UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Integración de herramientas para gestión de proyectos con sistemas de control de versiones: un caso práctico.

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Arnau Ruiz Fuster

Tutor: Patricio Letelier Torres

Curso 2020-2021

Resum

A mesura que els projectes van fent-se cada vegada més i més complexos i amb un número major d'integrants a l'equip, la implementació dels sistemes de control de versions es converteix en indispensable per a qualsevol grup de treball que vulga accelerar la producció de codi i simplificar dràstricament la gestió d'aquest. Paral·lelament, les ferramentes software de gestió àgil del treball estàn sent cada vegada més utilitzades a mesura que les empreses van adonant-se de la importància de l'organització i estructuració de la càrrega de treball. En aquest treball de fi de grau es realitza un anàlisi sobre les interfícies de programació d'aplicacions (API) que els serveis de control de versions ens ofereixen a l'hora de contruir aplicacions; un estudi sobre les integracions existents d'aquests sistemes en les ferramentes de gestió àgil del treball, i per últim la construcció d'un prototip de gestió de projectes que integra un sistema de control de versions específic. Aquesta aplicació farà ús de l'API de GitHub i del *framework* Angular.

Paraules clau: Sistemes de control de versions, ferramentes de gestió àgil del treball, GitHub, GitLab, Jira, PivotalTracker, Angular

Resumen

A medida que los proyectos se vuelven cada vez más y más complejos y con un mayor número de integrantes del equipo, la implantación de los sistemas de control de versiones se convierte en indispensable para cualquier grupo de trabajo que quiera acelerar la producción de código y simplificar drásticamente la gestión de este. Paralelamente, las herramientas software de gestión ágil del trabajo están siendo cada vez más utilizadas a medida que las empresas van advirtiendo la importancia de la organización y estructuración de la carga de trabajo. En este trabajo de fin de grado se realiza un análisis sobre las interfaces de programación de aplicaciones(API) que los servicios de control de versiones nos brindan a la hora de construir aplicaciones; un estudio sobre las integraciones existentes de dichos sistemas en las herramientas de gestión ágil del trabajo y por último la construcción de un prototipo de gestión de proyectos que integra un sistema de control de versiones específico. Dicho trabajo hará uso de la API de GitHub y del *framework* Angular.

Palabras clave: Sistemas de control de versiones, herramientas de gestión ágil del trabajo, GitHub, GitLab, Jira, PivotalTracker, Angular

Abstract

As projects become more and more complex and with a larger number of team members, the implementation of version control systems becomes indispensable for any working group that wants to accelerate code production and drastically simplify code management. In parallel, software tools for agile work management are being increasingly used as companies realise the importance of organising and structuring the workload. This Final Degree Project discusses the programming interfaces that version control services provide us with when building applications; a study of the existing integrations of these systems into agile work management tools and finally the construction of a project management prototype that integrates a specific version control system. This application will make use of the GitHub API and the Angular framework.

Key words: Version control systems, agile work management tools, GitHub, GitLab, Jira, PivotalTracker, Angular

Índice general

Índice general v
Índice de figuras vii

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 1
1.3 Estructura del trabajo . . . 2
2 Sistemas de control de versiones 3
2.1 Introducción . . . 3
2.2 Análisis de las API . . . 3
2.2.1 GitLab . . . 4
2.2.2 GitHub . . . 6
2.3 Conclusiones . . . 9
3 Estudio de integraciones existentes 11
3.1 Introducción . . . 11
3.2 Jira . . . 12
3.3 PivotalTracker . . . 16
3.4 Conclusiones . . . 20
4 Desarrollo de integración para Worki 21
4.1 Introducción a Worki . . . 21
4.2 Descripción del problema . . . 21
4.3 Requisitos . . . 22
4.4 Diseño . . . 22
4.5 Pruebas . . . 25
4.6 Cronología . . . 30
5 Conclusiones finales y trabajo futuro 33
Referencias 35

Índice de figuras

2.1 Respuesta a la llamada que lista todos los *commits* en GitLab 5
2.2 Respuesta a la llamada que lista la información de un único *commit* en GitLab 6
2.3 Respuesta a la llamada que recupera todos los *commits* de un repositorio en GitHub 7
2.4 Respuesta a la llamada que recupera la información de un *commit* específico en GitHub 8
2.5 Respuesta a la llamada que recupera todos los *commits* de un fichero en GitHub 9

3.1 Integraciones de Jira 12
3.2 Inicio de sesión de GitHub en Jira 13
3.3 Detalle de la pestaña *Git Roll Up* 13
3.4 Detalle de la pestaña *Git Commits* 14
3.5 Página del *diff* de un *commit* 14
3.6 Detalle de la página de confirmaciones 15
3.7 Creación de ramas desde Jira 15
3.8 Ramas asociadas al proyecto 15
3.9 *Merge requests* de las ramas asociadas a la tarea 16
3.10 Creación de un proyecto en PivotalTracker 17
3.11 Ventana de configuración de la integración en PivotalTracker 17
3.12 Ventana de configuración de la integración en GitHub 18
3.13 Flujo del estado de las tareas en PivotalTracker 19
3.14 Creación de dos ramas para una tarea 20
3.15 Asociación de las ramas creadas en GitHub a la UT 20

4.1 Esbozo inicial del prototipo 23
4.2 Pantalla principal del prototipo 23
4.3 Ventana de tarea 24
4.4 Detalle de un *commit* en la página de GitHub 24
4.5 Detalle de un fichero en la página de GitHub 24
4.6 Pruebas unitarias de *app.component* 26
4.7 Pruebas unitarias de *ut-list.component* 27
4.8 Pruebas unitarias de *ut-body.component* 28
4.9 Pruebas unitarias de *gfetch.service* 29
4.10 Tests con éxito en Jasmine 30

CAPÍTULO 1
Introducción

1.1 Motivación

Los sistemas de control de versiones juegan un papel fundamental en el desarrollo de cualquier aplicación software ya que ayudan a la gestión del código y de los documentos, permitiendo una forma sencilla, pero efectiva, de organización, administración y coordinación del equipo.

La gestión de proyectos es la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades de un proyecto con el objetivo de satisfacer los requisitos establecidos [4].

Ambos ámbitos de las herramientas mencionadas, aunque no estrechamente relacionados en la práctica, tienen un objetivo común: la organización del trabajo en un equipo de desarrollo. Es este objetivo común la motivación de este trabajo de fin de grado, una respuesta a la pregunta de cómo podríamos unir las dos áreas mencionadas: un sistema de control de versiones integrado con una herramienta de gestión del trabajo, que permita a los componentes de un equipo de trabajo disponer de la información de la gestión del proyecto así como la actividad del repositorio de código.

Dichos datos que nos proporcionan los sistemas de control de versiones nos podrían generar nueva información que nos sería útil en varios campos, ya sea a la hora de organizar y planificar pruebas de regresión, especialmente si son manuales, o analizar el éxito o fracaso del proyecto, ofreciendo otro campo para la retrospección y autocrítica.

El prototipo que se ha realizado es una propuesta de integración para Worki [1], una herramienta de gestión ágil del trabajo que implementa un variado abanico de funcionalidades, pero que no obstante se podría beneficiar de la integración mencionada y de la información que puede aportar.

1.2 Objetivos

Los objetivos de este trabajo son múltiples, por una parte el estudio de las funcionalidades ofrecidas por los sistemas de control de versiones más populares en la actualidad, GitLab y GitHub; el análisis de los mecanismos de integración que nos ofrecen las herramientas de gestión de proyectos, es decir, un análisis del estado del arte respecto a los programas software en el ámbito especificado y por último el desarrollo de una aplicación que integre las dos herramientas anteriores, como propuesta de integración para una herramienta de gestión ágil de proyectos, en nuestro caso, Worki.

[1] http://www.tuneupprocess.com/

El prototipo en cuestión deberá ser capaz de:

1. Pedir y recibir la información deseada desde los servidores de GitHub.
2. Procesar dicha información en base a los requisitos descritos en el capítulo 4.
3. Actualizar la interfaz con dicha información.

1.3 Estructura del trabajo

El siguiente trabajo ha sido dividido en los siguientes capítulos:

En el primer capítulo se introduce el contexto de este trabajo, la motivación, los objetivos propuestos y esta misma estructura.

En el segundo capítulo se exponen los sistemas de control de versiones, qué son y por qué son importantes en la actualidad. Seguidamente se hace un estudio de las API de dos servicios seleccionados, GitHub y GitLab, explicando las llamadas más relevantes para el proyecto.

En el tercer capítulo se analiza el estado del arte sobre las integraciones de los servicios anteriormente mencionados con respecto a dos herramientas seleccionadas, explicando con detalle su configuración.

En el cuarto capítulo se describe el desarrollo de la propuesta de integración, su diseño, los requisitos que ha de cumplir y las pruebas unitarias una vez finalizado el desarrollo. Adicionalmente, se introduce la herramienta Worki y su relación con el proyecto.

En el quinto y último capítulo, se recapitula sobre el trabajo, se extraen las conclusiones y se discute sobre el trabajo futuro.

CAPÍTULO 2
Sistemas de control de versiones

2.1 Introducción

La necesidad de saber quién ha modificado qué en un sistema organizado de ficheros existe desde hace décadas, y es más relevante hoy en día, en la era de la información, donde esa misma necesidad se ha vuelto mucho más compleja que antaño. Por esta misma razón se originan los sistemas de control de versiones, en adelante también referidos como SCV, un ámbito de herramientas *software* que permiten ayudar a administrar y gestionar las modificaciones realizadas en el tiempo de un fichero o conjunto de ficheros determinado, y de esta forma, se tiene un registro completo de cuándo ha sido modificado, quién lo ha hecho y qué se ha editado. Las ventajas del control de versiones no acaban en la trazabilidad de autoría de las modificaciones de un registro, también hacen inmensamente fácil la resolución de conflictos a la hora de realizar cambios de dos fuentes distintas a un mismo archivo; ofrece la posibilidad de revertir a una versión anterior del fichero, por si se quiere volver a una versión sin errores o a una más estable; además, el almacenamiento centralizado de los ficheros de un proyecto hace más sencilla la organización y la gestión de ficheros entre los miembros de un equipo, además de servir como una copia de seguridad externa a fallos e inconvenientes humanos [3].

En lo que a sistemas de control de versiones se refiere, es conveniente integrar en la herramienta uno que sea popular y familiar entre los usuarios que vayan a utilizarlo, por lo que se van a estudiar las interfaces de acceso que ofrecen los servicios más utilizados actualmente: Github [1] y GitLab [2]. Se han elegido dichas páginas por considerarse las más conocidas entre los desarrolladores de *software*, ya sean casuales o en el ámbito profesional, poniendo como ejemplo a GitHub, que en el momento de la confección de este documento tiene más de 55 millones de usuarios y más de 198 millones de repositorios creados.

2.2 Análisis de las API

Una API es una biblioteca de métodos y funciones que define las interacciones entre uno o más sistemas *software*, es decir, es un intermediario para la comunicación y para el traspaso de información entre dos componentes, en este caso, la aplicación a desarrollar y un servidor. El objetivo final de una API es muy variado, pero la funcionalidad clave es que estas nos proporcionan un nivel de abstracción respecto a la programación que permiten a los desarrolladores crear y extender funcionalidades complejas de manera

[1] https://github.com/
[2] https://gitlab.com/

sencilla y simplificada sin requerir el acceso al código fuente, y aunque hay varios tipos de API, el análisis que se llevará a cabo estará centrado en el tipo específico de las interfaces provistas, en este caso, APIs web. Las interfaces que vamos a analizar son todas servicios web RESTful, es decir, basadas en REST (*REpresentational State Transfer* o Transferencia de Estado Representacional), término que se refiere a un estilo de arquitectura *software* basado en una serie de operaciones con las que se puede actuar sobre unos recursos web, es decir, mediante unas llamadas específicas, se puede listar, modificar, borrar y recuperar información de un servidor [6].

Para la creación del prototipo del proyecto necesitaremos la ayuda que nos ofrecen los servicios de control de versiones, así que con el objetivo de comprobar qué utilidades nos podrán ser útiles, vamos a analizar únicamente las más relevantes de dos de los servicios más utilizados por desarrolladores en el mundo, siendo uno de ellos el que emplearemos en este proyecto: GitHub y GitLab.

Siendo unas utilidades tan populares, todas las APIs de los servicios que vamos a comparar ofrecen una extensa lista de *endpoints*, o puntos de acceso, para utilizarla, complementada con una documentación que explica qué hace cada uno de ellos, así como ejemplos para algunos. Sin embargo, para el alcance y objetivos de este TFG, no nos harán falta una gran parte de estos métodos, y únicamente se expondrán los que se consideren relevantes.

