Trabajo Fin de Grado en
Ingeniería de las Tecnologías de
Telecomunicación

Visualización del rendimiento en equipos de desarrollo Software a través de sistemas de control de versiones Git.

Autor: María Luz Aniceto Caba.

Tutor: Isabel Román Martínez.

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado en
Ingeniería de las Tecnologías de Telecomunicación

Visualización del rendimiento en equipos de desarrollo Software a través de sistemas de control de versiones Git.

Autor:
María Luz Aniceto Caba

Tutor:
Isabel Román Martínez
Profesor colaborador

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Visualización del rendimiento en equipos de desarrollo Software a través de sistemas de control de versiones Git.

Autor: María Luz Aniceto Caba

Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El secretario del Tribunal

A mi familia.

Agradecimientos

No puedo dejar este trabajo sin agradecer la infinita paciencia de todos los que me rodean: gracias por apoyarme y animarme cada vez que las fuerzas flaqueaban entre intento e intento. Gracias a mis padres que han estado apoyándome siempre.

Gracias a Jesús, Ale, Luis, Franxi y Pablo con los que he emprendido un camino de aventuras.

A mi compañera de batallas en la escuela.

A mí tutora, que siempre creyó en esta idea, que ha cuidado cada revisión y me ha animado en cada una.

Y al más paciente de todos Jesús, que tu curiosidad no deje de ilusionarte siempre.

Gracias.

Que a toda verdad demos la bienvenida.

María Luz Aniceto Caba

Sevilla, 2020

Resumen

La transformación digital está llevando a un crecimiento exponencial en la creación de empleo de desarrollo Software. Muchas empresas son las que, adaptándose al marco de las nuevas tecnologías, requieren de departamentos técnicos o servicios software externos. Nos encontramos por ello ante un interés por aplicar metodologías y herramientas que ayuden a los gestores de proyectos software a **optimizar** el proceso de desarrollo de forma continua. La calidad de este proceso debe ser medible y fiable.

En este trabajo se presenta el estudio y desarrollo de una aplicación web que permite conocer de forma visual y sencilla la calidad e impacto de las tareas realizadas en un equipo de desarrollo Software. De modo que los datos representados puedan ayudar a un equipo de desarrolladores a tomar mejores decisiones, detectar problemas, fomentar el uso de buenas prácticas…

Abstract

The digital transformation is leading to an exponential growth in the creation of software development jobs. Many companies are adapting to the framework of new technologies, require external technical departments or software services. There is a great interest in applying methodologies and tools that help project managers to optimize the process of development and to improve continuous improvement. The quality of this process must be measurable and reliable.

This paper presents the study and development of a web application that allows to know in a visual and simple way the quality and impact of the tasks performed in a software development team. So that the data represented can help a team of developers to make better decisions, detect problems, encourage the use of good practices ...

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE DIAGRAMAS

ÍNDICE DE CÓDIGOS

ÍNDICE DE CAPTURAS DE LA APLICACIÓN

1 INTRODUCCIÓN

Lo que no se mide, no se controla, y lo que no se controla, no se puede mejorar.

- Peter Drucker-

La transformación digital está llevando a un crecimiento exponencial en la creación de empleo de desarrollo Software. Muchas empresas son las que, adaptándose al marco de las nuevas tecnologías, requieren de departamentos técnicos o servicios software externos. Nos encontramos por ello ante un interés por aplicar metodologías y herramientas que ayuden a los gestores de proyectos software a **optimizar** el proceso de desarrollo de forma continua. La calidad de este proceso debe ser medible y fiable.

La **ingeniería del software** [1] estudia las metodologías para el desarrollo y mantenimiento de sistemas software. Los objetivos de este tipo de ingeniería son:

- Mejorar el diseño
- Promover mayor calidad al desarrollar aplicaciones complejas
- Detectar problemas y proponer mejoras
- Mejorar la rentabilidad al desarrollar un proyecto.
- Mejorar la organización de los equipos de trabajo.

Hay distintos modelos del proceso software que proponen diferentes aproximaciones para enfrentar las actividades implicadas en el proceso software con el fin de obtener un producto de calidad, como por ejemplo el modelo de proceso en Espiral:

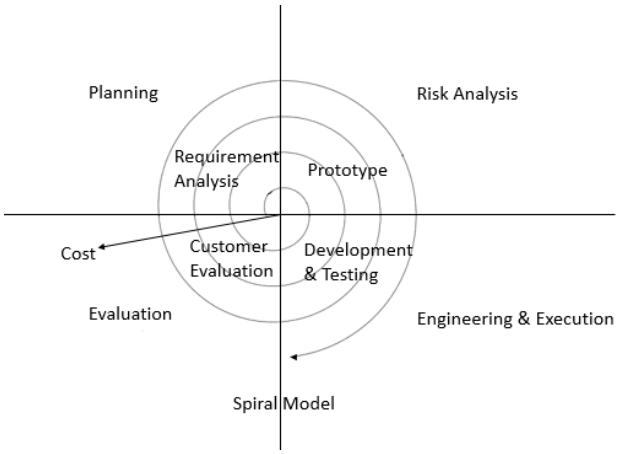


Figura 1: Modelo de proceso software en espiral

En cada una de las fases se pueden usar aplicaciones y herramientas que ayudan a los ingenieros a cumplir los objetivos de manera más fácil. Además de buscar la optimización del producto, la mejora continua del proceso es otro objetivo. Como *sin métricas no podemos mejorar,* necesitamos conocer datos y medidas del proceso software, para conseguir evaluarlo y mejorarlo.

En este proyecto queremos centrarnos en las fases de **evaluación, programación de pruebas y desarrollo** e intentar medir **cómo** se está ejecutando este proceso de una manera fiable. Así podremos tener mejor conocimiento sobre todo el flujo de trabajo.

1.1 Motivación

La etapa de implementación o desarrollo conlleva una verificación continua de la calidad del software y del propio proceso ¿Cómo podemos conocer cómo está trabajando nuestro equipo y cómo es la **calidad** de sus tareas? ¿Cómo sabemos lo bien que funciona un equipo?

La calidad del proceso debe ser medible, y de forma fiable. Uno de los parámetros que es muy importante para valorar la esta calidad es el rendimiento de los desarrolladores. Pero esto no es algo fácil de evaluar. El trabajo de desarrollo software conlleva la toma muchas decisiones que pueden tener efectos secundarios. Medir, exclusivamente, la cantidad de código que genera un desarrollador de forma numérica sería inapropiado.

Una de las métricas que más se valoran en la industria del software es la cantidad de **líneas de código añadidas** *(LoC: Lines of Code)* **por ingeniero en un día** usada, por ejemplo, para la medida del rendimiento en el desarrollo en Facebook [2]**.** No obstante, esta métrica genera los siguientes problemas:

- Se puede deformar fácilmente, un desarrollador puede añadir líneas extras para alcanzar números altos. Básicamente la conocida Ley de Campbell [3] que nos dice "que cuanto más usado sea un indicador cuantitativo para la toma de decisiones más probable será que se corrompa la medida de este indicador".
- El número de líneas escritas no es un parámetro fiable para medir la calidad de una tarea. Se pueden estar añadiendo líneas inútiles, que solo están complicando el código y provocando el alejamiento del principio KISS (En inglés: Keep It Simple, Stupid!).
- Copiar y pegar código evita que se reestructure el mismo y nuevamente se introduzca código inútil.

Esta métrica sólo es interesante si no la aislamos, es decir, si la ponemos en contexto con otros datos del trabajo de un desarrollador, como veremos más tarde.

Esto que ocurre con las líneas de código pasa con otros tipos de métricas, como contar fallos arreglados, tareas completadas, número de revisiones… Todas estas métricas son susceptibles de deformar la toma de decisiones al usarse de forma incorrecta.

En GitHub se nos muestra un mapa de calor sobre la actividad de una persona, y es una métrica fiable de la **cantidad de tareas realizadas,** pero no de la calidad de las mismas. La unión de calidad y cantidad nos daría un indicador más cercano a la realidad del trabajo de un desarrollador. Como se explicará más tarde: existen muchos factores importantes que no estamos evaluando.

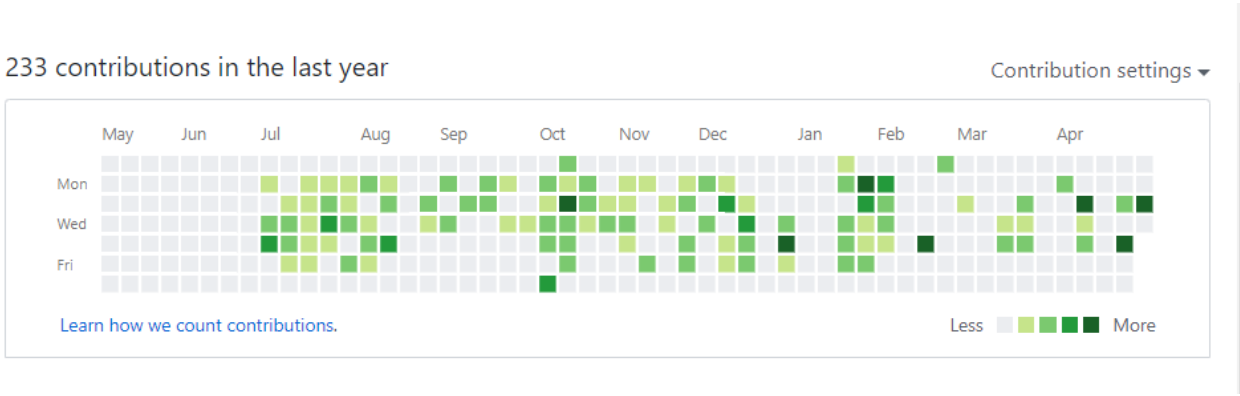


Figura 2: Mapa de calor de las contribuciones realizadas en GitHub

No valorar de forma adecuada el desempeño de los integrantes de un equipo y el progreso de un proyecto nos puede llevar a generar **deuda técnica.**

La deuda técnica es entendida en el sector como malas decisiones técnicas, código extremadamente complejo, falta de documentación… Se reconoce, entre los desarrolladores, que podría ser solventado con una buena metodología de trabajo [5].

La deuda técnica [4] se genera cuando se toman decisiones incorrectas, impidiendo en un futuro avanzar por tener que solucionar los defectos introducidos. Normalmente estas decisiones se toman por presión en las fechas de entrega o por no estar siguiendo una adecuada metodología de buenas prácticas.

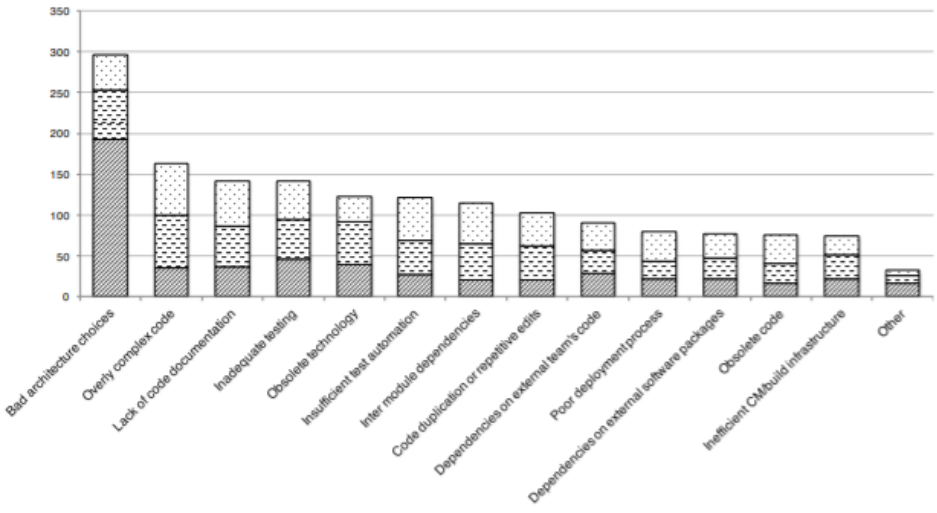


Figura 3: Ranking de causas de deuda técnica más comunes

La falta de datos fiables y con ello la deuda técnica, nos lleva a encontrarnos con problemas mayores como: no estar valorando el talento de tu equipo, no llegar a tiempo a fechas de entrega, estar construyendo un software de mala calidad, etc. Todo esto, acaba inexorablemente generando grandes pérdidas a las empresas.

1.2 Solución propuesta

Conociendo estos problemas, las métricas que deberíamos valorar serían unas que estén respaldadas en una buena metodología de trabajo, de tal forma que un gestor de proyectos pueda tener datos e información real con la que conocer mejor a su equipo y su proyecto. Si las métricas que tenemos se basan en buenas prácticas serán más útiles y fiables.

Como hemos visto, los datos cuantitativos no nos dicen nada eficaz de por sí, poder mostrar datos relevantes y darles contexto a los datos cuantitativos será misión de este proyecto.

En este trabajo se propone una solución para ayudar al gestor de un proyecto a tomar decisiones considerando información que le facilite la evaluación del desempeño de los miembros de un equipo de desarrollo y además conocer cómo se está desarrollando el proyecto en cuestión.

Se mostrará un marco teórico en el que se justifiquen mejores métricas para evaluar el rendimiento, respaldadas por literatura técnica y buenas prácticas.

Y como solución final se desarrolla una aplicación web que mediante datos de control de versiones git, estima la productividad y crea visualizaciones sencillas de entender con las que se pueden tomar mejores decisiones.

1.2.1 Objetivos

Los objetivos a los que se desea llegar en este proyecto son los siguientes:

Tabla 1: Objetivos *OBJ-01* - Desarrollar un algoritmo que estime el valor de una tarea

OBJ-01	Desarrollar un algoritmo que estime el valor de una tarea
Descripción:	Desarrollo de un algoritmo que a través de la API de GitHub consiga los datos necesarios y calcule de manera justificada el valor de cada tarea en un proyecto/repositorio.
Comentarios:	Este objetivo se apoya en un marco teórico para explicar las variables que se usarán para fundamentar las ponderaciones y los valores que se darán a cada tipo de tarea.

Tabla 2: Objetivos *OBJ-02* – Visualización General

OBJ-02	Visualización General
Descripción:	Visualización del trabajo total durante un proyecto.
Comentarios:	La misión de esta visualización será dar información sobre el desarrollo de un proyecto y en qué momentos se ha trabajado más o menos.

Tabla 3: Objetivo *OBJ-03* –Visualización Individual

OBJ-03	Visualización Individual
Descripción:	Visualización del trabajo individual de cada miembro del equipo en un proyecto
Comentarios:	Estará puesta en conjunto con todos los integrantes para poder comparar.

Tabla 4: Objetivo *OBJ-04* – Visualización del tiempo en terminar tareas

OBJ-04	Visualización del tiempo en terminar tareas
Descripción:	Visualización que muestre cuánto es el tiempo medio que se tarda en finalizar una tarea por usuario, para así poder comprobar si se están cumpliendo las metas del Proyecto.
Comentarios:	

Tabla 5: Objetivo *OBJ-05* – Visualización de ficheros más modificados

OBJ-05	Visualización ficheros más modificados
Descripción:	Muestra qué ficheros son los más modificados en el proyecto.
Comentarios:	Visualización para perfiles técnicos que permite detectar qué ficheros son más conflictivos.

2 MARCO TEÓRICO

Analizar y justificar las métricas con las que valoraremos el desempeño de un desarrollador es un pilar fundamental de este trabajo, ya que es la base para que un gestor de proyectos pueda obtener conclusiones sobre los integrantes de un equipo. En este apartado se detallan los estudios que se han seguido para evaluar la productividad de desarrolladores software. Además de una introducción al control de versiones Git y a la organización del trabajo basada en Pull Request.

2.1 Introducción

En primera instancia vamos a conocer en qué consiste git y GitHub. También se detallará la metodología de organización en equipos software basada en *Pull Request*.

Además, el OBJ.01 necesita un trabajo de investigación, una justificación teórica, que se llevará a cabo en el apartado 2.5.

2.2 Git

Git [6] es un sistema de control de versiones que ayuda y mejora el proceso de desarrollo software. Estos sistemas sirven para almacenar, recuperar, comparar y fusionar versiones del código fuente, pero guardan también una gran cantidad de información sobre otros aspectos que se pasan por alto. Obtener estos datos y poder visualizarlos será misión de este trabajo.

Git permite a cada uno de los colaboradores de un proyecto tener una copia local del mismo, y poder trabajar independientemente de si otras personas lo están haciendo a la vez.

Ayuda a etiquetar y separar las tareas, para trabajar en equipo de una forma ordenada y eficiente.
A continuación, dejamos unos conceptos básicos de Git que se usarán durante todo el proyecto y nos ayudarán a comprender mejor cómo funciona.

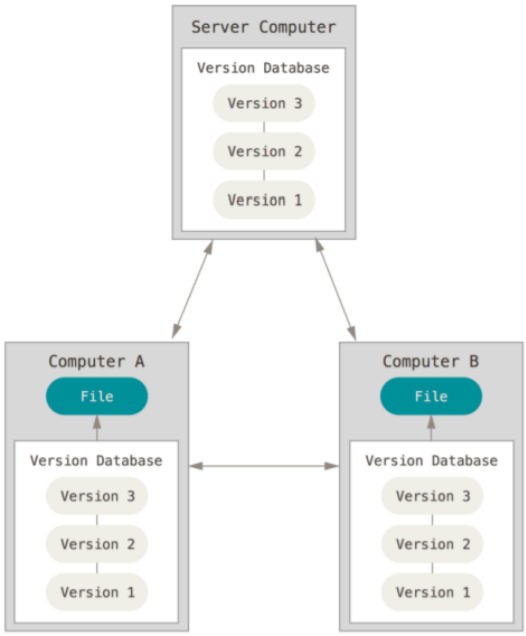


Figura 4: Esquema de control de versiones git.

- **Repositorio**: es el conjunto de versiones que conforman el proyecto en el que se está trabajando. El repositorio base desde donde todos los desarrolladores se descargan una copia. Este puede estar en la nube o en un servidor propio.
- **Commit**: cuando ya se han realizado los cambios necesarios en los ficheros, estos se confirman mediante el comando "commit". Cada commit crea una versión del código.
- **Rama**: Una rama es un conjunto de versiones etiquetadas con un nombre que hace referencia a los cambios o tareas en los que se centra. Una rama comienza desde un commit concreto. A partir de este punto creamos una ramificación donde están aislados los cambios que se hacen, entre otros beneficios, esto facilita la vuelta atrás si es necesario. En una rama puede haber varios commits. Las ramas permiten el trabajo en paralelo de varios desarrolladores en el mismo proyecto.

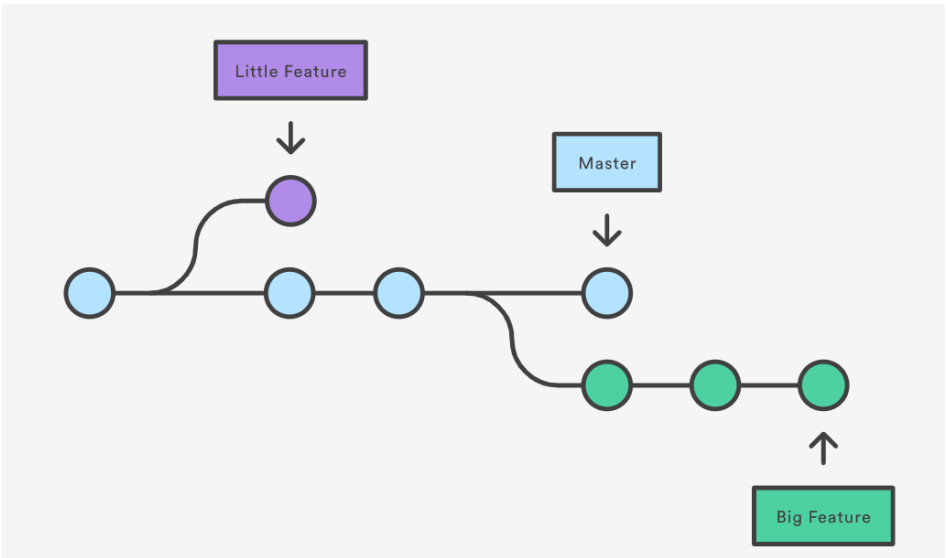


Figura 5: Representación gráfica de ramas (*líneas)* y commits (*círculos*)

- **Merge**: llamamos merge a fusionar versiones de varias ramas. Por lo general tendremos una rama donde estará la versión principal a la que se añaden los cambios, con el comando merge podemos fusionar una rama con la otra. Es decir, nos permite añadir los cambios de una versión a otra de distinta rama.
- **Push**: git es una herramienta que se usa en equipos de trabajo de más de una persona, por lo tanto, es normal que tengamos un repositorio remoto alojado en un servidor, con esta acción (push) podremos enviar los cambios que tenemos en local al repositorio remoto.

2.3 GitHub

GitHub es una plataforma de alojamiento de código para el control de versiones y colaboración a través de git. Actualmente es el más usado con más de 96 millones de repositorios alojados y más de 31 millones de desarrolladores usando esta plataforma [7].

2.4 Modelo de desarrollo "Pull-based"

2.4.1 Introducción

El modelo de desarrollo "pull-based" es muy usado en el mundo del desarrollo software [8].

Se popularizó años atrás en los repositorios de código abierto, ya que los métodos clásicos que se tenían para colaborar en estos proyectos eran abrir un ticket o pedir acceso directo al repositorio.

El modelo pull-based, resumidamente, permite tener una copia local del repositorio al se quiere contribuir, realizar tus cambios independientemente y luego pedir que tus cambios se incluyan en el repositorio general. La petición de incluir tus cambios en el repositorio general se llama **Pull Request** su traducción aproximada puede ser "petición de validación". Esta petición debe pasar una revisión antes de ser aceptada y que se cree una nueva versión, con la fusión de los cambios introducidos.

Desde 2011 el número de este tipo de colaboraciones en repositorios de código abierto ha ido creciendo casi exponencialmente, el año pasado, se hicieron más de 87 millones de Pull Requests en GitHub. [7].

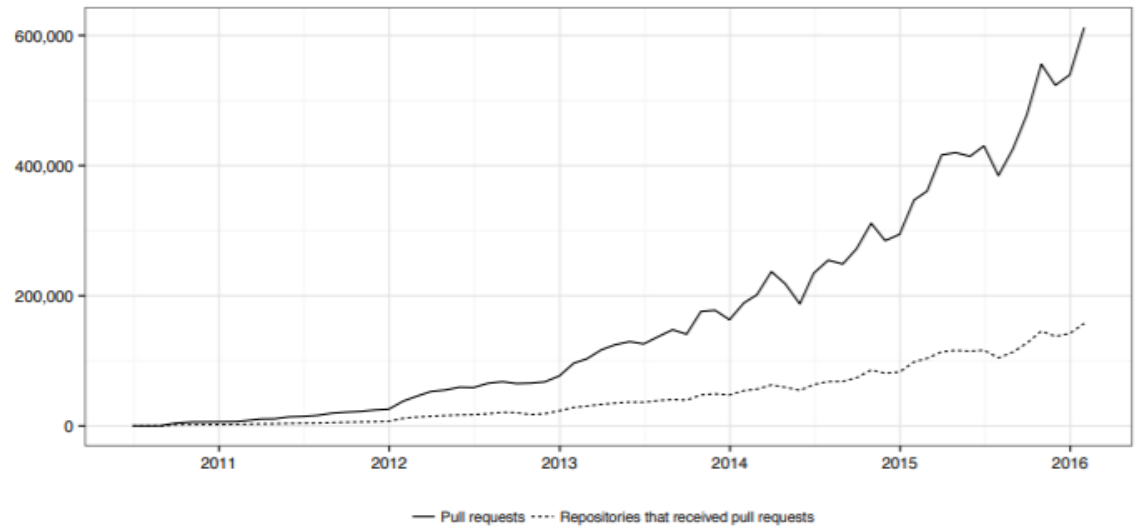


Figura 6: Crecimiento del uso de Pull Request en GitHub durante los años 2011 al 2016 [9]

2.4.2 Proceso en detalle

Partimos del escenario en el que tenemos un repositorio en GitHub y un equipo de desarrolladores que modifican el código usando como control de versiones git. Cada uno de los desarrolladores tiene una copia local en su ordenador del repositorio en la última versión que se esté manejando. Cada vez que un desarrollador quiere añadir una nueva versión o tarea al repositorio remoto se usan los Pull Request. Este proceso en detalle es el siguiente:

1. En primer lugar tendremos al desarrollador trabajando en su repositorio **local**. Tendrá creada una rama y sobre esta, estará realizando commits. es decir, está creando una nueva versión local del código.

