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

Contents

List of Figures

Acronyms

BSON Binary JSON. 26

CMS Content Management System. 19

DDNS Dynamic Domain Name System. 38

DOD Document-Oriented Database. 26

e2e End To End. 25

JSON JavaScript Object Notation. 26

ORM Object-Relational Mapping. 26

SEO Search Engine Optimization. 18

SQL Structured Query Language. 26

SSR Server-Side Rendering. 18

1 Introducción

La mayoría de las veces un proyecto comienza como una prueba de concepto. Escribiendo algunas líneas aquí y allá para ver finalmente si funciona, ver cómo la audiencia lo recibe y actuar según la retroalimentación. Esto permite un enfoque de bajo riesgo para un beneficio potencial. Sin embargo, este código generalmente se realiza a un ritmo rápido con poca o ninguna calidad. Podemos entender el código de calidad como cualquier código que siga las buenas prácticas, sea mantenible, se pruebe automáticamente y aproveche la integración y la implementación continuas. En resumen, es un código que es menos propenso a errores y se escala fácilmente.

Esta prueba de concepto puede tener éxito y si no se tiene en cuenta la calidad, también puede terminar en un código de crecimiento muy caro. La calidad no es opcional en productos medianos o grandes, ya que permite mantener un buen ritmo con las actualizaciones con la confianza de que casi no tendrá errores y será fácil de depurar.

La mayoría de las veces, establecer un buen entorno para el código de calidad significa duplicar o incluso triplicar el esfuerzo para productos pequeños, como pruebas de concepto. Por lo tanto, también es común ver que estos pequeños proyectos de investigación se creen sin buenas prácticas ni pruebas. Si el experimento es un fracaso, no pasa nada. Sin embargo, si el resultado es un éxito, es aún más costoso reconstruir la prueba de concepto en un producto escalable, comprobable y automatizado.

En este trabajo voy a tratar de aliviar un poco la carga para que la configuración de un entorno de código de calidad sea más fácil para un proyecto pequeño. El objetivo es crear un marco de trabajo que pueda aplicar automáticamente las buenas prácticas, crear un flujo de trabajo de integración continua y un esqueleto de pruebas para un proyecto para ayudar con este difícil proceso y hacer que los experimentos sean más baratos sin arriesgar la escalabilidad de nuestro exitoso experimento. Para demostrar que la producción de código de calidad vale la pena incluso en los proyectos más pequeños, voy a probar este nuevo marco en un ejemplo y medir los tiempos y los errores, para que podamos verificar como de costoso es codificar bien desde el principio.

2 Introduction

Most of the times a project starts as a proof of concept. Typing a few lines in here and there to finally see if it works, see how audience receives it and act on the feedback. This allows a low risk approach to a potential profit. However, this code is usually done in a fast pace with little or no quality at all. We can understand quality code as any code that follows good practices, is maintainable, automatically tested and takes advantage of continuous integration and deployment. To sum it up, is code that is less prone to errors and scales easily.

This proof of concept may succeed and if quality is not taken care of, it may also end in a very expensive code to scale. Quality is not accesory in medium or big sized products, as it allows to keep a good pace with updates with the confidence that it will have almost no bugs and will be easy to debug.

Most of the times, setting up a good environment for quality code means to double or even triple de effort for small product, such as proofs of concept. Thus, it is also common to see these small research projects to be created without good practices nor testing. If the experiment is a failure, nothing happens. However, if the result is a success, it is even more expensive to rebuild the proof of concept into a scalable, testable and automated product.

In this thesis I am going to try to lift the load a bit so setting up a quality code environment is easier for a small project. The aim is to build a framework that can automatically enforce good practices, create a continuous integration workflow and a testing skeleton for a project to help with this tough process and make experiments cheaper without risking the scalability of our successful experiment. To demonstrate that producing quality code is worth the effort even in the tiniest projects, I am going to test this new framework into an example and measure timings and errors, so we can check how expensive is to code well from the beginning.

3 Estado del arte

Existen numerosos marcos de trabajo para todo tipo de lenguajes de programación. La mayor parte de estos marcos producen esqueletos rápidos para ponerse a trabajar con un conjunto de librerías en poco tiempo. Todos ellos tienen su utilidad y forman parte el día a día de muchos desarrolladores.

En cuanto a calidad de código, existen también numerosas herramientas que facilitan las tareas. Existen analizadores estáticos de código, herramientas de integración contínua, marcos de trabajo para pruebas que incluyen el uso de imágenes instantáneas automáticas y existen generadores automáticos de código, que producen esqueletos de componentes específicos en el momento en que se necesitan.

Los desarrolladores, a lo largo de su carrera, van probando unas tecnologías y otras para ir decidiendo cuáles usar en un proyecto nuevo y, con práctica, hilar todas estas herramientas en un solo paquete se vuelve una tarea relativamente rápida. Sin embargo, cada proyecto tiene unas necesidades y no siempre sale barato el esfuerzo de unificar todas estas herramientas que nos facilitan la vida.

En este trabajo se ha dedicado tiempo a entender qué tecnologías se utilizan y de qué forma en el mundo del desarrollo web y, como sucede normalmente, existen todas las herramientas mencionadas por separado, pero no existe ningún marco que las unifique de forma que el código de lanzamiento que hay que escribir sea mínimo. Gracias al soporte de las herramientas a ficheros estáticos de configuración, con un simple catálogo de tecnologías, un usuario podría llegar a configurar en proyecto bajo un enfoque holístico sin escribir una sola línea de código. Ese marco de trabajo, esa herramienta de unificación, es precisamente el hueco en el mercado que voy a intentar explotar con este proyecto.

4 Estudio de mercado de las tecnologías disponibles

4.1 Introducción

El sistema que se plantea ha sido diseñado mediante la siguiente lista de componentes:

1. Tecnologías de lado de cliente
2. Tecnologías de lado de servidor
3. Control de versiones
4. Gestión de paquetes
5. Pruebas unitarias
6. Pruebas de integración
7. Pruebas funcionales
8. Gestor de Base de Datos
9. Capa de datos
10. Integración contínua

La figura Figure 4.1 muestra dónde está cada componente y cómo se relaciona con los demás. Por un lado, en la máquina de desarrollo, se podrían encontrar las tecnologías de lado de cliente y servidor (Aplicación y API), las tecnologías de base de datos y capa de datos y todas las tecnologías de pruebas. La única que quedaría fuera de la máquina de desarrollo es la integración contínua. Además, el entorno de desarrollo cuenta con tecnología de linting, que permite restringir las subidas de código siempre que este no cumpla normativas de limpieza y coherencia.

Una vez el código cumple los estándares propuestos por el linter y se sube a un repositorio mediante herramientas de control de versiones, se inicia un procedimiento automático descrito en el capítulo de Flujo de despliegue - Integración continua. En este procedimiento, se utiliza una entidad llamada servidor de integración contínua, que es la responsable de que el flujo se cumpla. En esta entidad vuelven a estar visibles todas tecnologías de pruebas. La máquina de desarrollo tiene acceso a las tecnologías de pruebas como herramienta de desarrollo. Sin embargo, la máquina de integración contínua tiene acceso a estas tecnologías como parte del flujo de integración contínua, y si estás pruebas no son satisfechas, el código no se puede publicar ni en la rama principal ni en el servidor de la aplicación.

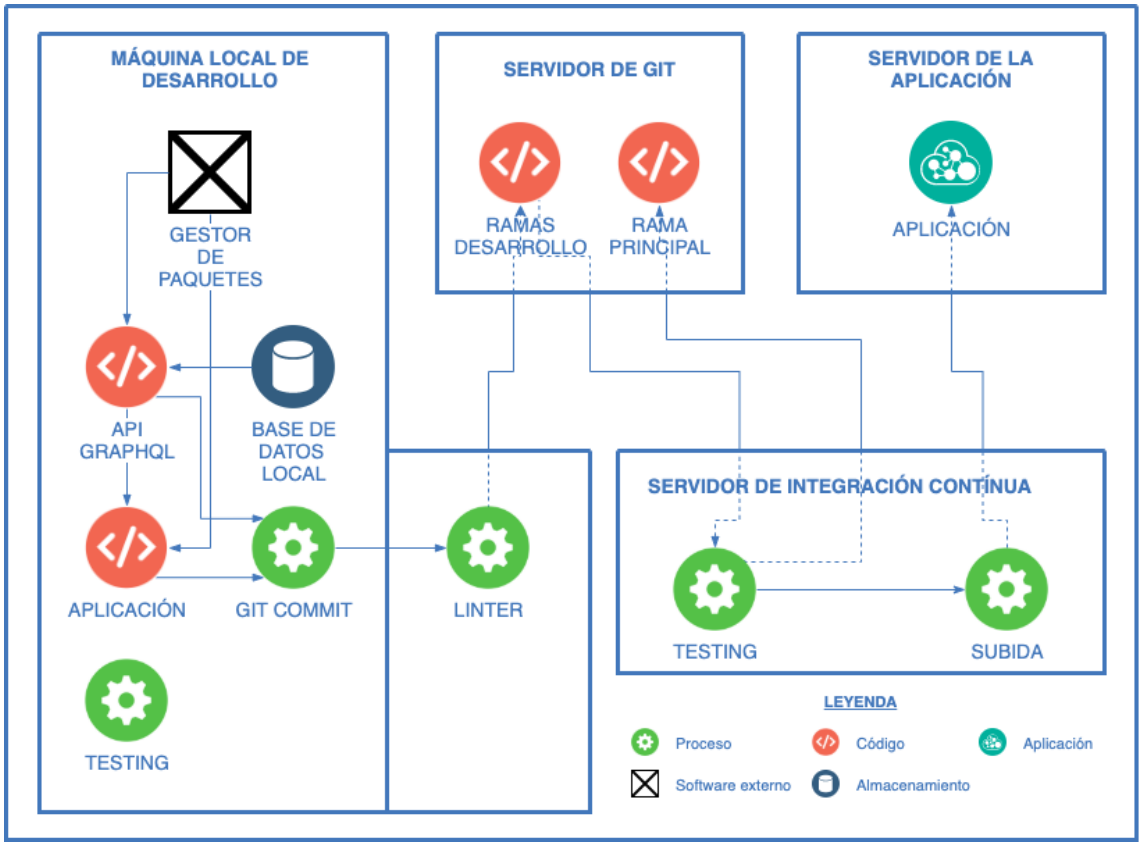

Figure 4.1: React Rocket Generator - Architecture

Por último, se encuentra el propio servidor de la aplicación, que contiene una versión comprimida del código, junto con la base de datos y la API. Este servidor es una entidad más compleja, dado que pueden ser varios servidores con un balanceador de carga. Es por eso que ese servidor no entra dentro del marco de desarrollo que se propone y se ha utilizado un icono distinto para representar toda esta complejidad. Cabe destacar que el servidor de integracón contínua y el de la aplicación pueden ser la misma máquina. Sin embargo, este no es el caso habitual y por eso se han marcado como entidades distintas.

