



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Máster Universitario en Ingeniería en Informática



**TFM del Máster Universitario en
Ingeniería Informática**

**Análisis Visual de Revisiones de
Código**



Presentado por Mario Juez Gil
en Universidad de Burgos — 27 de junio de 2017
Tutores: Dr. Carlos López Nozal, Dr. Raúl
Marticorena Sánchez



**UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR**

Máster Universitario en Ingeniería en Informática



D. Carlos López Nozal y Raúl Marticorena Sánchez, profesores del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Exponen:

Que el alumno D. Mario Juez Gil, con DNI 71308224J, ha realizado el Trabajo final de Máster en Ingeniería Informática titulado Análisis Visual de Revisiones de Código.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 27 de junio de 2017

Vº. Bº. del Tutor:

Vº. Bº. del Tutor:

D. Carlos López Nozal

D. Raúl Marticorena Sánchez

Resumen

El proceso de revisión de código se considera una buena práctica de desarrollo ágil. Aporta beneficios como la transferencia de conocimiento, la reducción de errores o la mejora de la calidad de código.

Recientemente GitHub ha incorporado herramientas para realizar revisiones de código a través de su plataforma, y grandes proyectos de código abierto como Elasticsearch ya lo utilizan en su proceso de desarrollo.

Este trabajo se centra en la creación de una herramienta que permita obtener y visualizar datos sobre revisiones de código realizadas en GitHub para su análisis posterior.

Descriptores

Revisión de código, revisor, calidad de código, GitHub, API REST, aplicación web SPA.

Abstract

The code review process is considered an agile practice. It provides benefits as knowledge sharing, mistakes reduction, or better code quality.

GitHub has recently included code review tools for performing this process through their platform, which are currently being used by big open source projects as Elasticsearch in their development process.

The main goal of this project is to develop a tool which allows to obtain and visualize code review data from GitHub for its subsequent analysis.

Keywords

Code review, reviewer, code quality, GitHub, REST API, SPA web application.

Índice general

Índice general	III
Índice de figuras	v
Índice de tablas	vi
Introducción	1
1.1. Estructura de la memoria	2
Objetivos del proyecto	3
2.1. Objetivos generales	3
2.2. Objetivos técnicos	3
Conceptos teóricos	5
3.1. Conceptos relacionados con las revisiones de código	5
3.2. Conceptos relacionados con buenas prácticas ágiles	10
3.3. Otros conceptos	12
Técnicas y herramientas	14
4.1. Metodologías ágiles	14
4.2. Git	14
4.3. GitHub	15
4.4. Node.js	15
4.5. TypeScript	16
4.6. jQuery	16
4.7. Sammy.js	16
4.8. Semantic-UI	17
4.9. D3.js + C3.js	17
4.10. Visual Studio Code	17
4.11. Google Drive	18
4.12. MongoDB	18
4.13. Herramientas de integración continua	19
4.14. Heroku	20
4.15. LaTeX	21

ÍNDICE GENERAL

IV

4.16. Skype Empresarial	21
4.17. Postman	22
Aspectos relevantes del desarrollo del proyecto	23
5.1. Ciclo de vida del proyecto	23
5.2. Estableciendo la base	25
5.3. Obteniendo datos de GitHub	29
5.4. Visualización de los datos	39
5.5. Otros aspectos	40
Trabajos relacionados	42
6.1. GHTorrent	42
6.2. ReDA - Review Data Analyzer	43
6.3. GiLA - GitHub Label Analyzer	43
Conclusiones y Líneas de trabajo futuras	44
7.1. Conclusiones	44
7.2. Líneas de trabajo futuras	45
Bibliografía	47

Índice de figuras

1.1. Ejemplo de análisis gráfico de un experto revisor.	2
3.2. Operaciones de la inspección de software.	6
3.3. Revisiones de código en git.	8
3.4. Ejemplo de los dos patrones de integración existentes.	9
5.5. Porcentaje de <i>sprints</i> dedicados a cada fase.	24
5.6. Una tarea de GitHub con ZenHub habilitado.	24
5.7. Ejemplo de uso de rama <i>dev</i> en GitHub.	25
5.8. Anvireco como aplicación distribuida.	26
5.9. Página web tradicional vs Aplicación SPA [28].	26
5.10. Entornos de producción y desarrollo del proyecto.	27
5.11. Proceso de integración continua con Travis CI.	27
5.12. Posibles estados de Travis CI	28
5.13. Evolución del GPA de la última semana.	29
5.14. Ejemplo de <i>pull request</i>	31
5.15. Ejemplo de revisión aceptada.	31
5.16. Ejemplo de comentario de revisión.	32
5.17. Error 403 API GitHub.	33
5.18. Ejemplo de cola de tareas FIFO.	35
5.19. Posibles errores en la obtención de datos.	35
5.20. Ejemplo de reintento tras error 403.	36
5.21. Planificación de la cola de tareas.	37
5.22. Interfaz web para crear tareas.	38
5.23. Mensajes de información sobre la correcta o incorrecta creación de tareas.	38
5.24. Los tres posibles estados del gestor de tareas.	39
5.25. Ejemplos de los tres tipos de gráfico.	40

Índice de tablas

5.1.	Tipos de entidades de GitHub	30
5.2.	Repositorios encolados (20-06-2017).	37
6.3.	Comparativa de las características de Anvireco y GHTorrent.	42
6.4.	Comparativa de las características de Anvireco y ReDA.	43
6.5.	Comparativa de las características de Anvireco y GiLA.	43

Introducción

El desarrollo de software es un proceso que siempre se encuentra en constante evolución. Las metodologías existentes se caracterizan por su diversidad y flexibilidad, y es común la aparición de nuevas técnicas y tendencias con objetivos como, por ejemplo, incrementar la calidad del software desarrollado.

Las metodologías de desarrollo ágil y colaborativo actualmente están incorporando, entre sus buenas prácticas, las revisiones de código como vía para crear código de forma más eficiente, con menos defectos, con una mantenibilidad mayor, y en definitiva, mejor calidad.

El concepto de revisión de código es propuesto por M.E. Fagan en 1976 [10], pero su práctica no estaba muy generalizada. Sin embargo, actualmente se puede observar que cada vez son más los proyectos que incorporan las revisiones de código a su ciclo de desarrollo.

GitHub [17], la mayor plataforma de desarrollo colaborativo, ha incorporado a principios de 2017 funcionalidades para realizar revisiones de código a través de su sistema de *pull requests*.

Grandes proyectos como Elasticsearch [9] de Elastic o WebFundamentals [19] de Google ya están utilizando activamente las revisiones de código de GitHub. También existen herramientas para este propósito específico como Gerrit Code Review [25] de Google utilizada en proyectos como Android [4], Linux Foundation [13] o Eclipse [8].

Para que las revisiones de código sean útiles deben llevarse a cabo por revisores expertos. Actualmente la evaluación y análisis de la calidad de revisiones y revisores es una práctica muy poco común, sin embargo, podría contribuir a una mejor selección de revisores para cada caso concreto, haciendo de la revisión de código un proceso más eficaz y fiable.

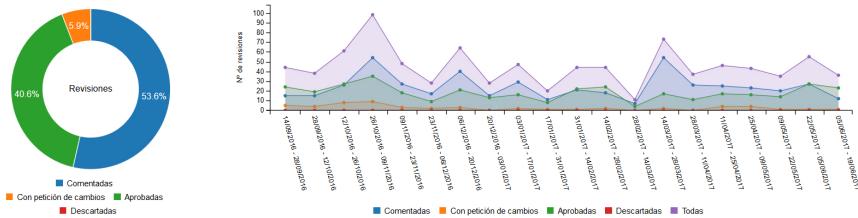


Figura 1.1: Ejemplo de análisis gráfico de un experto revisor.

Este proyecto se centra en la creación de una herramienta que permita obtener y representar gráficamente datos sobre revisiones de código realizadas en repositorios de GitHub a través de *pull requests*. Con estas representaciones se facilita un instrumento para análisis de revisiones en un proyecto y su relación con el equipo de desarrollo.

1.1. Estructura de la memoria

La memoria tiene la siguiente estructura:

- **Introducción:** esta parte de la memoria aborda el contexto del problema y una breve explicación de la solución desarrollada.
- **Objetivos del proyecto:** en esta parte de la memoria se exponen los objetivos perseguidos con la realización del proyecto.
- **Conceptos teóricos:** apartado de la memoria donde se exponen una serie de conceptos teóricos relacionados con el proyecto.
- **Técnicas y herramientas:** esta parte de la memoria está dedicada a la descripción de las diversas técnicas y herramientas empleadas durante el desarrollo del proyecto.
- **Aspectos relevantes del desarrollo del proyecto:** parte de la memoria que aborda los aspectos más destacables del desarrollo del proyecto.
- **Trabajos relacionados:** exposición de trabajos relacionados con el proyecto.
- **Conclusiones y líneas de trabajo futuras:** en esta parte de la memoria se detallan las conclusiones obtenidas tras el desarrollo y qué líneas de trabajo se podrían seguir a continuación.

