquier entorno. Dado que esta herramienta es la más popular en el mercado, está diseñada por los creadores de React y resuelve los problemas que explica Scott [Sco18] en su artículo, es la herramienta elegida para el marco que se ha desarrollado.

Sin embargo, no siempre se necesita capa de datos en el cliente y, por tanto, la posibilidad de utilizar GraphQL será una configuración opcional. En la configuración inicial, al desarrollador se le preguntará si desea capa de datos (GraphQL) en su entorno y este podrá sencillamente contestar si lo desea o prefiere desarrollar su producto controlando la capa de datos de forma personalizada, ya sea mediante otra herramienta o cuidando el estado y las propiedades de sus componentes de React.

2.10 Flujo de despliegue - Integración continua

La integración contínua es una disciplina que permite automatizar el flujo de código desde el momento en que un desarrollador lo genera hasta que llega a la rama principal de código del repositorio, asegurando por el camino que es código de calidad mediante pruebas automáticas mencionadas en las secciones Pruebas unitarias, Pruebas de integración y Pruebas funcionales. Además, uniéndolo con el despliegue contínuo, se puede lograr que haciendo solo un click el código sea verificado, enviado a un servidor y expuesto al público. Django Stars [Dja17] explica este procues en su artículo, mostrando además dos gráficos muy representativos de cómo funcionan la integración contínua (Figure 2.10) y el despliegue contínuo (Figure 2.11).

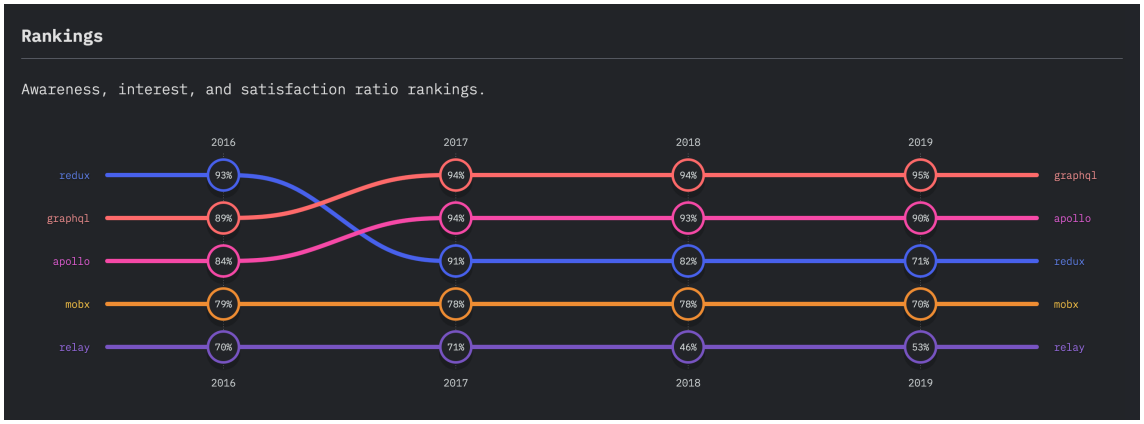

Figure 2.9: 2019 - Opinión popular de las herramientas de gestión de la capa de datos

En la figura Figure 2.10 se puede observar cómo múltiples desarrolladores pueden colaborar simultáneamente enviando sus aportaciones a un servidor de control de versiones como GitHub. Mediante un simple evento PUSH en una rama de desarrollo, los cambios irán al repositorio, que emitirá la nueva versión a un servidor de integración contínua. El servidor verificará que el código cubre todos los casos de prueba y, en caso de fracasar, notificará a los desarrolladores involucrados en la nueva versión, que podrán arreglar los fallos y volver a intentar una subida. En caso de éxito, podrá notificar a los desarrolladores también. Sin embargo, lo más habitual es que un éxito desencadene el siguiente proceso, el despliegue contínuo, que se encargará de subir la nueva versión a un servidor público de forma inmediata. Cabe remarcar que este servidor público puede estar duplicado (uno para desarrollo y otro para producción), generando dos procesos de integración contínua que pueden estar tanto encadenados como construidos de forma independiente.