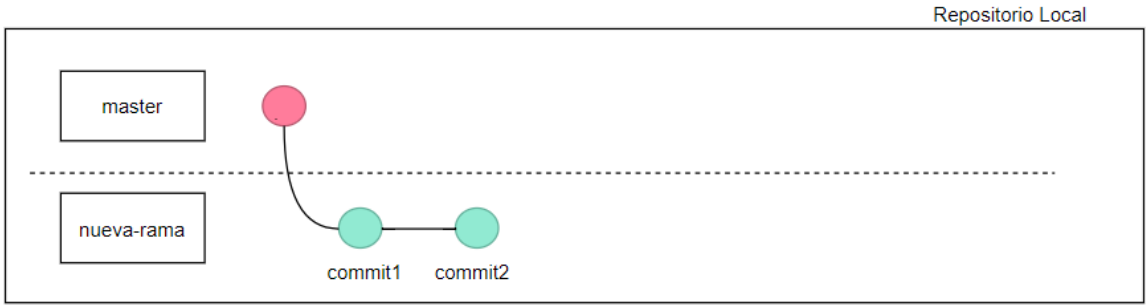


Figura 7: Representación gráfica de la creación de una nueva rama en git.

2. Cuando el desarrollador cree que la tarea, en su rama local, está lista para ser revisada debe incorporar la rama al repositorio remoto, para que su equipo pueda verla. En este caso el repositorio remoto está en GitHub.

Para ello se utiliza el comando push, que hemos visto antes. Esto hace que la *nueva-rama* esté ahora en el repositorio remoto.

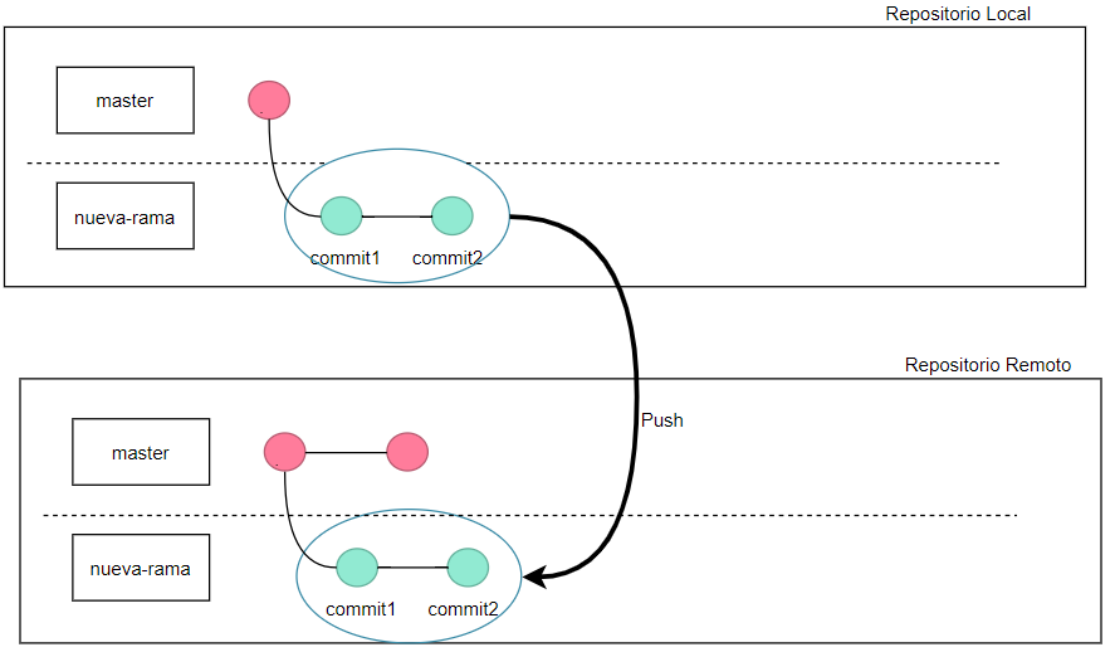


Figura 8: Representación gráfica del comando *push* en git

Ahora en el repositorio remoto de GitHub está la rama *nueva-rama.*

Pero todavía no está creada la petición de validación, esto se lleva a cabo a través de la interfaz de GitHub, el desarrollador deberá pulsar en la pestaña de Pull Requests y crear una petición nueva. El fin de un pull request es que se fusione esta nueva versión con una de las ramas remotas, por lo que la misma interfaz, en primer lugar,

te da a escoger entre las ramas disponibles con las que crear el pull request.

Como vemos en la imagen, se nos muestra *nueva-rama* en la parte superior. Los siguientes pasos son añadir título y descripción de la rama.

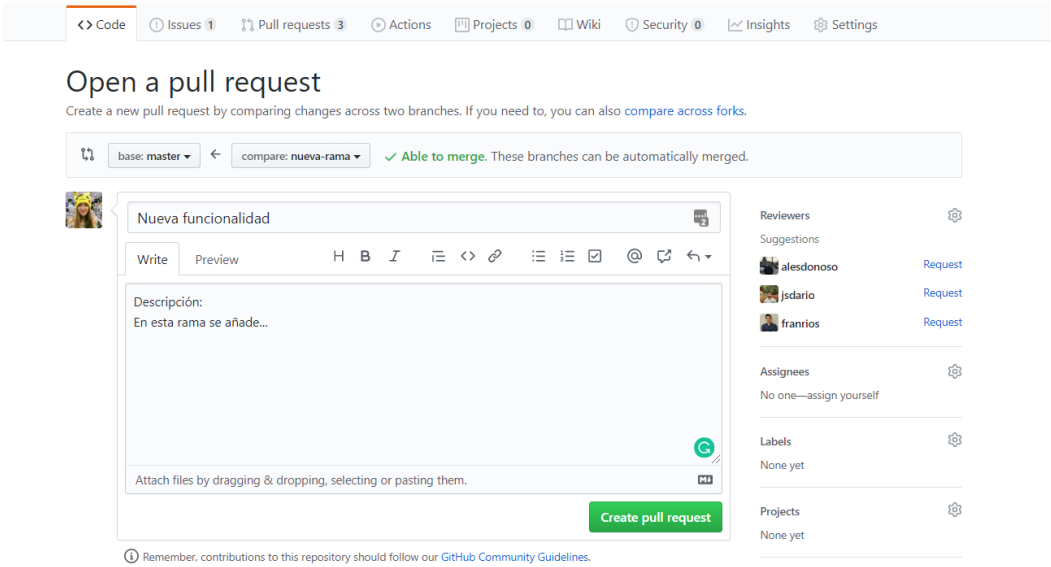


Figura 9: Interfaz gráfica de GitHub para la creación de un Pull Request

3. Otro desarrollador tendrá que revisar la tarea y comprobar que está correcta. Puede dejar comentarios, sugerencias de cambios, aprobar la tarea…

Puede, incluso, hacer commits a la rama *nueva-rama* ya que con el comando pull puede descargar la rama a su repositorio local y hacer cambios propios. Después sólo tiene que hacer push para incluirlos tanto en la rama remota como en el pull request, ya que la rama está vinculada y sincronizada con el Pull Request.

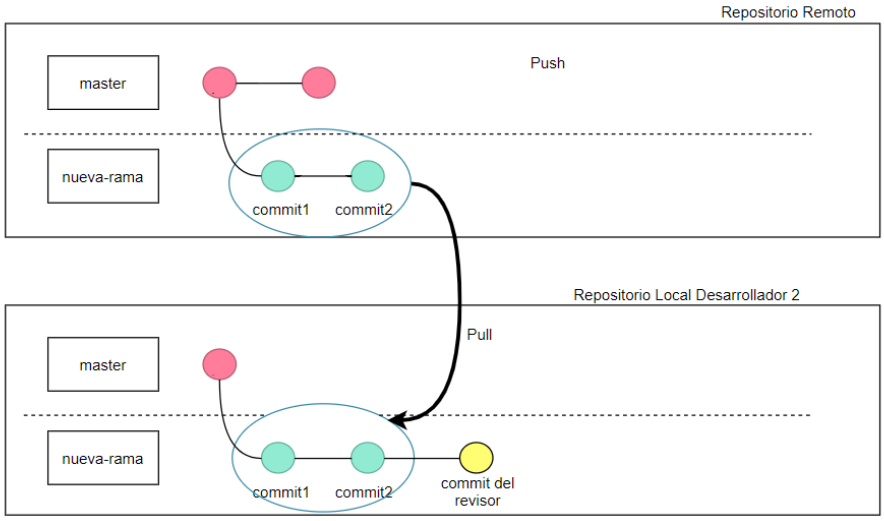


Figura 10: Representación gráfica del comando *pull* de git

4. Si todo está correcto la rama se fusiona con la rama remota, si se concluye que no se va a fusionar se cierra el Pull Request.

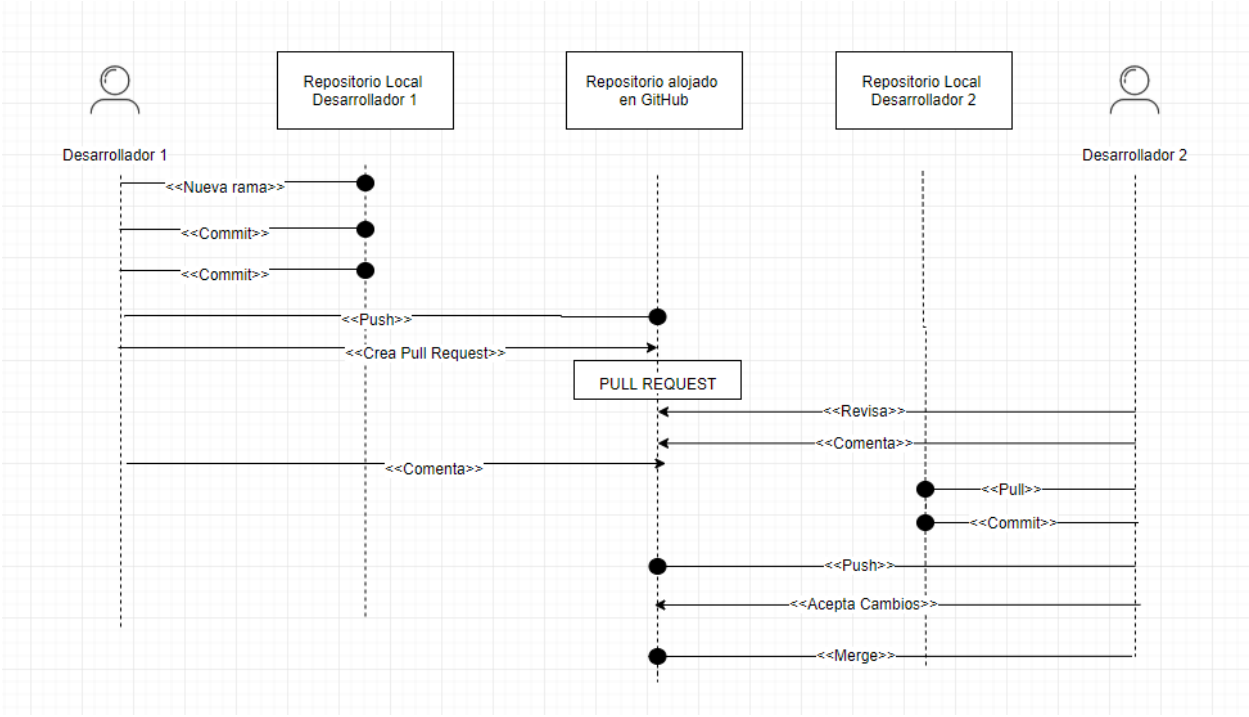


Figura 11: Resumen del proceso de trabajo basado en Pull Request [10]

Cabe destacar que al fusionarse la rama del pull request con la rama remota principal se puede hacer de dos formas diferentes:

- **Squash and merge**: todos los commits de la rama nueva se incluyen en la rama remota como un solo commit que tendrá como mensaje descriptivo el título del pull request.
- **Merge:** todos los commits de la rama se añaden a la rama remota.

2.5 Métricas destacables

Como bien hemos visto en la introducción, tener sólo datos cuantitativos no nos hace valorar justamente el trabajo de un desarrollador. A través de los datos que nos ofrece la API de GitHub estimaremos el impacto de una tarea o una revisión para poder mostrarlo en un gráfico.

Después de haber explicado el modelo pull based destacaremos tres unidades de trabajo básicas que podremos evaluar: pull request, revisión y revisión con commit. Vamos a elegir estas tres unidades de trabajo ya que son los elementos más destacables del método pull-based.

A continuación, detallaremos qué tipo de métricas se asocian a cada una de estas unidades de trabajo, y como podemos interpretarlos de la forma correcta. Para así ponderarlos y conseguir una puntuación más cercana al rendimiento real de un desarrollador en un proyecto.

2.5.1 Pull Requests

Esta unidad de trabajo se asocia a la persona que ha creado una petición de validación. Está caracterizado por:

Título, descripción, commits, líneas añadidas, líneas modificadas, ficheros modificados y finalmente si se ha

aprobado o no después de la revisión. Veamos qué impacto tienen estas variables en el valor final del pull request.

2.5.1.1 Título y descripción

Un pull request sin título o descripción no debería ser considerado de manera positiva. El trabajo en equipo es fundamental en el desarrollo software y no explicar o documentar una tarea influye a la hora de revisar el código, ya que el desarrollador que lo hace debe averiguar por su cuenta en qué consisten los cambios que el autor del pull request ha realizado.

En repositorios de código abierto un factor muy importante para que el pull request no se acepte y se fusione con la versión principal, es tener una descripción incompleta [11].

2.5.1.2 Commits

Un pull request, al ser una rama puede contener varios commits. Los commits tienen también un título, por lo que le pasa igual que al punto anterior. También, según un estudio, se puede vincular que los commits que no contienen un mensaje descriptivo están relacionados con fallos de compilación [12].

Además, se puede relacionar el tipo de tarea según el mensaje del commit, por palabras claves como error, añadir, arreglo, introducción…

2.5.1.3 Líneas y ficheros modificados

Estos datos como hemos visto antes, por sí solos, no nos dicen nada. Un pull Request que modifique muchos archivos añada y borre muchas líneas de código puede repercutir de forma negativa en el tiempo que se tardará en revisarlo y con ello en terminar la tarea.

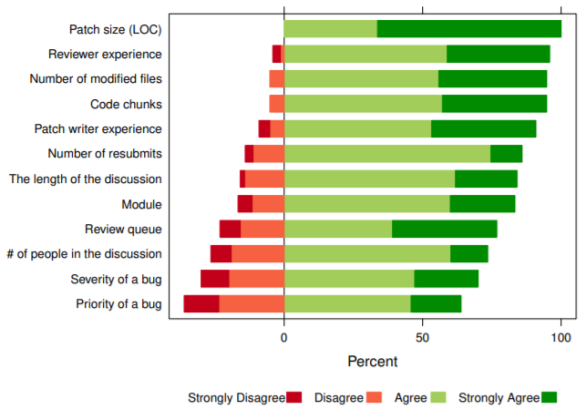


Figura 12: Factores que influyen en los tiempos de revisiones, en primer lugar, líneas modificadas.

En varios estudios [13] [14] sobre la experiencia de los desarrolladores de Mozilla [15] y Shopify Inc sobre cómo veían la revisión de código, se preguntó sobre cuales creían que eran los factores que más repercuten en el tiempo de revisión. Se obtuvo como factor más importante el volumen de líneas modificadas y en tercera posición en el ranking el número de ficheros modificados. Al mismo tiempo, también se les preguntó sobre qué elementos influyen en la calidad de las revisiones, determinando que el número de líneas era el segundo motivo que tenía un fuerte impacto en la calidad de éstas. Es decir, a mayor número de líneas que revisar, menor era la calidad de la revisión, y esto puede llevar a la introducción de fallos en el código.

Esto también se pudo ver en otro caso de estudio en los equipos de desarrollo de Cisco [16] se encontró que a medida que aumentaban las líneas de código que revisar, menos comentarios recibían y menos fallos se localizaban. A partir de las 400 líneas de código la calidad de la revisión disminuye considerablemente.

Por lo que podemos decir que los pull request que son más pequeños son revisados antes y con más atención que los que modifican muchas líneas y ficheros al mismo tiempo.

2.5.1.4 Si se ha fusionado o no el pull request con la rama principal

Si se ha aceptado o rechazado el trabajo de un desarrollador es un parámetro a tener en cuenta a la hora de evaluar el conjunto de tareas que haya realizado.

2.5.2 Revisión

La revisión de código en un equipo de desarrollo software es una buena práctica, hace que el equipo se comunique, se evitan fallos y se puede mejorar el código. Esta técnica está muy bien valorada entre los desarrolladores [16].

Es uno de los procesos imprescindibles en el modelo pull-based, y por eso es una de las unidades de trabajo que vamos a evaluar.

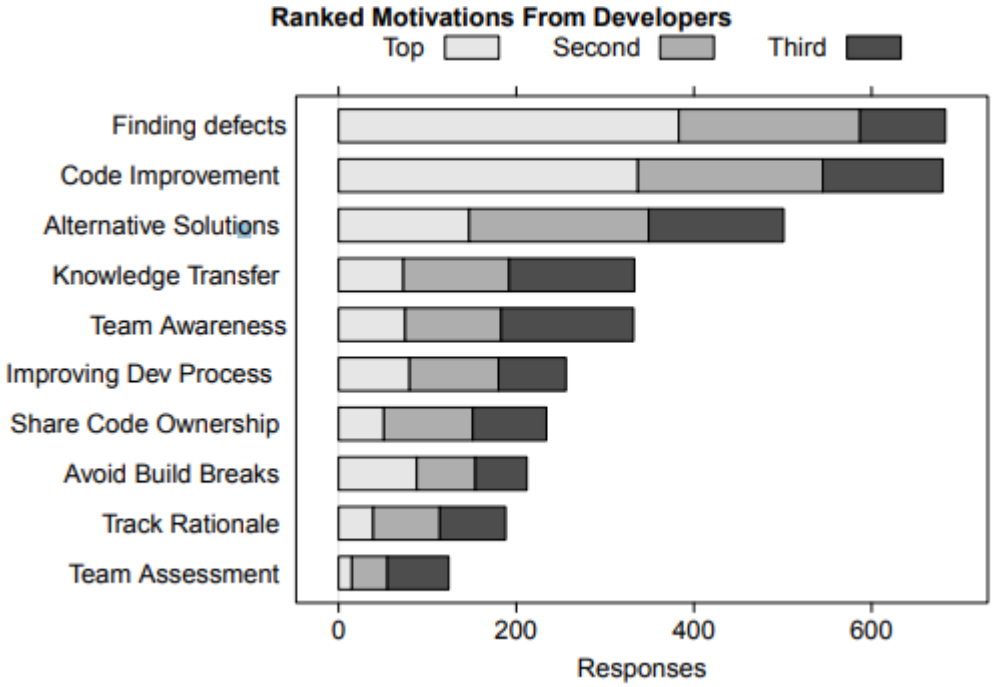


Figura 13: Motivación de los desarrolladores para hacer revisiones.

En los entornos de desarrollo actuales, la revisión de código debe ser un componente imprescindible en los procesos ya que nos aporta beneficios como: mejorar la calidad del código, mejorar la comunicación entre los desarrolladores, educar en buenas prácticas a los programadores que están empezando… Estos beneficios repercuten positivamente en un código más mantenible, tiempos más cortos de desarrollo y finalmente en ahorro de costes.

Poder mostrar si este proceso se está llevando a cabo podría ayudar a conocer mejor cómo se está trabajando, de manera que podamos fomentar una cultura de revisión en nuestros equipos.

GitHub nos da la opción de dejar un comentario en el pull request que estamos revisando.

El número de comentarios debe ser considerado en las métricas, ya que demuestra la implicación de un desarrollador a la hora de revisar un pull request.

Las revisiones se crean en la interfaz gráfica de GitHub y se pueden dejar también en líneas concretas del Pull Request.

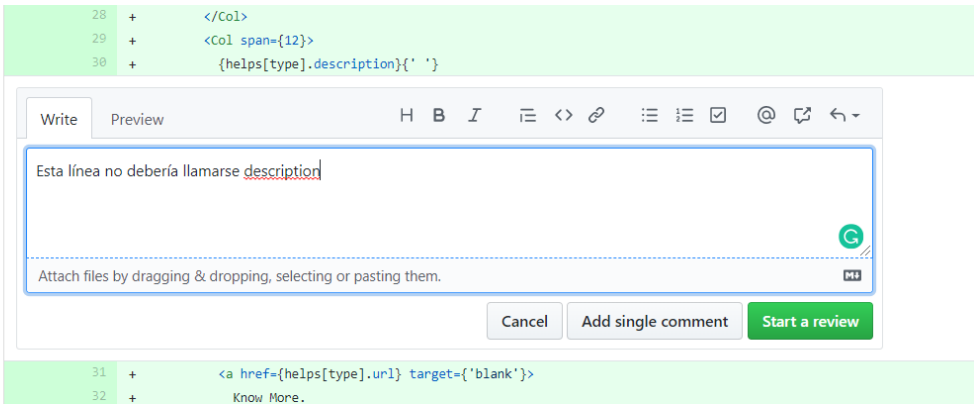


Figura 14: Interfaz gráfica de GitHub para dejar comentarios en líneas de código

Como dato, puede marcarse como aprobado un Pull Request sin dejar ningún comentario, esto no debería ser considerado una buena revisión.

2.5.3 Revisiones con commit

Se ha separado este tipo de revisión como unidad de trabajo ya que conlleva más implicación en la revisión que simplemente comentar en la interfaz de GitHub. Hemos visto en el proceso en detalle que el mismo revisor puede descargarse la rama en su repositorio local, hacer modificaciones y enviarlas al remoto, es decir, incluir commits propios en el Pull Request de otra persona como revisión.

Si una persona ha revisado la tarea y ha añadido código, podemos decir que ha entendido la tarea, y que se ha implicado en añadir más valor o corregir algún fallo del autor de esta. Es por esto por lo que se separa como una revisión especial y la llamamos “Revisiones con commit”, porque es muy interesante conocer qué personas en el equipo son comprometidas en la revisión.

En el commit podemos valorar algunos de los parámetros que ya vimos para pull requests, obtendremos su título, sus líneas y ficheros modificados.

3 ALGORITMO Y VISUALIZACIONES

3.1 Introducción

En este capítulo detallaremos las visualizaciones y métricas que se han propuesto en los objetivos. Es necesario también explicar el algoritmo que considerará las métricas destacables del apartado anterior y así justificar el cálculo que muestra el impacto de los pull requests.

3.2 Visualización General

La métrica principal que se obtiene del algoritmo explicado en este capítulo (*OBJ-01)* se muestra en una visualización (*OBJ-02*) que consiste en una sección por desarrollador que incluye círculos que representan las unidades de trabajo realizadas. Es una visualización que ayuda a ver en conjunto cuánto se ha hecho. Pero la cuestión está en representar también el impacto de cada una de esas unidades de trabajo. Para ello el tamaño del círculo representará el impacto. Este tamaño se calcula con un algoritmo que **considera las métricas que seguían buenas prácticas** explicadas en los apartados anteriores.

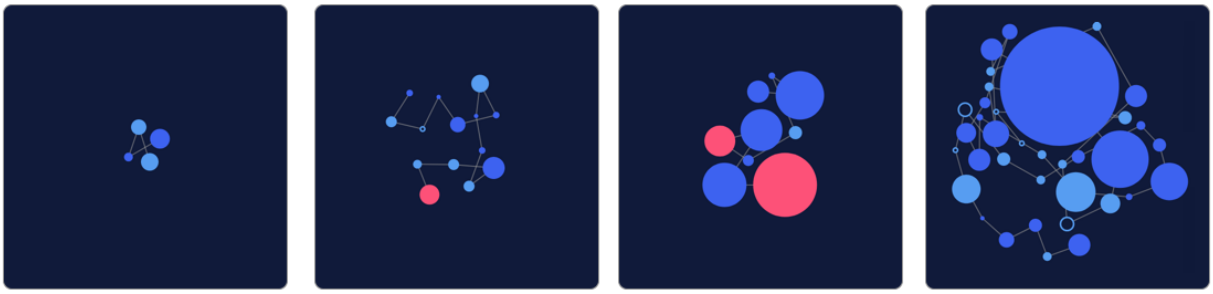


Figura 15: Ejemplo de visualización principal

En el ejemplo de la Figura 15 se muestra la evolución de un desarrollador en el que cada cuadrado representa un mes y los círculos unidades de trabajo. Se suele empezar con pocas contribuciones de poco valor y se acaba haciendo mucho más y de más valor.