Todo el marco está diseñado para que todo lo que hay en la máquina local sea automáticamente configurado mediante una serie de preguntas sencillas. Sin embargo, todo lo que corresponde a las otras máquinas pueden requerir configuración manual. El marco dispone de guías paso por paso para poder configurar todas las conexiones e integraciones descritas en la figura, así como los ficheros de configuración que alimentan el servidor de integración contínua. Todo esto queda explicado más adelante en detalle. Concretamente en el capítulo Desarrollo del marco de trabajo.

A lo largo de este capítulo se van a explicar en más detalle todas las tecnologías enumeradas y se va a desarrollar qué implementación o implementaciones van a recibir soporte directo en el marco propuesto, así como los motivos por los que se han seleccionado o descartado cada una de las implementaciones. Cabe destacar que este trabajo parte de la premisa de que el marco es solo una propuesta inicial y que, tal y como se explica en el capítulo Próximos pasos, es una propuesta preparada para su expansión.

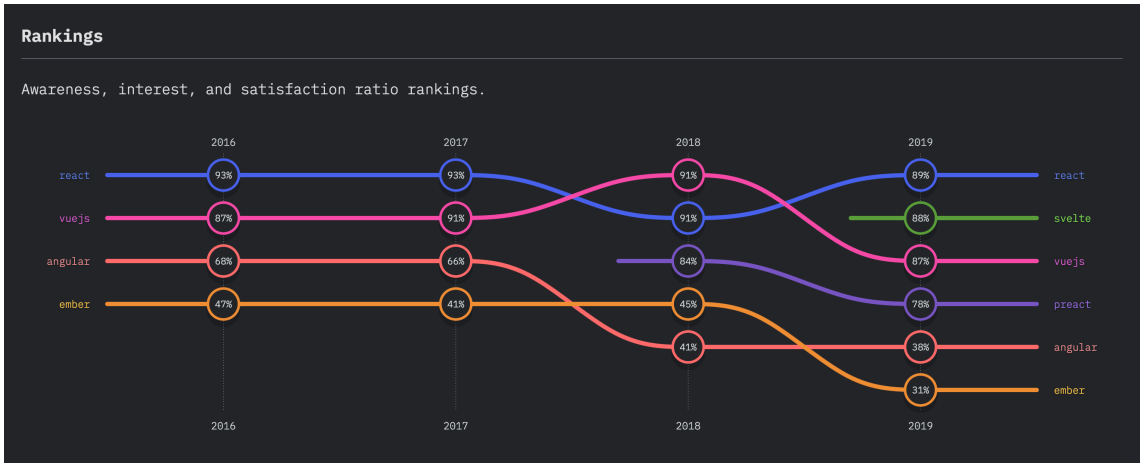

Figure 4.2: 2019 - Opinión popular de los entornos de trabajo front-end

4.2 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 4.2 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.

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.

Esto tiene repercusiones sobre el esto de toma de decisiones. La figura Figure 4.3 intenta ilustrar la cantidad de tecnologías que un desarrollador de React debería conocer para dominar todo su potencial. Aun así, esta figura no es más que un mapa conceptual, dado que durante el resto de capítulos se ha ido haciendo una investigación individual de cada tecnología. Aun así, el resultado de las tecnologías en el lado del cliente ha sido bastante parecida a lo que se puede observar en la imagen, así que la incluyo de referencia.

4.3 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 puede apreciar en la figura Figure 4.4 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:

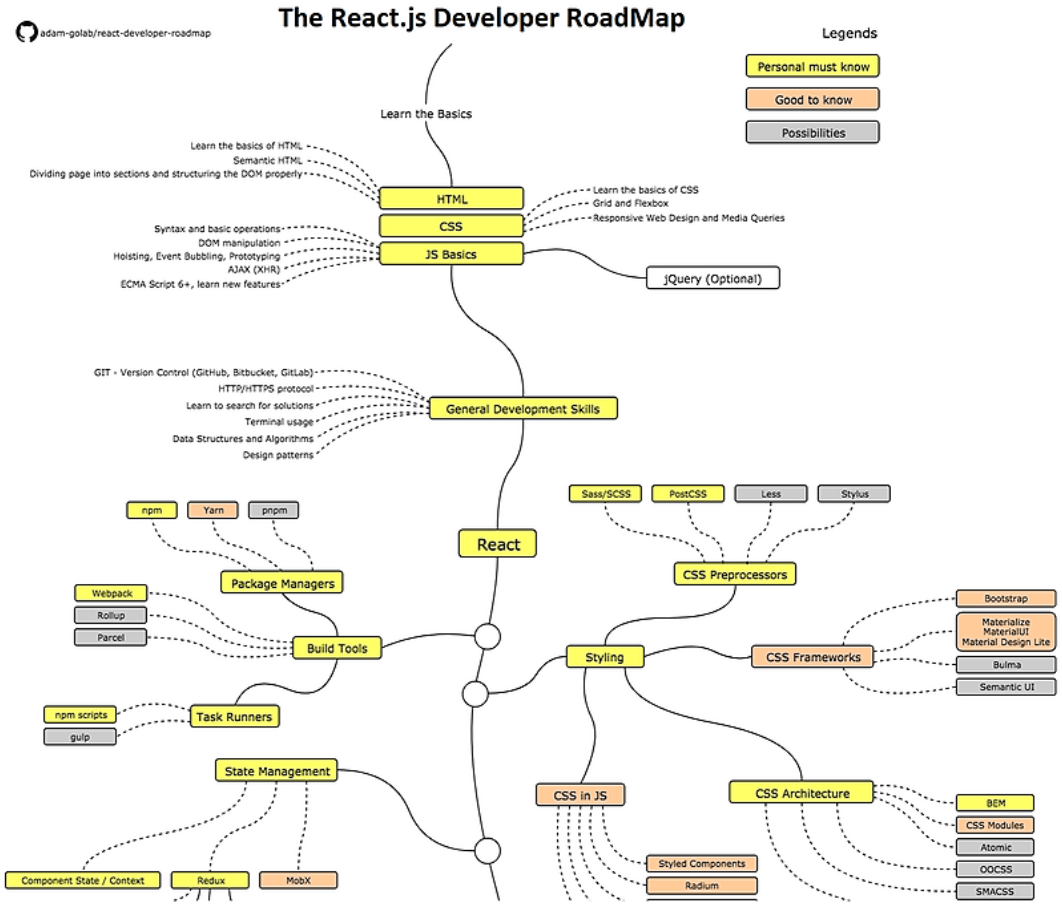

Figure 4.3: 2020 - El camino del desarrollador de React en 2020

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.

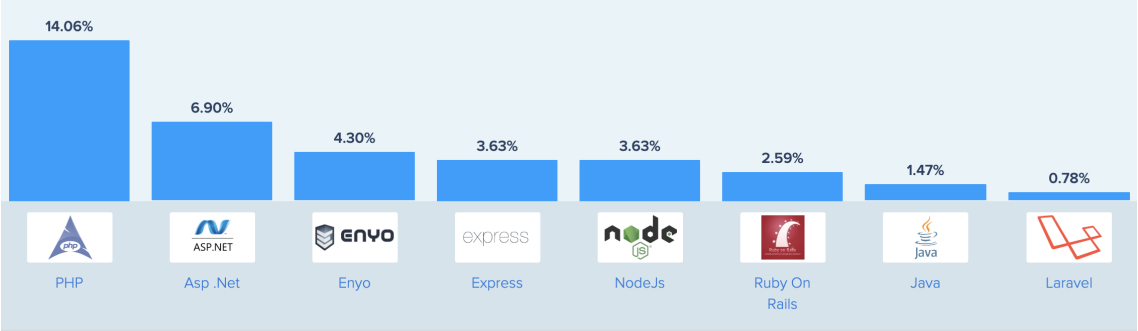

Figure 4.4: Leading Framework technologies share on the web - Top 10K Sites

Figure 4.5: 2019 - Opinión popular de los entornos de trabajo back-end

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 4.5. 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.

4.4 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 4.6, 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 4.7. 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.

4.5 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 4.8). 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.