Objetivos del proyecto

Este apartado se ha desglosado en dos secciones. En la primera se enumeran los objetivos de carácter general asociados a los requisitos del proyecto. En la segunda se describen aquellos objetivos técnicos relacionados con las tecnologías y metodologías a utilizar.

2.1. Objetivos generales

Se desea desarrollar una herramienta con dos funcionalidades claramente diferenciadas:

- Obtener y almacenar datos sobre revisiones de código realizadas en repositorios alojados en la plataforma GitHub.
- Representar gráficamente los datos en bruto a través de gráficas de diferentes tipos.

Con el desarrollo de una herramienta de este tipo se persigue lo siguiente:

- Obtención de conocimiento e información útil sobre revisiones de código y revisores mediante el análisis visual de los datos.
- Creación de un almacén de datos cuyo contenido pueda ser utilizado en procesos de minería de datos en ámbitos de investigación.
- Contribuir a una mejora de las revisiones de código haciendo de éste un proceso más importante y útil para el desarrollo de software de calidad.

2.2. Objetivos técnicos

A nivel técnico, con la realización de este proyecto se busca aplicar los conocimientos obtenidos a lo largo del grado y máster, así como el aprendizaje de nuevas tecnologías y metodologías utilizadas en la actualidad:

- Uso de la metodología ágil Scrum.
- Profundizar en la utilización de Git como sistema de control de versiones, y de forma concreta, la herramienta GitHub y sus funcionalidades particulares.
- Aprendizaje y uso de integraciones de GitHub como ZenHub para la gestión de proyectos.
- Aprendizaje y uso de técnicas de integración continua con herramientas como Travis, Heroku o Codebeat.
- Aprendizaje y uso de TypeScript (JavaScript tipado) junto con node.js.
- Aprendizaje y uso de jQuery.
- Utilización de bases de datos no relacionales (NoSQL), concretamente MongoDB.
- Creación de una aplicación distribuida cuyos elementos sean independientes formada por:
 - API REST en el lado del servidor.
 - Cliente de tipo SPA (*Single Page Application*).

Conceptos teóricos

En este apartado se van a exponer una serie de conceptos teóricos relacionados con las revisiones de código y con buenas prácticas de desarrollo ágil. Cada concepto contiene una definición y una breve descripción de su relación con el trabajo.

3.1. Conceptos relacionados con las revisiones de código

A continuación se definen diversos conceptos relacionados con las revisiones de código, las cuales son la base de este trabajo.

Revisión de código

La revisión de código es un proceso de ingeniería a través del cual se realiza una inspección del código fuente por otros desarrolladores diferentes al autor del mismo. Resulta útil para reducir los defectos de código así como mejorar la calidad del software [1].

Frente a las revisiones de código altamente estructuradas propuestas por Fagan [10], hoy en día se están adoptando metodologías más livianas con el fin de solventar las ineficiencias de las inspecciones de código. Las denominadas *Modern Code Reviews* son informales, hacen uso de herramientas, y se utilizan regularmente.

Mediante el uso de estas nuevas metodologías, las revisiones de código ofrecen un mayor número de beneficios a los equipos de desarrollo como transferencia de conocimientos, visión de equipo, o mejores soluciones a los problemas [5].

Las revisiones de código son el elemento principal de este trabajo, donde se van a obtener datos de las mismas realizadas en diversos repositorios de software.

Inspección de software

La inspección de software, también conocida como inspección de Fagan, consiste en la búsqueda de defectos en documentos de desarrollo (código, especificaciones, diseño, etc.) por parte de personas con diversos roles mediante un proceso estructurado y bien definido, con el fin de disminuir el número de errores y aumentar la calidad del producto final [10].

Las fases de dicho proceso son las siguientes [11]:

- **Planificación:** en esta fase, los materiales que van a ser inspeccionados deben coincidir con los criterios de entrada de la inspección. Tiene lugar la elección de los participantes de la inspección. También se realiza la organización de las reuniones (hora y lugar).
- **Información general:** en esta fase se comunica a los participantes cuales son los resultados esperados. También se realiza la asignación de roles.
- **Preparación:** en esta fase tiene lugar el estudio del material por parte de los participantes y su preparación para cumplir sus roles.
- **Reunión de inspección:** esta es la fase donde se lleva a cabo la búsqueda de defectos.
- **Repetición del trabajo:** el autor debe solucionar los defectos detectados, tras ello todo el proceso vuelve a comenzar desde la fase de planificación.
- **Seguimiento:** el moderador o equipo completo de inspección debe verificar que todos los cambios son correctos y no introducen nuevos defectos.

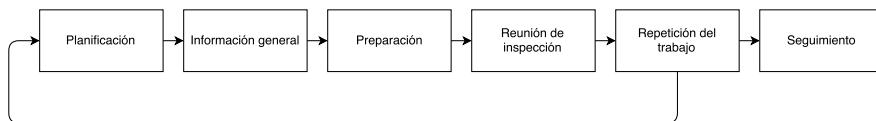


Figura 3.2: Operaciones de la inspección de software.

Además, durante el proceso de inspección de software intervienen los siguientes roles:

- **Autor:** responsable del desarrollo del documento a inspeccionar (código).
- **Lector:** responsable de leer el documento como si fuese a desarrollarlo él mismo.
- **Revisor:** responsable de examinar e inspeccionar el documento con el fin de identificar posibles defectos.
- **Moderador:** responsable de coordinar la reunión de inspección.

Revisor de código

El revisor de código tiene como responsabilidad principal examinar el código y detectar posibles defectos, ofreciendo comentarios y sugerencias de cambio al autor del código que permitan la mejora de la calidad del mismo.

Los aportes del revisor de código también pueden proporcionar nuevos conocimientos al autor del código revisado, mejorando así sus habilidades lo cual se puede traducir en una mejora progresiva de la calidad del software que desarrolle.

Se pueden distinguir dos tipos de revisores de código:

- **Revisor experto:** es una persona, idealmente con experiencia y amplios conocimientos (aunque no es estrictamente necesario).
- **Robot revisor:** son herramientas que permiten la revisión automática de código mediante la comprobación del código fuente para garantizar que cumpla un conjunto de reglas predefinidas, así como detectar errores [29].

Actualmente es común que las revisiones cuenten con ambos tipos de revisores, en primer lugar se realiza una revisión automática, y los resultados son utilizados por el revisor experto para ofrecer una mejor revisión con un mayor nivel de detalle.

El revisor de código es una figura importante de este trabajo. Nos interesa extraer los comentarios que registra en cada cambio puesto que pueden resultar útiles para obtener métricas que podrían permitir evaluar la calidad de los comentarios o incluso del propio revisor.

Sistema de control de versiones

Un sistema de control de versiones se encarga de registrar los diferentes cambios de un fichero o un conjunto de ficheros a lo largo del tiempo, permitiendo así recuperar versiones específicas en cualquier momento [6].

Gracias a este tipo de sistemas es posible revertir los cambios realizados en un fichero o incluso en un proyecto completo a un estado anterior, comparar los cambios, comprobar la autoría de los mismos, etc.

Los sistemas de control de versiones están en constante evolución, actualmente herramientas como Github están empezando a implementar funcionalidades para facilitar las revisiones de código. Por tanto, en este trabajo vamos a utilizar este tipo de sistemas como fuente de datos a extraer.

A continuación se muestra un diagrama donde se muestra la figura de la revisión de código y su relación con diferentes elementos de los sistemas de control de versiones (concretamente Git).

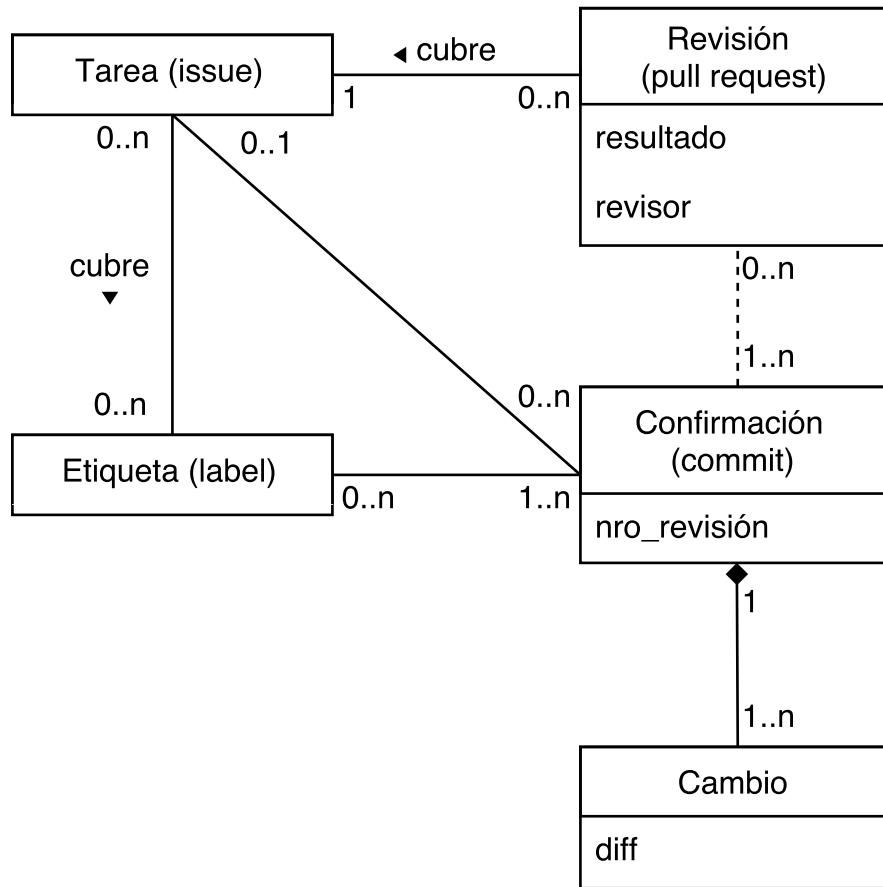


Figura 3.3: Revisiones de código en git.