En este apartado definiremos qué tipo de elemento representará cada unidad de trabajo y cómo calcularemos el tamaño de la circunferencia.

3.2.1 Cálculo del impacto

En primer lugar, vamos a estimar el tamaño del círculo que representará el impacto de un pull request. Queremos calcular una puntuación base a la que sumemos o restemos puntos considerando que contengan o no elementos de buenas prácticas vistos anteriormente.

Partiremos de una ecuación general que desarrollaremos en este apartado:

$$PesoTotal = F(modificaciones, archivos) + G(líneas_{añadidas}, líneas_{borradas}) + V$$

Cada una de estas variables significa:

- **Modificaciones**: la diferencia entre líneas borradas y líneas añadidas.
- **Archivos**: el número de ficheros modificados.
- **Líneas añadidas**: Líneas nuevas insertadas en el código
- **Líneas borradas**: líneas borradas en el código.

La primera función estima una puntuación inicial en función de los archivos y la diferencia entre líneas borradas y modificadas.

$$F(modificaciones, archivos) = \frac{archivos * modificaciones}{archivos + 1 + modificaciones}$$

Las **modificaciones** pueden valer cero en el caso que tengamos exactamente las mismas líneas añadidas que borradas, por ejemplo, en el caso de que hagamos un buscar y reemplazar en muchos ficheros, el número de modificaciones es el mismo siempre. Nos quedan otros valores en la función del calculo que suman puntos.

Con las métricas anteriores vemos que las líneas de código son un parámetro crítico ya que influye en la calidad y tiempo de revisión.

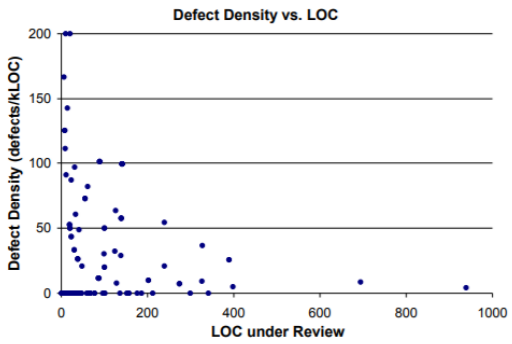


Figura 16: Densidad de fallos encontrados conforme aumenta LoC(*Líneas de código*)

Un número óptimo de líneas modificadas para tener una buena revisión estaría sobre las 250 líneas [16]. Este primer valor que obtendremos será a través de la variable **modificaciones**.

Cuando este número es enorme, el algoritmo en esta función debe valorarlos de manera logarítmica, en vez de escalar. Las modificaciones automáticas como buscar y reemplazar no aportan mucho valor. Por ejemplo, cambiar 2000 líneas se establecería en aproximadamente 8 puntos.

Para compararlo con este límite, 250, se lleva a cabo una simple comparación: si el número de modificaciones es mayor de 250 o el número de ficheros mayor de 100 se considera que debemos calcular el impacto de manera logarítmica. Lo que nos ayudaría a estabilizar la puntuación que se obtiene en esta función.

$$F(modificaciones, archivos) = \log\left(\frac{archivos * modificaciones}{archivos + 1 + modificaciones}\right)$$

La segunda función es una puntuación del número de líneas diferentes que hay una vez el pull request es fusionado. Es decir, el valor absoluto de la diferencia entre las líneas añadidas y las líneas borradas.

Esta puntuación será muy fácil de calcular y también se estimará con un logaritmo ya que es necesario que a mayor número de líneas modificadas crezca el impacto total de la tarea, pero sea continua en valores altos.

$$G(líneas_{añadidas}, líneas_{borradas}) = \log\left(|líneas_{añadidas} - líneas_{borradas}| + 1\right)$$

La última variable de la ecuación será una serie de puntos adicionales que se sumarán o restarán en función de que el pull request cumpla una serie de requisitos que hemos visto en el apartado *Métricas destacables*.

Tabla 6: Valores posibles de la variable V y su descripción

Variable	Descripción	No contiene	Contiene
Título Tarea	Si el pull request carece de título, o sólo tiene una palabra se considera una mala práctica.	-1	1
Descripción	Es una buena práctica que este campo no esté vacío. Ayuda al revisor a entender la tarea lo que conlleva que se agilice el proceso de revisión.	-1	1
Commit 'Merge Master'	Si en el commit se incluye 'merge master' simplemente significa que hemos añadido los cambios de la rama principal a la nuestra por lo que este commit no debería influir en el algoritmo	-1	0
Commits vacíos	Si contiene commits vacíos sin descripción	1	-1

El cálculo del impacto se ha descrito hasta aquí, ahora pasaremos a describir cómo se distinguen los tipos de trabajos en la visualización:

Tabla 7: Colores y significado de la visualización principal

Elemento	Descripción	Tipo
Círculo amarillo	Este elemento representa las revisiones del tipo comentario	Revisión
Círculo vacío	Este elemento significa que la persona en cuestión ha añadido un commit a una rama que no era suya	Revisión
Círculo Rosa	Este elemento representa una tarea que todavía no se ha cerrado o fusionado con la rama principal.	Pull Request
Círculo celeste	Este elemento representa una tarea que se ha finalizado.	Pull Request

Un ejemplo, de la visualización principal podría ser este:

Figura 17: visualización de ejemplo de un desarrollador.

Esta visualización ayuda a un gestor a determinar varios elementos que pueden ayudarle:

- Detectar las contribuciones de una persona de manera general de un simple vistazo
- Detectar qué perfiles son más proactivos en la revisión o qué perfiles menos.
- Determinar la magnitud de las contribuciones que se están realizando, gracias a los tamaños de los círculos.
- Motivar a los desarrolladores con las visualizaciones ya que se han pensado con este aspecto para que sean más amigables.
- Determinar si se están haciendo contribuciones muy grandes y se deben hacer tareas más pequeñas.

- Determinar la evolución de un desarrollador.

Ejemplo: Una representación de un proyecto real de código abierto: Stripe, una famosa plataforma de pago online.

La visualización nos muestra un cuadrado por desarrollador, de un solo vistazo podemos conocer la dinámica de este equipo: vemos que solo el usuario *remi-stripe* es el que hace pull requests los otros dos desarrolladores se enfocan más en revisar las tareas. Probablemente, el usuario con un solo pull request habrá hecho una contribución puntual, esto es debido a que el proyecto que estamos analizando es de código abierto y cualquier persona podría crear un pull request.

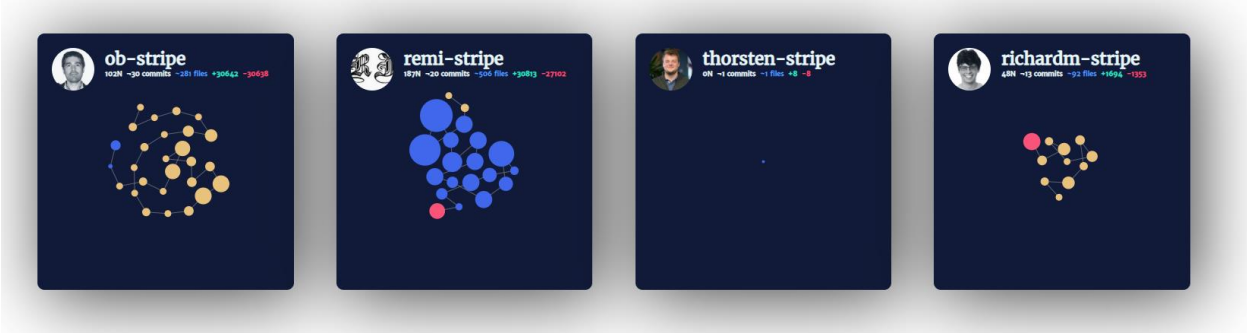


Figura 18: Varios desarrolladores y sus contribuciones a un proyecto de código abierto

3.3 Visualización: tiempo en terminar las tareas

Esta visualización tendrá dos modos:

- Tiempo desde que se abre un pull request hasta que se cierra o se fusiona con la rama principal.
- Tiempo desde que se empieza una revisión hasta que se fusiona o cierra el pull request.

Los objetivos de esta visualización son los siguientes:

- Mostrar qué tareas están tardando más tiempo en ser finalizadas. De forma que podamos detectar si tal vez es necesario separar tareas muy grandes en tareas pequeñas, o si un desarrollador está desmotivado o tiene problemas con esa tarea en concreto.
- Mostrar qué revisiones tardan más en concluirse, esto puede significar que no se aplican los cambios que se piden en la revisión o que se están haciendo Pull Requests demasiado grandes y tediosos de revisar.
- Enseñar qué tareas pueden estar siendo conflictivas en los ciclos de revisión.

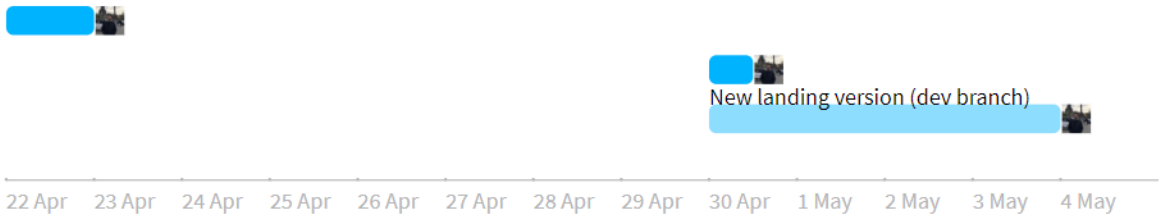


Figura 19: Visualización que muestra cuanto se ha tardado en realizar tres pull requests

Las visualizaciones son dinámicas por lo que en el ejemplo de la Figura 19 veremos los títulos de las tareas al

pasar el ratón por encima del rango de tiempo.

3.4 Visualización: Ficheros más modificados

Esta visualización muestra los ficheros del código que más se modifican, con el objetivo de conocer mejor qué puede estar siendo conflictivo en nuestro código.

Las ventajas de esta visualización son:

- Conocer mejor qué zonas del código se modifican con más frecuencia.
- Detectar qué zonas del código no se atienden adecuadamente. Por ejemplo: si los ficheros de test no se están modificando, pero sí ficheros de funciones, significa que se están añadiendo funcionalidades que no están siendo probadas.
- Detectar problemas y ficheros conflictivos.

La visualización que se ha elegido para estos datos es en forma de árbol circular, de todos los directorios del proyecto hasta los ficheros finales. Los caminos grises significan que no se han modificado, los verdes que se han modificado y los rojos los que más modificaciones tienen.

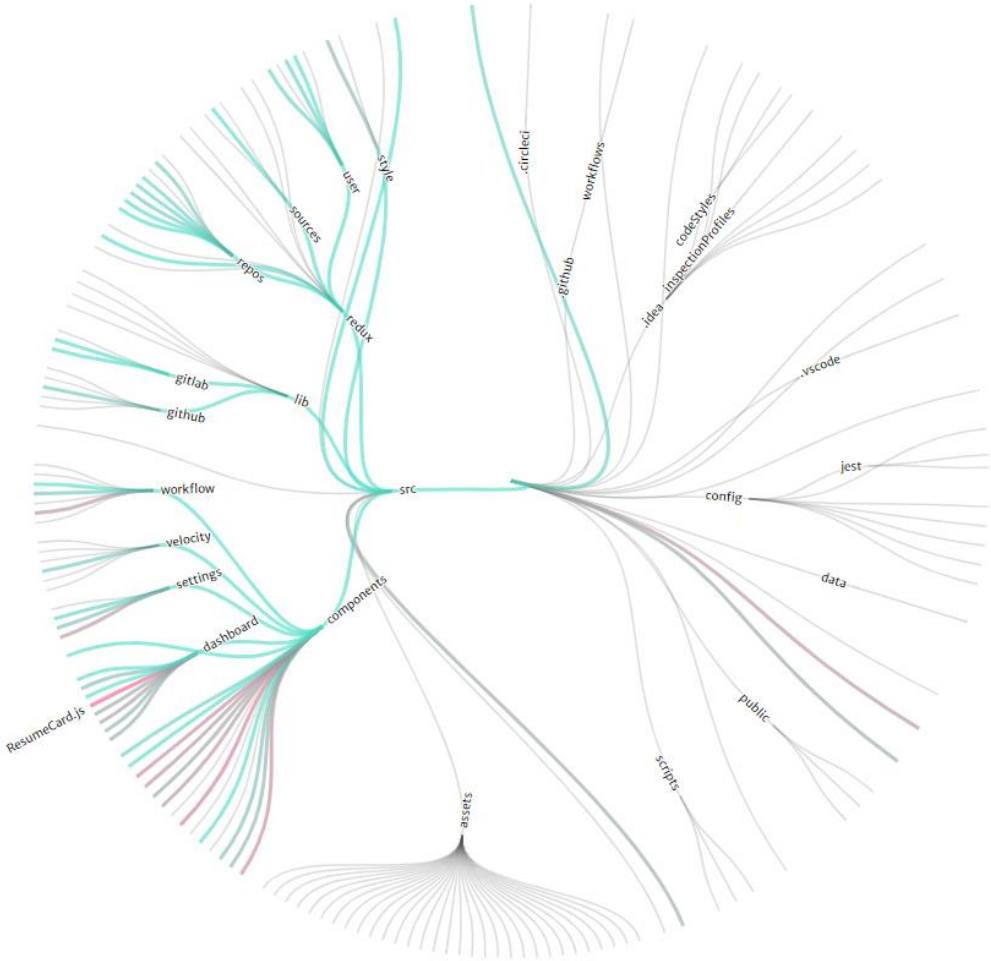


Figura 20: Visualización de ficheros más modificados.

En la imagen podemos ver diferentes colores como las líneas rosas representando los ficheros más

modificados, como por ejemplo el fichero *ResumeCard.js*.

Al igual que la visualización anterior, esta será dinámica y mostrará los títulos de los ficheros al pasar el ratón por encima de las líneas.

4 TECNOLOGÍAS USADAS

En esta sección se describen de forma breve las tecnologías utilizadas para la implementación y desarrollo de la aplicación web.

4.1 Entorno de desarrollo

Las herramientas y los servicios que se han utilizado para el desarrollo e implementación del código son los siguientes.

4.1.1 Javascript

Es un lenguaje de programación interpretado, esto significa que no requiere de compilación si no que es ejecutado por otro programa en este caso el navegador. Javascript está diseñado para ejecutar código en el lado cliente, pero existen formas de ejecutarlo en el servidor.

Como características podemos destacar que es un lenguaje prototipado y de objetos, es decir que no tiene clases y el sistema de herencias no funciona igual.

Javascript cuenta con una gran comunidad y muchas librerías que simplifican funcionalidades que serían muy tediosas de implementar desde cero. En este proyecto se usan varias librerías de las que destacaremos React y Redux.

Para poder ejecutar el código de JavaScript en este proyecto hemos usado Node.js, de esta forma sí que podemos compilar el proyecto y subirlo a un servidor que lo sirva al navegador.

4.1.1 Node.js

Cómo JavaScript era un lenguaje exclusivo del lado cliente y por tanto solo podía ser ejecutado por un navegador en 2009 [18] se anunció Node.js que permitía ejecutar código JavaScript fuera de un navegador. Se basa en el motor Javascript V8 que desarrolló Google, que es independiente a los navegadores.

Node.js Entorno de ejecución desarrollado con Javascript [21], es de gran utilidad en la creación de aplicaciones escalables, como servidores. Cabe destacar entre sus características que es asíncrono y consigue así tener un gran rendimiento.

Otro dato relevante de esta tecnología es que tiene su propio gestor de librerías npm (*Node Package Manager*). Este es el entorno de librerías más grande del mundo y mediante este podemos descargar librerías, gestionar las

dependencias…

4.1.2 React

React [18] es una librería de Javascript que nos permite desarrollar aplicaciones web de una manera sencilla. Nació por necesidades específicas de Facebook, al ser Open Source el proyecto ha crecido considerablemente, y es muy popular actualmente. Plataformas como Instagram, Dropbox o Netflix están hechas en React. Además, muchos contribuidores crean librerías con diferentes funcionalidades compatibles con React. Una de sus grandes ventajas es su rendimiento y que puede usarse tanto en el lado servidor como en el lado cliente.

React crea interfaces de usuario de forma eficiente usando un estilo declarativo. El código declarativo describe *qué* hacer, en vez de *cómo* hacerlo. Esto significa que la interfaz reacciona de manera automática ante los cambios del estado de la aplicación. El estado de la aplicación contiene los datos y variables que la construyen. Podemos decir que es una instantánea de la aplicación en cierto momento. React tiene un algoritmo que determina las diferencias en el estado y actualiza lo necesario.

Los componentes proporcionan interfaces reusables que permiten separar el código de la aplicación en bloques que actúan independientes unos de otros. Estos componentes tienen dos parámetros imprescindibles las propiedades y el estado. El estado es local al componente, pero podemos organizar el código de forma jerárquica de manera que el estado se pase por propiedades de manera unidireccional hacia los componentes inferiores.

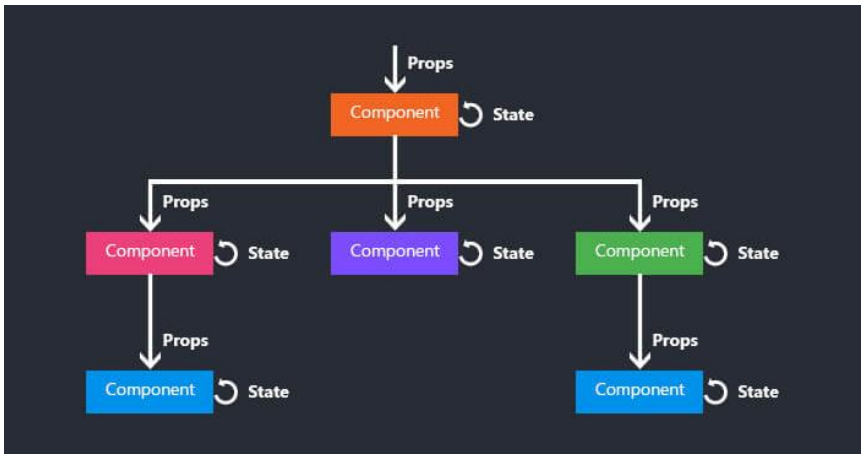


Figura 21: Diagrama jerárquico del flujo de las propiedades a través de los componentes en React.

4.1.3 Redux

React funciona con componentes de forma jerárquica, como hemos visto en el punto anterior, esto repercute en tener que estar pasando propiedades y variables de un componente a otro. Aquí es donde entra en juego Redux:

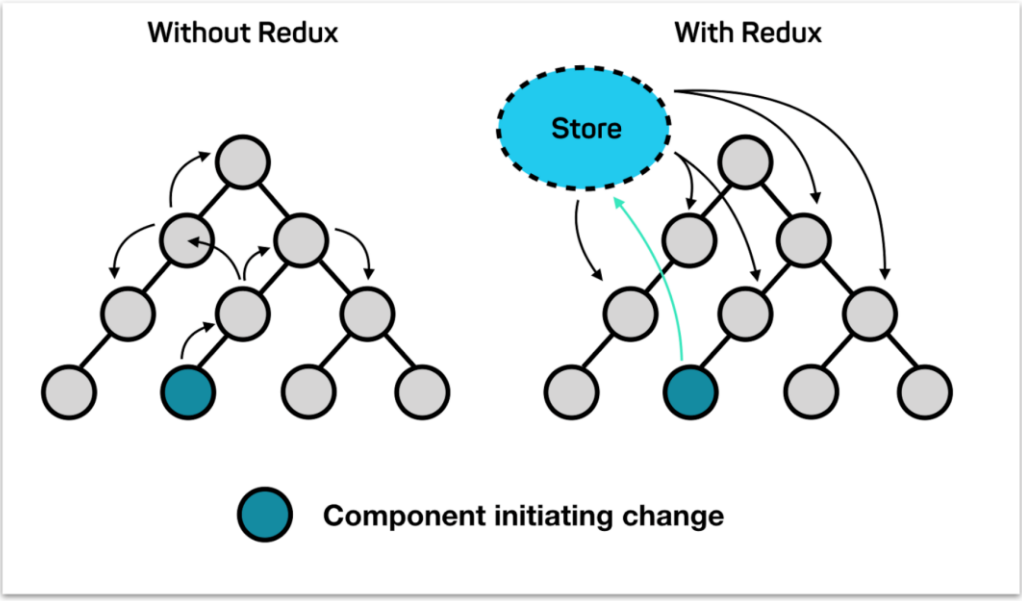


Figura 22: Paso de propiedades por componentes usando Redux vs sin esta herramienta.

En la Figura 22 podemos ver que si no tenemos Redux hay que pasar las propiedades de componente a componente si queremos acceder a alguna de ellas. Si queremos pasar una propiedad de un componente a otro que no estén directamente unidos en padre e hijo, tendríamos que dar dos saltos y pasar la información por todos los componentes, esta práctica no se recomienda en la documentación de React ya que es una mala práctica y muy difícil de mantener. Redux ayuda a el manejo de la información entre componentes.

Con Redux tenemos un almacén global, conocido en inglés como ***Store,*** donde guardamos todas estas propiedades, es decir, el estado de la aplicación. De esta forma al definir un nuevo componente podemos subscribirlo al Store y llamar a las funciones de Redux que nos permitan interactuar con el Store global. En la Figura 22 al tener Redux, si se modifica la Store se actualizan los componentes subscritos a él de manera automática y no hay que preocuparse de pasar en cascada los datos de unos a otros.

Redux es un contenedor del estado [24] de aplicaciones JavaScript, nos ayuda a tener un estado consistente de la aplicación gracias a este estado global, somos capaces de leer estas variables y propiedades desde todos los componentes del código. Redux se apoya en tres conceptos básicos:

- **Store**: es de tipo objeto, donde se encuentra almacenado el árbol de estado de la aplicación. Tiene varios métodos, los que más usaremos serán `getState()` y `dispatch(action)`.

  En un principio necesitamos declarar un store inicial con los valores por defecto.

```
const initialState = {
     objeto: {
        atributo1: null,
        atributo2: null
     }
}
```

Código 1: Ejemplo de declaración del Store inicial.

- **Acciones:** bloque de información que contiene, como mínimo, el tipo de acción también puede contener información extra que se necesite. Son objetos planos de JavaScript. Esta información se envía al Store mediante la función `store.dispatch(accion)`

```
store.dispatch({
     type: "TIPO_ACCION",
     atributo1: "valor1",
      atributo2: "valor2"
})
```

Código 2: Ejemplo del envío de una acción al Store.

- **Reductores:** son funciones que permiten definir cómo devolverá el estado cuando una acción se ha lanzado.

  Una propiedad del estado de Redux es la inmutabilidad, es decir que no se puede modificar o cambiar. Por esta razón los reductores **no** están modificando el estado si no que están creando un estado nuevo y devolviéndolo. Es por eso por lo que en cada una de las opciones del código se devuelve *state* que contiene el estado anterior. Los puntos suspensivos son un operador de propagación de JavaScript que nos permite hacer una fusión entre el estado antiguo y los nuevos atributos.

```
function reducer (state, action) {
   switch(action.type) {
     case 'AÑADIR_ATRIBUTOS:
       return {…state,
       atributo1: action.valor1,
       atributo2: action.valor2 },

     case 'BORRAR_ ATRIBUTOS:
       return {…state,
       atributo1: null,
       atributo2: null},



     default:
       return state;
  }
}
```

Código 3: Ejemplo de función reductor que recibe como parámetros el estado y la acción lanzada.

En nuestra aplicación Redux nos ayudará a compartir entre componente los datos que usaremos para llevar a cabo los cálculos del algoritmo además de propiedades que les pasaremos a los componentes.