2.2.1. GitLab

El apartado de la API de GitLab [3] ofrece una documentación estructurada y completa en secciones según el ámbito de la información que queramos obtener, desde seguimiento de errores, hasta la obtención de la iteración actual del proyecto, pasando por las llamadas básicas que recuperan información de usuario, de un repositorio, o de los *commits* realizados.

Como un apunte de uso básico, cualquier solicitud que se realice deberá ir precedida de la URL de GitLab, *https://gitlab.com*, de la etiqueta *api* y la versión actual de la API, actualmente *v4*, por lo que las llamadas que se expongan en este apartado deberán incluirlo en un caso de uso real, y en este trabajo se ha optado por poner cada llamada como el método REST correspondiente seguido por la continuación pertinente de la URL. Dicho esto, un ejemplo de llamada a un repositorio específico mediante *curl* [4], un comando para la transferencia de datos desde una dirección URL, sería:

curl https://gitlab.com/api/v4/projects/6853087

Debido al objetivo del proyecto, únicamente se expondrán las solicitudes que tengan que ver con la recuperación de información referente a los *commits*, ya que estas nos ofrecen la información necesaria para implementar las metas del trabajo a realizar.

Para empezar, una de las llamadas que nos será útil será la de listar todos los commits de un repositorio determinado:

GET /projects/*id*/repository/*commits*

Dicha llamada necesitará un *id*, el identificador del repositorio objetivo; además también se le puede incluir una serie de atributos opcionales para que el servicio refine la

[3] https://docs.gitlab.com/ee/api/
[4] https://curl.se/

respuesta dada, como el nombre de una rama concreta, una fecha para indicar que sólo devuelva los *commits* realizados antes o en esa fecha o la disposición del orden de los *commits*.

Figura 2.1: Respuesta a la llamada que lista todos los *commits*.

Como se puede comprobar en la Figura 2.1, la respuesta a la llamada en cuestión devuelve una lista de los *commits* realizados en el repositorio, con una lista de valores que nos indican el id del *commit*, el nombre e *e-mail* del autor, el mensaje asociado y la fecha de creación, entre otros.

Otra de las llamadas que más útiles nos serán por la información que nos brinda es la recuperación de un *commit* específico que se realiza con la siguiente llamada:

GET /projects/*id*/repository/commits/*sha*

Para que la llamada sea efectiva, aparte del *id* del repositorio objetivo, necesitaremos una referencia al *commit*, en este caso el *hash* [5] *sha*. La respuesta a esta solicitud devuelve una información muy parecida a la de la anterior llamada, pero con unos campos añadi-

[5] Un valor *hash* es el resultado de la aplicación de una función *hash*, cuyo finalidad es la generación de una cadena única de caracteres a partir de unos valores determinados de entrada.

dos, como el número de líneas modificadas y borradas, y la rama a la cual pertenece el *commit*, además de su estado.

A continuación se lista una respuesta ejemplo de una hipotética solicitud de este tipo:

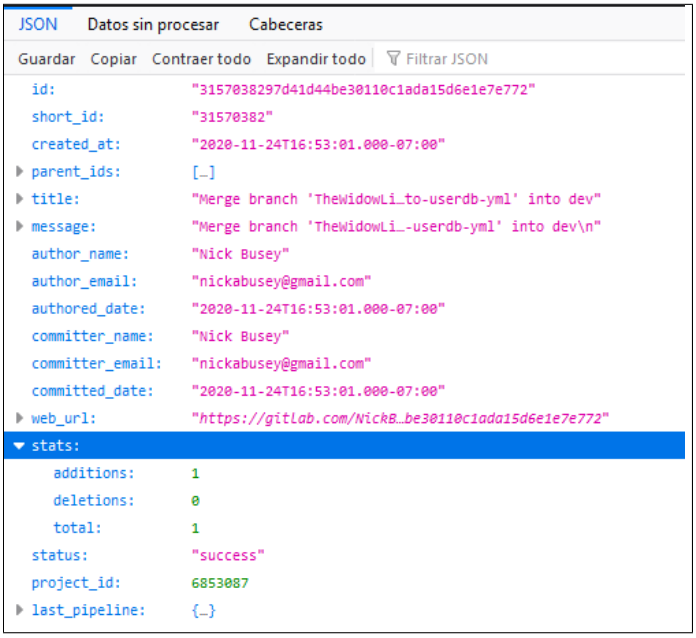

Figura 2.2: Respuesta a la llamada que lista la información de un único *commit*.

2.2.2. GitHub

La página de la API de GitHub [6] es extensa, está bien organizada y ofrece una serie de pequeñas guías para empezar a usarla, como por ejemplo un manual básico para autenticarse a través de una aplicación o un tutorial para empezar a manejar la API y varios otros para construir aplicaciones de distintos ámbitos.

Las siguientes llamadas que se exponen nos devuelven información proveniente de los servidores del servicio, y por motivos de seguridad cada respuesta viene acompañada además de un objeto con una serie de campos y valores destinados a verificar la firma del paquete de información entrante, aunque dicho objeto se omitirá porque se aleja del alcance del análisis de las interfaces.

Como en el apartado de GitLab, un apunte de uso: cada llamada deberá ir acompañada de la URL de la API, en este caso *https://api.github.com/*. Por lo tanto, un ejemplo de caso de uso real mediante *curl* sería el siguiente:

curl https://api.github.com/repos/owner/repo/commits

Para extraer la información que requerimos de los servidores de GitHub, nos será necesaria una llamada que recupere la información que necesitamos, en este caso los *commits* realizados en una rama específica del repositorio. Para ello, la API de GitHub nos ofrece una serie de llamadas para listar y comparar los *commits* de un repositorio.

GET /repos/*owner*/*repo*/commits

[6] https://docs.github.com/en/free-pro-team@latest/rest

Esta llamada devolverá un objeto que contendrá todos los *commits* realizados en el repositorio repo del usuario owner. La respuesta proveniente de los servidores a las llamadas que listan información contiene además un objeto con una serie de campos y valores destinados a verificar la firma del paquete de información entrante, aunque el estudio de dicho objeto se omitirá ya que se aleja del alcance del análisis de las interfaces. En la Figura 2.3 tenemos una respuesta a la anterior llamada, y como se puede comprobar, el servicio nos devuelve el listado de todos los *commits* de un repositorio específico. En dicha figura se puede observar que el repositorio del que estamos pidiendo su información únicamente tiene cinco *commits*, y la información provista para cada *commit* es limitada, únicamente nos muestra su identificador, el mensaje asociado, la fecha e información básica del autor.

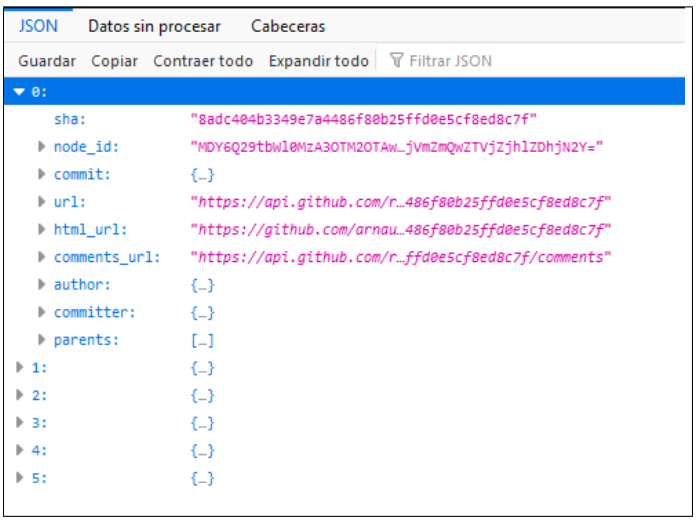

Figura 2.3: Respuesta a la llamada que recupera todos los *commits* de un repositorio.

Otra llamada que nos será útil será la de recuperar una referencia específica de un *commit*, cuya respuesta nos ofrecerá mucha más información relevante que con la anterior llamada, como por ejemplo los ficheros modificados y el sumario de cambios realizado; dichos datos nos serán útiles para el procesamiento de información que haremos en el prototipo.

GET /repos/*owner*/*repo*/commits/*ref*

Dicha solicud requiere, aparte del nombre del autor y del repositorio en cuestión, una referencia al *commit* que queramos recuperar, que es el *hash* de dicho *commit*, que, como ya se ha dicho, es una cadena de caracteres única para cada *commit*. Como se puede observar en la Figura 2.4, la respuesta a dicha llamada nos ofrece variada información y de muchos campos diferentes, por ejemplo nos lista todos los datos del usuario que ha realizado el *commit*, su nombre de usuario, su identificador y las varias URL de su página de usuario; también nos ofrece un listado de ficheros modificados, y un sumario de las modificaciones para cada uno de ellos, datos que no serán más tarde muy valiosos.

Por último, una llamada que podríamos destacar debido a la cantidad de información útil que nos aporta es aquella que lista los *commits* realizados en un fichero, es decir, todas las modificaciones que ha sufrido.

GET /repos/*owner*/*repo*/commits?*path*

La llamada necesita del *path* o ruta del archivo desde la raíz del repositorio, es decir, si el fichero el cual queremos extraer dicha información se llama *textfile_4.txt*, y dicho

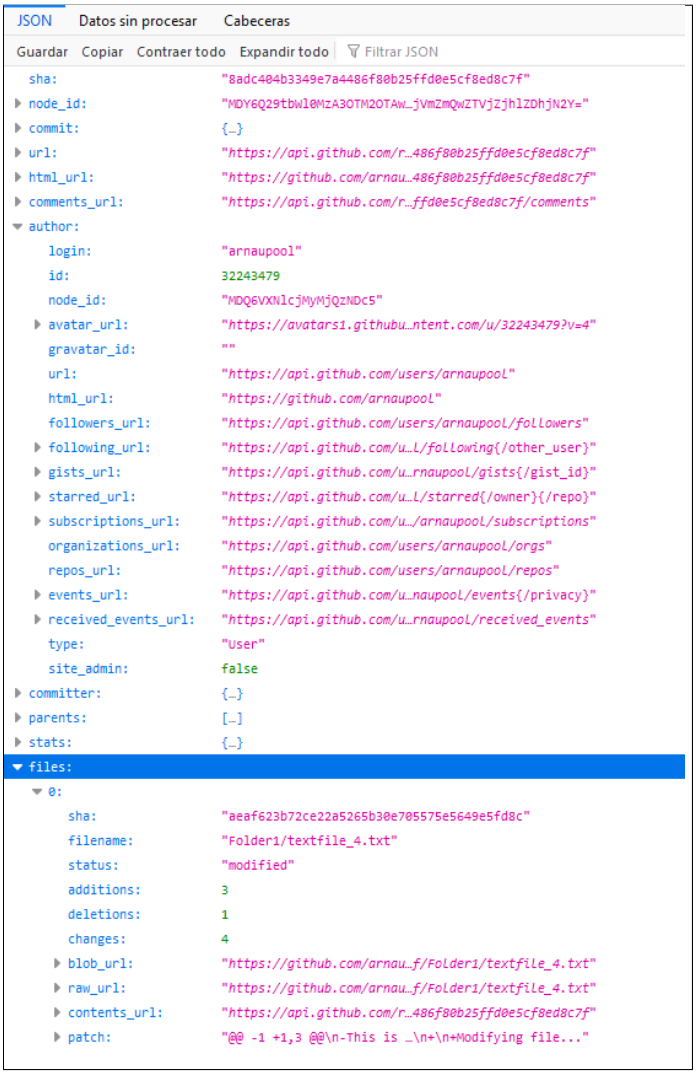

Figura 2.4: Respuesta a la llamada que recupera la información de un *commit* específico.

fichero se encuentra en la carpeta *Folder1*, la llamada que deberíamos hacer mediante *curl* sería:

curl https://api.github.com/repos/owner/repo/commits?path=Folder1/textfile_4.txt

Esta misma llamada nos ofrecerá una respuesta parecida a la de la Figura 2.5, que como se puede observar, nos entrega un número de objetos según los commits que haya tenido el fichero en cuestión. Dentro de cada objeto tiene toda la información que pudiéramos necesitar, la referencia o *hash* del *commit*, el autor, el mensaje asociado y la URL del *diff* o la comparativa entre su estado anterior y posterior al *commit*.