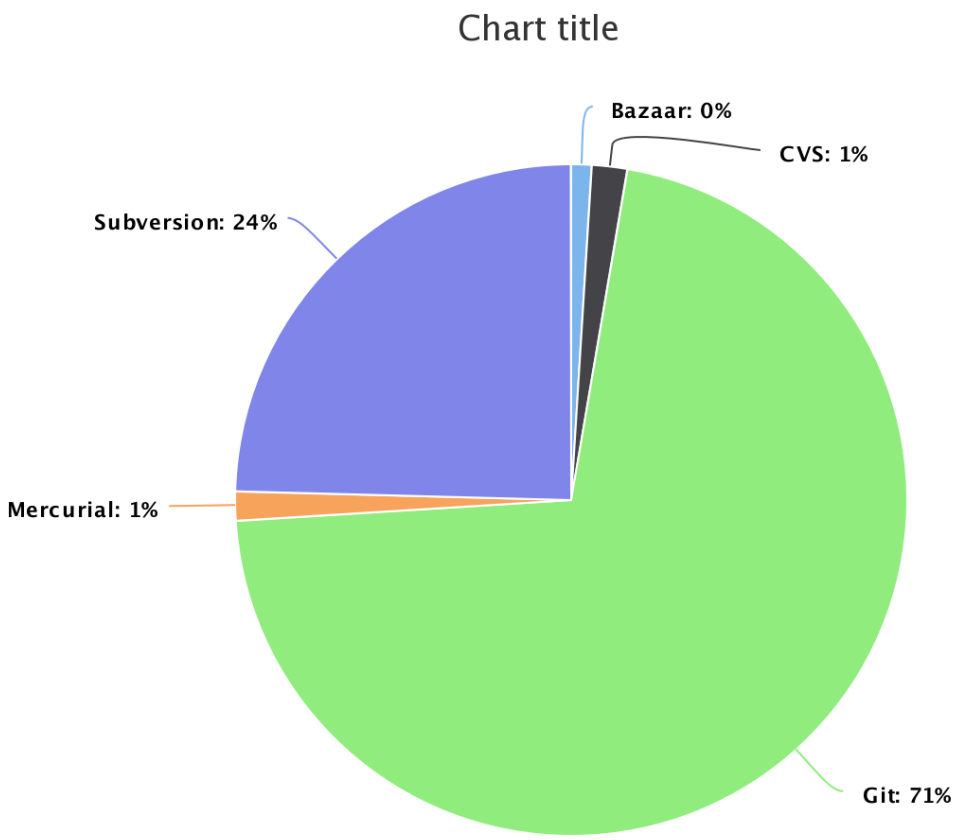

Figure 4.6: 2019 - Compare Repositories

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.

4.6 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.

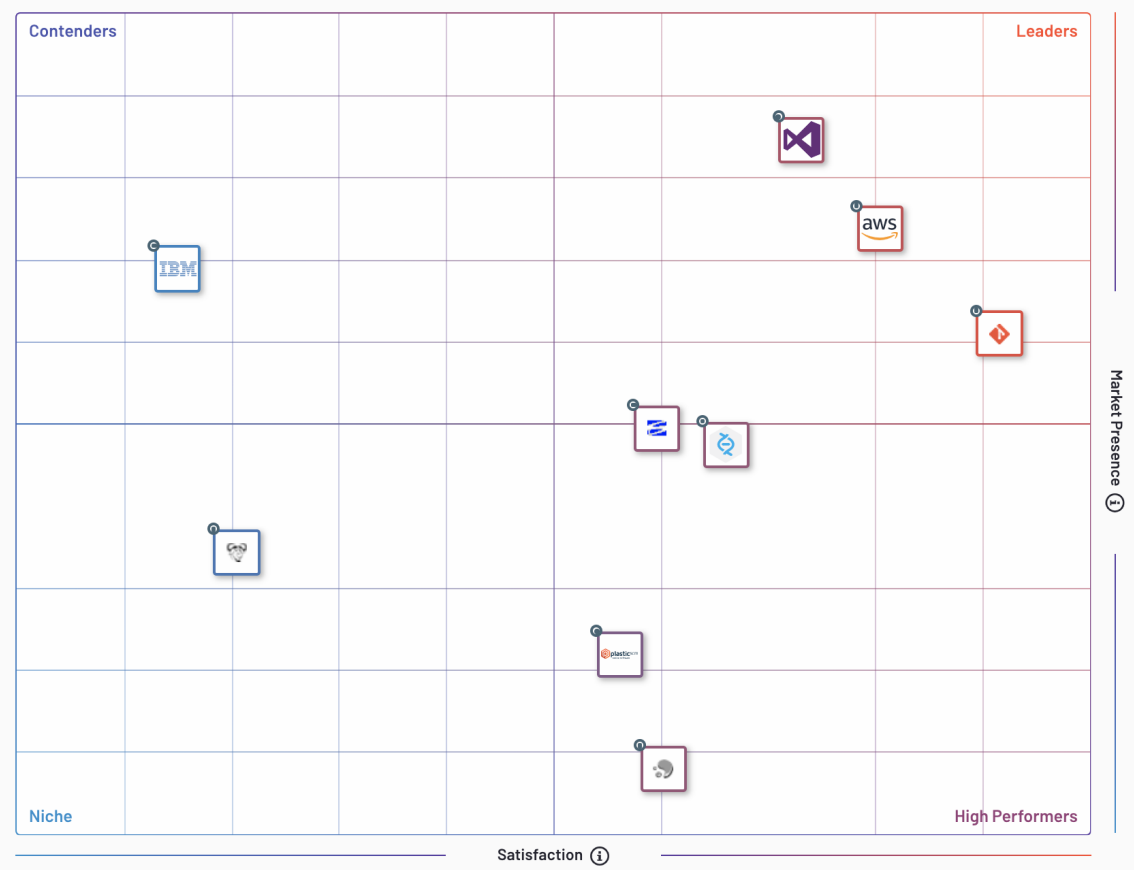

Figure 4.7: 2019 - Best Version Control Systems

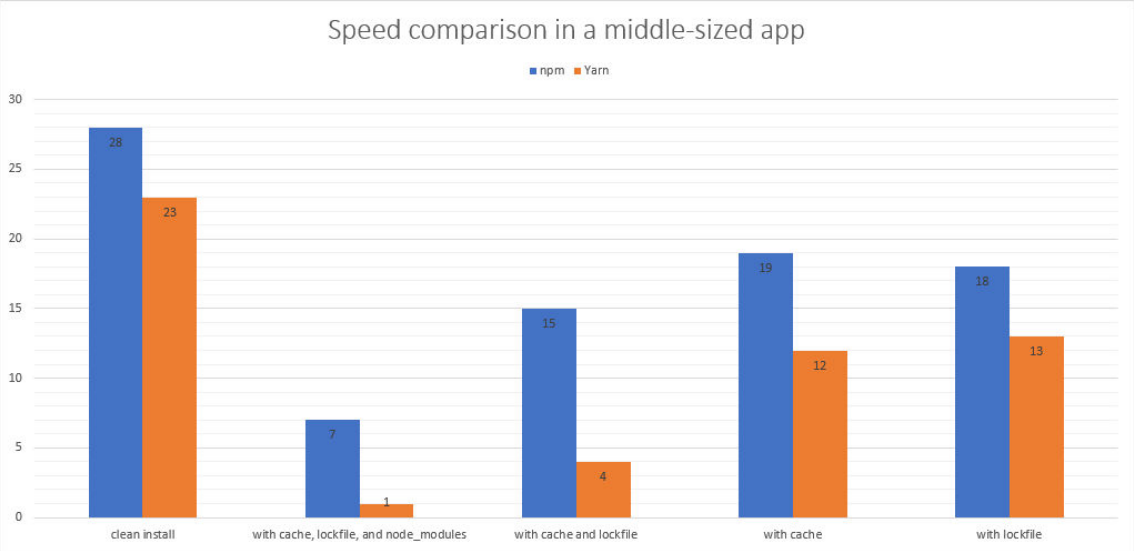

Figure 4.8: NPM vs Yarn. Speed comparison in a middle-sized app

Figure 4.9: 2019 - Opinión popular de las herramientas de pruebas unitarias

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 4.9.

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.

4.7 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].

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.

4.8 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.

4.9 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 4.10), 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

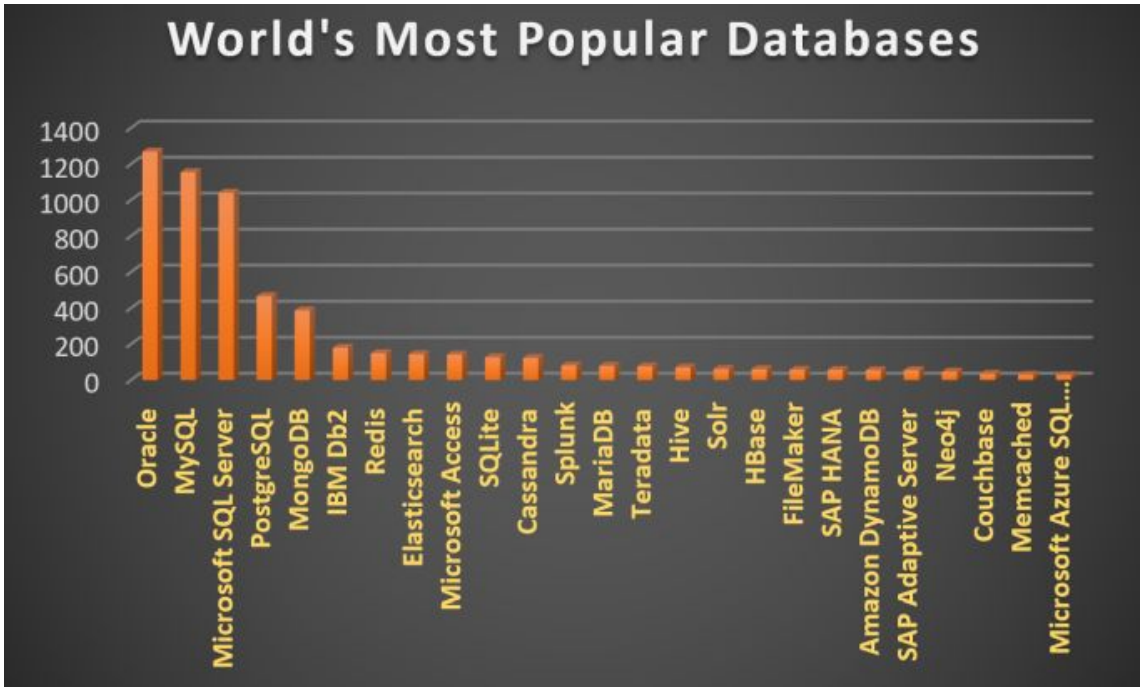

Figure 4.10: 2019 - Most Popular Databases In The World

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.