De forma que el esquema resumen del funcionamiento de React + Redux es el siguiente:

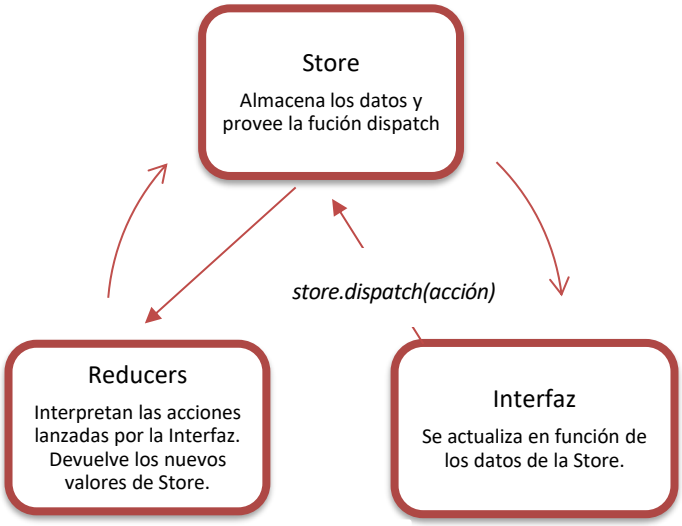


Figura 23: React + Redux esquema resumen

4.1.4 D3.js

Para crear las visualizaciones hemos usado D3 [20]: también es una librería de JavaScript que permite crear gráficos basándose en HTML, CSS y SVG. Es una de las más populares actualmente. También es de código libre, haciendo así mucho más fácil encontrar respaldo de una comunidad que mantiene esta librería.

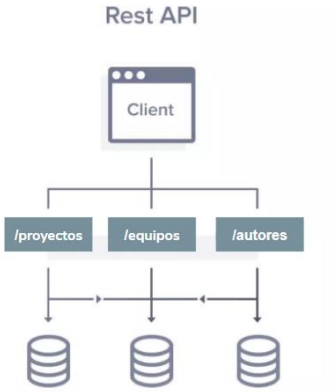
4.1.5 API GitHub + GraphQL

API, *Application Program Interface* o *interfaz de programación de aplicaciones* en español, define el conjunto de funciones, conectores y reglas que un desarrollador debe seguir si quiere interactuar con los datos y funciones de otro software. Se ofrece como una capa de abstracción, de forma que simplifica el trabajo de programación.

La API de Github [22] permite obtener la información sobre el trabajo de los desarrolladores en un repositorio. GitHub ofrecía un modelo API REST y migró al modelo GraphQL en 2016 [24]

GraphQL es una nueva forma de ver las API, para explicarlo podemos separarlo en dos secciones:

- El lado **cliente**: usa un lenguaje de consultas [23], con el que podemos hacer consultas específicas de datos. Esto comúnmente se ha visto en consultas SQL, pero siempre en el lado del servidor, con esta tecnología podemos hacer consultas desde el lado del cliente.
- El lado **servidor**: entorno de ejecución que procesará esas consultas y devolverá los datos, en nuestro caso es GitHub. GraphQL es completamente independiente del lenguaje y base de datos usados, puede ser una base de datos SQL, o no relacional, es más podría ser una API REST. Con respecto a los lenguajes, es compatible con JavaScript, PHP…



La principal diferencia que tenemos con un API REST es que el cliente tiene muy limitados los parámetros de la petición, las opciones son muy rígidas, mientras que en GraphQL, al tener un lenguaje de consultas en el cliente, se puede pedir más información con mucha más flexibilidad. En GraphQL podríamos hacer *Joins* de la información en una sola petición, mientras que en REST tendríamos que unir nosotros la información de varias peticiones.

Por ejemplo, en la Figura 24 si quisiéramos obtener la información de un autor y la información sobre el equipo al que está asociado, el cliente necesita hacer dos peticiones: primero pediríamos el autor y con el identificador del equipo al que pertenece, después pediríamos la información del equipo.

Figura 24: Modelo API REST de GitHub

GraphQL usa un sistema de tipos para poder construir la estructura de la información que ofrece el servidor.

- **Queries:** Representan la petición de datos.
- **Mutations:** permiten modificar, borrar o actualizar datos de la base de datos del servidor.
- **Subscriptions:** Son eventos en tiempo.

En este proyecto solo utilizaremos ***Queries*** ya que lo que queremos es consultar la información, es decir hacer peticiones de datos.

A este conjunto, queries, mutations y subscriptions, se le llaman Schema, define al cliente cómo debe consultar la información. Cuando un cliente se quiere conectar a ese Schema para hacer peticiones estará definido cómo debe hacerlo en la documentación de la API. Se puede decir que el Schema es el modelo de datos usado en la interfaz cliente servidor.

Los tipos, definidos anteriormente, se conectan con los *Resolvers* que se conectan con el gestor de datos que se tenga, que puede ser SQL, no relaciona, GraphQL, api rest… estos *Resolvers* son funciones que usan el Schema y que devuelven al cliente los datos pedidos en la consulta.

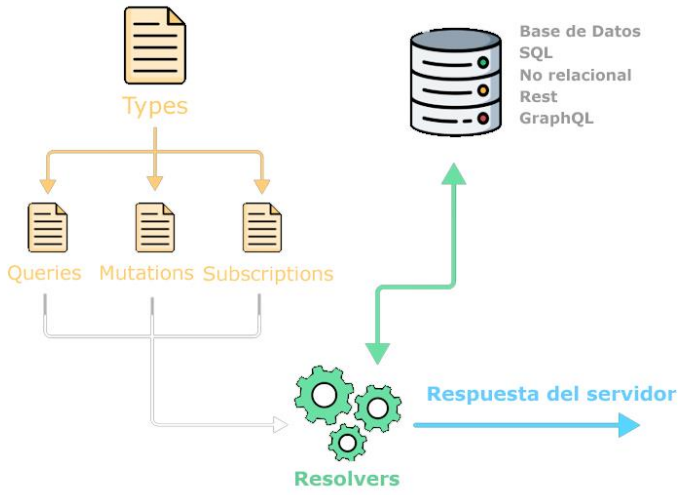
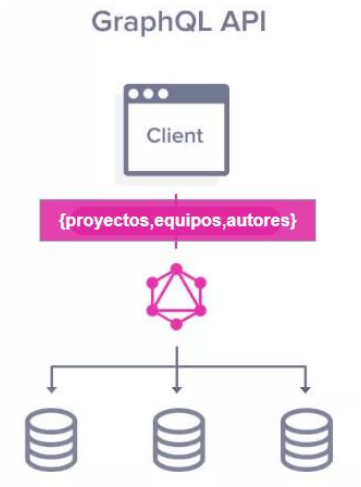


Figura 25: Funcionamiento del servidor que ofrece una API GraphQL



Por lo tanto, al poder el cliente construir la petición con todos los datos necesarios se ahorra conexiones con el servidor.

En el ejemplo anterior, si usamos GraphQL, el cliente sólo tiene que hacer una petición. Sólo tiene que construir la consulta en función del Schema que se le ofrece y el servidor le contestará con toda la información que ha pedido.

En nuestro caso es una gran ventaja ya que necesitamos consultar muchos datos. GraphQL nos ayuda a simplificar las peticiones y poder recibir la información agregada.

Figura 26: Modelo GraphQL de GitHub

```
query MyQuery {
  user(login: "Maluzzz") {
    id
    avatarUrl
    login
    repositories(last: 10) {
      edges {
        node {
          nameWithOwner
        }
      }
    }
  }
}
```


```
{
  "data": {
    "user": {
      "id": "MDQ6VXNlcjI0Nzg5NTk0",
      "avatarUrl":
"https://avatars2.githubusercontent.com/u/24789594?
u=4ec74d4c54324cc4d710724a2d649b7418c1ebfe&v=4",
      "login": "Maluzzz",
      "repositories": {
        "edges": [
          {
            "node": {
              "nameWithOwner": "Maluzzz/blog"
            }
```


Figura 27: Ejemplo de consulta y resultado en GraphQL a la API de GitHub

En la Figura 27 podemos ver una petición donde según el autor pedimos su id, su avatar y su nombre de usuario además pedimos sus diez últimos repositorios y un dato de estos repositorios. Como resultado nos da en formato JSON toda esta información unificada.

4.1.6 Jest

Jest [19] es una librería más de JavaScript que nos permite crear tests con los que verificar que el código se ejecuta de manera correcta.

Su sintaxis es sencilla y se implementa a través de scripts en JavaScript. Se puede testear tanto la interfaz como las funciones. Se ejecuta a través de la terminal de comandos. Con esta herramienta se pueden crear test unitarios que comprueben los árboles de React y verificar si el estado de la aplicación es el esperado.

Jest provee al usuario una sintaxis muy intuitiva, con la que construir los tests. En su documentación encontramos todos los métodos que podemos usar para codificar el test. Los nombres de las funciones que comprueban son muy intuitivos.

Para entender mejor el funcionamiento de esta tecnología y cómo la vamos a usar en este proyecto vamos a explicarlo con un ejemplo[19] :

Suponemos que nuestra aplicación tiene una función llamada **suma** que acepta dos parámetros, los sumandos, y devuelve el resultado de la suma. Tal como vemos en el siguiente código:

```
function suma(a, b) {
  return a + b;
}
module.exports = suma;
```

Código 4: Definición de una función suma simple.

Si queremos crear una prueba unitaria para comprobar que la función ***suma*** se comporta de manera adecuada, crearemos un test en el archivo **suma.test.js** que verificará esta función con Jest. Para ello seguiremos los siguientes pasos:

1) Para poder usar la función suma importamos el archivo donde se encuentra la declaración de la función a probar.

```
const suma = require('./suma');
```

Código 5: Importación del archivo donde se encuentra la función *suma*

2) Usamos las funciones *test, expect y toBe* proporcionadas por Jest. En primer lugar, *test* acepta dos parámetros: una cadena de caracteres para describir el test y una función, que, en nuestro ejemplo, llamará a la función *suma* y comprobará el resultado.

```
test('Sumar 1 + 2 debe ser igual a 3', () => {
 expect(suma(1, 2)).toBe(3); });
```

Código 6: Declaración del test para la función *suma*

Para declarar la función que comprobará *suma* usamos la función *expect* que acepta un parámetro, la función que debe ejecutar el test. Después usamos *toBe,* esta función recibe el resultado de *suma* y compara con el resultado que hemos declarado, en este caso 3.

3) Al ejecutar el script suma.test.js con Jest el resultado que debe darnos como salida en la terminal de comandos debe ser el siguiente

```
PASS  ./suma.test.js
✓ sumar 1 + 2 debe ser igual a 3 (5ms)
```

Código 7: Resultado correcto de la ejecución del test para la función suma.

Si el test no hubiera funcionado nos saldría error en rojo.

En este trabajo esta tecnología nos será de gran ayuda en la etapa de pruebas, para ejecutar tests unitarios de manera automática y probar que las funciones que hemos programado funcionan de manera adecuada, además también nos servirá para comprobar que Redux contiene los valores correctos después de lanzar una determinada función.

4.1.7 SDK de Autentificación y Autorización de Firebase

Es un servicio gratuito de Google [25] que nos sirve para llevar a cabo la integración de la autentificación y autorización de GitHub. La autentificación identifica al usuario y la autorización determina qué puede hacer. Necesitamos identificar al usuario de GitHub y determinar los permisos que tiene sobre la api.

Esto permitirá a los usuarios acceder a la aplicación web mediante su cuenta de GitHub. Google proporciona una serie de funciones que tienen ventajas para el desarrollo de la aplicación y para el usuario final:

- Es más seguro usar un servicio de autentificación externo, ya que este servicio brinda de manera sencilla y segura los métodos necesarios para gestionar usuarios. Evitando que tengan que desarrollarse métodos propios de seguridad desde cero.
- Para el usuario final la experiencia de usuario es mucho más rápida y sencilla, no necesita crear una cuenta con una nueva contraseña ni rellenar formularios de registro. Solo tiene que dar acceso a su cuenta y aceptar los permisos que se le piden.

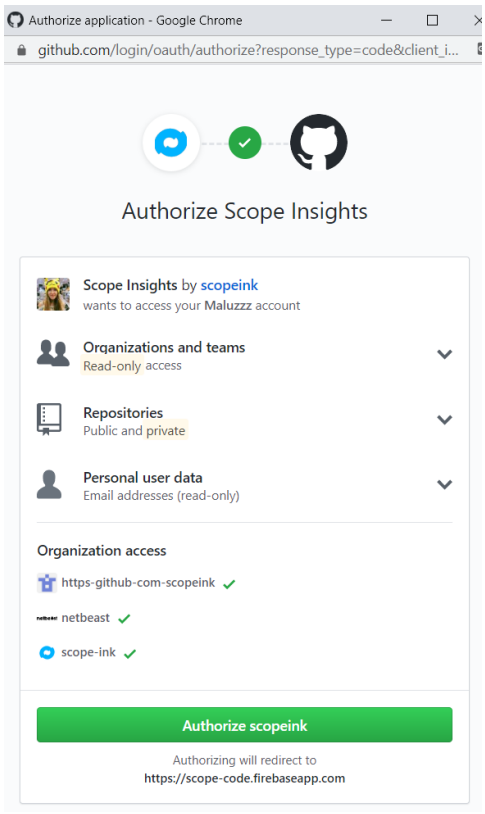


Figura 28: Ventana emergente de autorización para Github

También este servicio nos crea una base de datos con los emails que han iniciado sesión en la aplicación. Esta se crea de forma automática pero no la utilizaremos en este proyecto.

4.1.8 Visual Studio Code

Como entorno de desarrollo hemos usado Visual Studio Code de Microsoft [26]. Ofrece complementos que personalizan sus funcionalidades como la detección de errores de programación, formateo rápido, autocompletado inteligente, integración con Git… Es gratuito y de código abierto.

Por supuesto es compatible con el lenguaje que hemos usado para el desarrollo de la aplicación web.

4.1.9 Herramientas para depurar el código

Para poder depurar el código y resolver problemas, hemos utilizado un plugin para el navegador Google Chrome llamado **"React Developers Tool".** Esta herramienta nos permite:

- Comprobar las propiedades de los componentes de nuestro código
- Comprobar el estado de Redux
- Lanzar acciones desde su interfaz y comprobar cuáles se han lanzado.

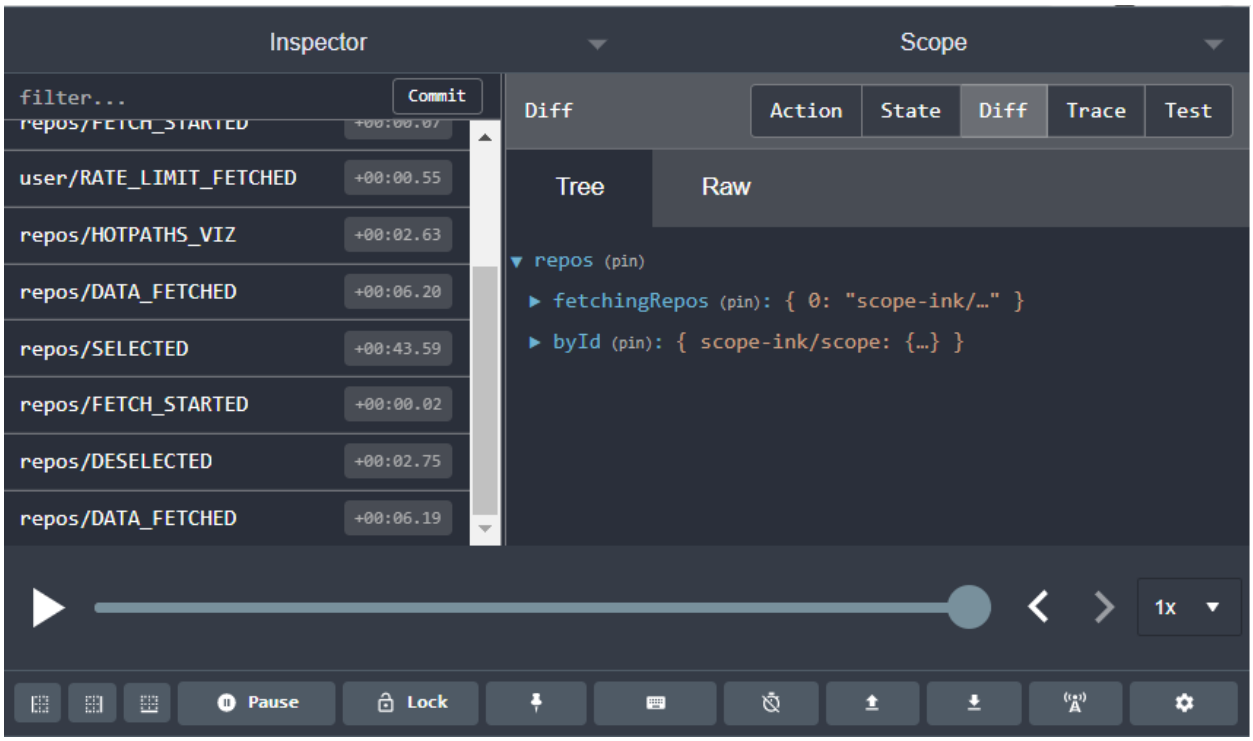


Figura 29: Interfaz de React Developers Tool en funcionamiento

Como complemento para poder parar la ejecución del código y trazar las variables hemos usado la consola que viene por defecto con el navegador Google Chrome. Se puede acceder en todos los navegadores Google Chrome pulsando F12.

Al tener la aplicación corriendo en localhost, accedemos con el navegador a la url *localhost:3000* y con esta herramienta podemos ver todo el código de nuestra aplicación en tiempo real.

Además, permite trazar el código añadiendo puntos de parada, y ejecutar comandos en la consola que aparece en la parte inferior. Esto es muy cómodo a la hora de conocer qué valores tienen las variables en cierto punto de la ejecución del código.

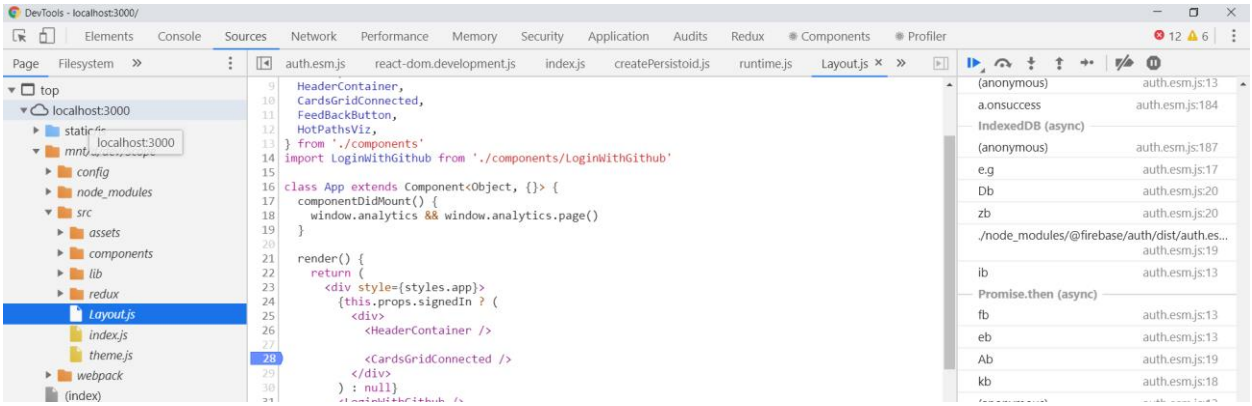


Figura 30: *DevTools* de Google Chrome. Ejemplo de un punto de parada en el código.

4.2 Herramientas de organización y control de versiones

En este apartado vamos a ver herramientas de organización de tareas y control de versiones que se han usado para el desarrollo del proyecto.

4.2.1 Git & GitHub

A la hora de organizar y tener almacenado el código de la aplicación se ha usado GitHub. Y como método de control de versiones Git.

4.3 Herramientas de integración continua y despliegue

4.3.1 CircleCI

Esta herramienta se integra con GitHub y nos permite automatizar la ejecución de las pruebas que hemos programado con Jest.

La hemos integrado con el repositorio remoto y como consecuencia cada vez que hay un nuevo commit o una nueva rama los tests se vuelven a ejecutar y nos alerta a través de la interfaz gráfica si los últimos cambios han pasado los test o no.

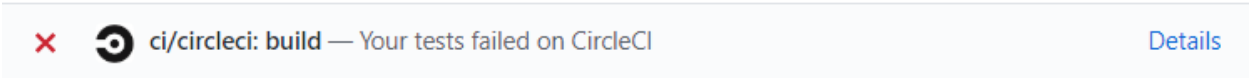


Figura 31: Mensaje que aparece en la interfaz gráfica de GitHub integrado con CircleCI cuando los tests no son correctos.

Para integrar esta herramienta con GitHub hemos tenido que añadir al sistema de ficheros de nuestro proyecto un fichero de configuración con el nombre *circle.ci*, que explicamos en más detalle a continuación. (Se resaltan los comentarios en gris para explicar código concreto)

```
# Javascript Node CircleCI 2.0 configuration file
#
# Check https://circleci.com/docs/2.0/language-
javascript/ for more details
#
version: 2
jobs:
  build:
    docker:
     #Versión de Node que queremos usar
      - image: circleci/node:10.9.0

   #directorio donde debe situarse CircleCI
    working_directory: ~/repo

    steps:
      - checkout

# Comprueba si han cambiado el package.json para instalar nuevas
dependencias o no
          keys:
          - v1-dependencies-{{ checksum "package.json" }}
          - v1-dependencies-
```

```
# Comando para instalar las dependencias
      - run: yarn install

      - save_cache:
          paths:
            - node_modules
          key: v1-dependencies-{{ checksum "package.json" }}

# Comando para ejecutar los tests
      - run: npm run test
```

Código 8: Fichero de configuración de CircleCI

4.3.2 Vercel

Esta herramienta es un servicio que también se integra con GitHub y nos permite compilar y desplegar, en los servidores de Vercel, la aplicación para poder testear de forma manual la misma. Si hay algún fallo en la compilación también manda alertas tanto por email como por la interfaz de GitHub. Por cada rama nueva se hace un despliegue, es decir en cada Pull Request tendremos la posibilidad de testear manualmente gracias al este servicio

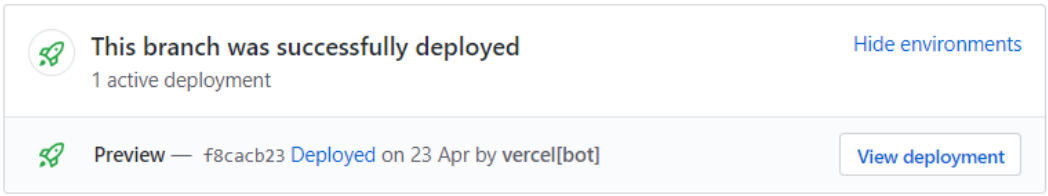


Figura 32: Mensaje integrado en un Pull Request de la integración de Vercel con GitHub

En la Figura 32 al pulsar en *View deployment* nos llevará a una url temporal donde estará desplegada esa rama del repositorio, es decir una versión de la aplicación, en este caso por ejemplo sería una url de este formato *https://scope-aatz3gjfpt.now.sh/*

Es una herramienta muy útil a la hora de revisar nuevas funcionalidades ya que no se necesita descargar al repositorio local los cambios, compilar y desplegar en localhost para comprobar si una rama está funcionando o no.

Además, evita tener servidores de pruebas, porque ellos se ocupan del despliegue automático después de configurarlo una primera vez.

Para llevar a cabo la integración con GitHub es necesario añadir un archivo de configuración en el directorio raíz de nuestra aplicación, el archivo es el siguiente:

- **Now.json:** este fichero define los parámetros de configuración del servicio. Vercel usará estas variables para compilar nuestra aplicación

  Le pasamos en el campo builds los parámetros necesarios: *src* es el nombre del archivo del que instalará las dependencias, *use* es la librería que usará Vercel para compilar la aplicación. Y por último hay que especificar en el parámetro config/distDir donde estará la carpeta que queremos desplegar, en nuestro caso esa carpeta se llama build.

```
{
```

```
  "version": 2,

# Nombre del despliegue
    "name": "Scope Testing",
    "builds": [
      {
        "src": "package.json",

# Módulo del servicio Vercel que compilará la aplicación
        "use": "@now/static-build",

# Directorio donde se crea la aplicación que se desplegará
        "config": {
                "distDir": "build"}
      }
    ]
  }
```

Código 9: Fichero de configuración del servicio Vercel.