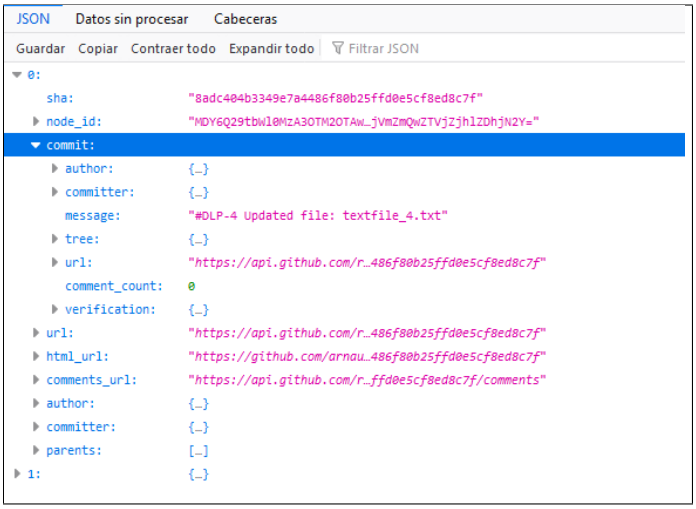

Figura 2.5: Respuesta a la llamada que recupera todos los *commits* de un fichero.

2.3 Conclusiones

En este punto hemos visto una muy pequeña parte de la gran variedad de métodos que nos ofrecen los servicios de control de versiones del *software* para poner solución a nuestros proyectos que los integren. Se ha hecho una revisión de las solicitudes más relevantes según los objetivos de este trabajo, se ha desglosado ligeramente la respuesta que ofrecen, y hemos visto que la información provista para un mismo tipo de consulta es ligeramente diferente según qué servicio.

Aparte de la diferencia en popularidad entre los dos servicios, la información que nos ofrece la API de GitHub es más completa y sencilla de implementar, y es por ello que este será el servicio elegido para su implementación en el prototipo, además de que nos brinda información muy relevante, por ejemplo la lista de ficheros modificados en cada *commit*, datos que nos serán fundamentales para el procesamiento de la información que más adelante expondremos.

CAPÍTULO 3
Estudio de integraciones existentes

3.1 Introducción

El término de agilidad en cuanto al desarrollo de proyectos se refiere es uno relativamente conocido desde hace décadas, al menos desde los años 70 [2], y surgió como una reacción a los métodos de desarrollo formales de la época, como el desarrollo en cascada, focalizados en el desarrollo ordenado y dividido en fases secuenciales de la carga de trabajo. Tales enfoques requerían de una previa planificación y estructuración altamente desarrollada para que el posterior progreso del proyecto se desarrollara de manera suave, ya que si los requisitos de diseño cambiaban estando ya en el desarrollo, se debía rediseñar y reprogramar la parte afectada, cosa que aumentaba los costes del proyecto. Por esta misma razón, las perspectivas tradicionales insistían en un desarrollo efectivo y eficiente del trabajo, en la que se debía definir antes que diseñar, y a su vez diseñar antes de codificar. Como respuesta a estos enfoques secuenciales y reacios a cambios surgen las metodologías ágiles, unas que abrazan el reencuentro de nuevos requisitos y la modificación de estos a medida que el proyecto se va desenvolviendo, y que tienen un desarrollo iterativo e incremental dividido en iteraciones, cada una de ellas con un límite de tiempo para su completitud que va desde una hasta las cuatro semanas, y que se divide a su vez en pequeños y manejables bloques, llamados unidades de trabajo o UT de ahora en adelante, que abarcan el análisis de requisitos, diseño, implementación y fase de pruebas. Algunas de estas metodologías son Scrum, Kanban y XP (*Extreme programming*) [1].

Como ya hemos dicho, uno de los objetivos básicos de estas metodologías es la flexibilidad ante cambios mientras se mantiene una planificación, y como consecuencia del nacimiento de dichas perspectivas y de estos objetivos, también surgen las herramientas destinadas a la organización y planificación del desarrollo del equipo, que ayudan a priorizar determinadas actividades, planificar cada iteración, analizar el progreso del proyecto y a observar una visión de conjunto del trabajo individual. Muchas de estas herramientas se basan en un tablero Kanban, un sistema de visualización del flujo de trabajo dividido en columnas que indican las fases del proceso, en el que los desarrolladores crean historias o *issues* que van trasladando a las columnas contiguas a medida que las completan.

El uso de este tipo de herramientas ayuda a la automatización de la planificación de cada iteración, incrementa la comunicación y transparencia entre integrantes del equipo y ofrece informes en tiempo real del estado del desarrollo del proyecto, como por ejemplo diagramas de flujo acumulado y de *burn down*. Por esta misma razón es interesante la integración con un sistema de control de versiones, ya que puede ofrecer una nueva dimensión en el análisis, además de mediciones del esfuerzo más precisas y concretas, al unir unidades de trabajo con ficheros e incluso con piezas de código concretas.

Jira [1] y PivotalTracker [2] son dos herramientas de gestión ágil de proyectos, y en este apartado se documentará el proceso de integración de un sistema de control de versiones, con el objetivo de contemplar el estado del arte respecto a las herramientas de gestión ágil del trabajo que se han seleccionado.

3.2 Jira

Para empezar, Jira no soporta de forma nativa la integración entre su servicio y las plataformas de control de versiones, por lo que es necesaria la instalación de un *plugin* que lo permita. Aunque haya un *addon* oficial para GitHub, nosotros vamos a utilizar uno mucho más completo y mejor valorado por la comunidad, desarrollado por la empresa BigBrassBand [3]. Para su instalación, accedemos a la tienda de aplicaciones de Atlassian, la empresa propietaria de Jira, e introducimos en el buscador "git"; automáticamente aparecerá en primer lugar la aplicación que queremos, *Git Integration for Jira* [4]. Por suerte, este *plugin* tiene un periodo de prueba de 30 días, por lo que podemos asociarlo a nuestra cuenta y empezar a trabajar con él. Una vez instalado, accedemos a Jira, iniciamos sesión con la misma cuenta con la que hemos asociado el *plugin* y, en la barra de arriba, en el desplegable de *Aplicaciones*, aparecerá una opción de gestionar los repositorios de la integración, o *Git Integration: Manage Git repositories*. Esa página nos llevará al apartado de configuración, donde podremos emparejar el sistema de repositorios que deseemos con Jira, mostrado en la Figura 3.1.

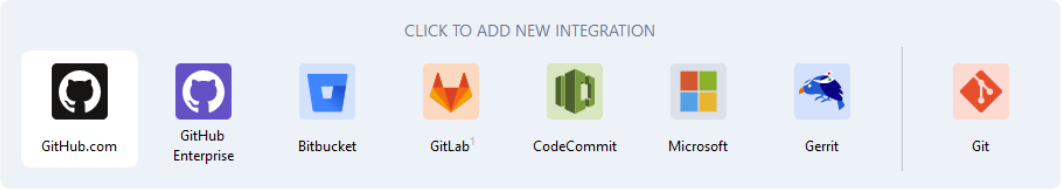

Figura 3.1: Apartado de integraciones a plataformas de control de versiones.

Presionamos sobre el botón de GitHub.com, y nos llevará a un página previa a donde podremos introducir las credenciales de nuestra cuenta de Github, como se muestra en la Figura 3.2.

La integración que acabamos de instalar y configurar tiene como objetivo listar los *commits* asociados a una unidad de trabajo determinada, por lo que, desde GitHub, será necesaria el uso de una sintaxis específica con el objetivo de que dicha integración sea capaz de atrapar los cambios y poder mostrarlos en la aplicación. Dicha sintaxis es muy sencilla: en el mensaje del *commit*, el ID de la tarea deberá ir en primer lugar seguido del comentario habitual. Si dicha etiqueta no es correcta o no se encuentra en el mensaje, Jira no podrá asociar la transacción a la tarea, por lo que no se mostrará en esta.

Una vez iniciada sesión, nos mostrará en pantalla una lista de los repositorios de nuestra cuenta, le damos al botón de *Next*, abajo a la derecha, y nos conducirá a una pantalla con un par de configuraciones, la primera la dejamos como está, sin marcar, y la segunda es para restringir acceso a los repositorios de GitHub desde los proyectos de Jira, es decir, que si seleccionamos un proyecto determinado de Jira en el desplegable, los

[1] https://www.atlassian.com/es/software/jira
[2] https://www.pivotaltracker.com/
[3] https://bigbrassband.com/
[4] https://marketplace.atlassian.com/apps/4984/git-integration-for-jira?hosting=cloud&tab=overview

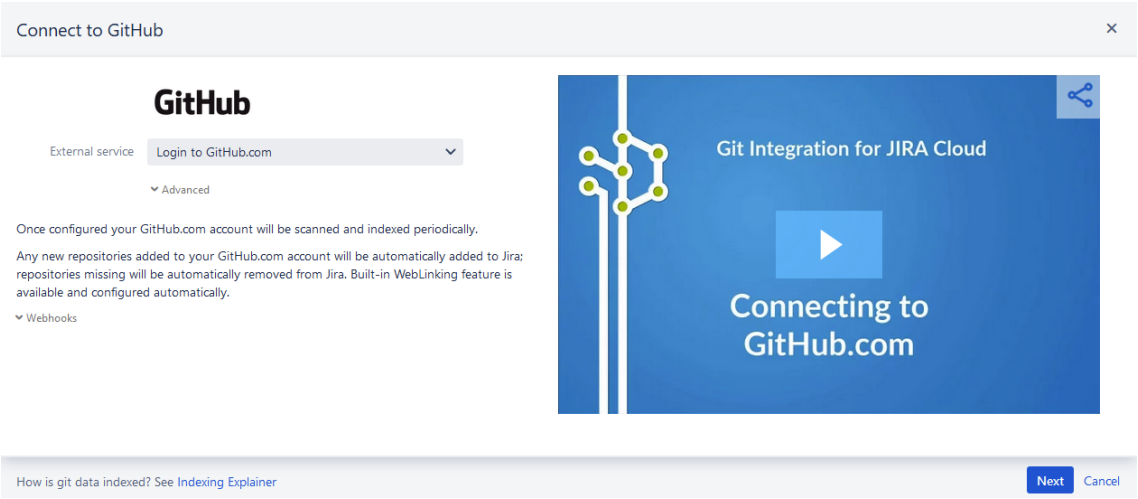

Figura 3.2: Página predecesora del inicio de sesión.

repositorios de GitHub de nuestra cuenta sólo se podrán visualizar desde el navegador de repositorios de ese proyecto; para nuestro caso, dejamos marcada la casilla de *Associate with all projects*, y pulsamos el botón de *Connect*, abajo a la derecha. Una vez acabada esta configuración, la integración con GitHub ya estará completada, y desde ahora, cada *commit* que se realice, siguiendo una sintaxis explicada más adelante, se mostrará en la UT correspondiente.

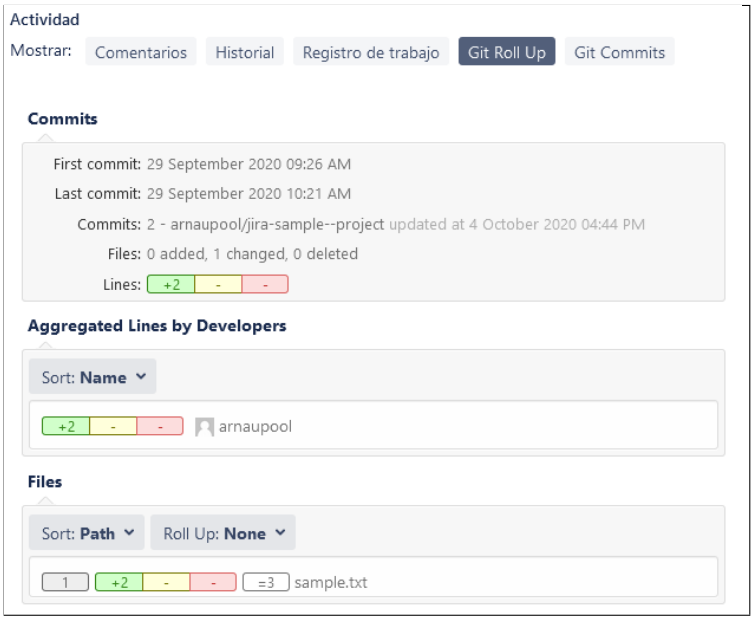

Figura 3.3: Detalle de la pestaña *Git Roll Up*.

Cada unidad de trabajo tiene su propia página desde la cual modificar aspectos de esta, como el responsable de la tarea o establecer la prioridad de la actividad, entre otros. Gracias al *addon* que hemos instalado, se nos han añadido una serie de opciones y pestañas, en particular la pestaña de *Git Roll Up*, mostrado en la Figura 3.3, donde se muestran diferentes estadísticas e información, como por ejemplo el número de *commits*, el número de ficheros añadidos, modificados y borrados, una clasificación de los desarrolladores que han tocado esa UT, junto con información de los cambios que han hecho en total, y por último una lista de los ficheros que se han tocado en la UT.