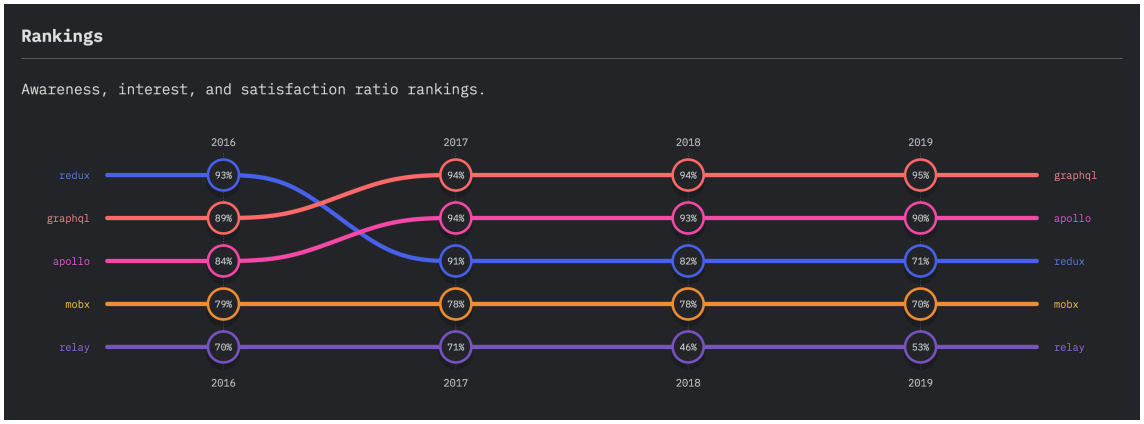

Figure 4.11: 2019 - Opinión popular de las herramientas de gestión de la capa de datos

4.10 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 4.11.

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.

4.11 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 4.12) y el despliegue contínuo (Figure 4.13).

En la figura Figure 4.12 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 4.13 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.

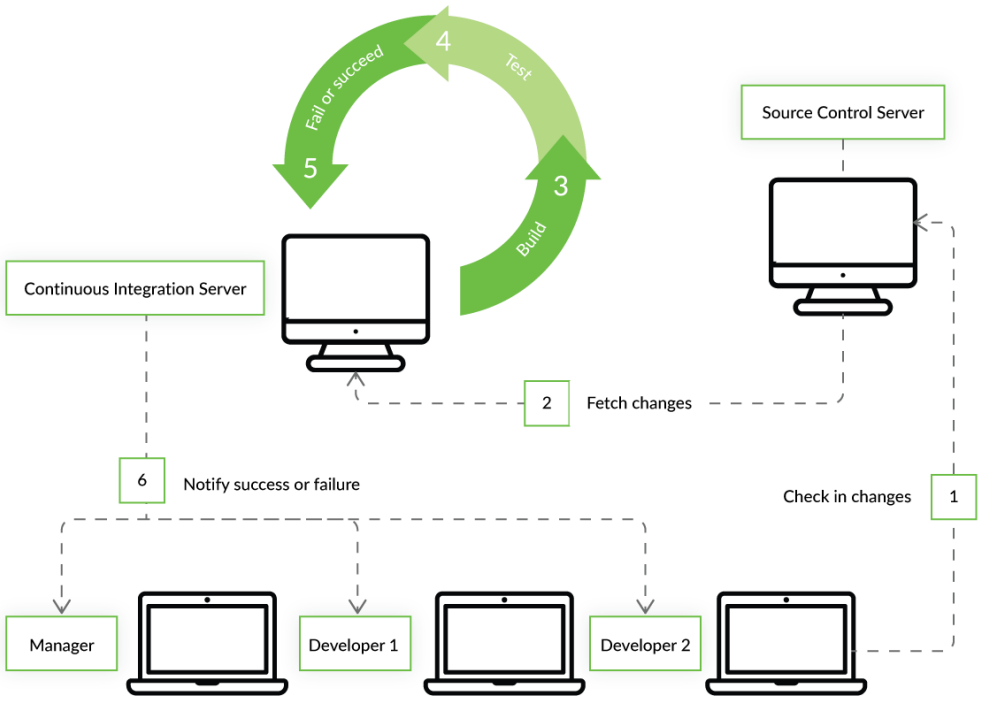

Figure 4.12: How Continuous Integration Works

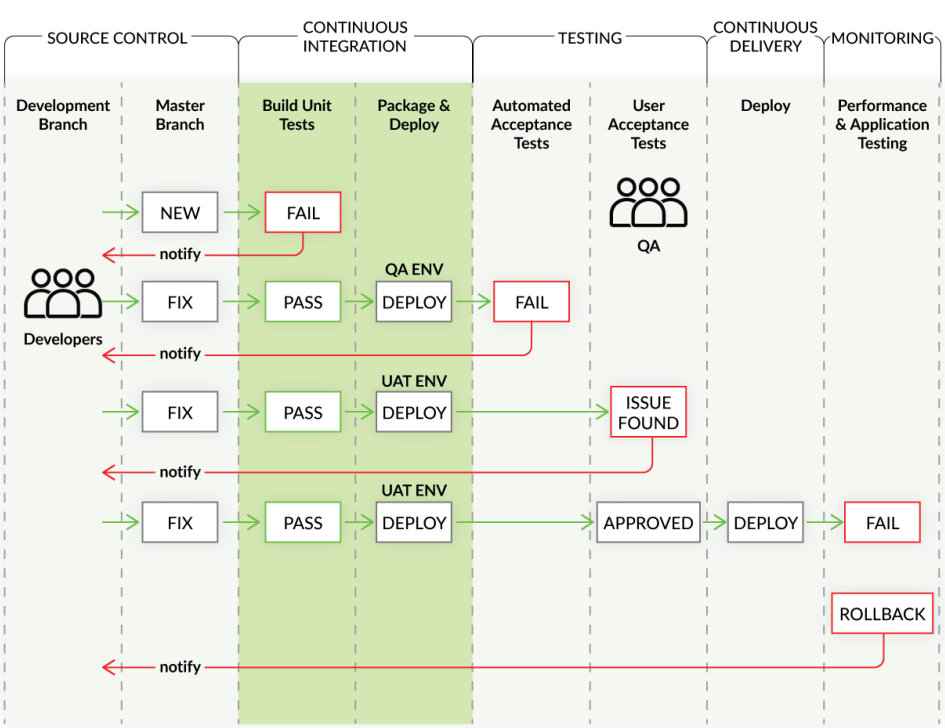

Figure 4.13: Release Workflow

5 Desarrollo del marco de trabajo

5.1 Funcionamiento general

El marco de trabajo ha sido diseñado para ser lo más automático posible. Para realizar todas estas automatizaciones, NodeJS dispone de un gestor de paquetes (que se ha discutido en Gestión de paquetes). Este gestor se encarga de tener controladas todas las dependencias del proyecto. Para ello, utiliza un fichero mantenido por el equipo de desarrollo llamado package.json. Además, se genera otro fichero package-lock.json a partir del primero, que contiene las versiones exactas del proyecto.

Este fichero package.json tiene un valor muy especial para los desarrolladores de NodeJS porque además contiene otros campos con otras finalidades. Uno de los más importantes es el campo scripts, que permite poner a disposición mandatos de consola que realizan ciertas funcionalidades. El objeto scripts es un conjunto de pares clave-valor, donde la clave es el nombre del mandato y el valor es el comando bash que se va a ejecutar al utilizar ese mandato. Tenga en cuenta que este comando bash permite el uso de binarios que se encuentren dentro de las dependencias del proyecto, por lo que la potencia de estos scripts es infinita.

Con esta base, se ha decidido incluir en el proyecto uno de estos mandatos: el mandato setupº, que se encarga de montar el proyecto en función de las preferencias del usuario. Para ello, utiliza la librería Inquirer [SBo20], que permite hacer una serie de preguntas al usuario mediante la consola. Una vez terminadas las preguntas, el mandato setup automáticamente elimina todos aquellos ficheros del repositorio que el usuario no va a utilizar (en función de sus preferencias), enlaza los que sí va a utilizar y deja preparados los scripts que el usuario pueda necesitar, de forma que el usuario pueda desde ese momento ponerse a trabajar. Se puede observar cómo se realizan las preguntas al usuario en las figuras Figure 5.1 y Figure 5.2. En la figura Figure 5.3 se puede ver el final del script con todas las preguntas realizadas al usuario.

Cabe destacar que el proyecto se ha planteado como un proyecto abierto y la intención es ir incrementando el número de opciones de las que dispone el usuario para que el marco vaya siendo cada vez lo más completo posible. Cómo se puede ampliar el proyecto y cómo se ha planteado la ampliabilidad del proyecto se detalla más adelante en el apartado Próximos pasos.

El marco cuenta con un proyecto modelo para todas las opciones que introduzca el usuario. El proyecto modelo es una simple lista de tareas (ver lista de tareas vacía en la figura Figure 5.4), en la cual el usuario puede añadir nuevas tareas (Figure 5.5), marcar y desmarcar las tareas como hechas (Figure 5.6), modificarlas y borrarlas. Este proyecto modelo ha sido realizado para todas las opciones de las que dispone el marco de trabajo: todos los tipos de base de datos incluídos en Gestor de Base de Datos y todas las capas de datos incluídas en Capa de datos. Además, se ha cubierto el proyecto modelo con un 98% de cobertura en pruebas para todos los tipos de proyecto modelo. Las pruebas se han realizado únicamente mediante Jest y React Testing Library (tal y como

```
[aalcazar ~/Projects/TFM/react-rocket-boilerplate (feature/rocket-options) $ npm run setup

> react-boilerplate@4.0.0 setup /Users/aalcazar/Projects/TFM/react-rocket-boilerplate
> node ./internals/rocket-options/index.js


Welcome to React Rocket Boilerplate! Please, answer a few questions about your new project.

? Which package manager you prefer? (Use arrow keys)
❯ npm
  yarn
```

Figure 5.1: React Rocket Generator - Asking about Package Manager

```
[aalcazar ~/Projects/TFM/react-rocket-boilerplate (feature/rocket-options) $ npm run setup

> react-boilerplate@4.0.0 setup /Users/aalcazar/Projects/TFM/react-rocket-boilerplate
> node ./internals/rocket-options/index.js


Welcome to React Rocket Boilerplate! Please, answer a few questions about your new project.

? Which package manager you prefer? npm
[? Do you want to use a back-end server with Node.js and Express? Yes
? Which data layer do you prefer for your application?
❯ GraphQL and Apollo
  REST and Redux
  Just REST
```

Figure 5.2: React Rocket Generator - Asking about the API

```
[aalcazar ~/Projects/TFM/react-rocket-boilerplate (feature/rocket-options) $ npm run setup

> react-boilerplate@4.0.0 setup /Users/aalcazar/Projects/TFM/react-rocket-boilerplate
> node ./internals/rocket-options/index.js


Welcome to React Rocket Boilerplate! Please, answer a few questions about your new project.

? Which package manager you prefer? npm
[? Do you want to use a back-end server with Node.js and Express? Yes
? Which data layer do you prefer for your application? GraphQL and Apollo
[? Do you want to use a database? Yes
? Which database do you prefer? MongoDB
[? Do you want Next.js for Server Side Rendering (SSR)? No
? Which testing library will you use to set up unit and integration tests? React Testing Library
[? Do you want to use a end to end testing? Yes
? Which testing library will you use to set up end to end tests? Selenium
[? Will you use a continuous integration server? Yes
? Which continuous integration server will you use? Jenkins
```

Figure 5.3: React Rocket Generator - All questions together

Figure 5.4: React Rocket Todo List - Empty List

se comenta en Pruebas unitarias), pero entra dentro del ámbito del proyecto (tal y como se comenta más adelante en Próximos pasos) dar también soporte a Enzyme. Esto implicará repetir todas las baterías de pruebas para Enzyme.