De este modo Vercel conoce los parámetros necesarios para desplegar la aplicación y podremos administrar este servicio desde su interfaz web.

Por ejemplo, al crear un pull request aparecerán este mensaje que nos facilita dos enlaces:

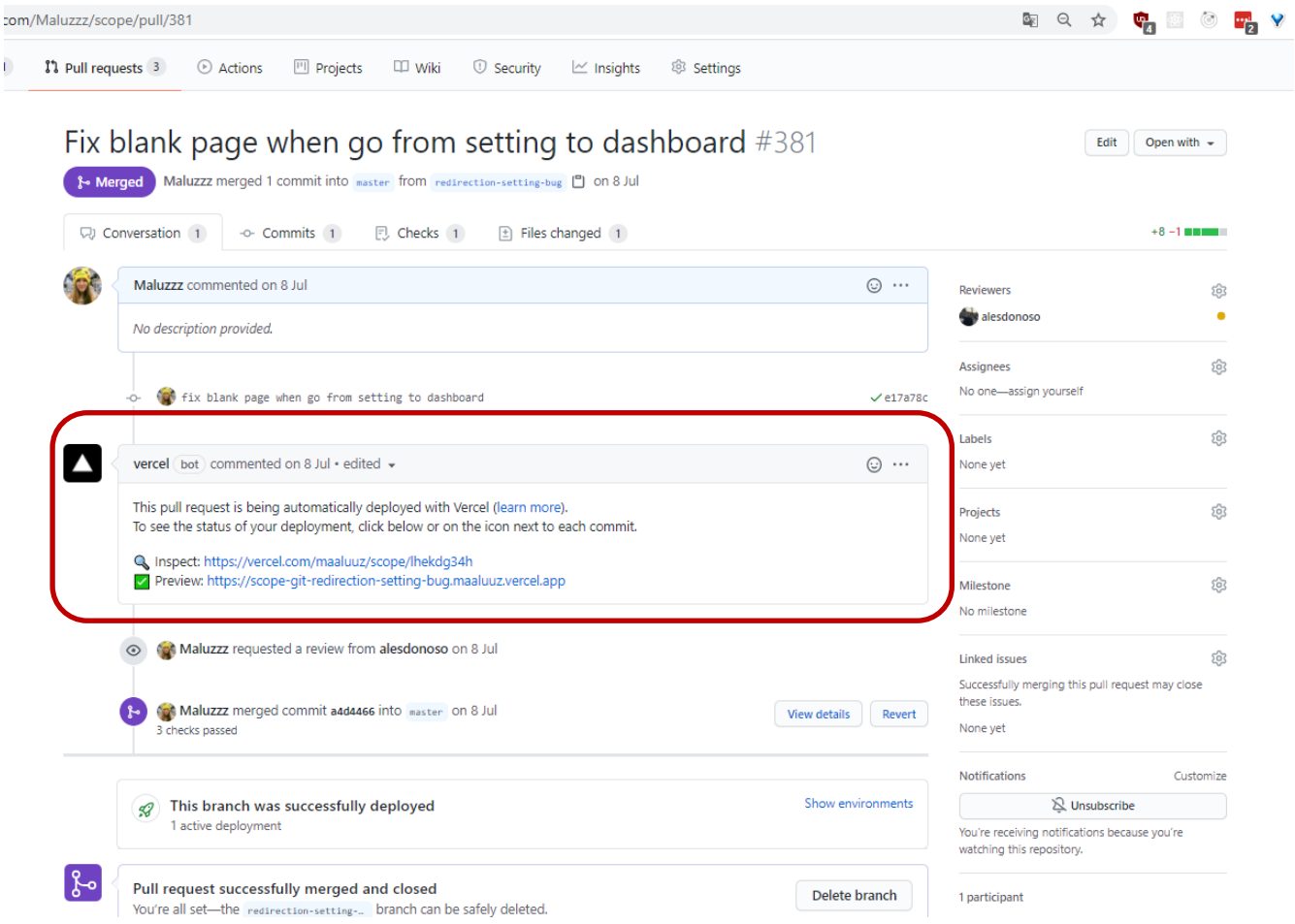


Figure 33: Interfaz gráfica de Github integrado con Vercel.

- **Preview:** te lleva al despliegue en sus servidores de la aplicación para poder testearla manualmente.
- **Inspect:** te lleva a su herramienta para poder ver los logs de esa compilación y despliegue en detalle.

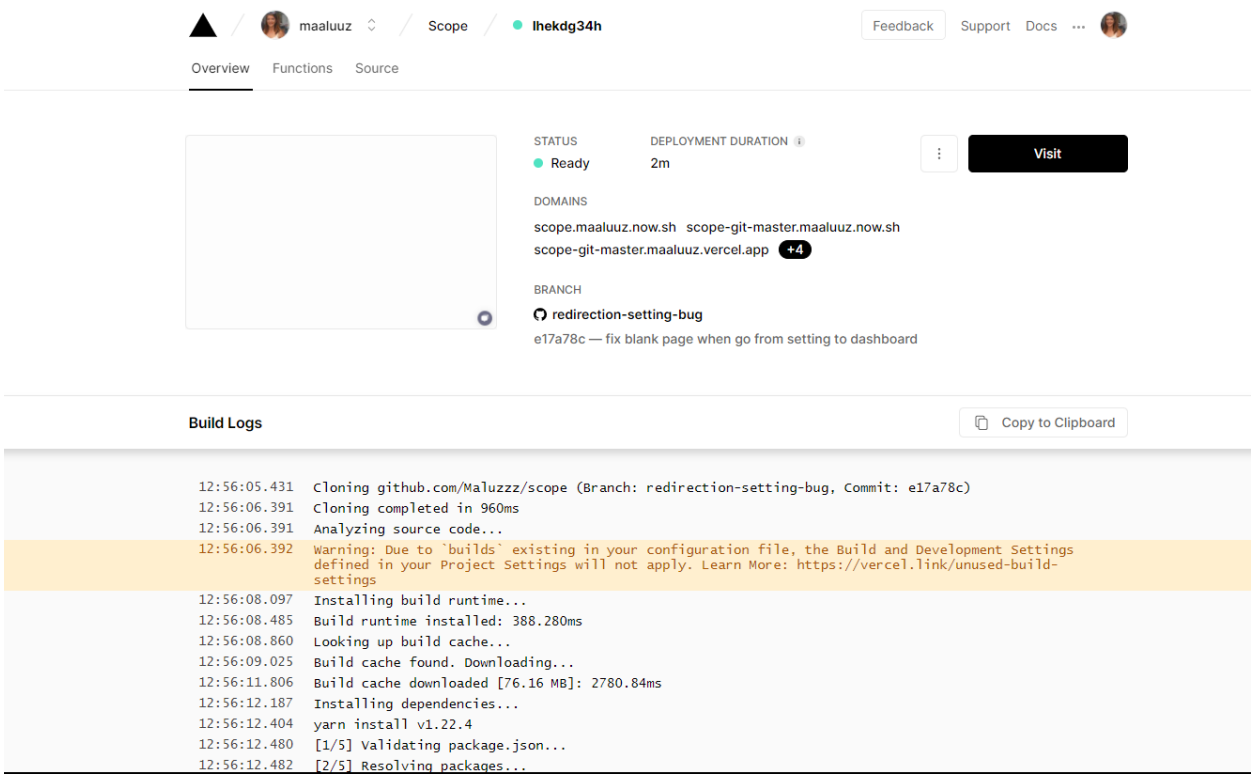


Figure 34: Herramienta Vercel Logs del despliegue de una rama en detalle.

En otro menú de la herramienta Vercel podemos acceder al despliegue que se ha realizado por cada rama y nos da su estado y el enlace para poder probar la aplicación.

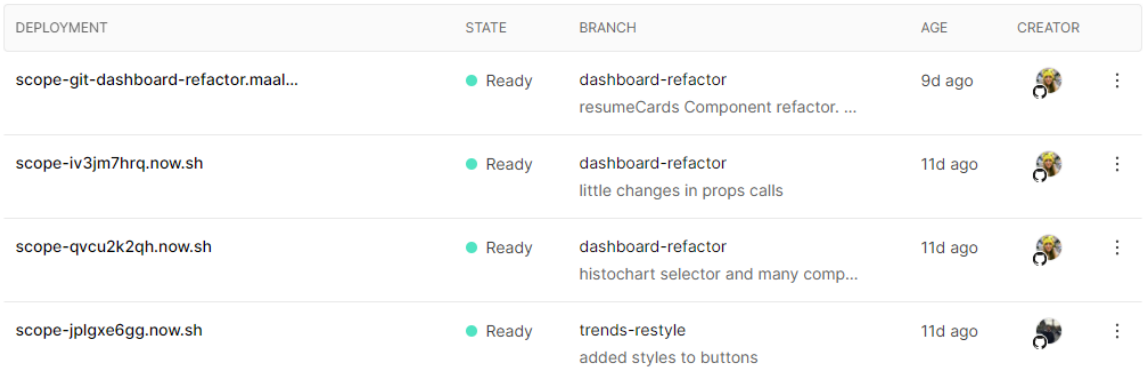


Figura 35: Interfaz gráfica del servicio Vercel, lista de despliegues.

Este servicio es gratuito para proyectos pequeños, si se quiere personalizar el dominio donde se despliega ya entra en un modelo del servicio de pago.

5 ESPECIFICACIÓN DE REQUISITOS

"Quality is never an accident; it is always the result of intelligent effort."
- John Ruskin -

5.1 Introducción

La especificación de requisitos define y detalla cómo debe ser el comportamiento de un sistema. En este capítulo se detallará la especificación de requisitos que debe cumplir la aplicación desarrollada.

Los objetivos se han definido en el apartado *1.2.1 Objetivos*, aquí desglosaremos qué vamos implementar, describiendo cómo se debe comportar el Sistema.

5.2 Actores

Los actores representan los elementos o entidades externos que interactuarán con nuestro sistema puede parecer que sólo sería el usuario, pero hay otros elementos con los que la aplicación se comunicará y son los siguientes:

Tabla 8: *Actores del sistema: ACT_01 Usuario*

ACT_01	Usuario
Descripción:	Usuarios de la aplicación. Cualquier persona que tenga una cuenta en GitHub con repositorios. En un equipo de trabajo pueden ser: - **Gestor de proyectos**: que necesitará las visualizaciones para conocer cómo va el proyecto, cómo están trabajando los desarrolladores del equipo. - **El equipo:** el equipo también puede entrar en la aplicación, cada desarrollador con su cuenta de GitHub. Estos usuarios pueden obtener motivación de conocer cómo están trabajando sus compañeros, cómo van las tareas…
Comentario:	Iniciarán sesión, buscarán repositorios, podrán generar visualizaciones y navegar por la aplicación.

Tabla 9: Actores del sistema: *ACT_02 API de GitHub*

ACT_02	API
Descripción:	Api que nos dará los datos de Git (GitHub)
Comentarios:	Nuestra aplicación debe conectarse con la API que nos dará los datos necesarios para poder generar las visualizaciones.

Tabla 10: Actores del sistema: ACT_03 Caché

ACT_03	Caché
Descripción:	Caché local en el cliente
Comentarios:	Guardará los datos a procesar, nos ayuda a no tener que repetir peticiones a la API cada vez que creemos visualizaciones y así podamos acelerar el proceso de generar visualizaciones.

Tabla 11: Actores del sistema: ACT_04 Firebase

ACT_04	Firebase Autentificación
Descripción:	Firebase es el actor que nos dará Autorización y Autentificación con GitHub.
Comentarios:	Este actor se encarga de mediar con GitHub para que nosotros obtengamos un token y mediante éste podamos realizar peticiones al ACT-02, es decir, la API de GitHub.

5.3 Requisitos generales

A continuación, se detallan los requisitos que derivan de los objetivos que se especifican en el apartado 1.3.1 Objetivos.

Tabla 12: Requisitos generales RG_001 - *Autentificación/Inicio de Sesión*

RG_001	**Autentificación/Inicio de Sesión**
Versión:	Versión 1.0
Dependencias:	No tiene
Descripción:	El sistema debe contar con una autorización y autentificación que nos inicie sesión con GitHub. - **Autentificación:** permitirá iniciar sesión con usuario y contraseña en GitHub y que

	se nos redirija a la aplicación. - **Autorización:** nos permite que el usuario decida si quiere dar a la aplicación los permisos necesarios para ejecutar las peticiones a la API. Si acepta ceder esos permisos se genera un token. Los permisos que debemos tener son acceso de lectura y escritura de los repositorios, para conseguir los datos de éste: colaboradores, pull requests…
Prioridad:	Alta
Comentarios:	El usuario puede ser tanto el gestor como un colaborador, si una persona tiene acceso a un proyecto en GitHub con este método de inicio de sesión se generará el token con los permisos pertinentes.

Tabla 13: Requisitos generales RG_002 -*Buscar entre sus repositorios*

RG_002	Buscar entre sus repositorios
Versión:	Versión 1.0
Dependencias:	*RG_001:* Autentificación/Inicio de Sesión
Descripción:	El usuario debe poder elegir entre sus proyectos mediante un buscador que autocomplete la búsqueda con los proyectos de su cuenta de GitHub.
Prioridad:	Alta

Tabla 14: Requisitos generales RG_003 – *Visualizaciones*

RG_003	Visualizaciones
Versión:	Versión 1.0
Dependencias:	• ***RG_001:*** *Autentificación/Inicio de Sesión.* • ***OBJ_01:*** *Desarrollar un algoritmo que estime el valor de una tarea* • ***OBJ_02:*** *Visualización general.* • ***OBJ_03:*** *Visualización Individual.* • ***OBJ_04:*** *Visualización del tiempo en terminar tareas.* • ***OBJ_05:*** *Visualización ficheros más modificados.*
Descripción:	La aplicación debe proveer al usuario de una interfaz gráfica donde se represente de manera visual el desempeño de los desarrolladores de forma objetiva y basándose en métricas reales.
Prioridad:	Alta

Tabla 15: Requisitos generales RG_004 - *Navegar entre visualizaciones*

RG_004	Navegar entre visualizaciones
Versión:	Versión 1.0
Dependencias:	• ***OBJ_02:*** *Visualización general.* • ***OBJ_03:*** *Visualización Individual.* • ***OBJ_04:*** *Visualización del tiempo en terminar tareas.* • ***OBJ_05:*** *Visualización ficheros más modificados.*
Descripción:	La interfaz gráfica de la aplicación debe permitir al usuario cambiar entre las diferentes visualizaciones que se proveerán.
Prioridad:	Alta

Tabla 16: Requisitos generales RG_005 - *Gestionar sesión*

RG_005	Gestionar sesión
Versión:	Versión 1.0
Dependencias:	• ***RG_001:*** *Autentificación/Inicio de Sesión.*
Descripción:	El usuario debe ser capaz de cerrar y abrir la sesión de la aplicación. Además, todos los datos deben ser borrados de la caché del navegador al salir de la aplicación.
Prioridad:	Baja

5.4 Requisitos Funcionales

En este apartado se detallará el conjunto de funcionalidades y servicios que debe facilitar el Sistema así como la descripción de las respuestas a las entradas al mismo

5.4.1 Requisitos de información

En esta sección detallamos la información que vamos a guardar en el estado global a través de Redux y que habremos obtenido a través de la API de GitHub. Se almacena como un objeto de tipo JSON.

Estos requisitos hacen referencia a la estructura de datos central que utilizará toda la aplicación a la hora de funcionar. Tendremos dos variables: usuario y repositorios. En cada una de esas variables se guardará un objeto con la información necesaria. Esta información es accedida por la aplicación fácilmente gracias Redux que hemos detallado en *Tecnologías Usadas.*

Tabla 17: Requisitos de información: RF.INF_001 - *Usuario*

RF.INF_001	**Usuario**
Versión	Versión 1.0
Dependencias	*RG_001:* Autentificación/Inicio de Sesión.
Descripción	Información que necesitará la aplicación para llevar a cabo el inicio de sesión.
Datos específicos	• **signedIn:** *Boolean* para saber si el usuario se ha autenticado o no. • **author:** *objeto* que contiene el nombre y el avatar de la cuenta de GitHub • **token:** *string* donde se guarda el token para poder construir las peticiones a la api
Requisitos hijos	*No contiene*
Prioridad	Alta
Comentario	Estos datos son los primeros en almacenar ya que del token depende que se puedan obtener los siguientes datos

Tabla 18: Requisitos de información: RF.INF_002 - *Repositorios*

RF.INF_002	**Repositorios**
Versión	Versión 1.0
Dependencias	*RG-001:* Autentificación/Inicio de Sesión. *RF.INF_001:* Usuario
Descripción	Estos datos se guardan para poder realizar las visualizaciones.
Datos específicos	• **erroredRepos:** *array* con los mensajes de errores generado de cada repositorio • **reposFetched:** *array* con la información del repositorio ordenada por puntuación. • **visibleRepos:** *array* con los repositorios que se están mostrando. • **currentViz:** s*tring* para elegir el tipo de visualización • **repoErrorsById:** *array* con un log de todos los repos que han tenido error. • **byId:** datos ya procesados que usarán las visualizaciones.
Requisitos hijos	
Prioridad	Alta
Comentario	

5.4.2 Requisitos de conducta

Tabla 19: Requisitos de conducta: RFC_001 - *Mostrar y gestionar errores*

RFC_001	**Mostrar y gestionar errores**
Versión	Versión 1.0
Dependencias	• ***RG_003:*** *Mostrar contenido y visualizaciones.*
Descripción	Si hay errores a la hora de pedir los datos de la API es necesario mostrar un mensaje al usuario con el tipo de error. Se debe mostrar información sobre los siguientes tipos de errores: • El proyecto que el usuario ha introducido no existe. • La API de GitHub no se encuentra disponible. • El proyecto analizado no tiene datos. • La persona no tiene proyectos que analizar
Prioridad	Alta
Comentario	

Tabla 20: Requisitos de conducta: RFC_002 - *Visualización de red de nodos simple*

RFC_002	**Visualización de red de nodos simple**
Versión	Versión 1.0
Dependencias	• ***OBJ_02:*** *Visualización general.* • ***OBJ_03:*** *Visualización Individual.*
Requisitos hijos	• ***RFC_006:*** *Leyendas.*
Descripción	El sistema debe crear esta visualización con los datos obtenidos. Este gráfico presentará los datos con respecto a los tipos de unidades de trabajo de esta manera: • **Commits**: círculos sin relleno • **Revisiones**: círculos amarillos • **Pull Requests**: círculos azules ** El tamaño del círculo se calcula según se explica en el al apartado 3 de este trabajo. Además, cada uno de estos círculos debe facilitar un enlace a la tarea en GitHub
Prioridad	Alta
Comentario	

Tabla 21: Requisitos de conducta: RFC_003 - *Gráfico de ficheros más modificados*

RFC_003	**Visualización de ficheros más modificados**
Versión	Versión 1.0
Dependencias	• ***OBJ_05:*** *Visualización ficheros más modificados.*
Requisitos hijos	• ***RFC_006:*** *Leyendas.*
Descripción	Esta visualización debe mostrar los ficheros que más se han modificado del proyecto. A más modificaciones más intensidad de color debe tener la ruta del archivo.
Prioridad	Alta
Comentario	

Tabla 22: Requisitos de conducta: RFC_004 - *Gráfico estimación duración de las tareas*

RFC_004	**Visualización estimación duración de las tareas**
Versión	Versión 1.0
Dependencias	• ***OBJ_04:*** *Visualización del tiempo en terminar tareas.*
Requisitos hijos	• ***RFC_006:*** *Leyendas.*
Descripción	Esta visualización debe mostrar, en horas, la duración media desde que un pull request se abre hasta que se fusiona con la rama principal.
Prioridad	Alta
Comentario	

Tabla 23: Requisitos de conducta: RFC_005 - *Pantalla de carga*

RFC_005	**Pantalla de carga**
Versión	Versión 1.0
Dependencias	• ***RFC_003:*** *Gráfico de ficheros más modificados* • ***RFC_002:*** *Gráfico de red de nodos simple* • ***RFC_004:*** *Gráfico estimación duración de las tareas*
Descripción	Es necesario mostrar que se están cargando los datos al usuario. Esto se consigue mediante la interfaz, en el caso de que estemos pidiendo datos a la API de GitHub debe mostrarse un elemento dinámico que de a entender que el sistema está cargando.
Prioridad	Media
Comentario	

Tabla 24: Requisitos de conducta: RFC_006 - *Leyendas*

RFC_006	**Leyendas**
Versión	Versión 1.0
Dependencias	-
Descripción	Es de vital importancia hacer la aplicación al usuario fácil de entender, por cada una de las visualizaciones se mostrará qué representan los tamaños, las formas, los colores, etc. es decir, mostrar una leyenda.
Prioridad	Media
Comentario	

Tabla 25: Requisitos de conducta: RFC_007 - *Desplegar información del gráfico de nodos.*

RFC_007	**Desplegar información del gráfico de nodos**
Versión	Versión 1.0
Dependencias	-
Descripción	El usuario debe poder saber qué tareas ha hecho cada desarrollador, a partir del gráfico de nodos simple. Para ello la vista cambiará a una lista simple con el título de la tarea y la puntuación que se le ha estimado. Esta lista debe contener enlaces a los pull requests en GitHub, es decir que al pulsar en alguno de ellos te lleve a ese pull request en la web de GitHub.
Prioridad	Media

Comentario	

5.5 Requisitos no Funcionales

Los requisitos no funcionales no hacen referencia directa a las funciones específicas del Sistema a nivel usuario. Estos requisitos se refieren a las características del sistema como rendimiento, seguridad, disponibilidad.

5.5.1 Requisitos de Fiabilidad

Tabla 26: Requisitos de Fiabilidad: RNF_001 - Disponibilidad de los datos

RNF_001	**Disponibilidad de los datos**
Versión	Versión 1.0
Dependencias	• ***OBJ_02:*** *Visualización general.* • ***OBJ_03:*** *Visualización Individual.* • ***OBJ_04:*** *Visualización del tiempo en terminar tareas.* • ***OBJ_05:*** *Visualización ficheros más modificados.*
Descripción	La API de GitHub garantiza una disponibilidad del 99%. Nuestro sistema guarda los datos en la caché del navegador del cliente. Por lo que, si en siguientes peticiones GitHub fallara, tendremos una copia en caché de los datos que se puede mostrar. Aunque estos datos seguirán siendo los de la última consulta realizada con éxito.
Prioridad	Alta
Comentario	Todas las visualizaciones dependen de este requisito ya que sin datos no podemos mostrar nada.

5.5.2 Requisitos de Seguridad

Tabla 27: Requisitos de seguridad: RNFS_001 - *Autentificación y Autorización*

RNFS_001	**Autentificación y Autorización**
Versión	Versión 1.0
Dependencias	• ***RG_001:*** *Autentificación/Inicio de Sesión.*
Descripción	Se debe garantizar que la autorización y autentificación sea fiable. Se utilizará Google como proveedor de servicio para llevar a cabo esta funcionalidad. Se deberá avisar al usuario, en todo momento, que sus datos son totalmente privados y se quedan en su navegador. Y que en ningún momento nuestro sistema guarda datos del proyecto analizado.

Prioridad	Alta
Comentario	-

Tabla 28: Requisitos de seguridad: RNFS_002 - *Borrado de datos al cerrar sesión*

RNFS_002	**Borrado de datos al cerrar sesión**
Versión	Versión 1.0
Dependencias	• ***RG_001:*** *Autentificación/Inicio de Sesión.*
Descripción	Es importante que después de cerrar sesión se eliminen los datos del usuario que están en su navegador, así como el estado de Redux. En el estado de Redux se encuentra información sensible como es el token de autorización y no es conveniente que se quede guardado si el usuario ya no está dentro de la aplicación.
Prioridad	Alta
Comentario	-

5.5.3 Requisitos de Eficiencia

Tabla 29: Requisitos de eficiencia: RNFE_001 - *Tiempo de respuesta menor en posteriores peticiones*

RNFE_001	**Tiempo de respuesta menor en posteriores peticiones**
Versión	Versión 1.0
Dependencias	• ***RNFF_001:*** *Disponibilidad de los datos.*
Descripción	El tiempo de respuesta debe ser menor de 2 segundos si los datos ya se han consultado con anterioridad. Ya que debe usar parte de los datos en caché.
Prioridad	Media
Comentario	

6 DISEÑO DEL SISTEMA

Este apartado tiene como objetivo describir cómo está diseñado el sistema. Además de definir cómo se comporta el mismo con los sistemas externos, proveedores de servicios y la API.