En la pestaña de *Git Commits*, como se muestra en la Figura 3.4, la aplicación permite ver todos los *commits* que tengan que ver con la tarea en cuestión, es decir, que el comentario del cual venga precedido por el ID de la tarea, además de qué desarrollador los ha emitido. Aparte de tener enlaces hacia Github de los diferentes *commits*, también te permite ver, en la misma página, el *diff* del *commit*, es decir, la comparación entre los archivos antiguos y los nuevos, mostrando líneas eliminadas, añadidas o modificadas (Figura 3.5).

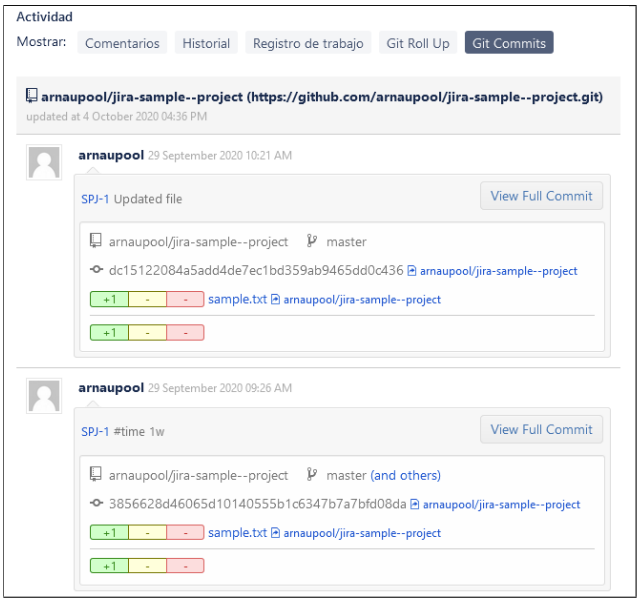

Figura 3.4: Detalle de la pestaña *Git Commits*

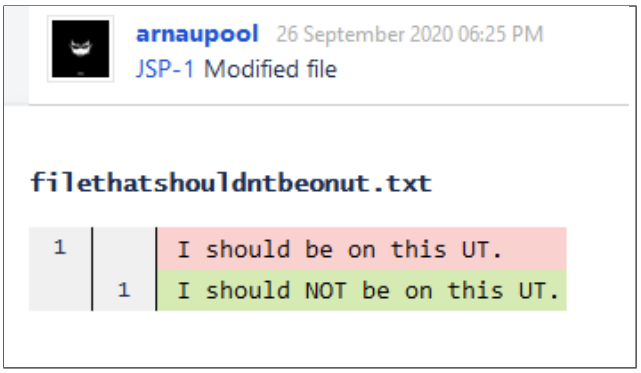

Figura 3.5: Página del *diff* de un *commit*.

Jira guarda una lista de los *commits* que se han hecho a una determinada UT, además de los archivos asociados a cada *commit*, como se puede observar en la Figura 3.6. Desgraciadamente, este *addon* no ofrece ninguna manera de diferenciar entre archivos de diferentes UT en un mismo *commit*, por lo que si creas o modificas archivos que no son de la misma unidad de trabajo y subes los cambios (referenciando las varias unidades de trabajo a las que pertenecen los archivos mediante su ID, separados por comas), Jira mostrará esos archivos en la tarea en la que no le corresponde además de en la que sí.

Este *addon* permite la creación de ramas por UT desde la misma aplicación, mostrado en la Figura 3.7, permitiendo crear flujos de trabajo paralelos al principal sin peligro a dañar los archivos originales.

Una vez creada la rama, podremos trabajar en ella y una vez hayamos acabado nuestro trabajo y sea estable y seguro, podemos unirla con la rama *master*, también desde la

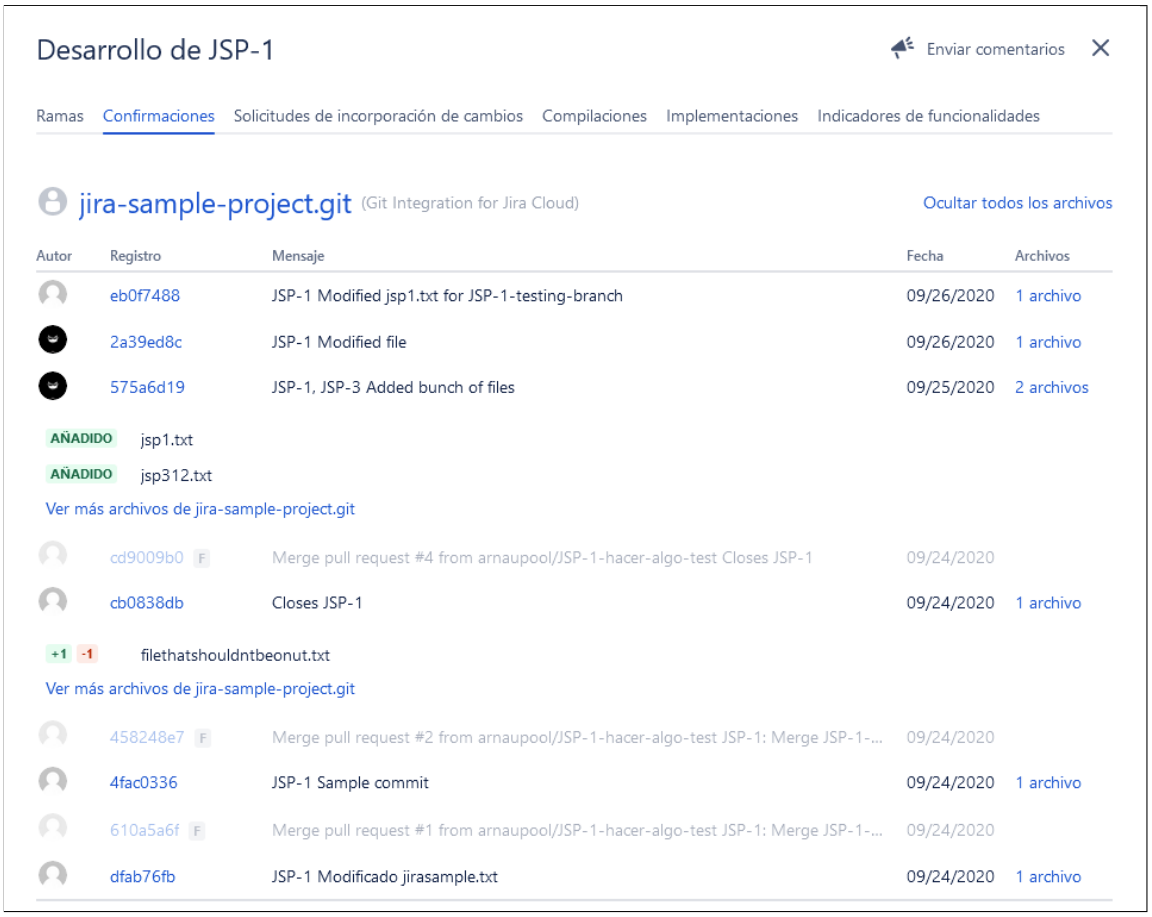

Figura 3.6: Detalle de la página de confirmaciones.

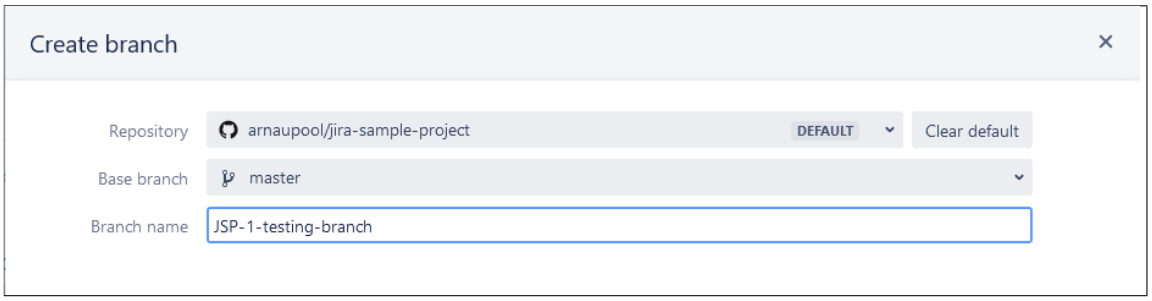

Figura 3.7: Creación de ramas desde Jira.

Figura 3.8: Ramas asociadas al proyecto.

propia aplicación, que además también mostrará el listado de solicitudes de fusionado de ramas o *merge requests* de manera automática, como se expone en la Figura 3.9.

Este complemento tiene una funcionalidad muy útil a la hora de realizar acciones a las unidades de trabajo en Jira desde los *commits*, y esta es los *Smart Commits*. A partir de una sintaxis específica en los mensajes del *commit*, Jira es capaz de ejecutar algunas acciones sobre la unidad de trabajo a la que se refiere el *commit*, como por ejemplo:

1. "SPJ-1 #time 1w 2d 5h Modified file1.hs": Este *commit* añadirá al registro de tiempo en Jira, para la tarea SPJ-1, una cantidad de una semana, dos días y cinco horas.
2. "SPJ-1 #comment This is a comment": Este *commit* añadirá el comentario descrito a la tarea SPJ-1.
3. "SPJ-1 #close Fixed file": Este *commit* cambiará el estado de la unidad de trabajo SPJ-1 desde su estado actual al estado de trabajo especificado, en este caso, a *close*.

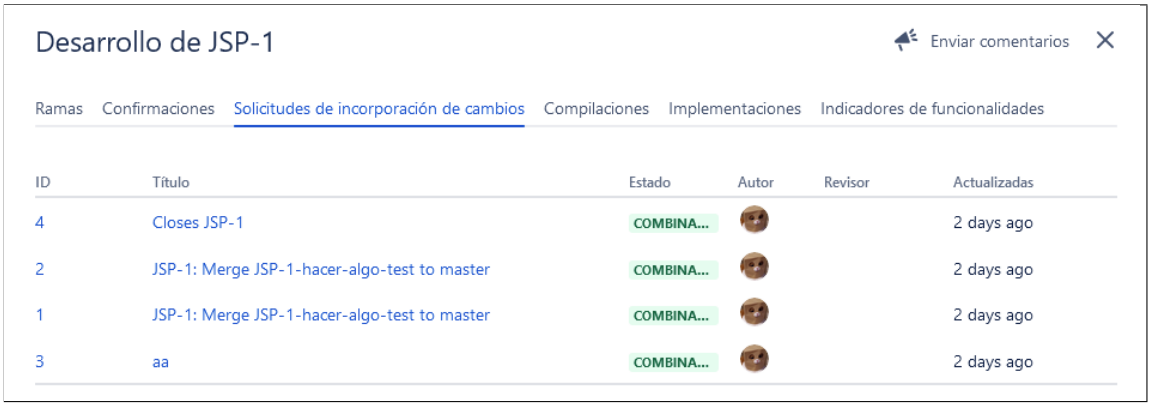

Figura 3.9: Lista de *merge requests* de las ramas asociadas a la tarea.

Además de estos ejemplos, también tiene muchas otras funcionalidades, como por ejemplo la asignación de un usuario a dicha tarea o la adición de una etiqueta concreta.

3.3 PivotalTracker

Para empezar el setup de un proyecto en PivotalTracker, empezaremos registrándonos en la página y pulsando el botón de *Create project*, creando un proyecto nuevo con un nombre, asociado a nuestra cuenta, y con una visibilidad ya sea pública o privada (Figura 3.10).

Una vez creado el proyecto, en la parte superior de la página, estarán visibles una serie de pestañas. Para empezar a integrar PivotalTracker con un servicio de control de versiones, haremos clic en la de *Integrations* y nos redigirá a una página simple con un único botón, el de *Add an integration*, lo pulsamos y nos mostrará una lista de los varios servicios que podremos escoger, entre los cuales estará GitHub.