Por último, el proyecto cuenta con integración continua integrada. Sin embargo, como esto depende de interacción explícita entre una cuenta registrada en un servidor de git (como Github) y una herramienta de desarrollo continuo (como Jenkins), ha sido imposible automatizar la instalación de la integración continua. Sin embargo, una vez están conectadas ambas herramientas mediante una cuenta con permisos suficientes, se ha logrado automatizar la tarea de integración continua mediante ficheros de configuración (tanto para Jenkins como para CircleCI), tal y como se especifica en el apartado Detalle de la integración continua.

Con todo esto, el usuario del marco debería tener que hacer los siguientes pasos para poner en mancha un proyecto con este marco:

1. Clonar el repositorio del proyecto.
2. Ejecutar npm setup.
3. Responder a las preguntas sobre su proyecto. Una vez las termine, el proyecto podrá ser utilizado de forma local con la aplicación modelo descrita anteriormente.
4. Continuar con las instrucciones del README para poner en marcha la integración continua del proyecto.

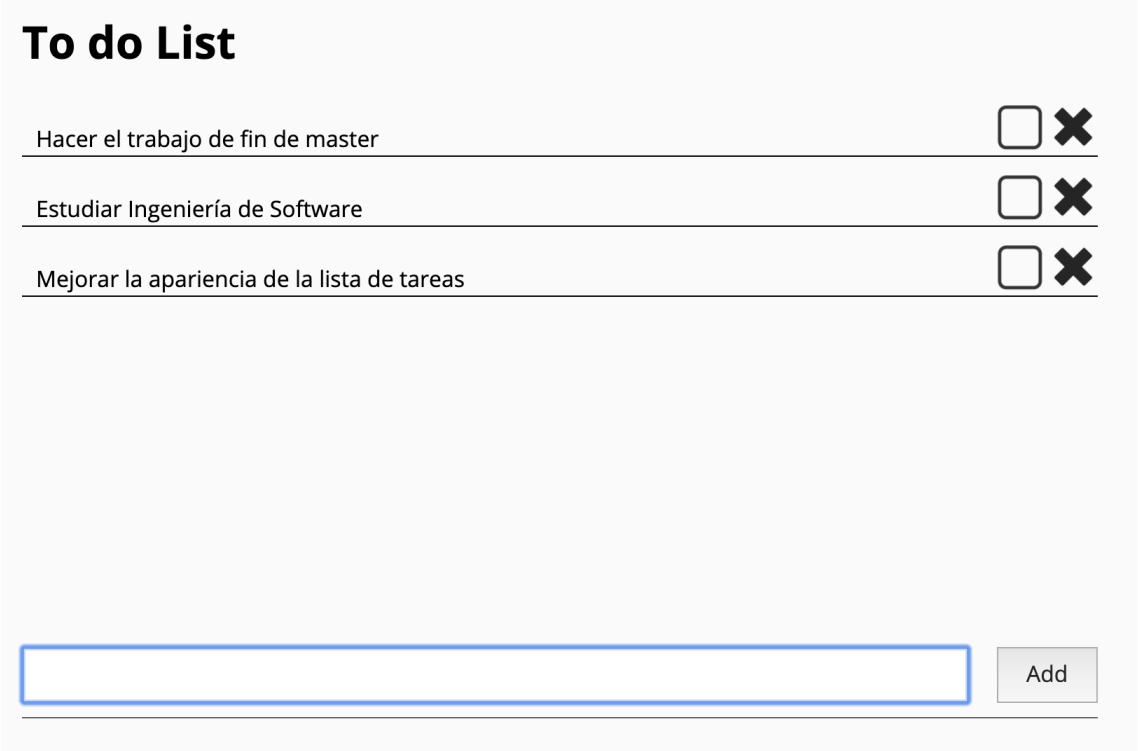

Figure 5.5: React Rocket Todo List - Generated 3 items

Figure 5.6: React Rocket Todo List - Item marked as done

Como se puede observar, el proyecto es usable con muy poco esfuerzo. La parte más costosa es la de poner en marcha la integración contínua, que es definitivamente algo opcional. Aunque, por otro lado, utilizar este marco carece de sentido si no se pretende realizar un proyecto de calidad con pruebas automáticas e integración contínua.

A lo largo de este capítulo se irá mostrando el detalle completo de todas las partes involucradas en el marco.

5.2 Detalle de la integración continua

Para el desarollo de la integración continua y el despliegue continuo se han utilizado dos herramientas libres: Jenklins y CircleCI. El objetivo era minimizar el esfuerzo de configuración, pero esta tarea se dificulta sabiendo que, tal y como explica el diagrama de la figura Figure 4.1, puede haber un total de 4 máquinas interviniendo en el despliegue. Una de las cuales es el servidor de git y se asume que se está utilizando Github, así que se descarta. La máquina que se encarga del desarollo continuo es necesaria solo si se utiliza Jenkins o si se utiliza el plan de pago de CircleCI. Como se ha ido comentando a lo largo del trabajo, se descarta cualquier opción de pago, así que se va a asumir que si se utiliza Jenkins, se dispone de una máquina para el servidor de integración continua y si se utiliza CircleCI no.

Tanto jenkins como CircleCI disponen de la posibilidad de configurar el proyecto mediante un fichero de configuración que se incluye dentro del repositorio. Esto implica que el usuario no tendrá que preocuparse por casi nada de la configuración del proyecto. En el caso de Jenkins, este fichero se llama Jenkinsfile y utiliza la sintaxis Groovy y en el caso de CircleCI, este fichero se llama .circleci/config.yml y utiliza la sintaxis YAML. Sin embargo, ambas herramientas carecen de un método para generar automáticamente el proyecto, así que se requiere un mínimo trabajo manual para generar el proyecto y conectarlo al repositorio. En el caso de Jenkins, además, es necesario el plugin de Github y el plugin SSH Publisher para poder realizar una integración completa. Jenkins es ligeramente más complicado de montar porque requiere tener una máquina disponible para hacer de servidor. Sin embargo, su instalación es directa y se ejecuta únicamente mediante el mandato jenkins –daemon.

Respecto a la máquina que va a alojar el código, se espera que posea por lo menos los siguientes paquetes:

1. git
2. node y npm
3. pm2: esta herramienta permite automatizar el despliegue en un solo fichero de configuración. El marco de trabajo asume que se utiliza pm2 para poder desplegar el código mediante un solo mandato.
4. ssh: es necesario para poder comunicarse con la máquina.
5. Visibilidad en la red: esto no es un paquete como tal, pero es un requisito indispensable. En el caso de Jenkins, la máquina objetivo debe estar dentro de la misma red (o ser la misma máquina) que la máquina que aloja Jenkins. En el caso de CircleCI, es imprescindible que la

máquina tenga una dirección IP estática o tenga asociado un nombre de dominio a una IP Dinámica mediante Dynamic Domain Name System (DDNS). Además, deberá tener abiertos el puerto 80 y el 8080.

Tal y como se comenta en Próximos pasos, se pretende dar soporte a Docker. Una vez se logre este hito, será indispensable que la máquina de la aplicación tenga instalado Docker. Para estos requisitos mínimos se ha elaborado un fichero de instalación que habrá que ejecutar una sola vez en la máquina que vaya a alojar el código, asumiendo que la máquina destino tiene acceso a apt. En caso de tener una distribución con otro servidor de paquetes, el usuario deberá buscar por su cuenta la forma de instalar los paquetes mencionados anteriormente. Dada una máquina con las características mencionadas, el fichero generado tanto para Jenkins como para CircleCI será capaz por sí mismo de ordenar a la herramienta en cuestión el despliegue automático en el servidor que vaya a alojar la aplicación.

Para simplificar el despliegue, se asume que la máquina que vaya a alojar la aplicación va a hacerlo utilizando el puerto 80 (o 443 en el caso de HTTPS) y que, la misma máquina va a utilizar el puerto 8080 para alojar la misma aplicación en preproducción. Además, se asume que la rama develop del servidor de git se conectará al servidor de preproducción y que la rama master se conectará al servidor de producción. Todo esto ya está contemplado en los ficheros de configuración tanto de Jenkins como de CircleCI.

Quiero aclarar que soy consciente de que esta es sin duda la parte más engorrosa del proyecto. Por su propia naturaleza, la variabilidad del entorno, la intervención de la red y el uso de herramientas muy gráficas como Jenkins y CircleCI, la cantidad de trabajo manual necesaria para poner en marcha el proyecto es mucho mayor de la que desearía. A lo largo del proyecto me he ido dando cuenta de la gran necesidad que hay de automatizar aun más todo este proceso y, por tanto, me he dado cuenta de que Docker debería haber sido parte del proyecto desde el principio. Sin embargo, mis recursos me han impedido añadirlo en esta primera fase y, por tanto, lo he dejado como máxima prioridad en el apartado de Próximos pasos. Docker no permitirá toda la automatización del sistema, pero sí permitirá una mejor integración con las herramientas mencionadas en este capítulo y reducirán mucho la carga de instalación de paquetes en la máquina que aloja la aplicación.