6.1 Diagramas de paquetes

Este diagrama nos muestra los paquetes que componen el sistema y su relación con los servicios externos, de forma que podemos ver cómo se compone el sistema en su totalidad de una forma simplificada.

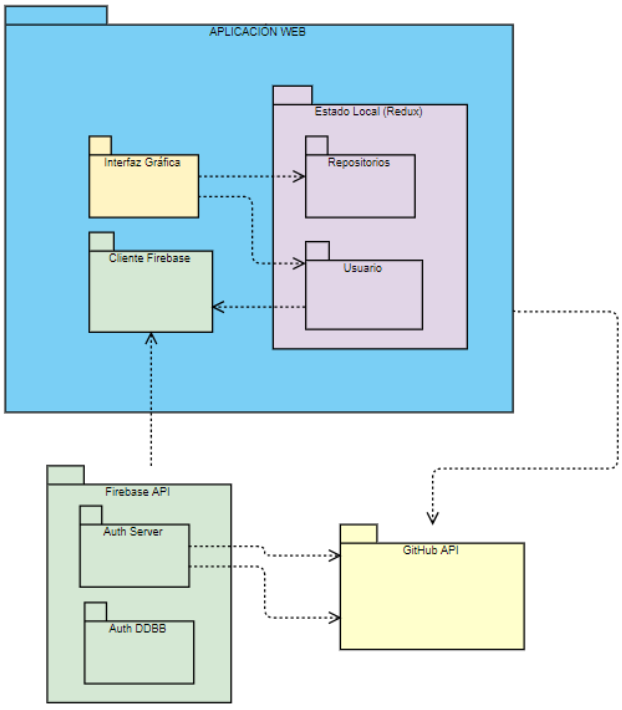


Diagrama 1: Diagrama de paquetes del sistema y relación con servicios externos.

Aquí se puede ver que en la aplicación se requiere un cliente Firebase que usará los servicios proporcionados por Google Firebase. Este es el encargado manejar la autorización y autentificación y devolver los datos necesarios a la aplicación web.

6.2 Diagrama de componentes

Como bien definimos en las tecnologías usadas, **React** se organiza en componentes de forma jerárquica, este diagrama describe qué componentes conforman la aplicación.

En este diagrama podemos distinguir las propiedades (*props*) que son las variables del componente, éstas se pueden pasar de un componente a otro, cómo es el caso de la relación entre el componente *Layout* y *Footer*. En el caso de los demás componentes estos se conectan con Redux y las propiedades son las que se pasan a través de la ***Store*.**

Es mejor conectar el ***Store*** de esta forma ya que si usamos mucho que las propiedades se pasen de unos componentes a otros puede causar fallos de rendimiento y actualizaciones de componentes innecesarias.

En el caso de que el componente use un estado propio (además del store de Redux) se detalla en *localState*.

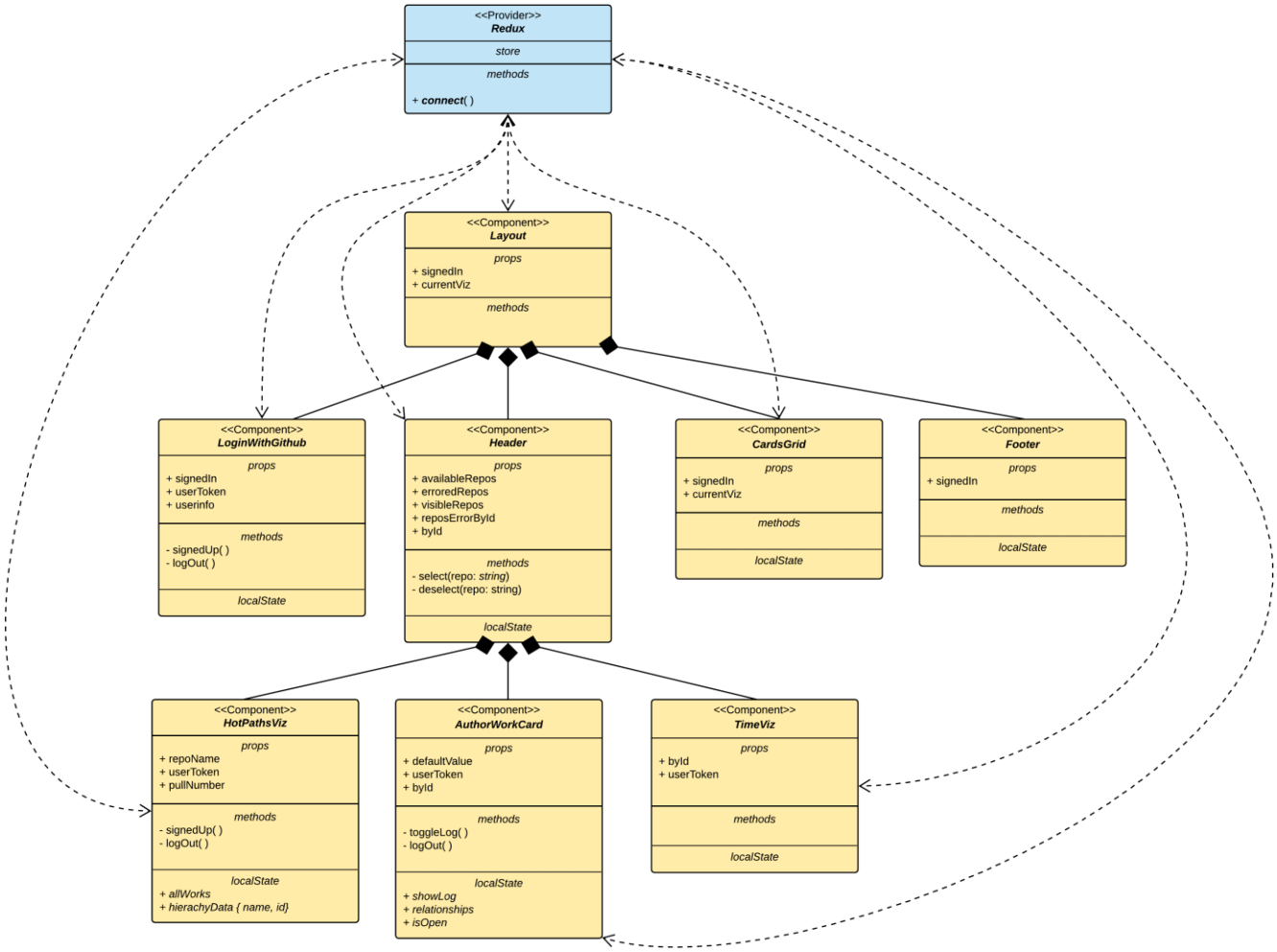


Diagrama 2: Diagrama general de componentes

6.3 Modelo del estado local

La información que muestra la aplicación se encuentra en **Redux**. Recordamos que este estado contendrá dos objetos principales: *user* y *repos*, como ya se especificó en los requisitos de información.

En el siguiente diagrama se define cómo será el estado inicial que tendrá la aplicación, las variables que guardará en Redux y los tipos de las mismas:

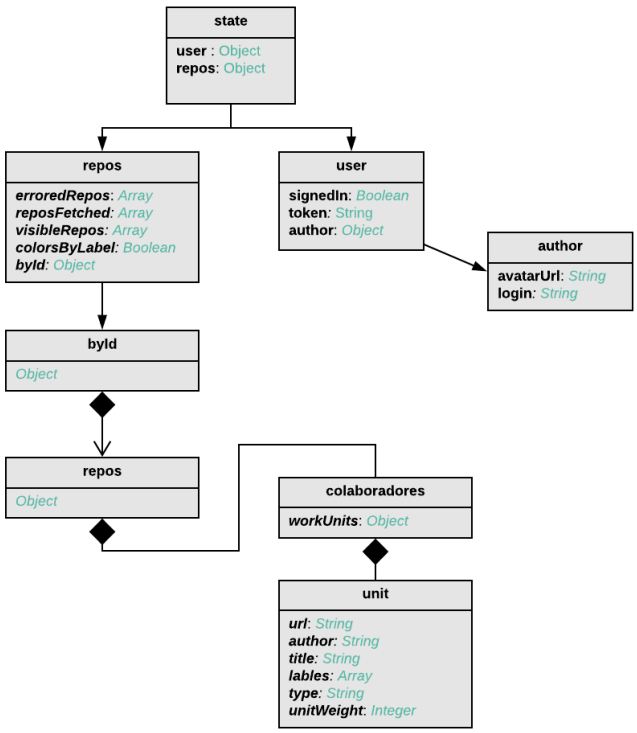


Diagrama 3: Diagrama de objetos del estado de Redux.

En el siguiente diagrama podemos ver un ejemplo de los valores por defecto que tendrá el estado de Redux si no se ha iniciado sesión.

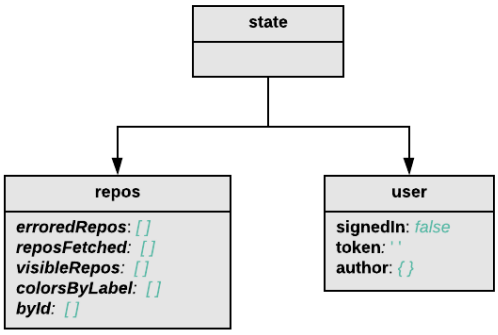


Diagrama 4: Diagrama de objeto del estado de la aplicación antes de iniciar sesión.

6.4 Diseño por casos de uso

6.4.1 CU.001: Inicio de sesión

En este caso vamos a describir este proceso con un diagrama de secuencia ya que muestra de manera clara como es la interacción del usuario con el servidor de autorización (Google) y de este con el proveedor de identidad (GitHub).

Para la aplicación el proceso de comunicación entre Google y GitHub es invisible. Si el usuario intenta iniciar sesión le aparecerá una ventana emergente que pedirá que inicie sesión con su cuenta de GitHub y acepte los permisos que necesita la aplicación para acceder a sus datos. Cuando acepta es Google el encargado de manejar este proceso con GitHub y devolvería a la aplicación el token. Después de obtener el token, es decir que el inicio de sesión se ha completado se cambiaría la interfaz al panel principal.

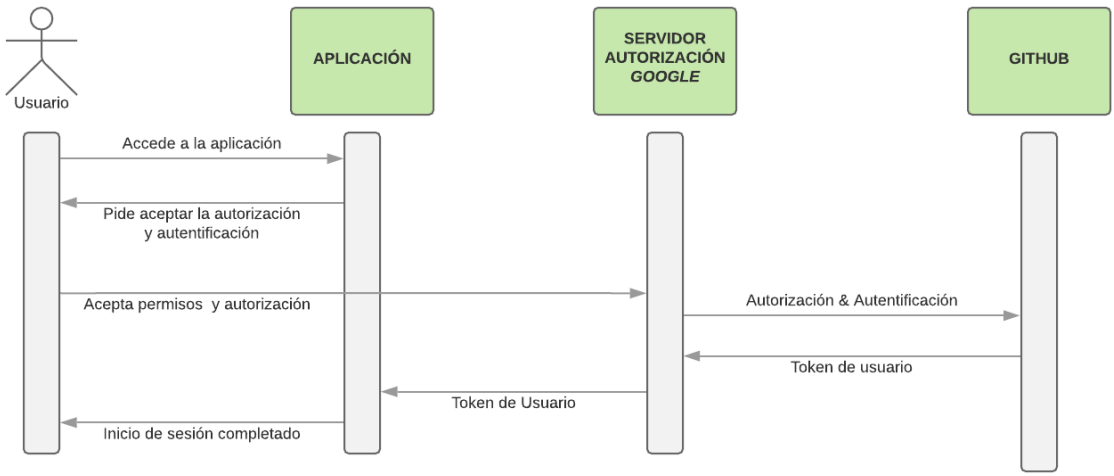


Diagrama 5: Diagrama de Secuencia: Inicio de Sesión

6.4.2 CU.002: Cierre de sesión

El siguiente diagrama de secuencia describe los mensajes que la aplicación intercambia con los servicios externos para cerrar sesión.

El usuario cerrará sesión y la aplicación usará la función *signOut* que provee el SDK de Google para cerrar la sesión y desconectar al usuario. Como vemos en el diagrama es necesario borrar el estado local cuando se cierra sesión cumpliendo así los requisitos de seguridad detallados en *RNFS_002.*

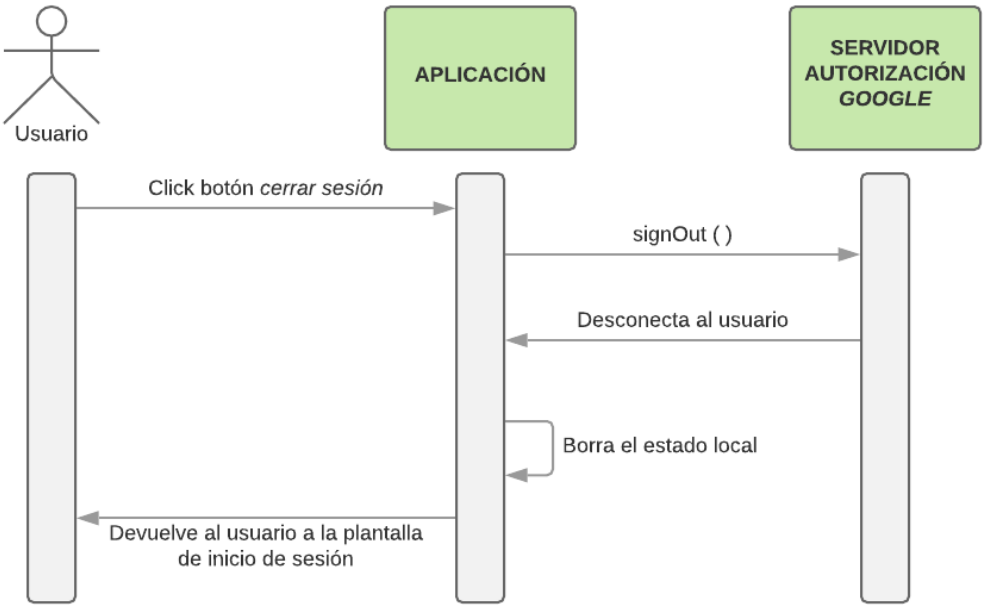


Diagrama 6: Diagrama de secuencia Cierre de Sesión

6.4.3 CU.003: Analizar un proyecto.

Los diagramas de actividad sirven para representar la lógica de un algoritmo, describir los pasos de un caso de uso, simplificar y clarificar los procesos, además de poder modelar elementos de la arquitectura del software como funciones y operaciones.

Nosotros lo vamos a usar para detallar qué lógica se da en la aplicación desde que se ha iniciado sesión hasta que se analiza un proyecto. Describiendo qué flujo se seguirá si hay fallos y si no los hay. Para que con este diagrama queden mucho más claros los pasos que se llevan a cabo para cubrir el requisito funcional detallado en *RFC_001: Mostrar y gestionar errores.*

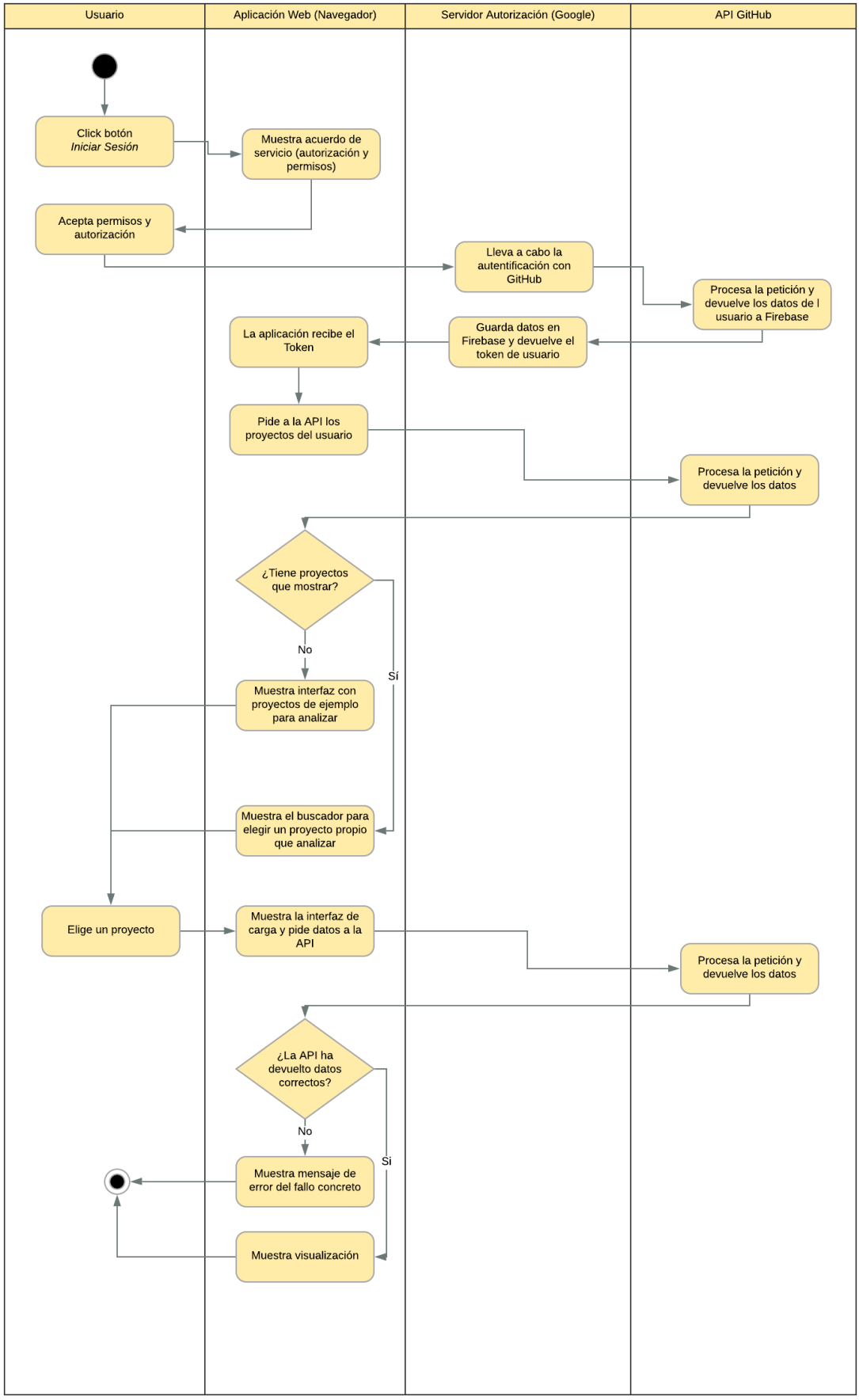


Diagrama 7: Diagrama de actividad de la aplicación desde el inicio de sesión al análisis un proyecto.

6.4.4 CU.004: Navegar entre visualizaciones

En este diagrama de actividades mostramos qué procesos internos se llevan a cabo cuando un usuario quiere cambiar la visualización que está viendo. Para ello hemos definido, con el fin de simplificar: interfaz, Store y reductores.

El caso representado es el siguiente: el usuario quiere cambiar la visualización, para ello tiene un desplegable que al seleccionar utiliza el método *dispatch* de Redux que lanzará una acción, en este caso, la acción contiene la información "CAMBIO_VISUALIZACIÓN". Como ya vimos en Redux, los reductores actualizan el estado, esta actualización repercute directamente en la interfaz que cambia y se configura de la manera adecuada al estado.

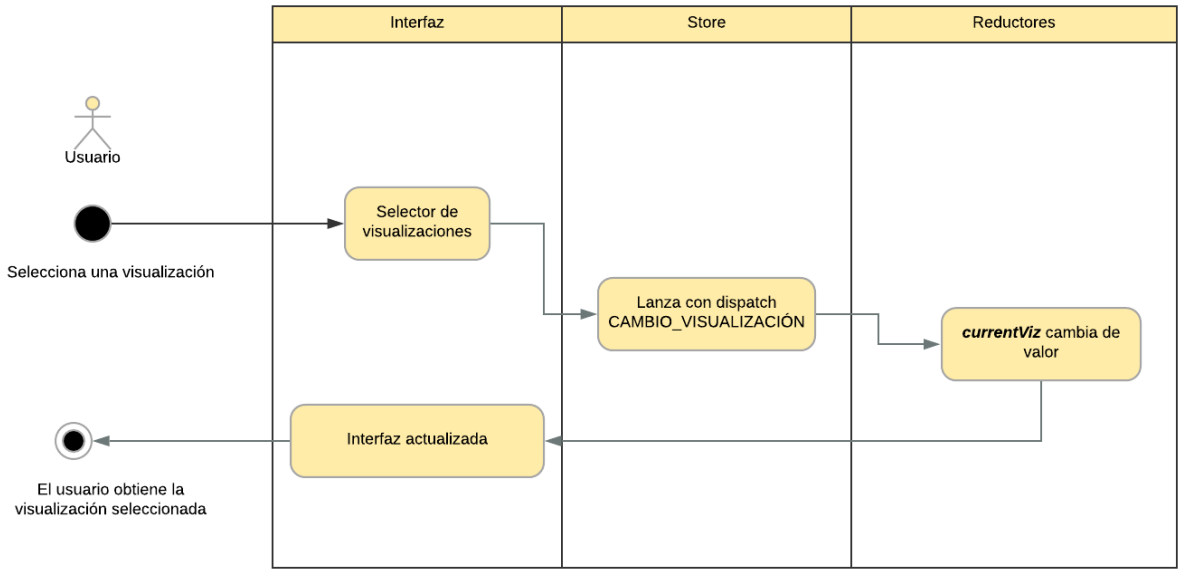


Diagrama 8: Diagrama de actividad Navegar entre visualizaciones

7 IMPLEMENTACIÓN

"Quality is never an accident; it is always the result of intelligent effort."
- John Ruskin -

La fase de implementación en el desarrollo de Software es el procedimiento de transformar el diseño del sistema en un sistema funcional. En este apartado vamos a describir cómo es la estructura del software que se va a implementar, las interfaces entre componentes y cómo hemos traducido los requisitos y los diseños del sistema en la aplicación.

7.1 Sistema de ficheros

Para el sistema de ficheros del proyecto, hemos usado una arquitectura que se adapte de manera adecuada al lenguaje escogido.

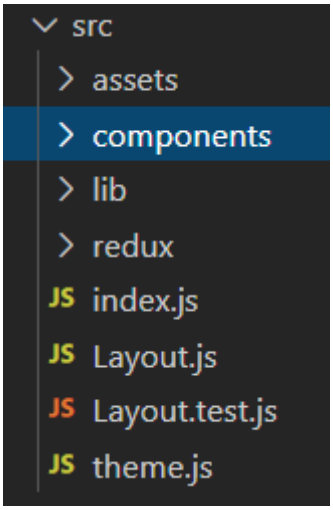


Figura 36: captura del directorio src/ de la aplicación

En la carpeta *src* del directorio principal se encuentra todo el código desarrollado, lo hemos dividido en las carpetas que se detallan a continuación:

- **assets**: esta carpeta contiene las imágenes que se van a usar, como logos, ilustraciones… ETC

- **components:** aquí se encuentran los ficheros que declaran los componentes de las interfaces. Estos están definidos en el apartado: "*Diagrama 2: Diagrama general de componentes".*

- **lib:** en esta carpeta encontraremos los ficheros que contienen las funciones que se usarán para extraer y procesar los datos de la API de GitHub.

- **redux:** como su nombre indica, en esta carpeta están los ficheros relacionados con redux, tanto las declaraciones de las acciones como los reductores y el store. Además de archivos adicionales como *thunks.js* que contiene funciones asíncronas que lanzan acciones.

index.js es el punto de entrada del código, el primero que se llama y *Layout.js* es el primer componente que se define y llama a todos los demás, como también declaramos en el apartado "*Diagrama 2: Diagrama general de componentes".*
El archivo *theme.js* nos servirá para guardar constantes de estilo, como por ejemplo los códigos de colores. Por

último, el fichero *Layout.test.js* contiene test de interfaz que explicaremos de manera detallada en el siguiente capítulo.

Como vimos en la introducción a React y redux, los componentes de la interfaz usarán sus propiedades para mostrar la información pertinente. Estas propiedades vienen del estado que guardamos en el Store de Redux.