Seleccionaremos el nombre de la integración que queramos, además del proyecto que queremos integrar, en nuestro caso, *pivotal-test*. De momento, para que se generen una serie los datos que se muestran en la Figura 3.11 que deberemos introducir en la página de GitHub, guardaremos esta configuración dándole al botón de *Save*, abajo a la derecha. Hecho esto, el siguiente paso será la creación de un *webhook*, un agente sensible a cambios específicos que ocurren en el repositorio, activando una respuesta en la página de Pivotal. Para ello, debemos ir a la página de GitHub, a nuestro repositorio, e ir a la configuración

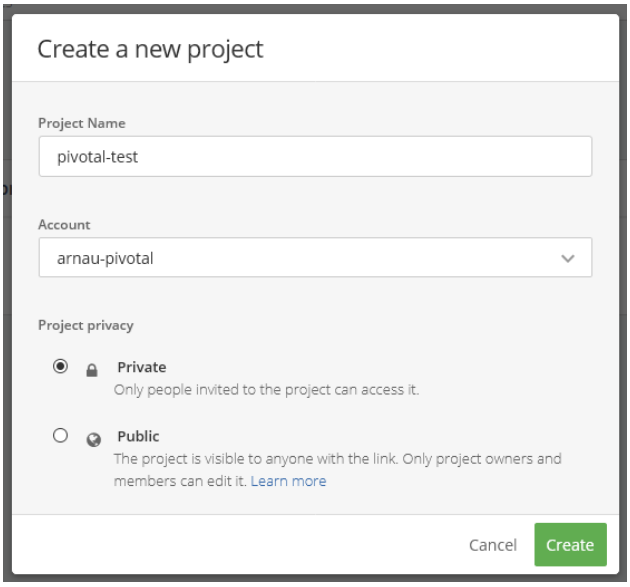

Figura 3.10: Pantalla de creación de un proyecto en PivotalTracker.

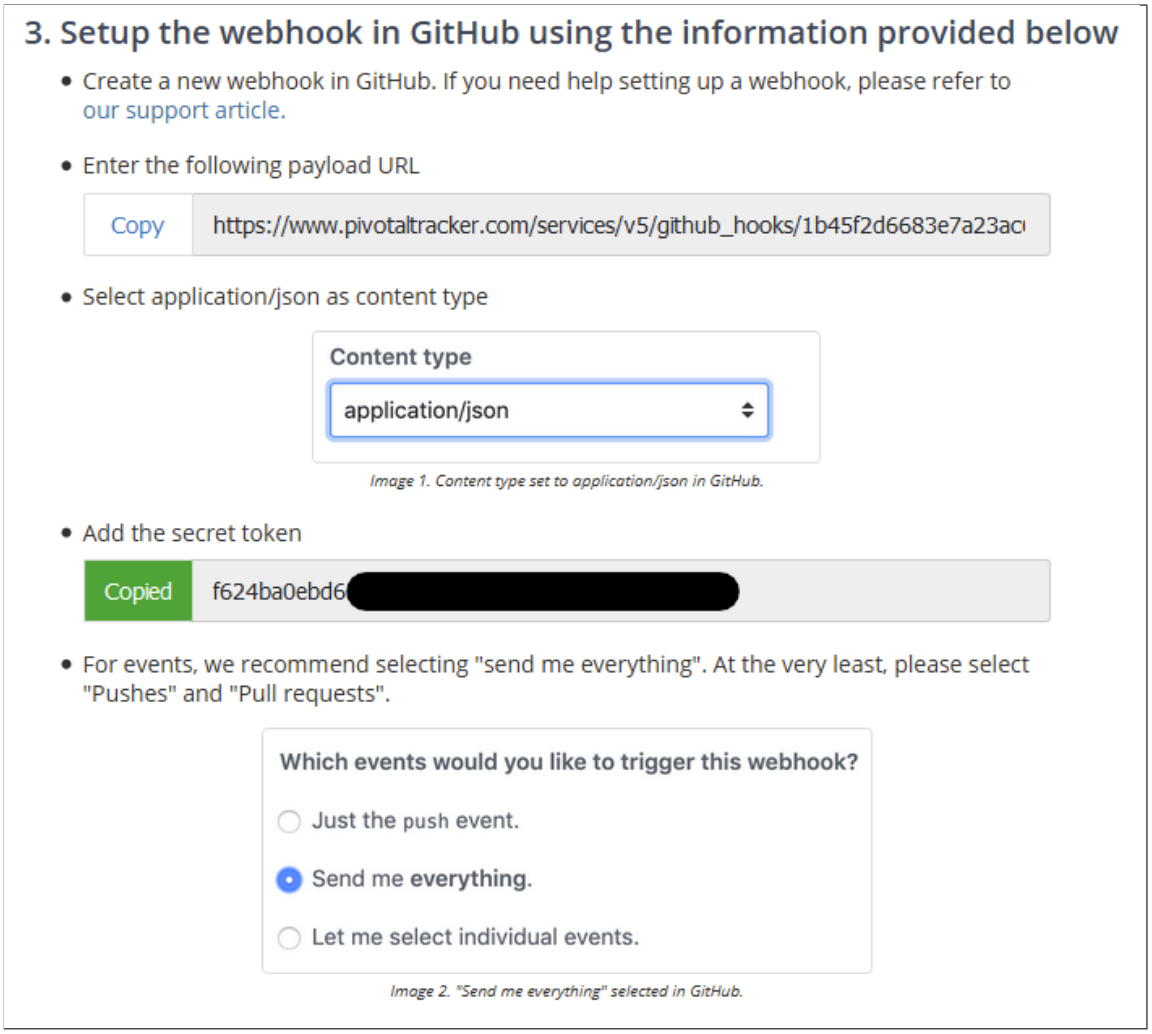

Figura 3.11: Ventana de configuración de la integración en PivotalTracker.

de este, más específicamente al apartado de *Webhooks*, y pulsar sobre el botón de *Add webhook*, arriba a la derecha.

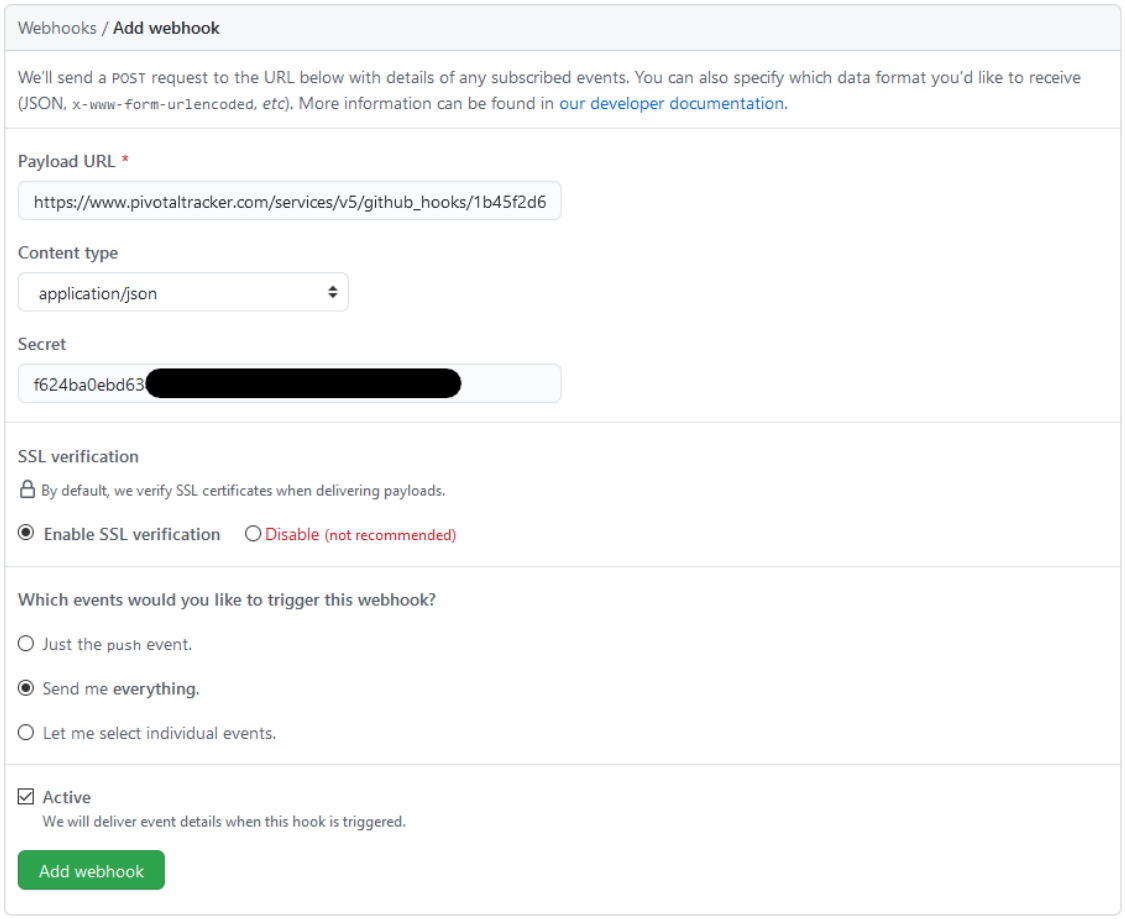

Figura 3.12: Ventana de configuración de la integración en GitHub.

En la página siguiente (Figura 3.12) y en los campos determinados, introduciremos los datos que PivotalTracker ha generado: la *Payload URL*, estableceremos el tipo de contenido a *application/json*, y el token secreto; además, marcaremos la opción de que todo lo que ocurra en el repositorio active el *webhook*, y nos aseguraremos de que la casilla de *Active* está marcada. Una vez hecho esto, guardaremos la configuración y nos redigirá a la página de ajustes, mostrando un mensaje de éxito en la creación del *webhook*, y ya tendremos lista la integración entre nuestro proyecto de Pivotal Tracker y nuestro repositorio de GitHub.

PivotalTracker también tiene el mismo sistema de cambios en la unidad a través del mensaje del *commit*, y aunque la sintaxis no sea compleja, al igual que con Jira es mínimamente tediosa, ya que para que Pivotal Tracker detecte el *commit* como un cambio en una tarea, el inicio del comentario ha de ser el ID de dicha tarea entre corchetes, por ejemplo: "[#175001097] Updated thisfile.txt", de no ser así, el *commit* no aparecerá en el apartado de *Activity* de la tarea.

Los estados de las tareas son un concepto importante a la hora de organizar el flujo de trabajo de una unidad, y en concreto en esta herramienta el flujo de estado de una UT se organiza según la Figura 3.13. En PivotalTracker, los diferentes estados en los que puede estar son:

1. No empezada (*Unstarted*): Estado en el que se encuentran las tareas en el backlog y en la iteración actual.
2. Planificadas (*Planned*): Estado en el que se encuentran las tareas que están en el la iteración actual de un proyecto manualmente planificado. Un proyecto manualmente planificado es aquel que se ha especificado en los ajustes del proyecto que no se planifique automáticamente.
3. Empezadas (*Started*): Estado en el que se encuentran las unidades de trabajo que están no empezadas o planificadas, una vez se pulse el botón de *Start*.
4. Acabadas (*Finished*): Una vez una tarea esté finalizada acorde a los estándares del equipo y del proyecto, los propietarios de dicha tarea podrán dar por acabada el desarrollo de ésta.
5. Entregadas (*Delivered*): Estado en el que se encuentran las tareas que se han desplegado a un entorno donde se les puedan hacer una serie de pruebas de aceptación. Este estado tiene dos subestados, *Accept* y *Reject*.
6. Denegadas (*Rejected*): Estado al que se envía una unidad de trabajo que ha tenido uno o varios problemas. Desde este estado, puede ir al de empezado (*Started*).
7. Aceptadas (*Accepted*): Una vez una tarea haya finalizado su producción y haya pasado su conjunto de casos de aceptación, está lista para ser lanzada a producción. Este es el estado que indica que la labor en esta unidad de trabajo ha finalizado.

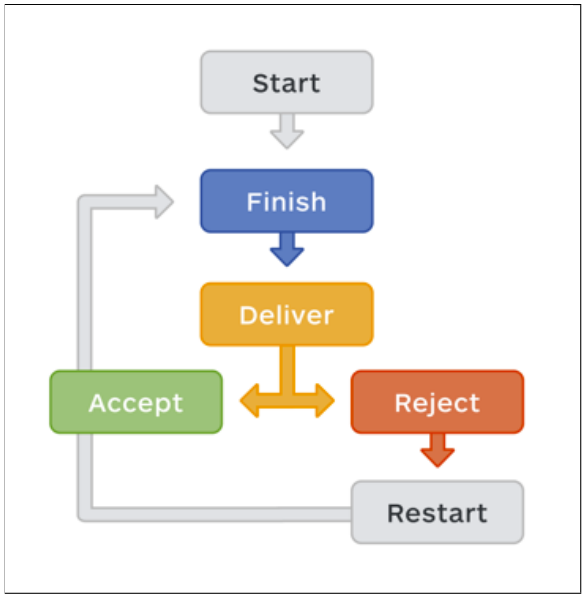

Figura 3.13: Flujo del estado de las tareas [7].