Repositorio

El repositorio es la parte central de un sistema de control de versiones, es el lugar donde se almacenan los datos del sistema, normalmente estructurados en forma de árbol de ficheros [24].

La característica principal de un repositorio es que mantiene todas las versiones de cada fichero, permitiendo al cliente recuperar estados previos de los mismos.

Cambio (diff)

Un cambio es una modificación concreta de un fichero alojado en un repositorio de un control de versiones [30].

Resulta útil para comparar dos versiones del mismo archivo.

Rama (branch)

Una rama es un apuntador a una confirmación. Por cada confirmación realizada, la rama avanza hasta la última confirmación. Por defecto existe una rama principal [6].

Crear otras ramas sirve para abrir nuevas líneas de desarrollo, por ejemplo una rama de pruebas, evitando así desordenar la principal.

Integración (merge)

Se entiende por integración a la unión de los cambios de dos ramas en una. Existen dos patrones de uso de esta operación [7]:

- Añadir los cambios realizados en una rama donde se están implementando nuevas características a la rama principal.
- Añadir los cambios de la rama principal en una rama donde se están implementando nuevas características con el fin de mantener la rama actualizada con los últimos parches y características. Esta práctica permite reducir el riesgo de conflictos¹ al unir la rama de desarrollo con la principal.

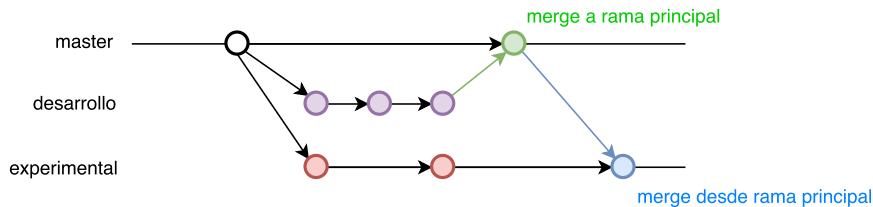


Figura 3.4: Ejemplo de los dos patrones de integración existentes.

Confirmación (commit)

Una confirmación (o *commit*) supone el almacenamiento en el repositorio de una nueva versión de los ficheros que contiene, en otras palabras, sirve para guardar los cambios [6].

Cada confirmación lleva asociados uno o varios ficheros modificados, un identificador denominado revisión y un mensaje donde se describen los cambios realizados.

¹Existe un conflicto cuando al integrar dos ramas, un mismo fichero ha sido modificado en ambas y por ello es necesario decidir que cambios deben mantenerse en la versión final.

Tag

La finalidad del uso de *tags* es dar nombre a alguna versión para que pueda ser localizada fácilmente en el futuro, es decir, permite identificar de forma sencilla revisiones importantes del proyecto [30].

Issue (bug tracker)

Las *issues* son una funcionalidad que ofrece la herramienta GitHub que permiten mantener la trazabilidad de las diferentes tareas, mejoras o fallos de los proyectos [18].

Pull request

Las *pull requests* son una característica particular de la herramienta GitHub. Sirven para mantener conversaciones sobre los cambios donde se pueden exponer ideas, asignar tareas, tratar detalles y hacer revisiones de código [16].

Etiqueta (label)

Las etiquetas (*labels*) de GitHub se pueden aplicar a las *issues* o *pull requests* como método para indicar prioridad, categoría, a qué requerimiento pertenece o cualquier otra información que pueda ser de utilidad.

Por defecto GitHub tiene siete etiquetas: *bug*, *duplicate*, *enhancement*, *help wanted*, *invalid*, *question* y *wontfix*.

3.2. Conceptos relacionados con buenas prácticas ágiles

Esta parte de conceptos teóricos está dedicada a buenas prácticas ágiles que se van a utilizar a lo largo del desarrollo del presente proyecto.

Para la definición de los siguientes conceptos se ha seguido el glosario de términos de Agile Alliance [2].

Integración continua

La integración continua es una práctica de desarrollo software donde los miembros de un equipo integran su trabajo frecuentemente, normalmente como mínimo una integración diaria [14].

Tiene dos objetivos principales:

- Minimizar la duración y el esfuerzo requerido por cada tarea de integración.

- Permitir la creación de una versión del producto candidata a ser publicada en cualquier momento.

Para la consecución de estos objetivos se requiere un proceso de integración reproducible y altamente automatizado. Para tal fin se pueden utilizar herramientas de control de versiones, diversas políticas y convenciones, así como las propias herramientas de integración continua.

Despliegue continuo

El despliegue continuo se puede entender como una parte de la integración continua que busca minimizar el tiempo que va desde el inicio del desarrollo de una característica, hasta que dicha característica es utilizada por el usuario final en un entorno de producción.

Construcción automática

Por construcción se entiende al proceso de convertir ficheros y otros elementos en el producto final. La construcción puede incluir:

- Compilar ficheros fuente.
- Empaquetar ficheros compilados.
- Crear de instaladores.
- Crear o actualizar el esquema o los datos de la base de datos.

Diagrama “Burn Down”

Un diagrama *Burn Down* permite relacionar la cantidad de trabajo restante (en el eje vertical) con el tiempo transcurrido desde el inicio del proyecto —*burn down* del producto— o un hito —*burn down* del *sprint*— (en el eje horizontal).

Desarrollo iterativo e incremental

Se dice que un proyecto ágil sigue un desarrollo iterativo cuando permite la repetición de actividades relacionadas con el desarrollo de software, así como la revisión del trabajo ya realizado mediante refactorizaciones por ejemplo.

El desarrollo iterativo también se puede caracterizar por contar con una serie de iteraciones con una duración prefijada, aunque el uso de ciertas técnicas de planificación como Kanban no requieren que esta parte se cumpla.

Actualmente, gran parte de los proyectos ágiles además de seguir técnicas de desarrollo iterativo, utilizan técnicas de desarrollo incremental, lo cual supone que cada versión sucesiva del producto debe ser usable y añadir nuevas funcionalidades visibles para el usuario (incrementos verticales).

Retrospectiva

La retrospectiva requiere que el equipo tenga reuniones regulares, normalmente dichas reuniones siguen el ritmo de las iteraciones. En ellas se deben reflejar los cambios más destacados tomando como referencia la anterior reunión, sirviendo como ayuda para la toma de decisiones.

Tablero de tareas

El tablero de tareas es una herramienta donde éste se divide en varias columnas, normalmente tres, cuyo fin es categorizar las diferentes tareas, por ejemplo en “pendientes”, “en progreso”, y “finalizadas”.

El tablero de tareas se debe actualizar con frecuencia para mantener cada tarea en su estado real. Normalmente al inicio de cada iteración el tablero se vuelve a poner en un estado inicial para reflejar la planificación de la iteración.

Pruebas unitarias

Una prueba unitaria es un pequeño fragmento de programa escrito y mantenido por el equipo de desarrollo, su misión es ejecutar una parte del código y comparar el resultado de la ejecución con el resultado esperado. La salida de una prueba unitaria es binaria, indicando que la prueba ha pasado si cumple las expectativas, o que ha fallado en caso contrario. Comúnmente se debe desarrollar un número de pruebas unitarias acorde al tamaño del código que va a ser probado, dichas pruebas son agrupadas en lo que se denomina conjunto de pruebas unitarias o *test suite*.

3.3. Otros conceptos

En esta parte se definen otros conceptos relacionados con el proyecto.

Herramienta de obtención de datos

Una herramienta de obtención de datos tiene como finalidad la descarga y almacenamiento de datos desde una fuente remota.

Dependiendo de la fuente remota, la obtención de datos puede tomar cierta complejidad. Si tomamos GitHub como fuente de datos, existe un límite de 5000 peticiones por hora a su API pública. Una herramienta de obtención de datos de GitHub deberá tener en cuenta dicha limitación.

Herramienta de visualización de datos

Una herramienta de visualización de datos tiene como finalidad la transformación de datos brutos en una representación gráfica de los mismos.