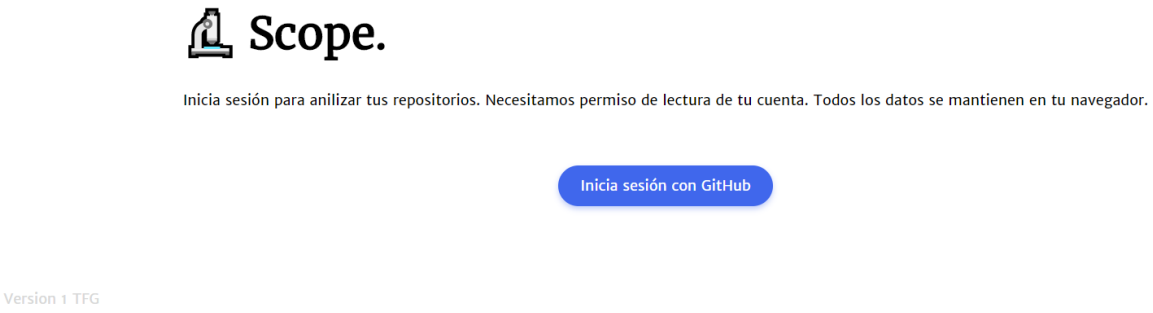
7.2 Interfaz de usuario

Para explicar la implementación vamos a recorrer las pantallas que ve un usuario al utilizar la aplicación web y describiremos qué funcionalidades y opciones se implementan en cada una de ellas.

7.2.1 Pantalla de inicio de sesión

Esta implementación está fundamentada en el requisito: ***RG_001: Autentificación/Inicio de Sesión*** y en los diagramas de diseño de secuencia *inicio de sesión y cierre de sesión.*

En esta pantalla la interfaz será sencilla tendrá el botón de iniciar sesión con GitHub y una pequeña descripción de qué hace la aplicación:



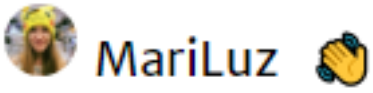
Captura 1: Interfaz de Inicio de sesión

Para la implementación de esta pantalla hemos declarado el siguiente componente:

```
export class LoginWithGithub extends PureComponent {…}
```

Este componente devuelve dos interfaces diferentes dependiendo de si ya se ha iniciado sesión o no. Esto lo comprueba mediante la variable del estado ***signedIn.***

- Si el usuario **no ha iniciado sesión**, la variable está a *false,* y devolverá el botón azul que vemos en la captura de arriba. Si pulsa sobre este, se mostrará la autorización de GitHub para iniciar sesión y lanzará la acción necesaria que cambie la variable ***signedIn*** a *true.*
- Si el usuario **ya ha iniciado sesión**, devuelve su nombre y avatar, y la funcionalidad al pulsar sobre este componente será la de cerrar sesión. Pulsando sobre la mano se cerrará sesión.



Captura 2: Información del usuario y botón de cerrar sesión

Las funciones de este componente son las descritas en *Diagrama 2: Diagrama general de componentes.*

7.2.2 Pantalla de sugerencias

Una vez autorizado el usuario puede ver la pantalla de sugerencias, las sugerencias variarán conforme al "*Diagrama de actividad: Analizar un proyecto*".

En esta pantalla debemos darle la posibilidad de:

- Introducir en el buscador de la parte superior el proyecto que el usuario quiera o elegir uno de la lista, si tiene proyectos disponibles.
- Salir de la aplicación.
- Recomendarle tres proyectos públicos si no tiene en su cuenta o suyos si tiene.

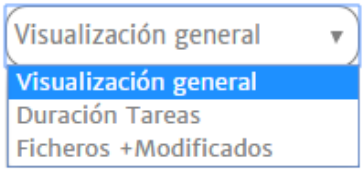
Solo se cambiará de pantalla si se cierra sesión o se analiza algún repositorio, ya sea pulsando en alguno de los tres botones o usando el buscador.

Si no tenemos repositorios esta será la interfaz que se le mostrará al usuario:

Captura 3: Interfaz mostrada cuando no hay ningún repositorio disponible

7.2.1 Navegación entre visualizaciones

Esto se llevará a cabo con un desplegable en el que se de la opción al usuario de elegir qué visualización quiere ver.



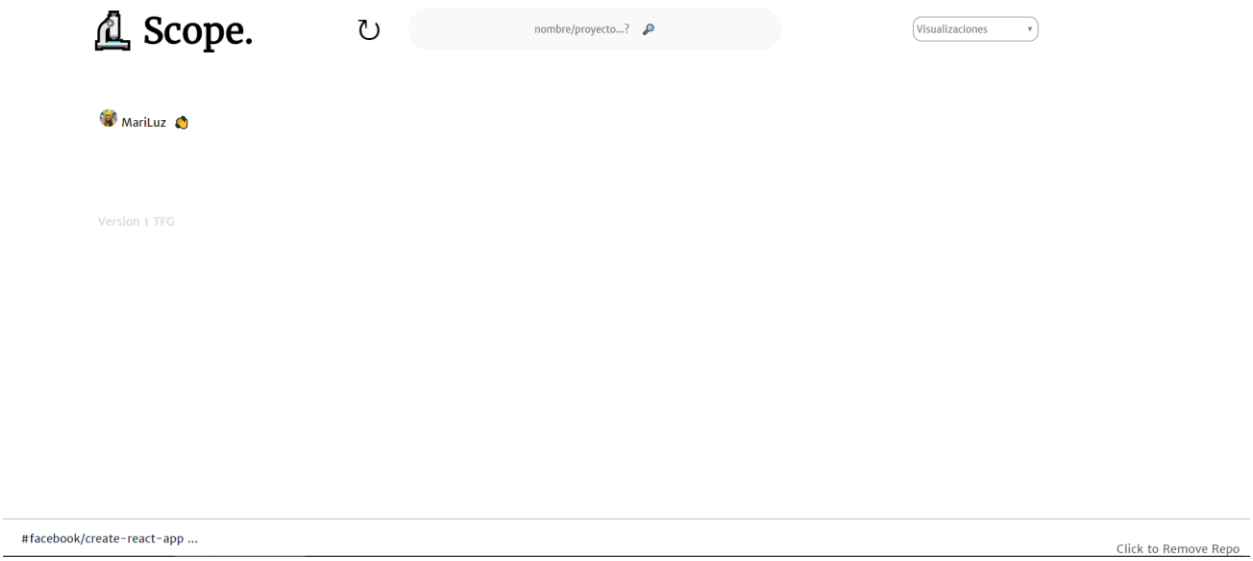
Captura 4: Desplegable para seleccionar la visualización

Esta interfaz es necesaria para cumplir el requisito del Sistema: RG_004: *Navegar entre visualizaciones.*

7.2.2 Pantalla de visualización general

Cuando se ha seleccionado un proyecto el usuario sabe que la aplicación está cargando los datos porque en la

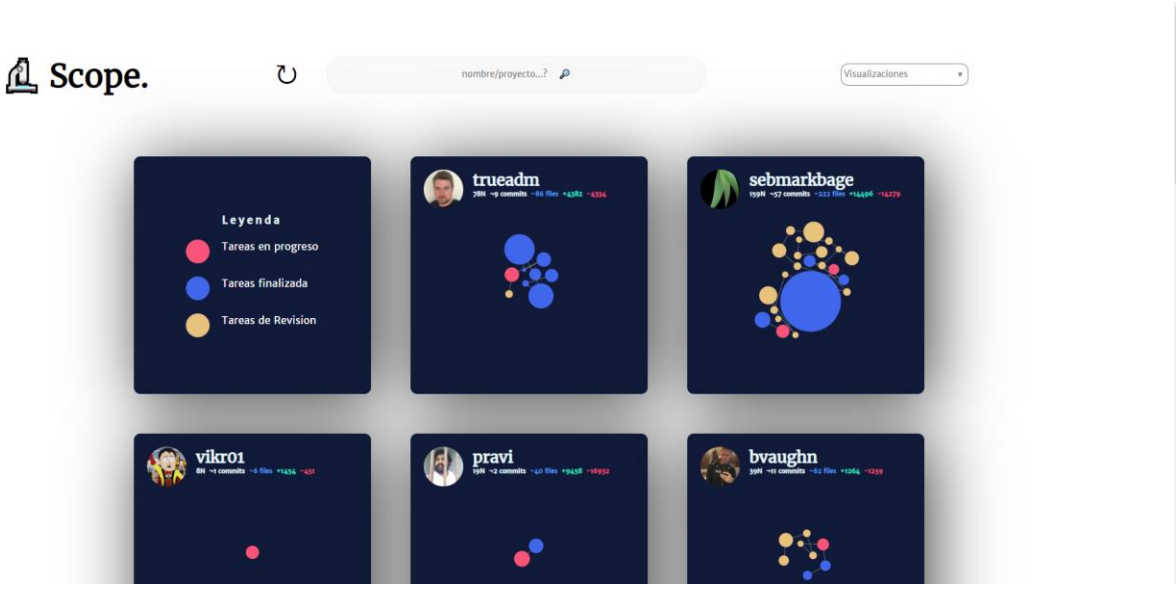
barra inferior habrá unos puntos suspensivos que van apareciendo de forma dinámica. Esto se basa en el requisito de conducta ***RFC_005: Pantalla de carga.***



Captura 5: Aplicación cargando los datos

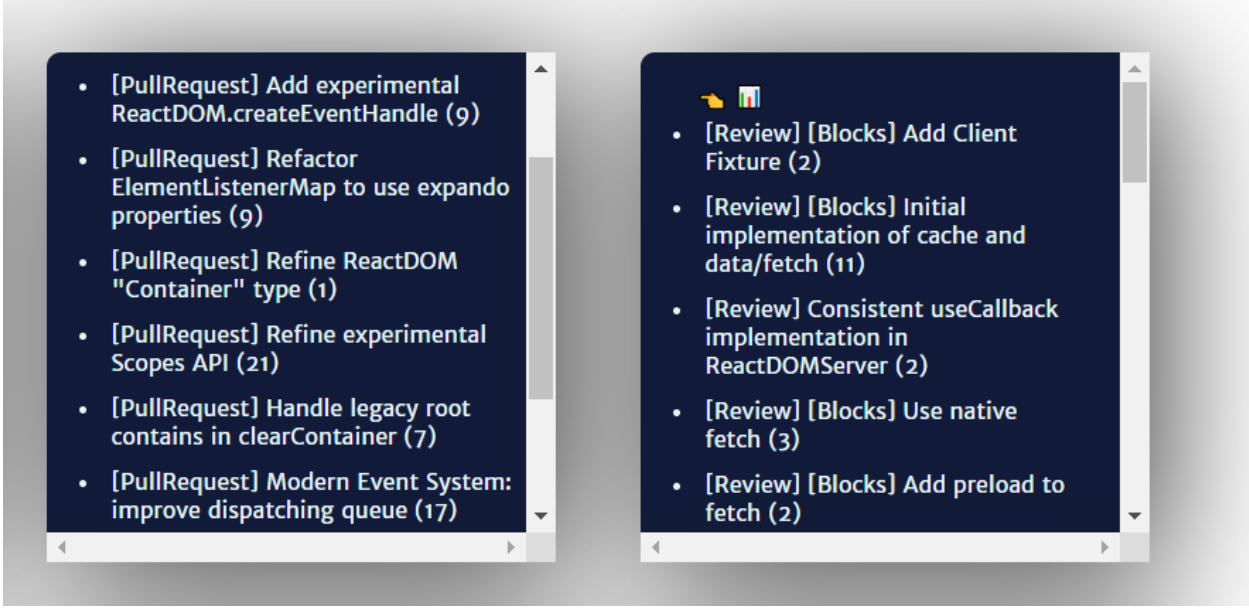
El siguiente resultado será la visualización general donde podremos ver los gráficos de cada desarrollador.

La siguiente interfaz cumple los requisitos y objetivos del proyecto: *OBJ-02: Visualización general, OBJ-03 – Visualización Individual, RFC_002 Gráfico de red de nodos simple y RFC_006: Leyendas,* la leyenda es lo primero que se muestra ya que así el usuario puede comprender desde primer momento qué es cada color.



Captura 6: Interfaz con la visualización principal

Y conforme al requisito *RFC_007: Desplegar información del gráfico de nodos,* podremos ver una lista de las tareas desempeñadas al pulsar sobre la puntuación del desarrollador:

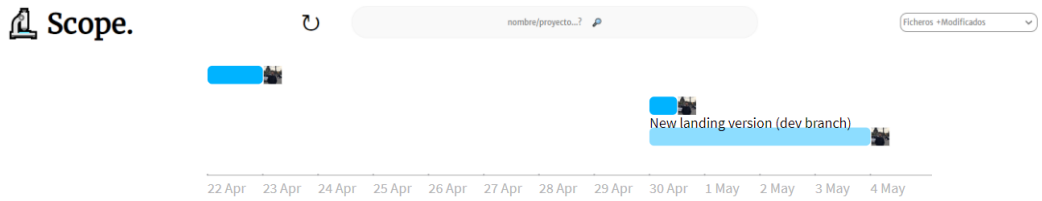


Captura 7: Componente que muestra en detalle cada una de las tareas

7.2.3 Pantalla de visualización tiempo

Esta visualización nos permite conocer la duración de las tareas. Es una visualización parecida a los diagramas de Gantt donde podemos ver en barras cuanto ha durado una tarea.

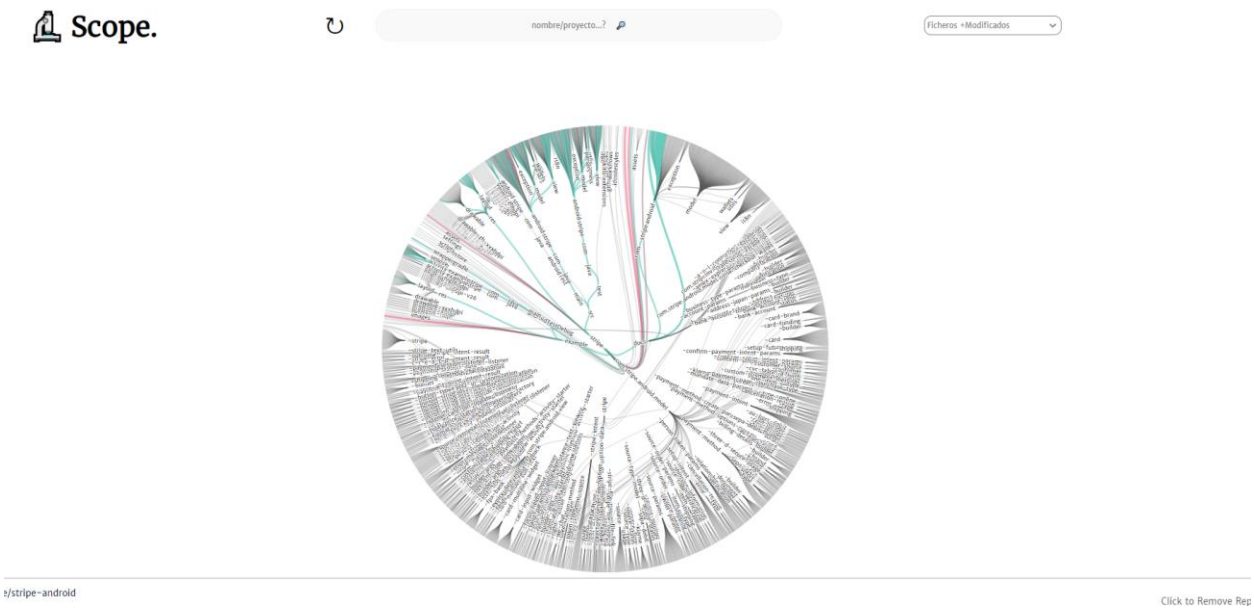
Al poner el ratón sobre los rectángulos se debe mostrar el título de la tarea en cuestión.



Captura 8: Interfaz con la visualización de tareas

7.2.4 Pantalla de archivos más modificados

Esta visualización nos permite conocer los ficheros más modificados del proyecto. Al cambiar de visualización con el desplegable comentado anteriormente lo que verá el usuario será la siguiente interfaz:

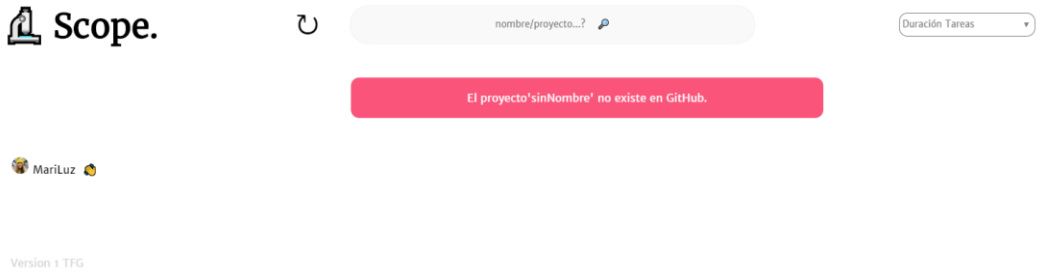


Captura 9: Interfaz de la visualización de ficheros más modificados

Además, al poner el ratón sobre los extremos de la visualización debe verse el nombre del fichero.

7.2.5 Alertas de errores

La gestión de errores está definida en el requisito general de conducta ***RFC_001:*** *Mostrar y gestionar errores,* donde están definidos los diferentes casos de error que se pueden dar en la aplicación. Es necesario que el usuario conozca qué tipo de error se ha producido, y la forma será la siguiente:



Captura 10: Mensaje de error cuando no existe el proyecto que se ha buscado

En esta captura se ha intentado analizar un proyecto que no existe en GitHub y el usuario lo sabrá por un mensaje en color llamativo en el centro de la pantalla.

Captura 11: Barra de proyectos cuando uno no se ha podido cargar

En la barra inferior aparecerá el título del proyecto que no se ha podido analizar correctamente también en color rosa. Hasta que no se deselecciona el proyecto, o se analice correctamente no se quitará el error de la pantalla.

Los demás errores se mostrarán de la misma forma con el texto descriptivo.

8 PLAN DE PRUEBAS

La fase de pruebas en el proceso software es muy importante ya que comprueba que se cumplen los requisitos tanto funcionales como los no funcionales y se verifica la calidad.

En este documento se detallarán las pruebas se van a realizar. Se han realizado pruebas **unitarias** que comprueban que las funciones básicas del sistema se comportan correctamente.

Las pruebas realizadas se han programado con Jest, descrita en **Tecnologías usadas**. Se han automatizado con la herramienta, también descrita, CircleCI.

También se llevarán a cabo una serie de tests manuales para comprobar que la interfaz muestra correctamente los datos. Para ello nos hemos ayudado de los despliegues de Vercel, agilizando el proceso de comprobar funcionalidades nuevas sin tener que desplegar manualmente el código.

8.1 Configuración de los tests y ejecución

Para el uso de tests en este proyecto hemos declarado dos ficheros scripts que contendrán los tests unitarios programados en Jest.

- **Layout.test.js**: contendrá los tests para pruebas de interfaz
- **tests.js:** contendrán los test que prueban las funcionalidades de la aplicación.

Para ejecutarlos de manera automática con un solo comando, se ha añadido la opción script en archivo *package.json,* este se encuentra en la carpeta raíz del proyecto, contiene datos de configuración y las dependencias de la aplicación que se instalan al ejecutar *npm install.* Con esta configuración Jest sabrá que archivo es el que se debe ejecutar cuando queremos lanzar los tests:

```
{
  "scripts": {
    "test": "node scripts/test.js --env=jsdom --forceExit"
  }
}
```

El resultado es que con solo llamar al siguiente comando tenemos Jest lanzado en el terminal.

```
npm run test
```

Cuando se ejecuta este comando aparecerá en la consola una serie de funciones a escoger, si queremos ejecutar todos los test a la vez nos interesará la opción *press a*, pero podemos escoger entre todas esas opciones dependiendo de las necesidades que tengamos en el momento.

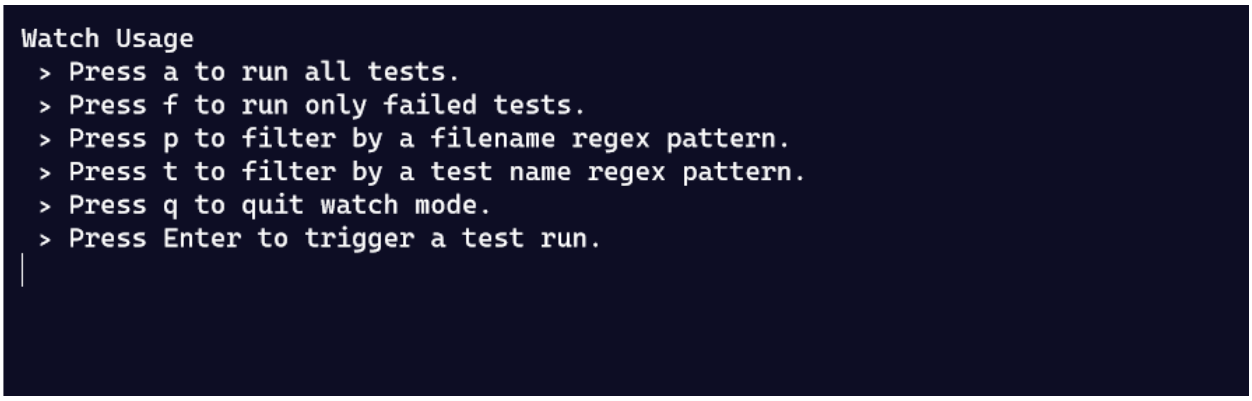


Figura 37: Salida del comando *npm run tests*

8.2 Pruebas unitarias

8.2.1 Introducción y descripción de los tests

Hemos creado tests que verifiquen que el programa actúa correctamente al pedir y recibir los datos. De forma que podamos probar las casuísticas del programa vistas en los casos de usos.

Hasta ahora hemos explicado cómo realizar la configuración e instalación de la herramienta Jest para integrar los test con nuestra aplicación y GitHub. Para un mejor entendimiento de estas pruebas unitarias vamos a describir cómo hemos llevado a cabo la codificación de los test. Vamos a detallar a nivel de código nuestras pruebas para comprender mejor esta sección.

El fichero *test.js* contiene los test que describiremos en la siguiente sección, estas pruebas se enfocarán en probar las funciones del programa que piden y procesan los datos. En el apartado *4. Tecnologías Usadas: Jest* probábamos la función *suma()* aquí probaremos las funciones de la aplicación.

En primer lugar, vamos a ver cómo se configuran las variables que necesitaran los tests.

```
// Necesitamos importar las funciones que vamos usar de los ficheros thunks y actions y la función
//configureStore()
// En thunks se encuentran las funciones que usa el programa
import {configureTestStore} from '../configureStore'
import * as thunks from '../thunks'
import * as actions from '../actions'

// Declaramos una variable global que guarde un token para hacer pruebas
const REFRESH_TOKEN = "AE0u-NeO3u_IVtOuT1hnkcD88MC5J1ud-"

// Declaramos una variable que tenga datos de verdad de un usuario para hacer pruebas
// No hay que modificarlo en cada sesión, para este test hemos usado nuestro usuario
const testUser = {
  token: '3f6da023f139cb9e5df0fccd448ad6ad1e8958',
  photoURL: 'https://avatars3.githubusercontent.com/u/43379210?v=4',
  email: 'maaluuz@gmail.com',
  username: 'maluzzz',
  refreshToken: REFRESH_TOKEN,
}
```

Código 10: Fichero script para los tests parte I

El token para los test lo hemos generado en a través de la interfaz de GitHub, de esta forma el token no caduca.