Una tarea puede estar en uno de los estados anteriormente mencionados, pero si en el *commit* se indica que está *Finished* or *Fixed*, la tarea automáticamente pasa de *Finish* a *Deliver*, significando esto que el desarrollo de la tarea ha finalizado y ahora está en la fase de pruebas; y si se indica que está *Delivered*, la tarea pasa automáticamente al estado

de elección entre *Accept* y *Reject*, indicando que las pruebas aplicadas a la tarea se han realizado y ahora se está a la espera de si pasan los estándares del proyecto o se han de reiniciar.

Pivotal Tracker contiene menos funcionalidades que Jira en cuanto a la integración con un servicio de control de versiones se refiere, pero no por ello se queda atrás, por ejemplo, cada unidad de trabajo contiene un campo *CODE*, que permite adjuntar el link de una rama o una *pull request*, y Pivotal se encargará de interpretarlo y asociarlo correctamente a la unidad, creando una etiqueta con un hipervínculo hacia la página en GitHub, de manera automática. Aparte de este método de asociación de ramas a unidades de trabajo, también se puede crear la rama directamente en GitHub, con una sintaxis específica, y automáticamente aparecerá en la unidad de trabajo correspondiente. Para ello, necesitamos el ID de la tarea, que colocaremos al principio del nombre de la rama, proceso que se muestra en las Figuras 3.14 y 3.15 respectivamente.

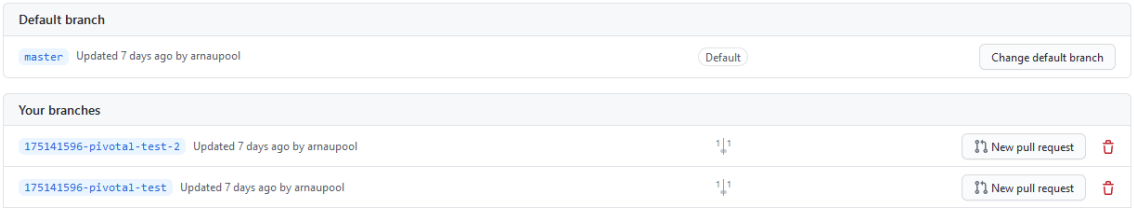

Figura 3.14: Creación de dos ramas para la tarea con ID 175141596.

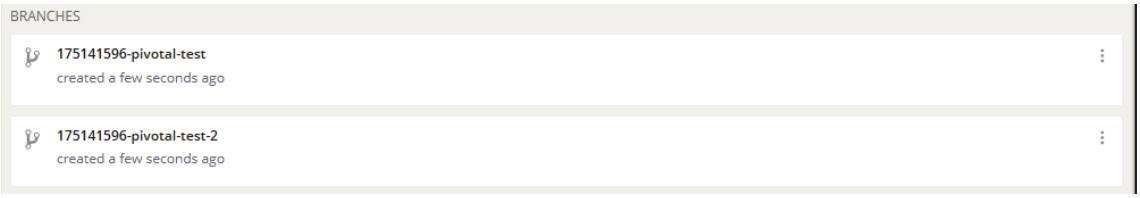

Figura 3.15: Asociación de las ramas creadas en GitHub a la UT.

3.4 Conclusiones

En este apartado se ha hecho un recorrido de la configuración de una integración de un sistema de control de versiones con las herramientas de gestión ágil de trabajo Jira y PivotalTracker, siendo en cada una de ellas la inicialización de la integración de una manera diferente, ya sea mediante la creación de un *webhook* o el acceso a la cuenta del usuario por medio de sus credenciales personales.

Con este análisis hemos aprendido algunos de los aspectos que hacen de estas integraciones una ayuda más tanto para los supervisores como para los desarrolladores, ya que dichos servicios ayudan a un registro y control más específico de cada una de las unidades de trabajo, como los *Smart Commits* de Jira o los cambios de estado de las UT de PivotalTracker.

Finalmente, hemos visto también que en ambos casos se ha de seguir una sintaxis que, aunque sencilla, es propensa a errores por parte del programador y requiere del identificador de la tarea en cuestión, no obstante, es un sistema simple y efectivo para la unión de las dos herramientas. Dicho sistema se implementará en el desarrollo de la aplicación.

CAPÍTULO 4
Desarrollo de integración para Worki

4.1 Introducción a Worki

TUNE-UP Process es una metodología cuyo objetivo es la implantación de prácticas ágiles, basada en las metodologías más populares en la actualidad, como por ejemplo Scrum, Kanban y XP, entre otros. Dicha metodología aplica una serie de ideas, como por ejemplo la explotación y análisis de la información generada por el equipo de trabajo (uniendo de esta forma la supervisión necesaria del trabajo realizado con la necesidad de independencia del equipo), la convivencia con y la evolución desde una metodología más tradicional hasta una ágil y la implantación de diferentes metodologías ágiles según las necesidades y contexto del equipo.

Worki es una herramienta *software* nacida para la implantación de esta metodología, que permite la gestión ágil del trabajo de un proyecto, estructurando la carga de trabajo en esprints, y estos a su vez dividiendo dichos esprints en unidades de trabajo individuales, que se le asignarán a un integrante del equipo para su realización. El prototipo desarrollado más adelante es expuesto como una propuesta de integración de un SCV, con el objetivo de una futura implantación en Worki.

4.2 Descripción del problema

Como ya hemos comentado anteriormente, uno de los objetivos base de este trabajo es el de unir dos áreas que existen para que el desarrollo de un proyecto se simplifique y sea eficiente, estas dos áreas son las de las herramientas de gestión ágil del trabajo con la de los sistemas de control de versiones. Los beneficios de unir estos dos sistemas son claros: ver cómo el desarrollo, en forma de código fuente, se desarrolla y toma forma; es una manera fácil y sencilla de asociar ficheros de código a una porción concreta de trabajo; podría ayudar a la planificación de pruebas de regresión, ya que aquellos responsables de su ejecución sabrían exactamente en qué fichero podría estar el problema si se llega a encontrar con uno; y por último, en los análisis que se realizan después de que el proyecto haya finalizado, es otra fuente de retroalimentación que podría mejorar la comprensión de por qué un determinado proyecto fracasa o tiene éxito.

Dicho esto, la descripción del problema es la construcción de una aplicación que emule una de gestión ágil del trabajo, en el sentido de que tenga una serie de unidades de trabajo, como si se estuviera en una iteración, y que cada una de esas UT tenga asociado una lista de los *commits* y una lista de los ficheros que se le han asociado a dicha tarea.

En el apartado técnico del proyecto, el SCV elegido es GitHub, ya que las funcionalidades que ofrece y la documentación exhaustiva de estas hacen de este servicio de los mejores, si no el mejor, para realizar una aplicación que lo integre. Para la construcción del prototipo se ha utilizado Angular [1], un *framework* de desarrollo de aplicaciones web muy popular en la actualidad entre los desarrolladores, escrito en TypeScript [2] y que cuenta con módulos, componentes y servicios como elementos básicos de construcción del programa. Las llamadas a la API y los métodos para la captura y el procesado de la información se han realizado con bibliotecas nativas de Angular, suprimiendo así la necesidad de un componente servidor externo a la aplicación.

4.3 Requisitos

Los requisitos que ha de cumplir el prototipo son los siguientes:

1. Listar una serie de unidades de trabajo ficticias, que recuperará de un fichero en memoria.

2. Al hacer click sobre una unidad de trabajo, se abrirá una ventana con un detalle de dicha UT, donde se mostrarán una lista de los *commits* asociados a dicha UT, y más abajo, una lista de todos los ficheros modificados asociados a los *commits* de la unidad de trabajo.

Para conseguir el desarrollo de estas funcionalidades, cada UT deberá tener un identificador único, que deberán hacer servir los *commits* a los que pertenezca la tarea, con el objetivo de que la herramienta pueda recuperar y enlazar dichos *commits* con la tarea que les corresponde. El prototipo deberá recuperar la información pertinente de los servidores del servicio mediante llamadas a su API, y procesarla en función de la unidad de trabajo que se haya seleccionado.

4.4 Diseño

Al inicio del proyecto, se intercambiaron ideas tempranas con el tutor sobre las funcionalidades básicas del prototipo, y se realizó el esbozo que se muestra en la Figura 4.1. Dicho *mock-up* contaba con ciertas funcionalidades adicionales, como la creación de nuevas unidades de trabajo y un apartado de ajustes, y una mayor cantidad de elementos en la interfaz, como una diferenciación entre las tareas del *backlog* y de los esprints. Después del envío del *mock-up* al tutor para su revisión, se acordó que la aplicación fuera aún más sencilla, eliminando las funcionalidades opcionales mencionadas y los elementos agregados de la interfaz, resultando en una interfaz que se mostrará a continuación.

Para los componentes de la interfaz, se ha optado por la la librería de Angular Material [3], que ofrece una gama de componentes con estética basada en el lenguaje de diseño *Material Design* [4].

Como se puede observar en la Figura 4.2, la interfaz principal es una lista de las unidades de trabajo creadas, con el identificador de la tarea como título, y su nombre como un subapartado más abajo.

[1] https://angular.io/
[2] https://www.typescriptlang.org/
[3] https://material.angular.io/
[4] https://www.material.io/

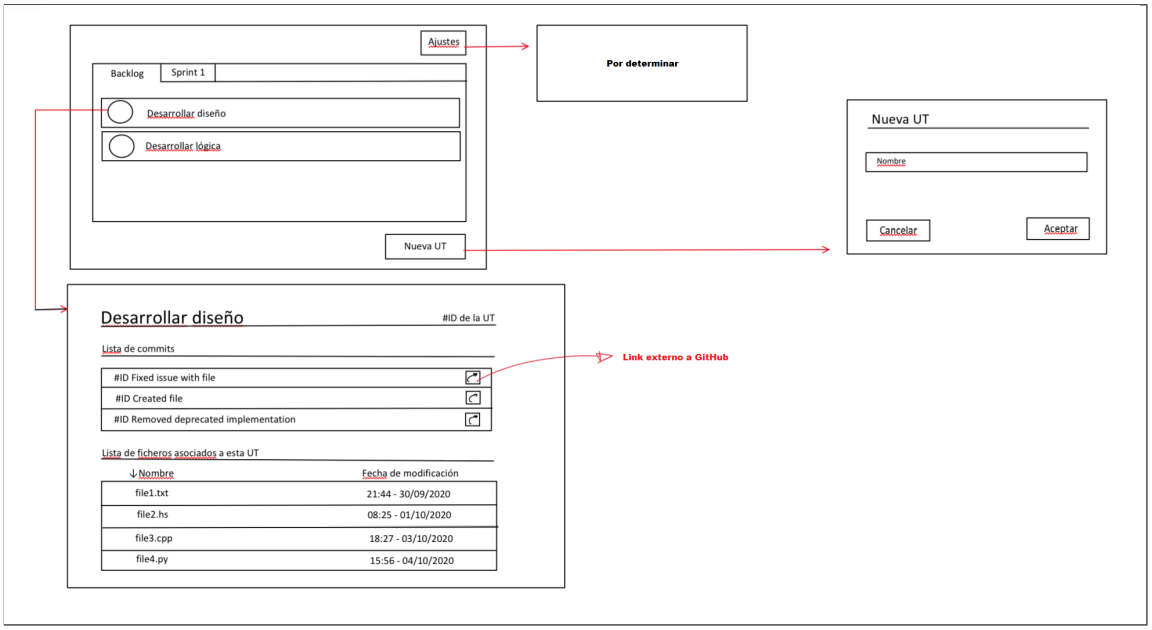

Figura 4.1: Esbozo inicial del prototipo.