API REST

El concepto REST (*Representational State Transfer*) es propuesto el año 2000 por Roy Fielding [12].

Se trata de un tipo de arquitectura de desarrollo web que hace uso del estándar HTTP [22]. Gracias a esto, cualquier cliente que conozca el protocolo HTTP puede hacer uso de aplicaciones o servicios REST. Por ello resulta mucho más simple que otras alternativas como XML-RPC o SOAP.

Aplicación web SPA

El término *Single-Page Applications* (o SPA) es introducido por Steve Yen en 2005.

Este tipo de aplicaciones se desarrollan para la web. Son accesibles a través de un navegador, como cualquier página web, pero ofreciendo interacciones más dinámicas como si de una aplicación de escritorio se tratase [26].

La mayor diferencia entre una aplicación SPA y una página web convencional es que la primera requiere una reducida cantidad de refrescos de página. Las aplicaciones SPA hacen un uso elevado de AJAX para obtener los datos necesarios para el funcionamiento de la aplicación.

Técnicas y herramientas

En este apartado se definen las diferentes técnicas y herramientas utilizadas a lo largo del desarrollo del trabajo.

4.1. Metodologías ágiles

Frente a las metodologías tradicionales, se ha optado por el uso de metodologías ágiles, las cuales priman [3]:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcional sobre documentación.
- Colaboración del cliente sobre negociación de contratos.
- Respuesta ante cambios sobre seguimiento de un plan.

Scrum

Concretamente se ha optado por el uso de Scrum como metodología ágil.

Scrum propone seguir un proceso de desarrollo iterativo e incremental a través de iteraciones denominadas sprints y de revisiones. Cada iteración debe finalizar con la entrega de una parte funcional del producto [23].

4.2. Git

Git es un sistema de control de versiones distribuido diseñado para manejar proyectos de cualquier tamaño con velocidad y eficiencia.

- <https://www.git-scm.com>

4.3. GitHub

GitHub es una plataforma que permite alojar proyectos en repositorios de código que utilizan el sistema de control de versiones Git. También cuenta con funcionalidades para realizar revisiones de código. En este trabajo se va a utilizar esta herramienta como repositorio de código y como fuente de datos sobre revisiones de código en diferentes repositorios.

- <https://www.github.com>

Alternativas estudiadas

Bitbucket es una alternativa a Github como herramienta de repositorio de código.

- <https://bitbucket.org>

En el ámbito de herramienta de revisión de código se estudiaron Gerrit Code Review y Review Board.

- Gerrit Code Review: <https://www.gerritcodereview.com/>
- Review Board: <https://www.reviewboard.org/>

4.4. Node.js

Node.js es un entorno de ejecución para JavaScript construido con el motor de JavaScript v8 de Chrome. En este trabajo se utiliza para ejecutar código JavaScript en el lado del servidor.

- <https://nodejs.org>

npm

Node.js cuenta con npm, un gestor de paquetes con un amplio número de librerías registradas.

- <https://www.npmjs.com/>

4.5. TypeScript

TypeScript es un superconjunto tipado de JavaScript que es compilado a JavaScript plano. Está desarrollado por Microsoft, y su uso puede mejorar la legibilidad y el entendimiento del código con respecto a JavaScript.

Se ha escogido este lenguaje de programación por que se va a tratar con documentos JSON (*JavaScript Object Notation*), y mediante JavaScript (o TypeScript) este cometido se simplifica.

- <https://www.typescriptlang.org/>

TypeDoc

TypeDoc es una herramienta para la generación de documentación para proyectos desarrollados en TypeScript. Su sintaxis es similar a la de JSDoc, pero simplificada ya que es capaz de leer e incluir los tipos de los parámetros definidos en el código.

- <http://typedoc.org/>

4.6. jQuery

jQuery es una librería de JavaScript, rápida, versátil, liviana y llena de características. Simplifica la manipulación de documentos HTML, el tratamiento de eventos, uso de AJAX, etc. Es compatible con multitud de navegadores.

- <https://jquery.com/>

4.7. Sammy.js

Sammy.js es un pequeño *framework* JavaScript que simplifica el desarrollo de aplicaciones web SPA. En el proyecto se utiliza para realizar el enrutamiento en el cliente.

- <http://sammyjs.org/>

Alternativas estudiadas

Como alternativas a Sammy se han estudiado AngularJS y React, dos conocidos *frameworks* para el desarrollo de aplicaciones web en el lado del cliente. Se ha escogido Sammy por su simplicidad frente a estos *frameworks*.

- AngularJS: <https://angularjs.org/>
- React: <https://facebook.github.io/react/>

4.8. Semantic-UI

Semantic-UI es un *framework* para maquetación web que hace uso de una sintaxis similar al lenguaje natural que ayuda a crear un HTML más intuitivo.

- <https://semantic-ui.com/>

Alternativas estudiadas

Como alternativa a Semantic-UI se ha estudiado Bootstrap. Elegimos el primero por que la versión actual de Bootstrap (3) fue liberada en 2013, y la nueva versión (4) aun se encuentra en fase de desarrollo.

- <http://getbootstrap.com/>

4.9. D3.js + C3.js

D3.js es una librería JavaScript para la manipulación de documentos basados en datos. Permite representar datos de diversas maneras mediante el uso de HTML, SVG y CSS.

C3.js es una librería que contiene una serie de gráficos pre-diseñados desarrollados en D3.js, y gracias a ella se representan los diferentes gráficos de la parte del cliente del proyecto.

- D3.js: <https://d3js.org/>
- C3.js: <http://c3js.org/>

4.10. Visual Studio Code

Visual Studio Code es un entorno de desarrollo integrado (IDE) multiplataforma desarrollado por Microsoft, cuenta con una herramienta integrada para el uso de Git, así como un elevado número de extensiones que permiten personalizar el editor para las necesidades concretas de cada proyecto.

- <https://code.visualstudio.com/>

4.11. Google Drive

Google Drive es un servicio de almacenamiento en la nube, ofrece 15 GB de almacenamiento gratuito y se utiliza en este trabajo como sistema de copias de seguridad y como medio para mantener el entorno de desarrollo sincronizado en cualquier máquina.

- <https://drive.google.com>

Grive2

Grive2 (fork de grive) permite sincronizar el contenido de la carpeta Google Drive desde Linux, plataforma para la que actualmente no existe cliente oficial.

- <https://github.com/vitalif/grive2>

4.12. MongoDB

MongoDB es un sistema gestor de base de datos NoSQL. Ofrece escalabilidad, rendimiento y gran disponibilidad. El motivo de uso de este tipo de SGBD sobre uno de tipo SQL en este trabajo es que MongoDB utiliza documentos JSON para almacenar los registros, el mismo formato en que se obtienen los datos que deseamos almacenar desde la API de GitHub.

- <https://www.mongodb.com>

Mongoose

Mongoose es un ODM (*Object Document Mapper*) para MongoDB y JavaScript. Incluye características como conversión de tipos, validación, construcción de consultas, etc.

- <http://mongoosejs.com/>

mLab

MLab es una herramienta de la familia DaaS (*Database As A Service*) permite alojar bases de datos MongoDB en la nube, su plan gratuito permite la creación de varias bases de datos con hasta 500 MB de espacio de almacenamiento cada una.

- <https://mlab.com/>

Alternativas estudiadas

MongoDB Atlas ofrece bases de datos MongoDB como servicio pero su plan gratuito solo permite la creación de un clúster.

- <https://www.mongodb.com/cloud/atlas>

4.13. Herramientas de integración continua

Las siguientes herramientas se utilizan en el proceso de integración continua.

Travis CI

Travis CI es un sistema de integración continua. Permite automatizar tareas como la construcción, ejecución de pruebas y despliegue de aplicaciones alojadas en Github.

- <https://travis-ci.org/>

Gulp

Gulp es una librería que permite automatizar diversas tareas comunes en el desarrollo de aplicaciones, como por ejemplo la compilación de código fuente. En este trabajo una de las tareas será compilar TypeScript a JavaScript.

- <http://gulpjs.com/>

Alternativas estudiadas

Grunt es la alternativa principal a Gulp como método de automatización de tareas.

- <https://gruntjs.com/>

Codebeat

Codebeat es una herramienta para la ejecución de revisiones automáticas sobre repositorios de código. Tras la evaluación ofrece una puntuación denominada GPA que va de 0 (peor) a 4 (mejor). Esta puntuación se calcula teniendo en cuenta aspectos como complejidad, duplicación y seguimiento de buenas prácticas.

La motivación principal de su uso es que, al contrario que las principales herramientas, cuenta con un motor para el lenguaje TypeScript utilizado en este trabajo.

- <https://codebeat.co>

Alternativas estudiadas

Se han tenido en cuenta las herramientas SonarQube y Code Climate.

- SonarQube: <https://www.sonarqube.org/>
- Code Climate: <https://codeclimate.com/>

ZenHub

ZenHub añade funcionalidades que permiten la gestión de proyectos utilizando metodologías ágiles dentro de GitHub. En este trabajo se utiliza como tablero de tareas y para generar diagramas *burn down* de cada *sprint*.