En la figura Figure 2.11 se muestra el proceso completo, incluyendo la integración contínua y el despliegue contínuo. En este flujo pueden intervenir revisiones manuales de código y pruebas por parte de los usuarios. Este proceso asume que la aplicación es desarrollada en un entorno profesional, pero el marco que se plantea desarrollar no está pensado en un ámbito de empresa. El marco pretende abarcar proyectos pequeños (de pocos desarrolladores) con potencial de crecer. Por tanto, este flujo debe tenerse en consideración, pero no debe ser la principal preocupación del marco.

La idea es que el marco proporcione de forma sencilla tanto un entorno de integración contínua como un entorno de despliegue contínuo. Es objeto de estudio de este marco analizar las distintas herramientas que permiten hacer estas tareas y pensar en cómo podría un proyecto escalar a ámbito profesional sin modificar las herramientas que se utilizan en el proceso. Para poder realizar estas tareas, se han analizado las dos herramientas gratuitas que propone Django Stars [Dja17]: Jenkins y CircleCI. TravisCI se descarta por su carácter de pago, que rompe con la filosofía del marco.

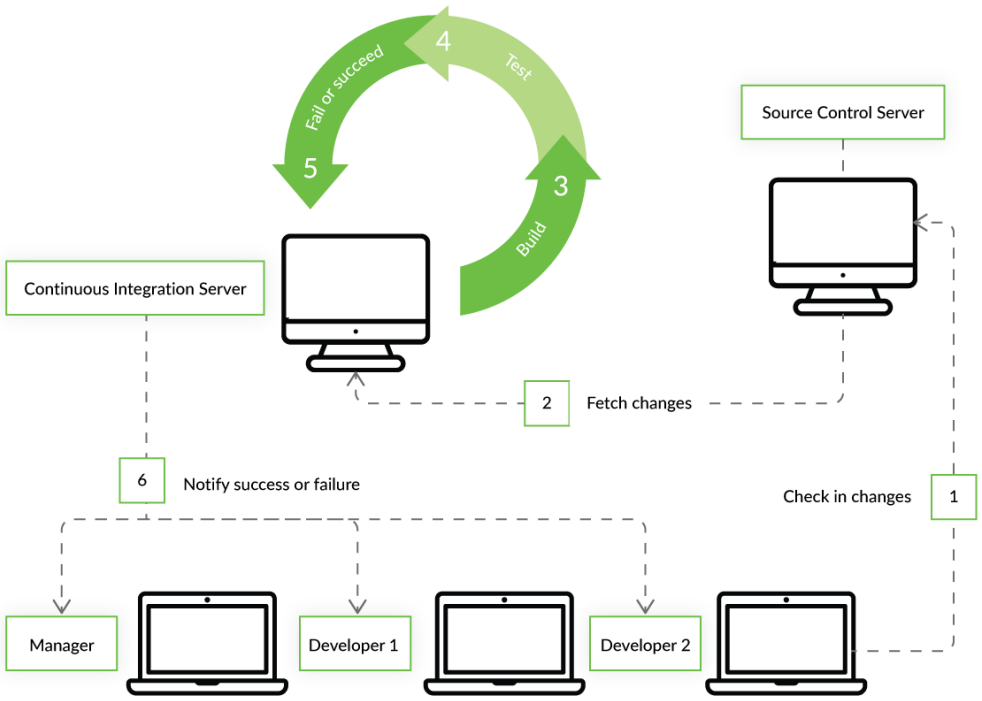

Figure 2.10: How Continuous Integration Works

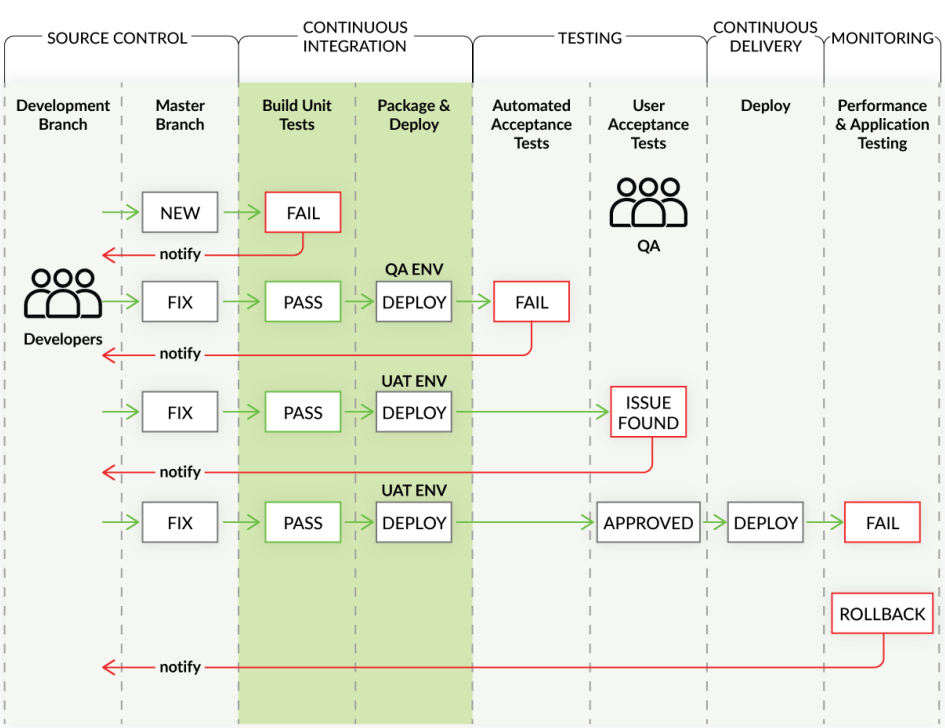

Figure 2.11: Release Workflow

3 Conclusiones

Bibliography