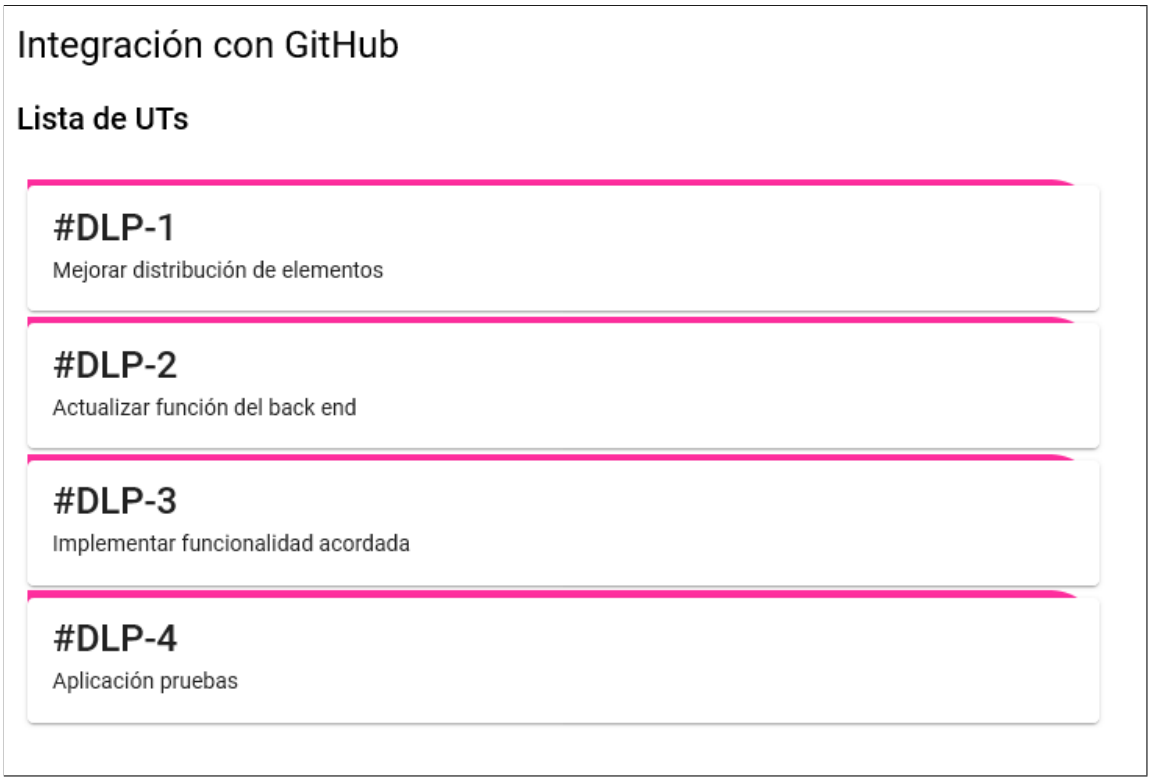

Figura 4.2: Interfaz inicial del prototipo.

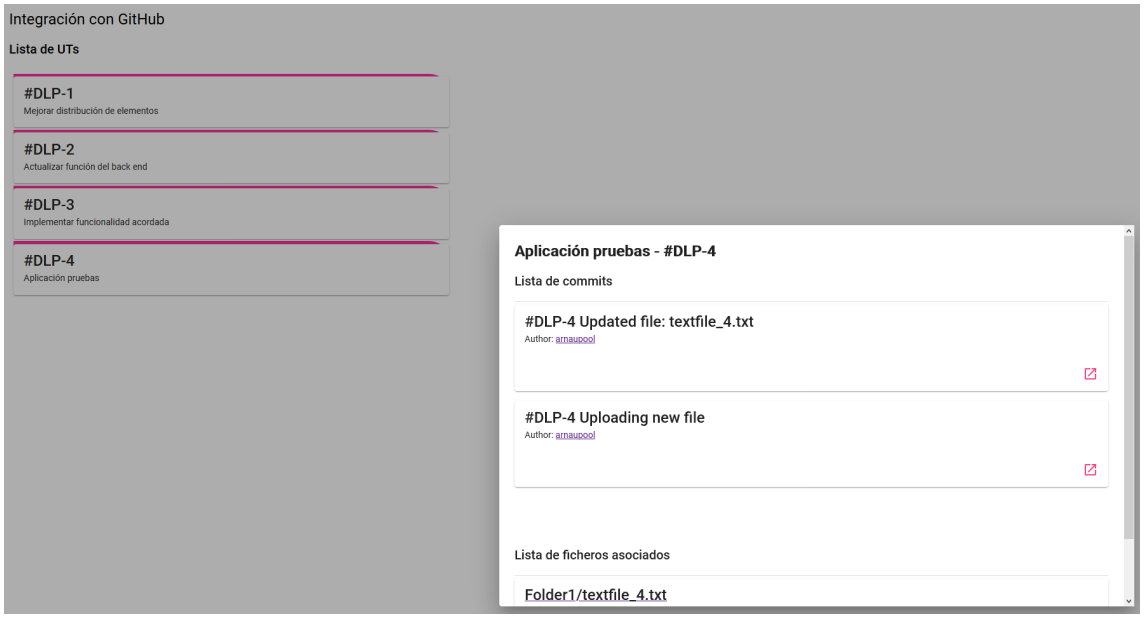

Figura 4.3: Ventana de la tarea #DLP-4.

Al hacer clic en cualquiera de ellas, aparece una ventana de diálogo con la información de dicha tarea, como se puede ver en la Figura 4.3, con una lista en orden cronológico descendente de los *commits* del repositorio objetivo asociados a la tarea en cuestión. La lista muestra el mensaje del *commit* como título, y como subtítulo se muestra el autor de dicha modificación en un hiperenlace hacia la página de GitHub del usuario. Además, en la parte inferior derecha, se encuentra un botón encargado de abrir la página de GitHub con el detalle de los modificaciones realizadas en el *commit* específico (Figura 4.4).

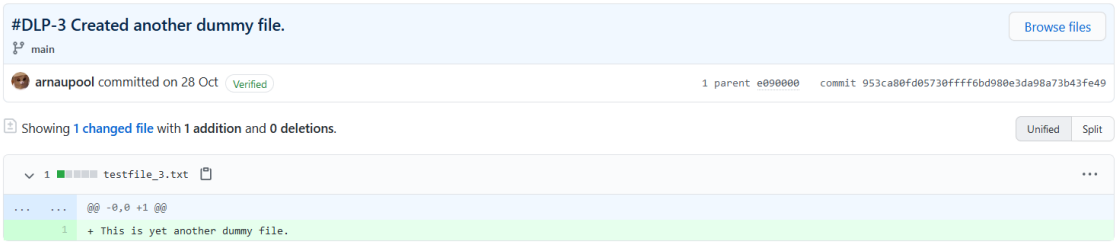

Figura 4.4: Detalle de un *commit* en GitHub.

En la parte inferior del diálogo de detalle de la tarea se encuentra una lista con la ruta y el nombre de todos los ficheros modificados en los *commits* asociados a dicha tarea específica. Si el usuario hace clic en alguno de los ficheros, una nueva pestaña de GitHub se abrirá con los contenidos del fichero, como se muestra en la Figura 4.5.

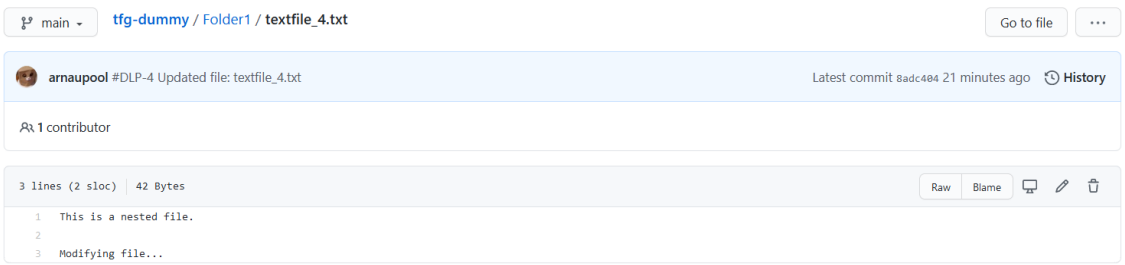

Figura 4.5: Contenido de un fichero en GitHub.

En cuanto al apartado del *back end*, para las solicitudes de información a la API se ha utilizado el cliente HTTP integrado en Angular, llamado HttpClient [5], y que contiene los métodos REST necesarios para la ejecución de llamadas. TypeScript, el lenguaje en el que está escrito Angular, es un lenguaje de programación asíncrono, es decir, la ejecución de las órdenes no tiene por qué ser de manera secuencial, y no se puede asegurar la inmediata disposición de la información solicitada a la API, por lo que en el paradigma de la programación asíncrona se puede dar el caso de que se ejecuten las funciones que requieran de estos datos sin tenerlos aún disponibles. Por ello, se ha necesitado de la librería RxJS [6] para el uso de los tipos de datos y de los métodos asociados contenidos en ella, y que permiten la ejecución secuencial de métodos.

4.5 Pruebas

Las pruebas unitarias o *unit tests* son un mecanismo para confirmar que una unidad de código o funcionalidad se ejecuta realmente como se espera y como esté programada, de manera automática y ágil. Como su nombre indica, dichas pruebas se ejecutan en una unidad o componente individual del sistema, para asegurarse de que funcionen una vez estén trabajando todas en conjunto. La ejecución de estas pruebas ofrece una serie de ventajas [5]:

1. Confeccionar pruebas unitarias para cada uno de los componentes requiere de menos casos de prueba totales que si se confeccionaran para todo el sistema en conjunto, ya que habría que considerar los diferentes flujos de ejecución.

2. Es una garantía de que el código hace realmente las funciones por las que fue escrito.

3. Si en el futuro la implementación de la funcionalidad cambiara, los tests para dicha unidad fallarían, por lo que son una protección contra la modificación de las asunciones inciales, tomadas en el momento de la programación del componente.

4. Sirven como documentación para la unidad de código a probar, ya que la misma prueba en cuestión evalúa la totalidad o parte de las funcionalidades en dicha unidad, ejemplificando su uso real.

Introducido el concepto de prueba unitaria y las ventajas que estas ofrecen, en este apartado se realizarán tales pruebas sobre cada una de las unidades y servicios que componen la aplicación desarrollada. Para el desarrollo y ejecución de estos tests, se ha utilizado Jasmine [7], un *framework* orientado a elaboración de especificaciones en lugar de test tradicionales, siendo una especificación una serie de instrucciones y de resultados esperados de la ejecución de dichas instrucciones. Adicionalmente se ha utilizado Karma [8], un *framework* que invoca al navegador para la ejecución rápida, sencilla y real de los tests.

Los componentes más relevantes de nuestra aplicación son cuatro:

1. *app.component*: Es la ventana inicial y principal de la interfaz, únicamente contiene el título de la aplicación y el componente de la lista de unidades de trabajo.

[5] https://angular.io/api/common/http/HttpClient
[6] https://github.com/ReactiveX/rxjs
[7] https://jasmine.github.io/
[8] https://karma-runner.github.io/latest/index.html

2. *ut-list.component*: Es el componente encargado de listar las unidades de trabajo en la interfaz.

3. *ut-body.component*: Este componente es el detalle de una unidad de trabajo específica. Se encarga del procesamiento de la información que se recibe, los *commits* y ficheros asociados a la UT, además de listar adecuadamente dichos datos mediante tarjetas en la interfaz.

4. *gfetch.service*: Este servicio hace el papel de intermediario entre la aplicación y la API. Se encarga de realizar las llamadas con los distintos métodos utilizados, y de la propagación de los datos recibidos a los componentes que lo necesiten.

En primer lugar, para el componente *app.component* se han realizado dos tests básicos, como se puede observar en la Figura 4.6; el primero comprueba que el componente se cree y se inicialice correctamente, y el segundo, que el título sea el indicado en código, en este caso *Integración con GitHub*.

```
import { TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have as title 'Integración con GitHub'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app.title).toEqual('Integración con GitHub');
  });
});
```

Figura 4.6: Pruebas unitarias de *app.component*.