- <https://www.zenhub.com>

Mocha + Chai + Sinon

Mocha es un *framework* de pruebas unitarias para Node.js y para navegadores que simplifica el desarrollo de pruebas para aplicaciones asíncronas.

Chai es una librería de aserciones con una sintaxis similar al lenguaje natural, facilitando la comprensión de las pruebas. También cuenta con un sistema de *plug-ins* que le permite aumentar su funcionalidad, un ejemplo es el *plug-in* chai-http que permite probar aplicaciones que funcionen mediante peticiones http (por ejemplo aplicaciones REST).

Sinon es una librería que funciona con cualquier *framework* de pruebas, y provee funcionalidades para el uso de *spies*, *stubs* y *mocks*.

- Mocha: <https://mochajs.org/>
- Chai: <http://chaijs.com/>
- Sinon: <http://sinonjs.org/>

4.14. Heroku

Heroku es una herramienta de la familia PaaS (*Platform As A Service*). Permite a los desarrolladores construir, ejecutar y operar con aplicaciones completamente en la nube. Soporta diferentes lenguajes como Java, Node.js, Scala, Clojure, Python, PHP o GO.

- <https://www.heroku.com>

Alternativas estudiadas

Como alternativa a Heroku se ha estudiado OpenShift, herramienta creada por Red Hat que hace uso de Docker. Actualmente los nuevos registros están deshabilitados y únicamente permite fases de prueba de un mes de duración.

- <https://www.openshift.com/>

4.15. LaTeX

L^AT_EX es un sistema de composición de textos orientado a la producción de documentación técnica y científica. Está formado por un conjunto de macros T_EX.

- <https://www.latex-project.org/>

Texmaker

Texmaker es un editor multiplataforma de L^AT_EX que integra todas las herramientas necesarias para desarrollar este tipo de documentos.

- <http://www.xm1math.net/texmaker/>

Alternativas estudiadas

Como editor L^AT_EX alternativo a Texmaker se valoró el uso de LaTeXila.

- <https://wiki.gnome.org/Apps/LaTeXila>

4.16. Skype Empresarial

Como herramienta de comunicación para llevar a cabo las reuniones en cada *sprint* se ha utilizado Skype Empresarial, incluida en el paquete de Office 365 ofrecido por la Universidad de Burgos. Permite programar y realizar videoconferencias con funcionalidades añadidas como compartir pantalla o mensajería instantánea.

- <https://www.skype.com/es/business/skype-for-business/>

4.17. Postman

Postman es una aplicación que permite la prueba y monitorización de API's mediante una interfaz que permite hacer peticiones de diferente tipo (GET, POST, PUT...), modificar las cabeceras, etc. Se ha utilizado en el proyecto para probar el funcionamiento de la API REST desarrollada.

- <https://www.getpostman.com/>

Aspectos relevantes del desarrollo del proyecto

En este apartado se recogen los aspectos más interesantes del desarrollo del proyecto.

5.1. Ciclo de vida del proyecto

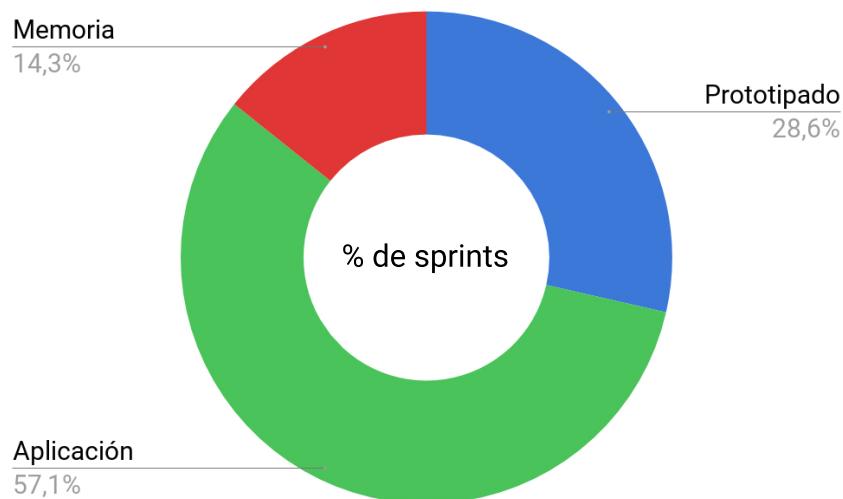
Este proyecto tiene una duración total de 4 meses y 22 días, con inicio el 10 de febrero de 2017 y fin el 3 de julio de 2017.

Como se ha utilizado la metodología ágil Scrum, el proyecto está dividido en *sprints* (o iteraciones), concretamente 14. Inicialmente las iteraciones tenían una duración de 2 semanas, y a partir del sexto *sprint* la periodicidad pasó a ser semanal.

Se pueden distinguir tres fases principales:

- **Prototipado:** en esta fase se deciden, estudian y prueban diversas tecnologías como paso previo a la decisión de cuales utilizar en el desarrollo del producto final.
- **Desarrollo de la aplicación:** se trata de la fase más importante, abarca todo el desarrollo de la aplicación que a su vez se podría dividir en dos: desarrollo de la parte *backend* (API REST) y desarrollo de la parte *frontend* (cliente SPA).
- **Desarrollo de la memoria:** esta fase consiste en escribir toda la documentación relativa al proyecto (memoria y anexos).

En el siguiente gráfico se muestra el porcentaje de *sprints* dedicados a cada una de las fases. Es un gráfico orientativo puesto que no hay un punto exacto en el cual termina la fase de prototipado y empieza la de desarrollo de la aplicación. Asimismo, en la fase de prototipado, una vez decididas las herramientas y tecnologías hubo avances en la fase de desarrollo de memoria.

Figura 5.5: Porcentaje de *sprints* dedicados a cada fase.

Gestión de *sprints* mediante GitHub y ZenHub

Cada *sprint* está formado por una serie de tareas concretas.

Para la gestión y planificación de dichas tareas se han utilizado las *issues* de GitHub junto con el *plug-in* de ZenHub que permite, entre otras cosas, incluir estimaciones de tiempo.



Figura 5.6: Una tarea de GitHub con ZenHub habilitado.

Se utilizan tres ramas:

- ***master***: rama principal, contiene los últimos cambios estables.

- ***dev***: rama de desarrollo, se va actualizando a medida que se realizan tareas. Es una rama inestable, la aplicación puede no tener el comportamiento esperado.
- ***memo***: rama de memoria, contiene los cambios no definitivos de la documentación del proyecto.

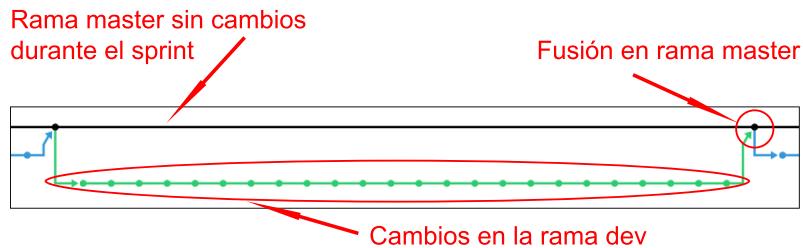


Figura 5.7: Ejemplo de uso de rama *dev* en GitHub.

El ciclo de vida de un *sprint* en GitHub consiste en:

1. Apertura de *issues* (tareas) a realizar durante la iteración.
2. Selección de rama *dev* o *memo*, dependiendo del tipo de tarea que se está llevando a cabo.
3. Desarrollo de las diferentes tareas.
4. Revisión y comprobación de que los cambios satisfacen los requisitos.
5. Apertura de *pull request* para ejecutar los *checks* de integración continua y revisión automática.
6. Cierre de *pull request* e integración de los cambios en la rama *master*.

5.2. Estableciendo la base

Antes de empezar con el desarrollo software del proyecto se tomaron ciertas decisiones que marcaron el resto del proceso.

Anvireco como aplicación distribuida

La herramienta desarrollada tiene una arquitectura cliente-servidor en la cual el servidor es completamente independiente del cliente.

Pese a que en este proyecto únicamente hay un tipo de cliente, en el futuro podrían existir otros, por ejemplo un consumidor de datos para realizar tareas de minería. Por ello es importante la independencia cliente-servidor.

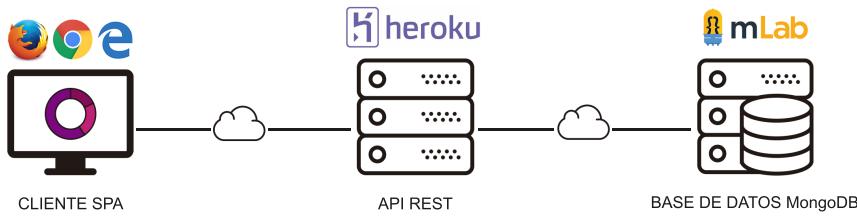


Figura 5.8: Anvireco como aplicación distribuida.