[Bla20] Black Duck. *Compare Repositories*. 2020. URL: https://www.openhub.net/repositories/compare (cit. on p. 16).

[Bui20] BuiltWith. *Framework Usage Distribution in the Top 1 Million Sites*. 2020. URL: https://trends.builtwith.com/framework (cit. on p. 14).

[Cha19] M. Chand. *Most Popular Databases In The World*. 2019. URL: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/ (cit. on p. 21).

[Das18] C. Dascalu. *Unit testing — why?* 2018. URL: https://medium.com/@corneliu/unit-testing-why-3490d08e89f2 (cit. on p. 19).

[Dja17] Django Stars. *Continuous Integration. CircleCI vs Travis CI vs Jenkins*. 2017. URL: https://djangostars.com/blog/continuous-integration-circleci-vs-travisci-vs-jenkins/ (cit. on pp. 23, 24).

[Dod18] K. C. Dodds. *Why I Never Use Shallow Rendering*. 2018. URL: https://kentcdodds.com/blog/why-i-never-use-shallow-rendering (cit. on p. 19).

[DZo19] DZone. *Integration Testing: What It Is and How to Do It Right*. 2019. URL: https://dzone.com/articles/integration-testing-what-it-is-and-how-to-do-it-ri (cit. on p. 19).

[G220] G2. *Best Version Control Systems*. 2020. URL: https://www.g2.com/categories/version-control-systems#grid (cit. on p. 16).

[Gar18] D. M. J. Garbade. *GitHub vs. GitLab — Which is Better for Open Source Projects?* 2018. URL: https://hackernoon.com/github-vs-gitlab-which-is-better-for-open-source-projects-31c45d464be0 (cit. on p. 16).