Respecto al componente *ut-list.component*, las pruebas unitarias realizadas se pueden comprobar en la Figura 4.7. El concepto de prueba unitaria choca con la dependencia de información de otro componente, ya que se está probando únicamente la funcionalidad de la unidad, por lo que en este caso ha sido necesario la sustitución de las dependencias que la creación de este componente requería por unas dependencias de prueba. En este caso, este componente tiene una función que requería de los datos de la unidad de trabajo seleccionada por el usuario, pero como no es realmente relevante para la ejecución y el éxito del test, dicha información se ha sustituido por *null*, o datos nulos. Por una parte, un test que se ha realizado ha sido la correcta inicialización del componente, incluso con los valores sustituidos comentados anteriormente; por otra, se ha probado que el componente creara y listara exactamente cuatro unidades de trabajo, que es el número definido en el fichero a partir del cual la aplicación recupera dichas UT.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MatDialogModule, MAT_DIALOG_DATA } from '@angular/material/dialog';
import { By } from '@angular/platform-browser';

import { UtListComponent } from './ut-list.component';

describe('UtListComponent', () => {
  let component: UtListComponent;
  let fixture: ComponentFixture<UtListComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ UtListComponent ],
      imports: [ MatDialogModule ],
      providers: [ {
        provide: MAT_DIALOG_DATA, useValue: null
      } ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(UtListComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should create four mat-cards', () => {
    const mat = fixture.debugElement.queryAll(By.css('.mat-card'));
    expect(mat.length).toBe(4)
  });
});
```

Figura 4.7: Pruebas unitarias de *ut-list.component*.

El componente *ut-body.component*, cuyos tests se ven reflejados en la Figura 4.8, ha requerido de una situación similar al componente anterior. En este caso, la creación e inicialización requería de los datos específicos de una UT, el nombre y su ID, y además requería del servicio *gfetch.service*, para poder pedir y recuperar la información que este módulo necesita. Por ello, se ha necesitado de la inicialización de una varible *data*, con la información de una UT específica, en este caso aquella con ID #DLP-2, y de una implementación parcial del servicio que se ha comentado anteriormente, que devuelve todos los *commits* del repositorio y el detalle de un *commit* específico. Dichos datos se recuperan de unos ficheros sin tener que realizar una solictud real a la API, ya que como se ha comentado anteriormente, se está probando la funcionalidad de este componente, sin depender de los servicios o información ajena a este.

Los tests que se han realizado también son dos y son similares al caso anterior, se comprueba que el componente se cree y se inicialice correctamente con los valores sustituidos que se han creado y que, en este caso concreto, se cree únicamente una tarjeta asociada a un *commit*, ya que sólo hay uno asociado a la tarea en cuestión.

El servicio *gfetch.service* también requiere de la ejecución de pruebas unitarias, en este caso son tres, como se puede comprobar en la Figura 4.9. Este servicio se encarga de solicitar y recuperar los datos de la API de GitHub, por lo que se ha necesitado de un controlador para contener y responder a dichas llamadas, *HttpTestingController* [9]. Este controlador permite imitar las llamadas HTTP y responder datos, sin realmente hacer ninguna llamada externa al componente.

Una de las pruebas realizadas es la de comprobar la inicialización del componente; otra ejecuta un par de tests sobre la función que permite recuperar todos los *commits* de un

[9] https://angular.io/api/common/http/testing/HttpTestingController

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MatDialogModule, MAT_DIALOG_DATA } from '@angular/material/dialog';
import { Observable, of } from 'rxjs';
import { GfetchService } from '../gfetch.service';
import { By } from '@angular/platform-browser';

import allcommits from '../all_commits_test.json';
import onecommit from '../one_commit_test.json';

import { UtBodyComponent } from './ut-body.component';

describe('UtBodyComponent', () => {
  let component: UtBodyComponent;
  let fixture: ComponentFixture<UtBodyComponent>;
  let data;

  beforeEach(async () => {
    data = {
      name: "Desarrollar unidad 2",
      id: "#DLP-2"
    }

    await TestBed.configureTestingModule({
      declarations: [ UtBodyComponent ],
      imports: [ MatDialogModule ],
      providers: [ {
        provide: MAT_DIALOG_DATA, useValue: data
      }, {
        provide: GfetchService, useClass: gServiceStub
      } ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(UtBodyComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should create only one mat-card', () => {
    fixture.detectChanges();
    const mat = fixture.debugElement.queryAll(By.css('.mat-card'));
    expect(mat.length).toBe(1);
  });
});

class gServiceStub {
  getCommits() { return of(allcommits); }
  getIndividualCommit(sha) { return of(onecommit); }
  getCommitsofFile(file) { return new Observable<any>(); }
}
```

Figura 4.8: Pruebas unitarias de *ut-body.component*.

repositorio, que requiere que el número total de *commits* recuperados sean seis, porque ese es el número total de *commits* del repositorio objetivo, y además que el objeto con los *commits* recuperados mediante la solicitud HTTP sea igual al objeto en un fichero en memoria, que contiene también todos los *commits* del repositorio; y por último también se aplica un test sobre la función que recupera la información de un único *commit*, dicho test se asegura que el objeto recuperado sea el mismo que el que hay en otro fichero local. Si los tests mencionados tienen éxito, significa que el servicio hace correctamente su función de solicitar y recuperar la información requerida.

```
import { TestBed } from '@angular/core/testing';
import { GfetchService } from './gfetch.service';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import allcomits from './all_commits_test.json';
import onecommit from './one_commit_test.json';

describe('GfetchService', () => {
  let service: GfetchService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [GfetchService]
    });
    service = TestBed.inject(GfetchService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  describe('#getCommits', () => {
    it('should return an Observable<any>', () => {
      const dummyCommits = allcomits;

      service.getCommits().subscribe(commits => {
        expect(commits.length).toBe(6);
        expect(commits).toEqual(dummyCommits);
      });

      const req = httpMock.expectOne(`${service.baseurl}repos/${service.reponame}/commits`);
      expect(req.request.method).toBe("GET");
      req.flush(dummyCommits);
    });
  });

  describe('#getIndividualCommit', () => {
    it('should return an Observable<any>', () => {
      const dummyCommit = onecommit;
      const sha = 'e090000f5673b8c0a1fed6850d47fbfb27141a84';
      service.getIndividualCommit(sha).subscribe(commit => {
        console.log(commit);
        expect(commit).toEqual(dummyCommit);
      });

      const req = httpMock.expectOne(`${service.baseurl}repos/${service.reponame}/commits/` + sha);
      expect(req.request.method).toBe("GET");
      req.flush(onecommit);
    });
  });


  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

Figura 4.9: Pruebas unitarias de *gfetch.service*.

En la Figura 4.10 se muestra el estado de los tests después de la ejecución de estos. Se puede observar que los nueve tests que se han implementado para los cuatro componentes pasan con éxito, lo que implica que las asunciones y las especificaciones definidas para cada uno de ellos se cumplen.

Como resumen, en este apartado se ha visto cómo se ha utilizado Karma y Jasmine para la confección de especificaciones y la ejecución de pruebas unitarias en los diferentes componentes que configuran el prototipo. Adicionalmente, se ha comprobado que los tests en cuestión se ejecuten de manera correcta.

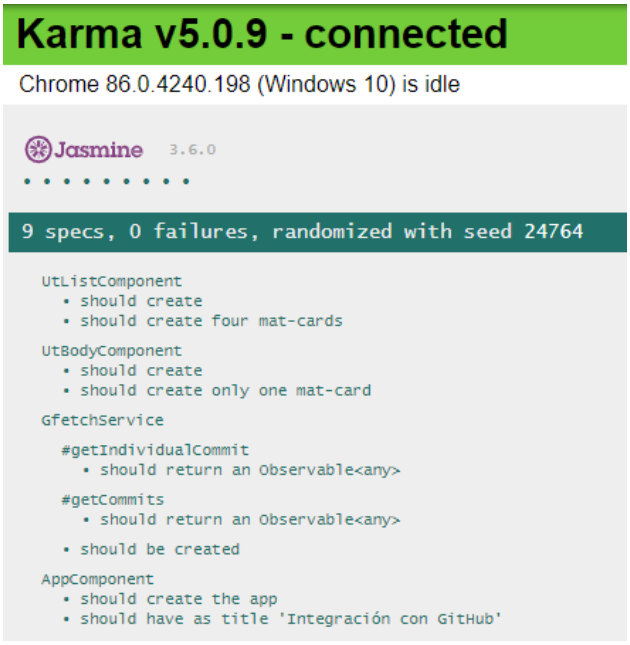

Figura 4.10: Tests con éxito en Jasmine.

4.6 Cronología

Para finalizar esta memoria, se adjunta una cronología por semanas del desarrollo de este trabajo, tanto de la memoria como del prototipo:

⇒ [03/08/2020 - 30/08/2020]

- Configuración del entorno de trabajo.
- Planificación del tiempo, estudio de alcance del proyecto.
- División de la carga de trabajo.
- Inicio del capítulo 1.
- Inicio del capítulo 2.

⇒ [31/08/2020 - 27/09/2020]

- Finalización capítulo 1.
- Inicio punto 3: Investigación sobre el estado del arte .

⇒ [28/09/2020 - 04/10/2020]

- Realizar mockup del prototipo.
- Finalización capítulo 2.

⇒ [05/10/2020 - 11/10/2020]

- Investigación sobre la tecnología a emplear.

⇒ [12/10/2020 - 18/10/2020]

- Finalización capítulo 3.
- Inicio de la construcción del prototipo.

⇒ [19/10/2020 - 25/10/2020]

- Desarrollo del prototipo.
- Inicio capítulo 4.

⇒ [26/10/2020 - 01/11/2020]

- Desarrollo del prototipo.

⇒ [02/11/2020 - 08/11/2020]

- Finalizar las acciones necesarias para el prototipo.

⇒ [23/11/2020 - 29/11/2020]

- Finalización capítulo 4.
- Inicio y fin del capítulo 5.
- Corrección de la memoria según las directrices del tutor.
- Corrección de bugs de última hora.

CAPÍTULO 5
Conclusiones finales y trabajo futuro

En el primer capítulo de este trabajo se establecían una serie de objetivos que, en finalizar el desarrollo, se deberían haber cumplido, y haciendo una mirada retrospectiva, se puede afirmar que dichos objetivos se han logrado exitosamente.

El estudio de las API realizado ha permitido un mayor entendimiento de los datos necesarios, y cómo su distribución y valores para una llamada que pida los mismos datos puede cambiar en dos servicios diferentes. Durante dicho estudio, se descubrieron una serie de solicitudes que más tarde nos serían fundamentales para los fines de la aplicación desarrollada, como por ejemplo el listado completo de *commits* o los *commits* asociados a un fichero específico.

El análisis del estado del arte en cuanto a las integraciones incluidas en herramientas de gestión ágil del trabajo proporcionó información muy valiosa sobre el proceso y el planteamiento que seguían tales herramientas para conseguir una integración sencilla de un SCV. En esta fase se analizaron las diferentes funciones que tenían las integraciones para hacer del desarrollo de *software* un proceso ubicuo y razonablemente fluido.

En lo que al desarrollo del prototipo respecta, habiendo entendido los datos a solicitar, la forma de solicitarlos y la respuesta que se nos daba en el segundo capítulo, y habiendo revisado los enfoques aplicados por otras integraciones en el tercero, el prototipo evolucionó uniendo los conocimientos adquiridos. En dicho apartado, se ha revisado el diseño y las funcionalidades de la interfaz, además de la aplicación de unas pruebas unitarias básicas a cada uno de los componentes.

Como resultado de todas estas fases, se ha tenido éxito en el desarrollo de la herramienta, que integra un sistema de control de versiones que cumple correctamente con los requisitos establecidos, y que aplica una serie de funcionalidades destinadas al control y a la trazabilidad de las modificaciones en las unidades de trabajo.

Adicionalmente, el desarrollo de este trabajo ha resultado en una profundización del entendimiento de términos, conceptos y herramientas que en este TFG se exponen, y ha permitido el aumento del conocimiento que ya se tenía previamente sobre Angular y TypeScript, disciplinas con las que se tuvo contacto en asignaturas del curso, como por ejemplo en Proyecto de Ingeniería de software. En el desarrollo del prototipo se han seguido las nociones aprendidas en la asignatura Diseño de Software, con el objetivo de que el código escrito fuera legible y de calidad. Los conocimientos de interfaces de aplicaciones y su funcionamiento que fueron enseñados en la asignatura Redes de Computadores también fueron una parte muy importante, ya que ayudaron a agilizar el entendimiento de su funcionamiento.

Como trabajo futuro, se propone una mayor refactorización y una revisión del código, ya que las herramientas utilizadas y las funciones implementadas se . Con una nueva especificación de requisitos, se podrían añadir una serie de operaciones destinadas a la mejora de la herramienta, como por ejemplo la habilidad de realizar *commits* desde la misma interfaz o la creación y gestión de ramas. Este trabajo ha sido una propuesta de integración para Worki, con lo que se espera que la información recopilada de los análisis y estudios realizados, además de las conclusiones a las que se ha llegado durante el desarrollo de la herramienta sean de utilidad para la implantación de este tipo de sistema en el producto.

Referencias

[1] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, Nils Brede Moe. A decade of agile methodologies: Towards explaining agile software development. En *Journal of Systems and Software*, 85:6:1213–1221, 2012.

[2] C. Larman y V. R. Basili. Iterative and incremental developments, a brief history. En *Computer*, 36:6:47–56, junio, 2003.

[3] Nazatul Nurlisa Zolkifli, Amir Ngah, Aziz Deraman. Version Control System: A Review. En *Procedia Computer Science*, 135:408–415, 2018.

[4] Project Management Institute. *A Guide To The Project Management Body Of Knowledge (PMBOK Guide)*. Newtown Square, Pennsylvania: Project Management Institute, quinta edición, 2013.

[5] Seshadri, S. *Angular: Up and Running: Learning Angular, Step by Step*. O'Reilly Media, Sebastopol, 2018.

[6] Richardson, Leonard and Amundsen, Mike and Ruby, Sam. *RESTful Web APIs*. O'Reilly Media, Sebastopol, 2013.

[7] Imagen sobre el flujo de estados en PivotalTracker. Disponible en https://www.pivotaltracker.com/help/articles/story_states/.