En el servidor se optó por implementar una API REST con la cual podrá comunicarse cualquier cliente que conozca el protocolo HTTP.

El cliente de visualización de datos es una aplicación web de tipo SPA por los beneficios que ofrece el uso de AJAX en cuanto a optimización de peticiones y disminución de los refrescos de página, ofreciendo así una mejor experiencia de usuario.

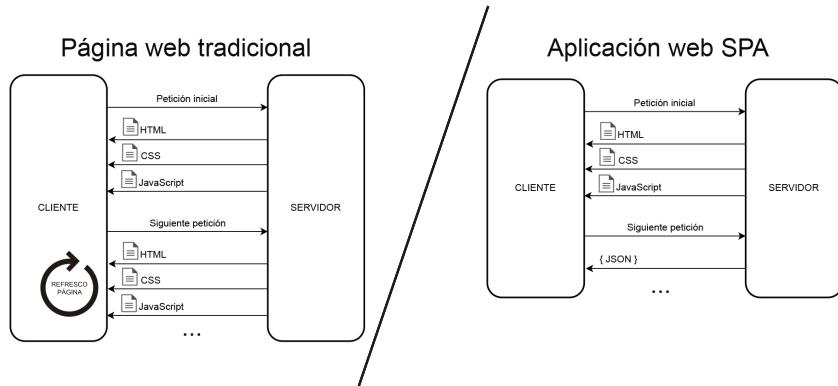


Figura 5.9: Página web tradicional vs Aplicación SPA [28].

Entornos: producción y desarrollo

En la sección del ciclo de vida del proyecto se ha descrito el uso de diferentes ramas Git durante el desarrollo. Esto se traduce en la utilización de dos entornos diferentes e independientes donde se despliega la aplicación:

- **Entorno de producción:** en este entorno se despliegan los cambios de la rama *master*, es decir, la aplicación en un estado estable.
- **Entorno de desarrollo:** en este entorno se despliegan los cambios de la rama *dev*, es decir, la aplicación puede tener un comportamiento inestable.

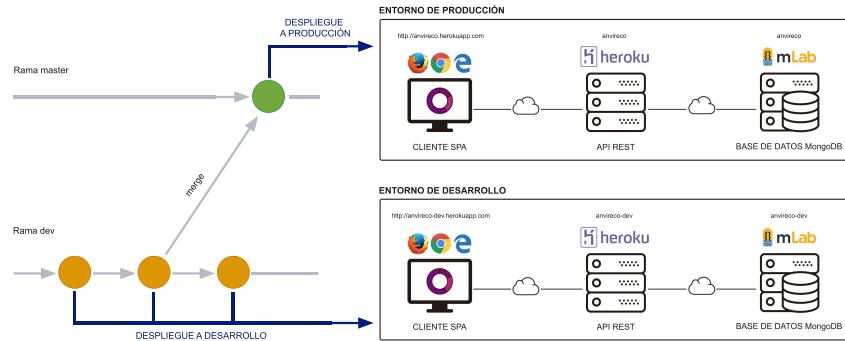


Figura 5.10: Entornos de producción y desarrollo del proyecto.

Travis y la integración continua

Mediante el uso de Travis CI, el despliegue de los cambios incluidos en las ramas *dev* y *master* se realiza de forma automática en su entorno correspondiente.

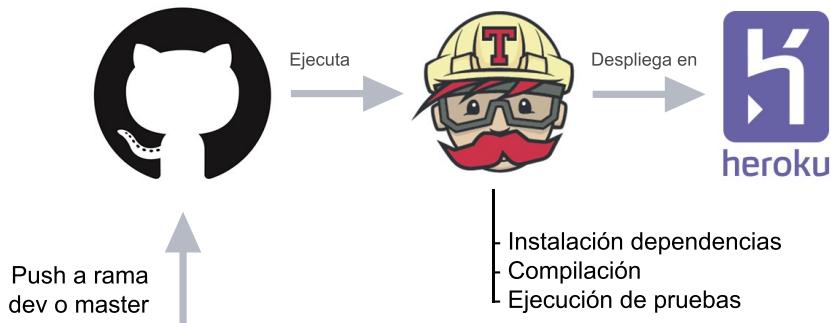


Figura 5.11: Proceso de integración continua con Travis CI.

En nuestro caso concreto, Travis está configurado para realizar una serie de comprobaciones previas al despliegue en Heroku:

1. Descarga e instalación de dependencias (`npm install`).
2. Compilación de código TypeScript a JavaScript (`gulp compile`).
3. Ejecución de pruebas unitarias (`npm test`).

Si alguno de estos pasos no se completa correctamente, el despliegue se cancela. En este caso se dice que la construcción con Travis ha fallado, y se muestra como tal en GitHub.

Otro estado posible es un error debido a que en ese momento Travis no funcionaba correctamente.

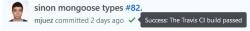
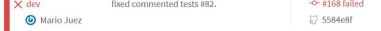
Estado de la construcción en GitHub	Estado de la construcción en Travis	
		Construcción Correcta
		Construcción Fallida
		Error de Travis

Figura 5.12: Posibles estados de Travis CI

Variables de entorno frente a fichero de configuración

El funcionamiento de este proyecto requiere de una serie de valores de configuración:

- Cadena de conexión a la base de datos.
- Nombre de la aplicación *GitHub Developer Application*.
- Identificador de la aplicación *GitHub Developer Application*.
- Clave secreta de la aplicación *GitHub Developer Application*.
- Número de puerto (opcional, por defecto 3000).

Una posible solución para fijar el valor de las diferentes variables de configuración es mediante un fichero de configuración. Esta estrategia es común en el desarrollo de aplicaciones ASP.NET, las cuales cuentan con un fichero `web.config` para tal propósito.

En nuestro caso concreto, esta estrategia no es útil por dos motivos principales:

- Tenemos dos entornos, cuyas variables de configuración deben tomar valores diferentes. En el proceso de integración continua se desplegaría el mismo fichero de configuración en ambos entornos.
- Las variables de configuración contienen información sensible como las credenciales de acceso a la base de datos. Para el proceso de integración continua es necesario que el fichero de configuración se encuentre publicado en el repositorio, lo que significa que la información sensible sería también pública.

El uso de variables de entorno es la alternativa al uso de un fichero de configuración. En cada entorno concreto se deben definir los valores de las variables, y dichos valores de las variables de entorno no son públicos, sino que únicamente son accesibles desde dentro del entorno. Ésto soluciona ambos problemas descritos anteriormente.

La desventaja principal del uso de variables de entorno es que el proceso de declaración de las mismas es diferente en cada sistema operativo (Windows, Linux, Mac).

Revisiones automáticas con Codebeat

Como medio para desarrollar un código con buena calidad y menos defectos se optó por el uso de un sistema de revisiones automáticas como Codebeat.

Este tipo de sistemas evalúan la calidad del código en base a ciertas métricas que penalizan, entre otras cosas, duplicidad de código, clases y métodos de gran tamaño, etc.

Codebeat evalúa la calidad del código de la rama master, y de los cambios que se están intentando introducir a dicha rama desde una pull request. En este proyecto, la calidad del código se ha evaluado de forma automática en el cierre de cada sprint.

Actualmente la herramienta desarrollada tiene una calidad de 3.21 GPA (sobre un máximo de 4), lo que supone una calificación B (donde A es la mejor y F la peor).



Figura 5.13: Evolución del GPA de la última semana.

En la ilustración anterior, además de la evolución del valor de GPA, también se puede observar un indicador de la cantidad de ficheros sin defectos (verde), con defectos leves (naranja), y con defectos graves (rojo).

5.3. Obteniendo datos de GitHub

Una de las dos funcionalidades principales de la herramienta desarrollada es la obtención de datos sobre revisiones de código realizadas en GitHub.

Para ello se ha hecho uso de la API pública de GitHub. Dicha API ofrece los datos de la plataforma en formato JSON.

¿Qué datos son necesarios?

Un repositorio de GitHub está formado por una serie de entidades, de las cuales solo algunas de ellas son de nuestro interés:

Tipo de entidad	Necesaria
<i>Gists</i>	No
<i>Gist Comments</i>	No
<i>Blobs</i>	No
<i>Commits</i>	No
<i>References</i>	No
<i>Tags</i>	No
<i>Trees</i>	No
<i>Issues</i>	No
<i>Issue Comments</i>	No
<i>Issue Events</i>	No
<i>Labels</i>	No
<i>Milestones</i>	No
<i>Organizations</i>	No
<i>Projects</i>	No
<i>Pull Requests</i>	Sí
<i>Reviews</i>	Sí
<i>Review Comments</i>	Sí
<i>Repositories</i>	Sí
<i>Branches</i>	No
<i>Commits</i>	No
<i>Releases</i>	No
<i>Users</i>	Sí

Tabla 5.1: Tipos de entidades de GitHub

Entidad *Pull Request*

Es la entidad de mayor importancia para nuestra herramienta. Las revisiones se realizan dentro de las *pull requests*.