6 Métricas del marco de trabajo

6.1 Introducción

Una vez ha sido construido el marco de trabajo, se ha realizado el proyecto ”TODO List”con él, tal y como se explica en el apartado Desarrollo del marco de trabajo. Este proyecto se ha realizado de dos formas distintas: primero se ha realizado sin el marco de trabajo, y luego se ha repetido utilizando el nuevo marco.

El objetivo de este experimento es comprobar cuánto esfuerzo supone realizar un proyecto muy pequeño de forma que el resultado tenga calidad y sea escalable y, de ese modo, comprobar desde qué tipo de proyecto puede empezar a merecer la pena aplicar dicho esfuerzo.

La motivación original de este proyecto era conseguir reducir este esfuerzo inicial que supone realizar este trabajo mantenible y escalable. Lo ideal para este experimento habría sido que hubiese una población más grande y dividirla en tres grupos de desarrollo:

1. Los que hacen el proyecto sin aportar calidad.
2. Los que hacen el proyecto a partir del marco de trabajo y sin conocerlo previamente.
3. Los que hacen el proyecto con calidad, pero sin utilizar el marco de trabajo.

De este modo, podríamos realizar un verdadero estudio estadísitico sobre lo que aporta de verdad el marco. Sin embargo, dados los recursos de este trabajo (tanto de tiempo como de capacidad de trabajo), tengo que reducir el experimento a un solo desarrollador (yo mismo) y prescindir del tercer grupo de trabajo. Además, una vez hecho el proyecto con marco de trabajo, al realizar el proyecto sin marco de trabajo ya habré pasado por un proceso previo de diseño y habré aprendido de mis errores en el desarrollo anterior. Por todo esto, la valided del experimento debe ser tomada con cuidado.

Aun así, se pueden sacar conclusiones de este experimento. El objetivo es tener una idea general de cuánto supone este esfuerzo, hacer una estimación de costes para una empresa pequeña y, de paso, comprobar si he echado de menos algo del marco de trabajo para poder tenerlo en consideración a futuro. Para poder sacar el máximo partido al experimento, en este apartado he realizado predicción sobre los resultados que esperaba y por qué antes de realizar dicho experimento. Una vez hecho el experimento, he complementado las predicciones con comentarios sobre las sorpresas que he recibido y las conclusiones que he sacado de los números obtenidos.

6.2 Predicciones: resultados esperados

Todas las estimaciones que se van a poder ver a continuación han sido realizadas bajo la asumción de que la persona que realiza las tareas domina las tecnologías que está utilizando. Para este proyecto se han estimado los siguientes tiempos sin utilizar el marco de trabajo ni aportando calidad al producto:

1. Servidor: 3 horas
    a) Montar el servidor (GraphQL y Express): 1 hora
    b) Añadir tarea: 0.5 hora
    c) Eliminar tarea: 0.5 hora
    d) Editar tarea: 0.5 hora
    e) Marcar tarea: 0.25 horas
    f) Obtener lista de tareas: 0.25 horas
2. Cliente: 5.5 horas
    a) Montar el cliente (Webpack): 2 horas
    b) Página de lista de tareas: 0.5 horas
    c) Componente de campo de texto: 0.5 horas
    d) Componente de botón de borrar tarea: 0.25 horas
    e) Componente de botón de editar tarea: 0.5 horas
    f) Componente de tarea: 0.25 horas
    g) Componente de caja de tareas: 0.25 horas
    h) Componente de botón de añadir tarea: 0.25 horas
    i) Integración con el servidor: 1 hora
3. Subir a producción: 30 minutos

En total, sin utilizar el marco de trabajo, se estima que se va a tardar 9 horas. En principio, en un día largo de trabajo podría abordarse el proyecto TODO List entero. El marco de trabajo se encargará de ahorrarnos las matesde montar el servidor, montar el cliente y la integración con el servidor. A cambio, nos exigirá un mínimo de cobertura en el código y nos facilitará engancharnos con un servidor de integración contínua. Por tanto, se espera que al utilizar el marco de trabajo el coste del proyecto sin preubas se reduzca en 3.5 horas aproximadamente. A cambio, se esperan los siguientes costes en pruebas:

1. Pruebas unitarias en servidor: 5 horas
2. Pruebas unitarias en cliente: 10 horas
3. Pruebas de integración: 4 horas

4. Pruebas funcionales: 6 horas

Por tanto, se espera que el proyecto va a costar 32 horas utilizando el marco de trabajo para aportar calidad. Hay que tener en cuenta que, una vez terminado el proyecto con el marco de trabajo, se asume integración contínua y automática con Github, Jenkins, entorno de preproducción y producción. Cuanto más se modifique este proyecto a futuro, más rentable saldrá montar todo este entorno de calidad.

Además de los tiempos, es muy importante medir errores. En un proyecto tan pequeño, se esperan en total 10 errores máximo sin utilizar el marco de trabajo. En el proyecto con marco de trabajo no se espera ningún error.

6.3 Resultados obtenidos

Los resultados del experimento se han recogido en la tabla Table 6.1. Como se puede observar, todas las tareas han implicado más esfuerzo debido a las pruebas unitarias que se han realizado haciendo el uso del marco. Sin embargo, la configuración inicial del proyecto mediante el marco de trabajo ha sido nula, tal y como estaba previsto. De esta tabla cabe destacar que el esfuerzo que nos ha ahorrado el marco de trabajo corresponde a las tareas de Montar el servidor y la integración con el servidor, que han sido un total de 3 horas.

Se han encontrado un total de 20 errores en la implementación sin marco, mientras que el marco nos ha evitado todos los errores. Estos errores han sido encontrados una vez he acabado los dos proyectos. He ido revisando todos los casos de prueba y realizándolos manualmente en el proyecto sin marco. El resultado ha sido que ha habido 20 problemas que, utilizando pruebas unitarias he identificado y resuelto con antelación, mientras que sin realizar estas pruebas se me han pasado por alto. El listado de errores concretos se puede encontrar en la tabla Table 6.2.

Una vez identificados los erroes, se ha dedicado esfuerzo a solucionarlos y se ha añadido una columna a la tabla Table 6.2 con ese tiempo. El tiempo total ha sido de más de 4 horas. En la figura Figure 6.1 se puede apreciar la diferencia del esfuerzo dedicado en el proyecto sin marco de trabajo y el proyecto con marco de trabajo, que casualmente ha sido de 4 horas.

En resumen, por un lado el proyecto sin marco de trabajo nos ha ahorrado 4 horas en total en desarrollo de pruebas unitarias. De las horas dedicadas, 3 han sido evitadas por el marco de trabajo. Es decir, que si hubiésemos realizado el proyecto sin código de calidad, pero utilizando el marco de trabajo, habríamos dedicado 8 horas en total. Por otro lado, se han producido 20 errores desarrollando sin pruebas automáticas, lo cual ha llevado 4 horas y 10 minutos solucionarlos.

6.4 Conclusiones

Lo primero que me ha sorprendido al ver los números finales es que se ha amortizado el tiempo dedicado a hacer las pruebas unitarias. He experimentado esta amortización en proyectos más grandes, pero no me esperaba que en un proyecto de esta escala las horas dedicadas a solucionar errores fuesen a ser mayores a las horas dedicadas a realizar pruebas untiarias. Hay que tener en cuenta que, tal y como se comenta en la introducción de Métricas del marco de trabajo, este

Table 6.1: Tabla de resultados

		Sin marco de trabajo		Con marco de trabajo	
Tarea	**Tiempo estimado**	**Tiempo empleado**	**Errores encontrados**	**Tiempo empleado (con marco)**	**Errores encontrados (con marco)**
Montar el servidor (GraphQL y Express)	1	1.5	2	0	0
Añadir tarea	0.5	1	0	1.5	0
Eliminar tarea	0.5	0.5	1	1	0
Editar tarea	0.5	0.75	4	1.5	0
Marcar tarea	0.25	0.25	1	0.75	0
Obtener lista de tareas	0.25	0.5	1	1	0
Montar el cliente (Webpack)	2	2	0	0	0
Página de lista de tareas	0.5	0.5	0	2	0
Componente de campo de texto	0.5	0.25	1	0.5	0
Componente de botón de borrar tarea	0.25	0.25	1	0.75	0
Componente de botón de editar tarea	0.5	1	2	2	0
Componente de tarea	0.25	0.5	4	1.25	0
Componente de caja de tareas	0.25	0.25	0	1.5	0
Componente de botón de añadir tarea	0.25	0.25	1	1.25	0
Integración con el servidor	1	1.5	2	0	0

Table 6.2: Tabla de errores

ID	**Tarea**	**Descripción del error**	**Tiempo de resolución (min)**
1	**Montar el servidor (GraphQL y Express)**	El esquema GraphQL se lee con una codificación no soportada	5
2	**Montar el servidor (GraphQL y Express)**	Error de CORS	5
3	**Eliminar tarea**	Eliminar tarea que no existe devuelve OK	15
4	**Editar tarea**	Editar tarea que no existe devuelve OK	15
5	**Editar tarea**	Editar una tarea marcada (toggled) quita la marca	5
6	**Editar tarea**	Editar una tarea sin texto devuelve OK	5
7	**Editar tarea**	Editar una tarea que tiene el texto repetido no funciona	10
8	**Marcar tarea**	Marcar una tarea con el texto repetido a veces no funciona	10
9	**Obtener lista de tareas**	Si la lista se acaba de vacíar tras eliminar tarea, devuelve error	30
10	**Componente de campo de texto**	Si la API devuelve error, muestra ese error en el campo de texto	5
11	**Componente de botón de borrar tarea**	Borrar una tarea recién reordenada borra una tarea distinta	10
12	**Componente de botón de editar tarea**	Editar una tarea recién reordenada edita una tarea distinta	10
13	**Componente de botón de editar tarea**	Editar una tarea sin cambiar el texto envía un vacío a la API	5
14	**Componente de tarea**	Reordenar una tarea marcada la desmarca	15
15	**Componente de tarea**	Editar una tarea la cambia de orden	10
16	**Componente de tarea**	Editar una tarea y no ponerle texto genera un error en consola	5
17	**Componente de tarea**	Marcar y desmarcar una tarea rápidamente la hace desaparecer	45
18	**Componente de botón de añadir tarea**	Añadir tarea sin escribir nada en el campo de texto envía una petición al servidor	5
19	**Integración con el servidor**	Inconsistencia de la cache de Apollo con respecto a los datos del servidor	30
20	**Integración con el servidor**	Error con el tamaño máximo de carga	10
Tiempo total (min)			250
Tiempo total (horas)			4.166666667

experimento ha sido muy limitado y hay que coger con pinzas estos resultados, pero ya nos hacemos a la idea que en ocasiones e incluso para proyectos pequeños, hacer pruebas unitarias puede salir rentable.