[Hri19] K. Hristozov. *MySQL vs PostgreSQL – Choose the Right Database for Your Project*. 2019. URL: https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres (cit. on p. 21).

[Kin19] E. Kinsbruner. *Cypress vs. Selenium: What's the Right Cross-Browser Testing Solution for You?* 2019. URL: https://www.perfecto.io/blog/cypress-vs-selenium-whats-right-cross-browser-testing-solution-you (cit. on p. 21).

[Kry19] D. Kryukov. *Yarn vs. npm in 2019: Choosing the Right Package Manager for the Job*. 2019. URL: https://blog.soshace.com/yarn-package-manager-in-2019-should-we-keep-on-comparing-yarn-with-npm/ (cit. on p. 17).

[Mon16] MongoDB. *What is a Document Database?* 2016. URL: https://www.mongodb.com/document-databases (cit. on p. 22).

[Pan20] A. Panwar. *Database Management Systems*. 2020. URL: https://www.c-sharpcorner.com/UploadFile/65fc13/types-of-database-management-systems/ (cit. on p. 22).

[Pol19] G. Polvara. *Difference between enzyme, ReactTestUtils and react-testing-library*. 2019. URL: https://stackoverflow.com/a/54152893/6282406 (cit. on p. 19).

[Rea18] React Team. *React Test Utilities*. 2018. URL: https://reactjs.org/docs/test-utils.html#overview (cit. on p. 19).

[Rog19] S. Rogers. *The pros and cons of Object Relational Mapping (ORM)*. 2019. URL: https://centralblue.co.uk/blog/2019/01/the-pros-and-cons-of-object-relational-mapping-orm (cit. on p. 22).

[Rus18] J. Russom. *Integration Testing in React*. 2018. URL: https://medium.com/expedia-group-tech/integration-testing-in-react-21f92a55a894 (cit. on p. 20).

[San18] R. Santor. *One Major Way React > VueJS*. 2018. URL: https://hackernoon.com/one-major-way-react-gt-vuejs-ba5c0332e75a (cit. on p. 14).

[Sas16] M. Sasidharan. *Should I Or Should I Not Use ORM ?* 2016. URL: https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce (cit. on p. 22).

[Sch19] J. Schae. *A RealWorld Comparison of Front-End Frameworks with Benchmarks*. 2019. URL: https://www.freecodecamp.org/news/a-realworld-comparison-of-front-end-frameworks-with-benchmarks-2019-update-4be0d3c78075 (cit. on p. 13).

[Sco18] J. R. Scott. *Goodbye Redux*. 2018. URL: https://hackernoon.com/goodbye-redux-26e6a27b3a0b (cit. on p. 23).

[Sha19] H. Shah. *React vs Vue – The CTOs guide to Choose the Right Framework*. 2019. URL: https://www.simform.com/react-vs-vue/#codequality (cit. on p. 14).

[Sim20] SimilarTech. *Top Framework Technologies*. 2020. URL: https://www.similartech.com/categories/framework (cit. on pp. 14, 15).

[Sin19] S. Singh. *The benefits and origins of Server Side Rendering*. 2019. URL: https://dev.to/sunnysingh/the-benefits-and-origins-of-server-side-rendering-4doh (cit. on p. 14).

[TFT19] TFT. *Top 10 Most Popular Software Testing Tools 2019*. 2019. URL: https://www.tftus.com/blog/top-most-popular-software-testing-tools-2019/ (cit. on p. 20).

[Tuc19] J. Tucker. *Revisiting React Testing in 2019*. 2019. URL: https://codeburst.io/revisiting-react-testing-in-2019-ee72bb5346f4 (cit. on p. 19).

[Tya19] N. Tyagi. *Top Front End JavaScript Frameworks*. 2019. URL: https://www.lambdatest.com/blog/top-javascript-frameworks-for-2019 (cit. on p. 13).

Todos los enlaces han sido comprobados el 22 de Febrero de 2020.