La obtención se realiza en dos fases:

- Una fase inicial a través de la cual se recuperan páginas de 100 entidades *pull request* parciales (no contienen todos los datos).

- Otra fase en la cual se obtienen las *pull request* completas una a una a partir de las parciales.

Cierre de Sprint 5 / Prueba codebeat #28



Figura 5.14: Ejemplo de *pull request*.

Son necesarias $\lceil \frac{n}{100} \rceil + n$ peticiones, donde n es el número de *pull requests* del repositorio.

Entidad *Review*

Una *pull request*, entre otros elementos, puede contener entidades de tipo *Review*. Este tipo de entidad contiene un comentario general sobre los cambios realizados, y un estado de revisión que puede ser:

- **Approved:** los cambios son aceptados por el revisor.
- **Commented:** el revisor hace un comentario sobre los cambios, pero ni los acepta ni los rechaza.
- **Changes requested:** el revisor rechaza los cambios y hace una petición de modificaciones que se deben realizar para que los cambios sean aceptados.
- **Dismissed:** es un estado especial, únicamente se puede fijar a través de la API. Descarta una revisión.

La obtención se realiza por cada *pull request*, en páginas de 100 revisiones.

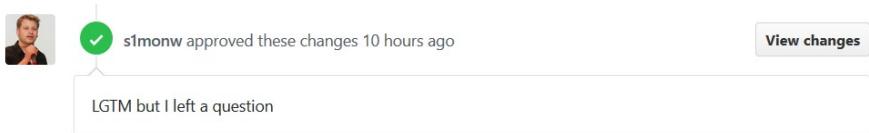


Figura 5.15: Ejemplo de revisión aceptada.

Son necesarias $\sum_{i=1}^n \lceil \frac{r_i}{100} \rceil$ peticiones, donde n es el número de *pull requests* del repositorio, i es la *pull request* actual, y r es el número de revisiones de dicha *pull request*.

Entidad *Review Comment*

Una revisión puede contener entidades de tipo *Review Comment*. Este tipo de entidad tiene como finalidad hacer un comentario en una parte específica de los cambios realizados, por ejemplo una parte de código de un fichero.

Es posible obtener todos los comentarios de revisión de un repositorio en páginas de 100 elementos.

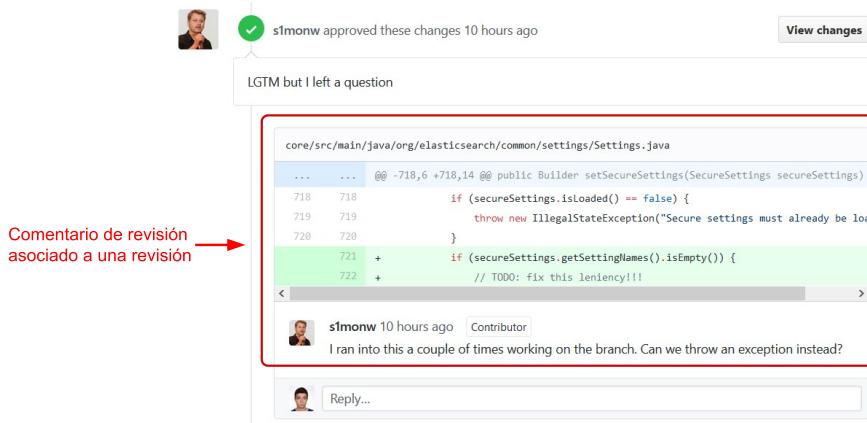


Figura 5.16: Ejemplo de comentario de revisión.

Son necesarias $\lceil \frac{x}{100} \rceil$ donde x es el número de comentarios de revisión que se han hecho en un repositorio.

Entidad *User*

Tanto *pull requests*, revisiones y comentarios de revisión, tienen asociada una entidad de tipo *User*. Este tipo de entidad es de nuestro interés por que un usuario que realiza una revisión es un revisor.

Por cada *pull request*, revisión y comentario de revisión se obtiene un usuario asociado.

Son necesarias tantas peticiones como usuarios diferentes tengan asociados los tres tipos de entidades.

Entidad *Repository*

Una entidad de tipo *Repository* contiene una serie de entidades *pull request*. Un repositorio incluye una serie de datos estadísticos útiles para la posterior visualización de los datos.

Se necesita una petición.

El problema del límite de 5000 peticiones por hora

La API REST de GitHub tiene una limitación de 5000 peticiones por hora, lo cual supone un problema para nuestra herramienta, ya que repositorios de gran tamaño precisan de un número de peticiones mucho mayor.

Una vez superado el límite de peticiones, la API responde con un código de error 403, e información acerca de cuando se reinicia el número de peticiones disponibles.

```
1  HTTP/1.1 403 Forbidden
2  Date: Sat, 24 Aug 2017 14:50:41 GMT
3  Status: 403 Forbidden
4  X-RateLimit-Limit: 5000
5  X-RateLimit-Remaining: 0
6  X-RateLimit-Reset: 1377013266
7
8  {
9      "message": "API rate limit exceeded for... ",
10     "documentation_url": "https://developer.github.com..."
11 }
```

Figura 5.17: Error 403 API GitHub.

Algoritmo de gestión de tareas

Como solución al problema descrito en el punto anterior, la obtención de datos se ha modelado como un problema de gestión de tareas tolerante a fallos.

La idea principal de nuestro gestor de tareas es que sea capaz de pausar su ejecución cuando no sea posible obtener datos, y reanudarla cuando sí sea posible. Si la aplicación se detiene, también debe ser capaz de reanudar el proceso cuando ésta se vuelva a ejecutar.

Por ello las tareas son persistentes, y además tienen estado. Como se ha descrito anteriormente, existen diferentes tipos de entidades a obtener desde GitHub, por tanto nuestro algoritmo trabaja con diferentes tipos de tareas.

Una tarea tiene la siguiente estructura (compartida por todos los tipos de tarea):

- **Tipo:** tipo de tarea, depende del tipo de datos que se obtienen.
- **Repositorio:** el repositorio del que obtener datos sobre revisiones. Está definido por el par *login* del propietario, nombre del repositorio.

- **Completada:** bandera que indica si la tarea está completada.
- **Fecha de creación:** fecha de creación de la tarea. Es necesaria en la política de planificación.
- **Fecha de inicio:** fecha en la cual la tarea comienza a ejecutarse.
- **Fecha de fin:** fecha en la cual la tarea termina su ejecución.
- **Página actual:** algunos datos se obtienen en páginas de 100 elementos. Esta propiedad permite conocer qué página se debe obtener cuando la tarea se reanuda.
- **Última entidad procesada:** algunas tareas iteran sobre una lista de entidades (por ejemplo *pull requests*). Esta propiedad permite saber cual fue la última entidad procesada para reanudarla a partir de ahí si fuese necesario.

Tipos de tareas

La herramienta cuenta con un total de ocho tipos de tareas diferentes, una tarea principal y siete derivadas. A continuación se enumeran en orden de ejecución:

1. **Principal:** obtención de todas las *pull requests* del repositorio con paginación de 100 entidades por petición. Las *pull request* obtenidas no contienen el conjunto de datos completo.
2. **Pull Requests:** itera sobre todas las *pull requests* obtenidas en la tarea anterior y las solicita una a una para obtener los datos restantes.
3. **Revisiones:** itera sobre todas las *pull requests* obtenidas y solicita sus revisiones por páginas de 100 elementos por petición.
4. **Comentarios de revisión:** obtención de todos los comentarios de revisión del repositorio con paginación de 100 entidades por petición.
5. **Usuarios de pull request:** itera sobre todas las *pull requests* obtenidas y solicita los usuarios creadores, de la base y de la cabeza de las mismas. Si un usuario se ha solicitado durante la ejecución de la misma tarea principal no se vuelve a solicitar.
6. **Usuarios de revisión:** itera sobre todas las revisiones obtenidas y solicita los usuarios creadores de las mismas. Si un usuario se ha solicitado durante la ejecución de la misma tarea principal no se vuelve a solicitar.
7. **Usuarios de comentario de revisión:** itera sobre todos los comentarios de revisión obtenidos y solicita los usuarios creadores de los mismos. Si un usuario se ha solicitado durante la ejecución de la misma tarea principal no se vuelve a solicitar.
8. **Repositorio:** el último tipo de tarea sirve para obtener los datos del repositorio.

No es posible crear un subconjunto de las tareas descritas anteriormente. Al crear una tarea de tipo principal se crean todas las subtareas correspondientes.

Política de planificación: FIFO

La política de planificación escogida para el gestor de tareas es la política FIFO (*First Input First Output*), es decir, la tarea más prioritaria es la más antigua no completada.



Figura 5.18: Ejemplo de cola de tareas FIFO.