Para hacer justicia a la verdad, el tiempo total dedicado a realizar las pruebas automáticas ha sido de 7 horas. Sin embargo, con el descuento que nos hace el marco de trabajo con toda la configuración del entorno ha salido rentable. En proyectos más grandes esto es mucho más difícil de medir, pero los errores tienden a escalar en profundidad y suelen ser más difíciles de depurar. Además, a esto habría que sumar pruebas de integración y funcionales, que no se han realizado por la simplicidad del proyecto.

Otro dato que me ha sorprendido es la cantidad de errores que se han producido. 20 errores en un código de aproximadamente 5600 líneas de código implica a un error cada 280 líneas de código. Es de esperar que a este ritmo, si se comienza una prueba de concepto sin código de calidad, a la hora de realizar el escalado cualquiera de estos muchos errores va a pasar factura más adelante. Los errores predecidos eran 10, así que aunque el proyecto sea bien conocido y de dimensiones

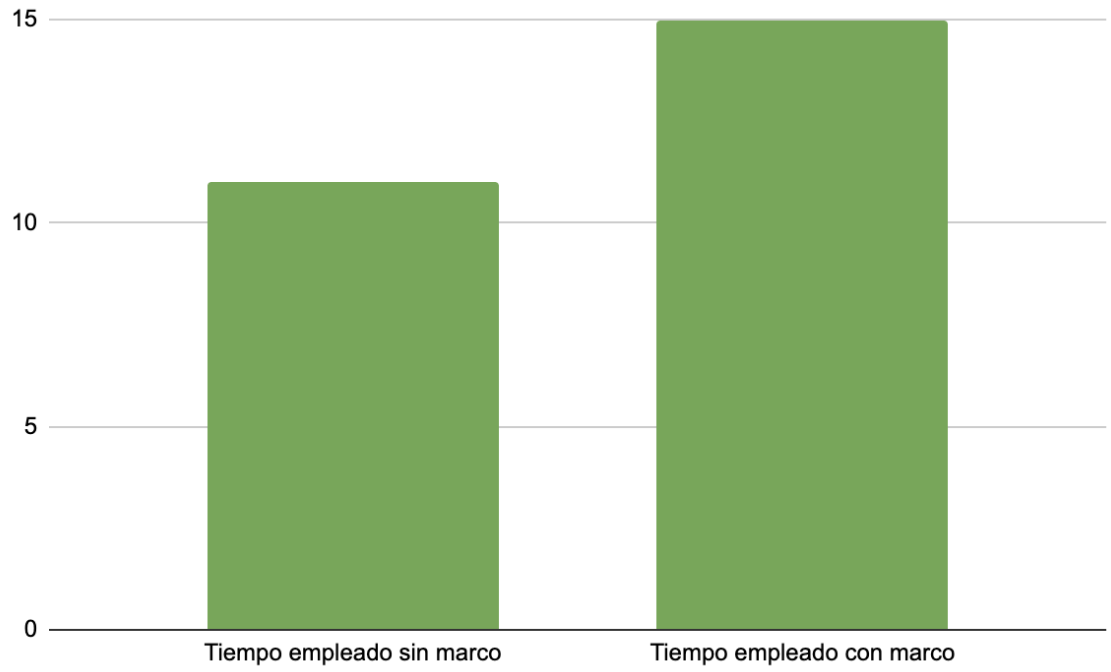

Figure 6.1: Comparación entre el tiempo dedicado con y sin pruebas

limitadas, los errores están a la orden del día. En un proyecto de prueba, podríamos llegar a esperar un error cada 150 líneas de código. Esto hace mucho más acuciante la necesidad de utilizar código de calidad.

El tiempo que nos ha ahorrado el marco de trabajo ha sido de 3 horas, pero no es un dato muy representativo dado que gran parte de la potencia del marco no se ha utilizado. No se ha utilizado ningún servidor de integración contínua ni se ha desplegado el proyecto en ningún sitio, así que esas 3 horas de ahorro se pueden considerar como el mínimo. Además, estas 3 horas son lo que he tardado yo en configurar el proyecto, que soy la misma persona que ha realizado el marco de trabajo y, por tanto, se ha estudiado numerosas posibles configuraciones y comprende las interacciones de Express, Webpack, React y GraphQL con bastante profundidad. Una persona que está explorando un campo nuevo puede dedicar incluso una semana entera a configurar el proyecto, así que este ahorro no ha quedado bien reflejado en el experimento.

Así que, como conclusión final del experimento, este marco de trabajo ha sido muy positivo incluso para proyectos pequeños porque alivia la carga de configuración del proyecto, tal y como se pretendía, preparando el entorno para la realización de pruebas automáticas y evitando esos 20 errores que podrían costar una fortuna si se encuentran una vez el proyecto ha escalado y está en producción.

Aun así, como el experimento ha sido limitado queda por comprobar si realmente en un caso de uso real este marco de trabajo es de utilidad. Y la única forma de comprobarlo es con tiempo y esperando la retroalimentación de los usuarios del marco. La apuesta, sin embargo, es que este marco va a ser de gran utilidad siempre y cuando el compromiso de realizar código de calidad sea real.

7 Próximos pasos

7.1 Abrir el código

El alcance del trabajo está limitado a las horas impuestas y, dadas las circunstancias, se ha decidido acotar a unas pocas tecnologías elegidas generalmente mediante el criterio de la popularidad, de forma que este marco pueda ser útil al mayor número posible de desarrolladores. El código ya ha sido escrito con eso en mente y, por tanto, se ha preparado el terreno para dar un paso evidente en el ciclo: publicar el código como abierto.

Esto tiene las siguientes ventajas:

1. El código es revisado por numerosos desarrolladores, dando distintos puntos de vista y aportando diversidad.
2. El código se produce a más velocidad y los errores se resuelven más rápido, sin coste alguno.
3. El proyecto se puede reutilizar y se pueden generar otros marcos y otras ideas.
4. El marco se puede ampliar a más tecnologías porque más usuarios lo van a dar forma en función de sus necesidades.

Sin embargo, el código abierto no es un paso pequeño. Tal y como constata el estándar de código abierto [Ope20], tiene una serie de responsabilidades para el dueño originario del código para que pueda funcionar, que en este marco son necesarias. Estas responsabilidades son, en general, buenas prácticas para cualquier código, pero en este marco en particular son unos requisitos indispensables a ser abordados antes de poder publicar el código como abierto:

1. Todo el código debe estar revisado y comentado en función de una guía de buenas prácticas. La dificultad de esta tarea se va a reducir porque se utilizó Linter desde el primer momento, pero aun así una revisión de calidad es necesaria.
2. El marco debe estar documentado en su totalidad. En el estado actual del proyecto, toda la documentación se refiere a como construir un proyecto a partir del marco, pero el marco en sí debe ser documentado mediante el punto de vista del editor del marco. Esto implica documentar las tareas automáticas y una guía de amplicación de las tencologías disponibles. Para hacer estas guías, es razonable realizar una amplicación por mí mismo para comprobar que todo funciona como se había planteado.
3. Elaborar los documentos requeridos para el código abierto: Licencia, README, Guía de contribución y Código de conducta. El README ya está generado, pero requerirá una revisión y quizá nuevos apartados para la instalación desde un punto de vista de contribución.

En el momento en el que el código se transforme en código abierto, se habrá cumplido el verdadero propósito del proyecto y se considerará iniciada la primera versión oficial del producto. Por tanto, el código ha de estar totalmente completo desde todos los puntos de vista posibles. El estado actual del producto es que es completo y listo para ser utilizado como marco, como semilla de otros proyectos. Pero faltan una serie de pasos para poder ser ampliado libremente y esto requiere planificación y revisión, así que esto será, en definitiva, el próximo paso más inmediato.

7.2 Ampliabilidad del código

Uno de los pasos fundamentales para poder publicar el código abierto es que sea ampliable. Como la idea es que la tarea de ampliar el marco se va a repetir en numerosas ocasiones, se ha planteado la opción de automatizarla en la medida de lo posible. Por tanto, teniendo en cuenta el futuro, he diseñado el proceso de ampliación y las tareas que van a facilitar la tarea. Supongamos que un usuario desea ampliar el código con una tecnología particular -digamos que por ejemplo desea añadir Redis como base de datos- así que se clona el respositorio y se instala las dependencias. Lo ideal sería que ese usuario ejecutase una tarea framework:expand y que, mediante el mismo sistema que utiliza el marco para comenzar un proyecto, le hiciese unas preguntas sobre su ampliación.

Estas preguntas buscarán obtener la siguiente información:

1. Conocer el nombre de la tecnología a implementar. En este caso Redis
2. Conocer el tipo de tecnología a implementar. En este caso Base de datos.
3. Conocer qué ficheros implicados ha de poseer el marco, que deberá eliminar si el usuario no elige la tecnología.
4. Conocer qué tarea del package.json deberá aplicar al instalar el marco en caso de necesitarla. En este caso particular, el usuario puede querer producir una instancia de la base de datos en su instalación.
5. Conocer las opciones con las que se puede utilizar esta tecnología en el marco. Estas opciones serán luego ofrecidas al usuario final durante la fase de preguntas si elige utilizar esta nueva tecnología, Redis, y el resultado de la selección se le pasará a esa tarea de inicialización del package.json para que el usuario desarrollador pueda trabajar con ellas.