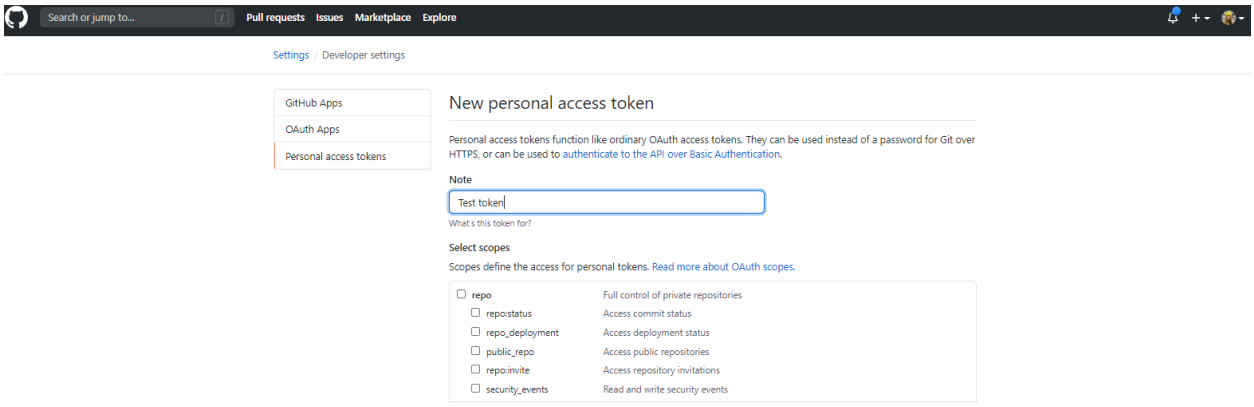


Figure 38: Interfaz de GitHub para genera un token

En estas pruebas Jest simula la ejecución del programa, por lo que necesitamos darle las variables necesarias, en este caso *testUser* y un token. Además, el programa centra todos sus movimientos en el Store, por lo que también es necesario usar la función *configureStore()*, esta función nos la provee el fichero configureStore.js, en este fichero hacemos uso de Redux para crear un Store inicial que contendrá todo lo necesario para que Jest pueda ejecutar la aplicación. Al usar esta función obtenemos el objeto Store que contiene los métodos necesarios como *dispatch* o *getState*, imprescindibles para la ejecución de estos tests

A continuación, vamos a describir cómo funciona paso a paso uno de los tests

1) Usamos la función *describe()* que nos provee Jest, esta nos permite pasarle un título y un bloque de varios tests

```
// Esta función nos permite ejecutar un bloque entero de tests
describe('redux/repos/tests', () => {
/*Bloque de tests*/
})
```

Código 11: Fichero script para los tests parte II

2) En el interior de esta función usaremos *it(name, fn)* que nos permite ejecutar un test, le pasamos como parámetros un título y *fn* hace referencia a la función que debe ejecutar Jest, si esa función contiene algún error fallará el test y nos lo dirá por pantalla.

   Esta función debe simular paso a paso las funciones que se ejecutan cuando un usuario usa la aplicación. De esta forma se ponen a prueba las funciones. Cuando las funciones se han ejecutado, ya por último, debemos comprobar que el resultado de la ejecución es el correcto y no otro.

```
// PRIMER TEST: dentro de describe() empezaremos a llamar a las funciones que deben ejecutarse para
// probar una funcionalidad concreta.
 it(
// Descripción del test como primer parámetro
    'Debe obtener los datos del proyecto público facebook/react',
```

```
    async () => {
// Obtenemos un store inicial
      const store = configureTestStore()
// Lanzamos la acción de iniciar sesión de nuestro testUser
      store.dispatch(actions.userSignedIn(testUser))
// Lanzamos la acción de pedir los datos en este caso es
// fetchSinbleRepoDataByLogin(nombreRepositorio)
      await store.dispatch(thunks.fetchSingleRepoDataByLogin('facebook/reac
t'))
// Llamamos al estado de la aplicación
      const state = store.getState()
// Si ha ido bien la petición en el estado de la aplicación la variable erroredRepos debe estar vacía.
      expect(state.repos.erroredRepos).toEqual([])
    },
    10000 // Temporizador, Si el tiempo de ejecución es mayor de 10 segundos se acaba el test y da
// error
  )
})
```

Código 12: Fichero script para los tests parte III

Como vemos en el código hemos definido paso a paso qué funciones se ejecutarán hasta llegar a obtener los datos de un proyecto de GitHub. En este test hemos llamado a*userSignedIn(user)* y

fetchSingleRepoDataByLogin(nombreRepositorio) .

De manera parecida a la que se muestra en *Código 12: Fichero script para los tests parte III* se han construido cuatro tests diferentes que verficarán que el proceso que hemos visto en los diagramas de actividades sea el correcto. Estas pruebas, como hemos visto antes, estarán enfocadas en que se consigan los datos y se procesen correctamente, además de probar que los mensajes de error que deban mostrarse sean los adecuados.

8.2.2 Obtención de datos

En esta sección describiremos cada prueba, tanto su objetivo como su código y cómo vamos a verificar si ha sido correcto.

PR_FUN_01	Obtiene los datos de la API correctamente de un proyecto privado
Objetivo	El objetivo de esta prueba es verificar que la aplicación es capaz de obtener los datos de un repositorio existente y privado sin fallos.
Descripción	Esta prueba llama a las funciones que obtienen y procesan los datos de la API de GitHub. Le pasamos como parámetro el nombre de un proyecto que sabemos que existe, es privado y contiene pull requests.

Código	``` describe('redux/repos/tests', () => { it('Debe obtener los datos del proyecto privado jsdario/yeti', async () => { //Descripción del test // Creamos un store con datos de ejemplo const store = conFiguraTestStore() // Lanzamos la acción de iniciar sesión de nuestro testUser store.dispatch(actions.userSignedIn(testUser)) // Lanzamos la acción de pedir los datos en este caso es // fetchSinbleRepoDataByLogin(nombreRepositorio) await store.dispatch(thunks.fetchSingleRepoDataByLogin(jsdario/yeti)) // Llamamos al estado de la aplicación const state = store.getState() // Si ha ido bien la petición en el estado de la aplicación la variable erroredRepos debe estar vacía. expect(state.repos.erroredRepos).toEqual([]) // por último se le da un tiempo de ejecución al test, en nuestro caso 10s }, 10000) ``` Código 13: Test que Obtiene los datos de la API correctamente de un proyecto privado
Resultado Esperado	Se espera que pueda obtener los datos sin problemas, esto se verifica cuando en el estado en la variable de errores no contiene ningún valor. Se hace referencia esta comprobación en la línea del código: `expect(state.repos.erroredRepos).toEqual([])`
Comentario	Para este caso hemos usado un token privado, definido anteriormente, que se obtendrá en el proceso de autorización de la aplicación.

PR_FUN_02	Obtiene los datos de la API correctamente de un proyecto público
Objetivo	El objetivo de esta prueba es verificar que la aplicación es capaz de obtener los datos de un repositorio existente y público sin fallos.
Descripción	Esta prueba llama a las funciones que obtienen y procesan los datos de la API de GitHub. Le pasamos como parámetro el nombre de un proyecto que sabemos que existe, es público y contiene pull requests.
Código	El código de este test es el descrito en ***Código 12: Fichero script para los tests parte III***

Resultado esperado	Esta prueba verifica si se pueden obtener los datos sin problemas. Se espera que pueda obtener los datos sin problemas, que en el estado en la variable de errores no tengamos ningún valor. Se hace referencia a este resultado en la línea: `expect(state.repos.erroredRepos).toEqual([])`
Comentario	No es necesario para este test el uso de un token.

PR_FUN_03	**No obtiene datos y se guarda un error si se introduce un proyecto que no existe**
Objetivo	El objetivo de esta prueba es verificar el caso en el que el usuario introduce en el buscador un nombre de un proyecto que no existe. En este caso el estado de la aplicación debe contener un mensaje de error en la variable correspondiente.
Descripción	Esta prueba llama a las funciones que obtienen y procesan los datos de la API. Le pasamos como parámetro a la función que obtiene los datos el nombre de un proyecto que sabemos que no existe.
Código	

```
describe('redux/repos/tests', () => {
  it('Debe guardar un error si el proyecto no
existe', async () => {
// Creamos un store con datos de ejemplo
    const store = conFiguraTestStore()
// Lanzamos la acción de iniciar sesión de nuestro testUser
    store.dispatch(actions.userSignedIn(testUser))
// Lanzamos la acción de pedir los datos en este caso es
// fetchSinbleRepoDataByLogin(nombreRepositorio)
    await store.dispatch(thunks.fetchSingleRepoDataByLogin('n
oexisto/noExisto'))
// Llamamos al estado de la aplicación
      const state = store.getState()
// Si ha ido bien la petición en el estado de la aplicación la variable erroredRepos no
debe estar vacía.

    expect(state.repos.erroredRepos).toEqual(['noexisto/noExi
sto'])
}, 10000)
```

Código 14: Código del test para un proyecto que no obtiene datos

Resultado esperado	Si el programa ha manejado con éxito el error en la variable ***erroredRepos*** debe haberse guardado el nombre del proyecto introducido. Se hace referencia a este resultado en la línea: `expect(state.repos.erroredRepos).toEqual(['noexisto/noExisto'])`
Comentario	No miramos el tipo de error en este test. Nos basta con comprobar que el programa reconoce que hay un error en este tipo de acción.

Si las pruebas se han realizado correctamente al ejecutarlos la salida por pantalla debe ser la siguiente:

```
 PASS  src/redux/repos/tests.js (10.543s)
  redux/repos/tests
    ✓ Debe obtener los datos del proyecto publico facebook/react (4160ms)
    ✓ Debe obtener los datos del proyecto privado jsdario/yeti (3625ms)
    ✓ Debe guardar un error si el proyecto no existe (691ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
```

Figura 39: Resultado de las pruebas unitarias para la obtención de datos del api

PR_FUN_04	**El proyecto no tiene pull requests que analizar**
Objetivo	Comprobar que el estado local tiene los valores correctos si se intenta analizar un repositorio sin pull requests. Es decir, que almacenamos el mensaje correspondiente al error
Descripción	Esta prueba llama a las funciones que obtienen y procesan los datos de la API. Le pasamos como parámetro el nombre de un proyecto que existe pero que sabemos que no tiene pull requests, por lo que en el estado se debe guardar error y el mensaje para identificarlo. Las variables encargadas de esto serán **erroredRepos** y **repoErrorsById**.

Código	

```
it('Debe guardar el mensage de error correcto cuando no hay p
ull requests', async () => {
 // Creamos un store con datos de ejemplo
    const store = conFiguraTestStore()
// Declaramos una variable para guardar el mensaje que debe almacenar el estado
// y así compararlo más tarde.
    const mensajeError = {"Maluzzz/phpmyadmin":
     {"message": "Reduce of empty array with no initial value"}}
// Lanzamos la acción de iniciar sesión de nuestro testUser
    store.dispatch(actions.userSignedIn(testUser))
// Lanzamos la acción de pedir los datos en este caso es
// fetchSinbleRepoDataByLogin(nombreRepositorio)
   await store.dispatch(thunks.fetchSingleRepoDataByLogin('Maluzzz/
phpmyadmin'))
    const state = store.getState()
// Si ha ido bien la petición y ha recibido el mensaje del API de GitHub en el estado
// de la aplicación la variable erroredRepos no debe estar vacía y repoErrorsById
// tampoco
   expect(state.repos.erroredRepos).toEqual(["Maluzzz/phpmyadmin"])
   expect(state.repos.repoErrorsById).toEqual(mensajeError)
  }, 10000)
```

Código 15: *Test para un proyecto que no tiene pull requests que analizar*

Resultado esperado	Se espera que se guarde el nombre del proyecto que tiene error. Además, se debe guardar el mensaje que devuelve la API cuando no hay pull requests, que es: **`"Reduce of empty array with no initial value"`** Se hace referencia a este resultado en las líneas: `expect(state.repos.erroredRepos).toEqual(["Maluzzz/phpmyadmin"])` `expect(state.repos.repoErrorsById).toEqual(mensajeError)`
Comentario	El Proyecto usado para esta prueba ha sido Maluzzz/phpmyadmin

8.3 Test unitario de interfaz

En el fichero *Layout.test.js* se encuentra la prueba de interfaz. La ejecución de esta prueba permite comprobar que los componentes no tienen variables que vacías o no definidas provocando fallos al cargarse. El código que permite probar la interfaz es el siguiente:

```
// En primer lugar necesitamos los imports de las librerías que usamos de React y Redux
import React from 'react'
import ReactDOM from 'react-dom'
import {Provider} from 'react-redux'

// Importamos al componente que instancia todos los demás componentes:  Layout
import Layout from './Layout'
// Y usamos configureStore, visto en el apartado anterior
import configureStore from './redux/configureStore'

// Comienza la declaración de la prueba con test y el mensaje descriptivo del test

test('Test: El componente Layout funciona correctamente',
// A partir de aquí es la declaración de la función que probará Layout
() => {
// Creamos un elemento div para pasarle los parámetros del Store
  const div = document.createElement('div')
// Declaramos la variable objecto store que pasaremos a nuestro componente Provider de Redux
  const {store} = conFiguraStore()
// La siguiente variable será la que pasameos a la función encargada de cargar la interfaz de manera
// aislada
  const testJsx = (
    <Provider store={store}>
      <Layout />
    </Provider>
  )
// ReactDOM.render() instanciará provider, layout y los sucesivos componentes con el store que hemos
// cargado anteriormente, si en este proceso algo falla el test nos avisara ya esta prueba está dentro de
// test()
  ReactDOM.render(testJsx, div)
})
```

Código 16: Código del script para la prueba de interfaz

La prueba que corresponde con el código visto anteriormente es el siguiente:

PR_INTF_01	Los componentes se cargan correctamente
Objetivo	La interfaz es lo más cercano al usuario, por ello es de vital importancia que no se encuentren fallos críticos en la misma. El objetivo de esta prueba es comprobar que toda la interfaz de la aplicación carga correctamente.
Descripción	Se ha programado un script con Jest que comprueba que pasado los parámetros a los componentes todos ellos cargan correctamente.
Verificación	El resultado esperado, es que el script programado se ejecute sin fallos. El resultado por pantalla al ejecutar los tests debe ser el siguiente:  Esto indicia que Jest se ha ejecutado sin errores y ha pasado el test sin ningún problema.
Comentario	-

8.4 Integración de pruebas continuas

CircleCI ha sido la herramienta elegida para que cada vez que se hacía un cambio en el código y se subía a una nueva rama en el repositorio remoto se volvieran a ejecutar las pruebas. Si falla alguno de los tests se manda una alerta al email del desarrollador, en este caso mi email y aparece un mensaje en rojo en la interfaz de GitHub.

Vercel nos provee de otra prueba más ya que esta herramienta es la encargada de compilar y desplegar la aplicación en entorno de pruebas. Por lo tanto, es una verificación más de que la aplicación funciona. Si algo falla, al tenerlo integrado con GitHub, también se nos manda una alerta al email y aparece un mensaje en rojo en la interfaz de GitHub.

9 MEJORAS Y CASO REAL

En este apartado comentaremos las posibles mejoras que tiene la aplicación y conclusiones sobre el uso comercial de la misma, ya que se ha probado en entornos laborales.

9.1 Mejoras

La aplicación como tal es simple, con tres visualizaciones concretas. Actualmente en el mercado hay aplicaciones y software de productividad mucho más completos, como *GitPrime* [28], que muestran simplemente métricas de la cantidad de trabajo, sin procesar los datos. Lo novedoso de este proyecto es ese procesamiento de los datos cuantitativos para ayudar al gestor de un proyecto. Aun así, hay mejoras que se podrían hacer en la aplicación. Al emprender con este proyecto, las ideas de mejoras han surgido conforme se encontraban clientes, ellos mismos han propuesto alguna al usar la aplicación en su día a día.

9.1.1 Mejoras comerciales

Ahora mismo la aplicación solamente la pueden utilizar usuarios de GitHub, pero no es la única plataforma en la que almacenar repositorios git, sabemos que existen otras muy famosas como GitLab con millones de usuarios, BitBucket, etc.

Integrar estas plataformas ensancharía el embudo de posibles clientes para esta aplicación.

9.1.2 Mejoras en el algoritmo

La construcción y desarrollo de código es más un arte que una ciencia, y el proceso de aprendizaje y mejora se consigue con la experiencia. Es así, que el algoritmo tiene potencial para incluir métricas de todo tipo. Las principales y más interesantes podría ser:

- Procesar los títulos y descripciones con machine Learning y obtener una ponderación en función de haber alimentado el Sistema con buenas descripciones.
- Añadir como parámetro si se han pasado los test o no en un pull request como valor positivo a la puntuación de la tarea. Mostrar el porcentaje de tareas que no pasan los Tests.

9.1.3 Mejoras en la aplicación

Durante el tiempo que se ha emprendido, con esta idea, la aplicación ha sido probada por más de 20 empresas distintas. Equipos de desarrollo que han dado su opinión y su experiencia de usuario con la aplicación. Entre ellas hemos recogido las sugerencias más recurrentes:

- Para un gestor es interesante poder descargar estos datos en un archivo formato excel. Es una mejora de funcionalidad bastante fácil de implementar.
- Concretar un filtro de fecha, para poder ver las visualizaciones en un periodo de tiempo concreto, o ver el rendimiento en días pasados.
- Filtro por autores, para poder quitar ciertos usuarios de una visualización.

Estas mejoras han sido escuchadas y se desarrollarán en futuros ciclos del proceso Software.

En versiones posteriores de la aplicación esto estará implementado.

9.2 Escenario real

Este proyecto nació como una herramienta interna en la Startup Yeti Smart Home como método para ayudar y gestionar el proyecto. Además de ser una herramienta que servía como motivación para el equipo. En 2019 esta Startup cerró y decidimos emprender con esta aplicación. Ya que si nos había servido y motivado a nosotros queríamos ver si tenía hueco en el mercado.

En marzo de 2019 nos escogieron para la iniciativa Andalucía Open Future [27], creada por Telefónica y la Junta de Andalucía, donde se da apoyo, oficinas y mentorías a startups. Después de casi un año emprendiendo hemos conseguido una tracción alta a la aplicación.

En total, desde el momento de poner en línea esta aplicación, en marzo de 2019, se han analizado 6000 proyectos y han iniciado sesión más de 329 usuarios.

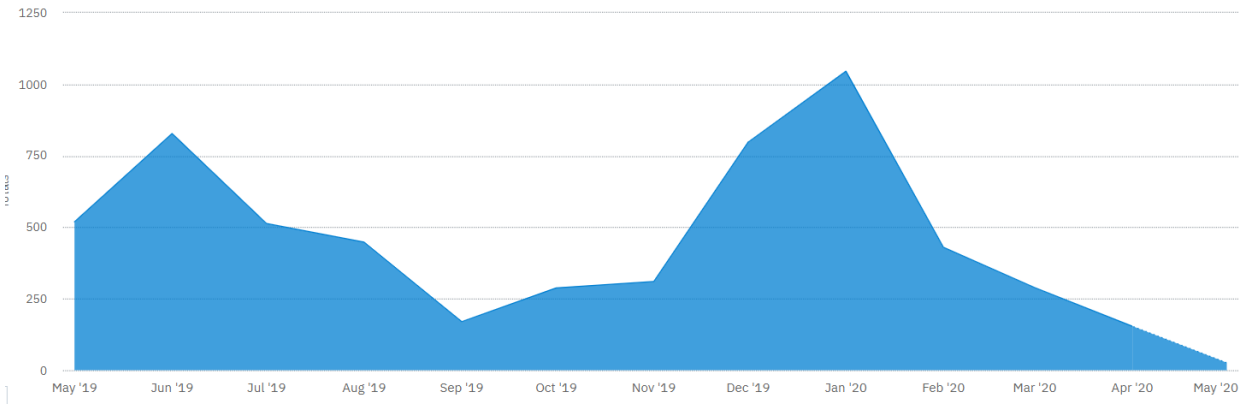


Figura 40: Proyectos analizados desde mayo del 2019 a mayo de 2020

Hemos conseguido tener clientes de pago, el primero fue el director de Ingeniería de una cadena de noticias de televisión americana, le siguió una empresa londinense que nos usa en su día a día, en un equipo de seis personas. En esta última el mismo gestor del proyecto nos envía reportes sobre qué le gustaría ver en la aplicación y qué fallos encuentra. Con los primeros clientes conseguimos validar el producto y nos animó a seguir emprendiendo, teniendo la certeza de que la aplicación aporta valor a los equipos de ingenieros, tanto en empresas grandes como pequeñas.

REFERENCIAS

[1] H. A. F. Fernández*, «Procesos de ingeniería de software,» *Vínculos V6, Nº1,* vol. 2, nº 13, 2009.

[2] M. D. M. G. L. W. K. B. a. T. Savor, « "Continuous deployment at facebook and oanda,",» de *Proceedings of the 38th International Conference on Software*, 2016.

[3] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Ley_de_Campbell. [Último acceso: 07 Abril 2020].

[4] S. B. Neil A. Ernst, «Measure It? Manage It? Ignore It?,» Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA, USA.

[5] A. Villar y S. Matalonga, «Definiciones y tendencia de deuda técnica: Un mapeo sistemático de la literatura.,» de *Computer Science; Published in CIbSE* , 2013.

[6] C. Scott y S. Ben , Pro Git, Apress.

[7] GitHub, «Octoverse,» GitHub, 2019. [En línea]. Available: https://octoverse.github.com/. [Último acceso: 2020].

[8] F. &. C. F. &. N. N. &. S. A. Ebert, «Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies,» 2019.

[9] G. Gousios, M.-A. Storey y A. Bacchelli, «Work Practices and Challenges in Pull-Based,» de *IEEE/ACM 38th IEEE International Conference on Software Engineering*, 2016.

[10] G. Gousios, M. Pinzger y A. v. Deursen, «An Exploratory Study of the Pull-Based SoftwareDevelopment Model,» de *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[11] P. Pooput y P. Muenchaisri, «FindingImpact Factors for Rejection of Pull Requests on,» de *Proceedings of the 2018 VII International Conference on Network, Communication and Computing*, 2018.

[12] E. A. Santos y A. Hindle, «Judgin a Commit by Its Cover,» de *MSR '16: Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[13] O. Kononenko, Baysal y M. Godfrey, «Code Review Quality: How Developers See It,» de *IEEE/ACM 38th IEEE International Conference on Software Engineering*, 2016.

[14] O. Kononenko, R. Tresa, B. Olga, M. Godfrey∗, D. Theisen y B. de Water, «Studying Pull Request Merges: A Case Study of Shopify's Active Merchant,» de *ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018.

[15] «Developers Mozilla,» [En línea]. Available: https://developer.mozilla.org/es/. [Último acceso: 14 Abril 2020].

[16] J. Cohen, S. Teleki y E. Brown, Best Kept Secrets of Peer Code Review, SmartBear Software, 2013.

[17] A. Bacchelli y C. Bird, «Expectations, outcomes, and challenges of modern code review,» de *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[18] «React 16.13.1,» [En línea]. Available: https://es.reactjs.org/. [Último acceso: 07 Abril 2020].

[19] «Documentación de Redux en español,» [En línea]. Available: https://es.redux.js.org/. [Último acceso: 08 Abril 2020].

[20] «D3 Data-Driven Documents,» [En línea]. Available: https://d3js.org/. [Último acceso: 07 Abril 2020].

[21] «Wikipedia: Node.js,» [En línea]. Available: https://es.wikipedia.org/wiki/Node.js. [Último acceso: 08 Abril 2020].

[22] «GitHub API Documentation,» [En línea]. Available: https://developer.github.com/v3/.

[23] G. Engineering, «The GitHub Blog,» 2016. [En línea]. Available: https://github.blog/2016-09-14-the-github-graphql-api/. [Último acceso: 21 Junio 2020].

[24] «Introduction to GraphQL,» [En línea]. Available: https://graphql.org/learn/. [Último acceso: 08 Abril 2020].

[25] «Wikipedia: Jest (JavaScript framework),» 20 Febrero 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Jest_(JavaScript_framework). [Último acceso: 7 Abril 2020].

[26] «Documentación de Firebase Authentication,» [En línea]. Available: https://firebase.google.com/docs/auth?hl=es. [Último acceso: 08 Abril 2020].

[27] «https://code.visualstudio.com/docs,» Microsoft, [En línea]. Available: https://code.visualstudio.com/docs. [Último acceso: 09 Abril 2020].

[28] Pluralsight, «GitPrime Pluralsight,» [En línea]. Available: https://www.pluralsight.com/product/flow.

[29] «Andalucía Open Future,» Telefónica, [En línea]. Available: https://andalucia.openfuture.org/.

[30] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.

[31] G. Gousios, «Work Practices and Challenges in Pull-Based,» de *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*, 2016.

[32] E. A. a. H. A. Santos, «Judging a Commit by Its Cover: Correlating Commit Message Entropy with Build Status on Travis-CI,» de *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin, Texas, 2016.

[33] D. Hu, T. Wang, J. Chang y Y. Z. Gang Yin, «Multi-discussing Across Issues in GitHub: A Preliminary Study,» de *25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018 .