Con toda esta información, el marco será capaz de mantener un registro de los ficheros asociados, podrá pasar pruebas automáticas con distintas configuraciones y podrá actualizar una documentación online sobre las tecnologías disponibles. Todas estas tareas quedan a responsabilidad del propio marco y no del usuario.

Como la intención no es restringir, si no ayudar, esta tarea debe ser una mera inicialización de la nueva tecnología, pero todo debe poder ser editado al vuelo. Por tanto, esta tarea no hará más que generar información en una serie de ficheros que podrán ser editados. Estos ficheros serán la clave de la ampliabilidad, dado que permiten documentar de forma automática el marco y, a la vez, automatizar todas las tareas de instalación.

Por otro lado, el usuario quedará responsable de cumplir los requisitos de cobertura para su tecnología en la aplicación de pruebas ”Lista de tareas”, generar las tareas automáticas de instalación y mantener los ficheros mencionados en el párrafo anterior. Aun así, una vez el código pasa esas pruebas y está

listo para ser subido, requerirá una revisión manual por parte de los encargados del mantenimiento del código, que será un equipo compuesto por personas de confianza que hayan contribuido al código. En la imagen inicial, estas ampliaciones solo requerirán de mi revisión manual.

La idea es que, de forma inicial, la ampliación del código sea lo más sencilla y automática posible. Y que, poco a poco, con contribuciones de distintos desarrolladores, el sistema se vaya cerrando más y quede totalmente automatizado. Todavía no se ha diseñado exactamente qué ficheros van a participar y qué estructura van a tener, pero esto será uno de los pasos requeridos antes de poder publicar el código como abierto.

7.3 Próximas tecnologías a abordar

Una vez se haya abierto el código, mi tarea con el marco no ha concluido. Además de las revisiones que tendré que hacer y de la resolución de errores que vayan surgiendo, quedará a mi responsabilidad contribuir a la ampliación del marco por mí mismo. De este modo, podré probar si todo lo que he diseñado para la ampliación funciona bien y tiene sentido. Además, se publicará una serie de tecnologías que han sido contempladas en la fase de investigación pero no han podido ser incluídas en el marco original por falta de tiempo y alcance en el trabajo.

Queda a mi responsabilidad el ir añadiendo poco a poco estas tecnologías a menos que algún contribuyente decida abordar alguna por sí mismo. Esta lista ya ha sido pensada en la fase de investigación, pero quedará expuesta a posibles cambios en función de la demanda en el apartado Issues de Github, que tendrá una etiqueta disponible para pedir ampliaciones de tecnologías. Mi tarea consistirá en ir añadiendo las más populares, independientemente de la lista que voy a añadir a continuación. Sin embargo, por el momento y mientras el marco no tenga suficiente público, se irán abordando en el orden en que considero más necesario a raíz de este trabajo.

Es evidente que la dirección concreta que va a tomar el marco depende en su totalidad de su popularidad. En el desafortunado caso de que este marco no logre darse a conocer o no logre ser de suficiente utilidad, el marco quedará completo cuando las tecnologías de la lista hayan sido añadidas. En caso de que el marco adquiera uso y la demanda aumente, es probable que la lista cambie de forma en función de esta demanda.

Así que, como último apartado de los últimos pasos, será añadir las siguientes tecnologías en el orden propuesto:

1. Docker: es imprescindible que el marco de soporte a contenedores Docker y, por tanto, esta será la máxima prioridad. Esto además tiene grandes repercusiones sobre la integración continua y facilita mucho la tarea del despliegue y el escalado.
2. Enzyme: numerosos desarrolladores están trabajando con Enzyme en lugar de React Testing Library. Me parece requisito indispensable que acabe entrando dentro del marco como máxima prioridad.
3. Redis: una base de datos libre que se utiliza en numerosos proyectos.
4. Vue
5. AngularJS

6. Go: esta idea todavía no la tengo clara, dado que todo el marco está en NodeJS. Sin embargo, sería adecuado si otros lenguajes pudiesen ser utilizados en el lado de servidor. Go parece ser la alternativa a NodeJS que más está entrando en el mercado. De todos modos, esta opción queda pendiente a ser evaluada cuando llegue el momento.

7. Kubernetes y, en el futuro, otras herramientas de escalado flexible: el objetivo es que el usuario comience su proyecto de forma rápida pero tenga acceso a todas las herramientas de escalado de forma relativamente sencilla. El escalado nunca es una tarea fácil, pero el mundo dispone de herramientas como Kubernetes que, una vez se ha integrado el proyecto con Docker, es más asequible que nuestro proyecto aumente de tamaño sin miedo a perder el control. Kubernetes se encarga de gestionar los distintos nodos que interactuan en nuestro proyecto, generando o eliminando instancias en función de la demanda. Por tanto, para que el marco de trabajo se pueda utilizar de forma profesional, debe haber un soporte adecuado para este tipo de herramientas.

8 Conclusiones del proyecto

Durante este proyecto se ha investigado una numerosa cantidad de tecnologías y se han recogido las más frecuentes en el ámbito del desarrollo web. Se ha analizado cuáles combinan bien entre ellas, se ha creado un entorno que facilita la configuración rápida del proyecto y se ha generado un proyecto de pruebas para cada tecnología seleccionada.

El objetivo era facilitar la producción de pruebas de concepto de calidad, de modo que pudiesen escalar de forma rápida y sin errores. Para eso hace falta seguir una serie de buenas prácticas y construir una buena batería de pruebas desde el primer minuto.

Para demostrar que este objetivo se cumple, se ha realizado un experimento sencillo: se ha creado un proyecto en una versión sin marco y otra versión con marco. Se han comparado, y ha resultado que la versión con marco, para este caso particular, nos ha ahorrado 20 errores en tan solo 4 horas más de codificación. Si consideramos el tiempo de resolver esos errores, el marco nos ha permitido dedicar el mismo tiempo que si lo hubiésemos hecho sin marco, pero teniendo la batería de pruebas generada y, por tanto, previniendo futuros errores que pudiesen surgir con la evolución del proyecto.

Después de todo el esfuerzo y viéndolo con perspectiva, he llegado a la conclusión de que la cantidad de tecnologías soportadas no es suficiente para cubrir la mayoría de necesidades del mundo del desarrollo web y, cuantas más tecnologías cubra el marco, más rápida va a ser la configuración de las pruebas de concepto y más útil va a ser el marco. Así que, para cubrir un mayor número de tecnologías, hace falta abrir el código y trabajar más en el producto.

Aun así, la experiencia de utilizar el marco de trabajo tal y como está ahora ha sido muy gratificante. Tener un esqueleto de proyecto que asegura calidad tan solo respondiendo unas cuantas preguntas, que ese esqueleto sea flexible a las tecnologías implicadas y que incluya ficheros de configuración para integración continua es exactamente lo que personalmente necesito para motivarme a que mis pruebas de concepto tengan calidad desde el primer momento.

Así que espero ser capaz de transmitir esta experiencia a otros usuarios y que poco a poco este marco se vaya transformando en una necesidad para todo aquél que desee empezar un proyecto de cero. También espero que el marco sea semilla de proyectos de calidad y, de este modo, animar al resto de desarrolladores a comenzar con buen pie cada idea que tengan por pequeña que sea.

Bibliography

[Bla20] Black Duck. *Compare Repositories*. 2020. URL: https://www.openhub.net/repositories/compare (cit. on p. 21).

[Bui20] BuiltWith. *Framework Usage Distribution in the Top 1 Million Sites*. 2020. URL: https://trends.builtwith.com/framework (cit. on p. 18).

[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. 26).

[Das18] C. Dascalu. *Unit testing — why?* 2018. URL: https://medium.com/@corneliu/unit-testing-why-3490d08e89f2 (cit. on p. 22).

[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 p. 29).

[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. 24).

[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. 25).

[G220] G2. *Best Version Control Systems*. 2020. URL: https://www.g2.com/categories/version-control-systems#grid (cit. on p. 21).

[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. 21).

[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. 26).

[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. 25).

[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. 21).

[Mon16] MongoDB. *What is a Document Database?* 2016. URL: https://www.mongodb.com/document-databases (cit. on p. 26).

[Ope20] Open Source. *Open Source Guide - Starting a Project*. 2020. URL: https://opensource.guide/starting-a-project/ (cit. on p. 45).

[Pan20] A. Panwar. *Database Management Systems*. 2020. URL: https://www.c-sharpcorner.com/UploadFile/65fc13/types-of-database-management-systems/ (cit. on p. 27).

[Pol19] G. Polvara. *Difference between enzyme, ReactTestUtils and react-testing-library*. 2019. URL: https://stackoverflow.com/a/54152893/6282406 (cit. on p. 24).

[Rea18] React Team. *React Test Utilities*. 2018. URL: https://reactjs.org/docs/test-utils.html#overview (cit. on p. 24).

[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. 27).

[Rus18] J. Russom. *Integration Testing in React*. 2018. URL: https://medium.com/expedia-group-tech/integration-testing-in-react-21f92a55a894 (cit. on p. 25).

[San18] R. Santor. *One Major Way React > VueJS*. 2018. URL: https://hackernoon.com/one-major-way-react-gt-vuejs-ba5c0332e75a (cit. on p. 18).

[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. 27).

[SBo20] SBoudrias. *Inquirer - NPM Library*. 2020. URL: https://www.npmjs.com/package/inquirer (cit. on p. 33).

[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. 17).

[Sco18] J. R. Scott. *Goodbye Redux*. 2018. URL: https://hackernoon.com/goodbye-redux-26e6a27b3a0b (cit. on p. 28).

[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. 17).

[Sim20] SimilarTech. *Top Framework Technologies*. 2020. URL: https://www.similartech.com/categories/framework (cit. on p. 18).

[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. 18).

[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. 25).

[Tuc19] J. Tucker. *Revisiting React Testing in 2019*. 2019. URL: https://codeburst.io/revisiting-react-testing-in-2019-ee72bb5346f4 (cit. on p. 24).

[Tya19] N. Tyagi. *Top Front End JavaScript Frameworks*. 2019. URL: https://www.lambdatest.com/blog/top-javascript-frameworks-for-2019 (cit. on p. 17).

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