Posibles errores y modo de tratarlos

Durante el proceso de obtención de datos pueden surgir dos tipos de error:

- **Error de API:** error al obtener datos desde la API de GitHub, por ejemplo cuando se excede el límite de peticiones.
- **Error de base de datos:** error al almacenar los datos en la base de datos, por ejemplo si el servicio, en nuestro caso mLab, no se encuentra disponible en ese momento.

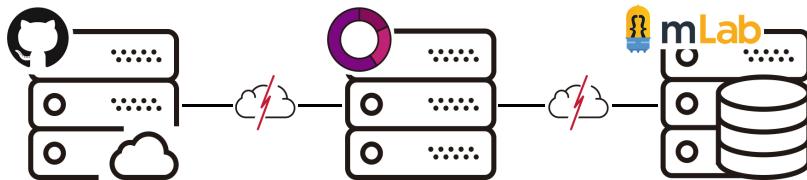


Figura 5.19: Posibles errores en la obtención de datos.

Se ha elegido una estrategia de reintento para el tratamiento de los errores, en la cual, se pausa la ejecución de tareas durante un tiempo y tras ello se reanuda el proceso. Dependiendo del tipo de error, el tiempo de espera será diferente.

- **Error 403 API:** indica que se ha excedido el límite de peticiones por hora a la API de GitHub. La cabecera del error contiene una marca de tiempo que indica en qué momento se pueden volver a realizar peticiones. El gestor de tareas pausará su ejecución hasta esa marca de tiempo.

- **Otros errores de API:** para el resto de errores de la API de GitHub, el gestor de tareas pausa su ejecución durante un minuto.
- **Error de base de datos:** para cualquier error de conexión a la base de datos, el gestor de tareas pausa su ejecución durante un minuto.

```
1  { [Error: {"message": "API rate limit exceeded..."}]
2    code: 403,
3    status: 'Forbidden',
4    headers:
5      { server: 'GitHub.com',
6        date: 'Tue, 20 Jun 2017 02:50:02 GMT',
7        'content-type': 'application/json; charset=utf-8',
8        status: '403 Forbidden',
9        'x-ratelimit-limit': '5000',
10       'x-ratelimit-remaining': '0',
11       'x-ratelimit-reset': '1497928505', } }
12 Going to retry on: Tue Jun 20 2017 05:15:05 GMT+0200 (CEST)
13 [Tue Jun 20 2017 05:15:05 GMT+0200 (CEST)] - Continuing...
```

Figura 5.20: Ejemplo de reintento tras error 403.

De forma adicional, antes de crear una tarea, se comprueba si el repositorio existe. Esta comprobación no se realiza a través de la API por que podría encontrarse bloqueada por haber superado el límite de peticiones. La comprobación se hace mediante una petición HTTP a la página HTML del repositorio. El contenido HTML de la página de un repositorio contiene ciertas meta etiquetas como por ejemplo "description" que no existe en el contenido HTML de la página de error 404, de ese modo se verifica la existencia de un repositorio.

Un ejemplo del algoritmo en funcionamiento

A continuación se describe un ejemplo del funcionamiento del algoritmo donde se puede observar cuánto tiempo lleva el proceso de obtención de datos.

Se han encolado los siguientes repositorios:

Repositorio	Pull Requests	Revisiones	C. revisión
elastic/elasticsearch	11698	8410	45274
pallets/flask	1139	243	722
Leaflet/Leaflet	2198	241	615
playframework/playframework	4863	1567	5798
google/WebFundamentals	2376	2141	3500

Tabla 5.2: Repositorios en colas (20-06-2017).

A continuación se muestra un gráfico que ilustra la duración de la ejecución de las tareas y subtareas. Cada repositorio tiene 8 divisiones que se corresponden con los 8 tipos de tarea descritos anteriormente (en el mismo orden):

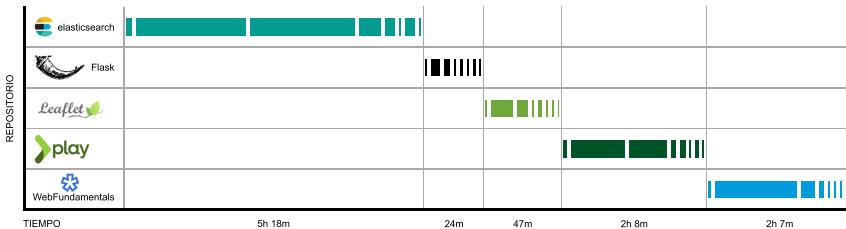


Figura 5.21: Planificación de la cola de tareas.

Interfaz para la creación de tareas

Se pueden crear tareas a través de la API, o desde el cliente web. La interfaz del cliente web es la siguiente:

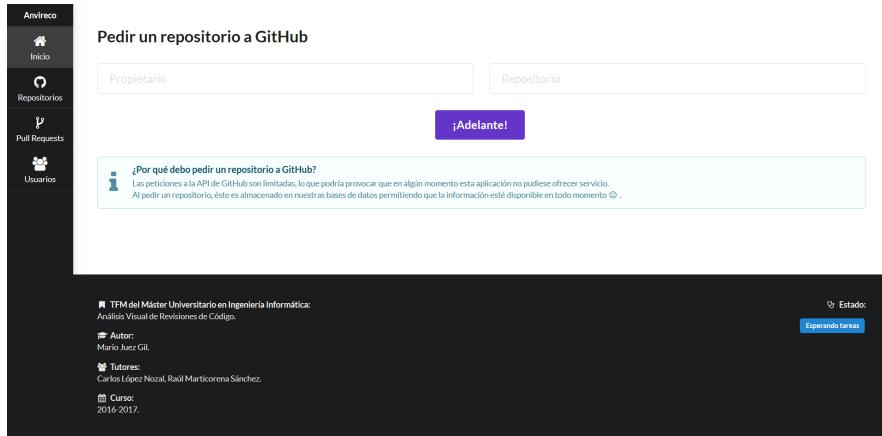


Figura 5.22: Interfaz web para crear tareas.

El formulario de la parte central cuenta con dos campos (propietario y repositorio), si se introduce un par de valores válidos, se encolarán las 8 tareas necesarias para obtener datos de dicho repositorio.

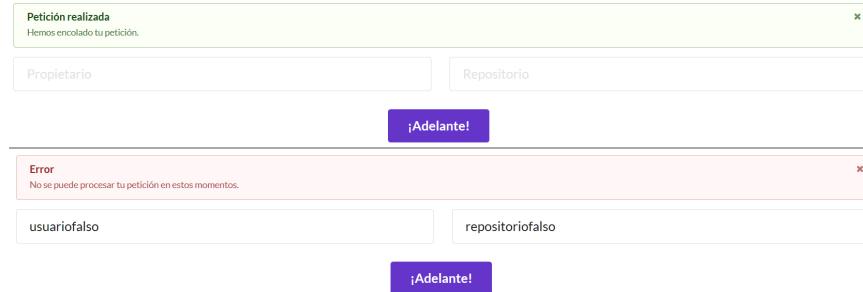


Figura 5.23: Mensajes de información sobre la correcta o incorrecta creación de tareas.

Estado del gestor de tareas

Una vez encolada una petición para obtener todos los datos de un repositorio, no es posible saber en qué momento va a finalizar. Sin embargo, en el cliente web se muestra información sobre el estado del gestor de tareas. Tiene tres posibles estados:

- **Esperando tareas:** actualmente no hay ninguna tarea en la cola pendiente de ser ejecutada.
- **Obteniendo datos:** se está ejecutando una tarea de obtención de datos de GitHub. Actualmente se están realizando peticiones a su API.

- **Error API GitHub:** hay tareas pendientes pero su ejecución está pausada debido a un error de la API de GitHub, posiblemente por exceder el límite de peticiones.

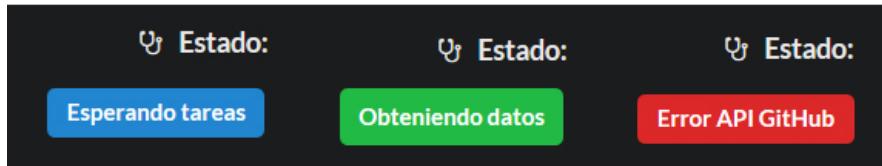


Figura 5.24: Los tres posibles estados del gestor de tareas.

5.4. Visualización de los datos

La segunda funcionalidad principal de la herramienta es la visualización de los datos obtenidos a través de gráficas.

La información se muestra en tres niveles:

- A nivel de repositorio.
- A nivel de *pull request*.
- A nivel de usuario.

Se han utilizado tres tipos de gráficos:

- **Gráfico de barras:** La finalidad de este tipo de gráfico es comparar una estadística concreta de un repositorio, *pull request* o usuario con el valor medio de todas las entidades almacenadas de ese tipo.
- **Gráfico donut:** La finalidad de este tipo de gráfico es comparar el porcentaje de ciertas entidades como *pull request* o revisiones dependiendo de su tipo.
- **Gráfico de área temporal:** La finalidad de este tipo de gráfico es mostrar el número de entidades creadas en un periodo concreto. El gráfico se divide en un máximo de 20 periodos.

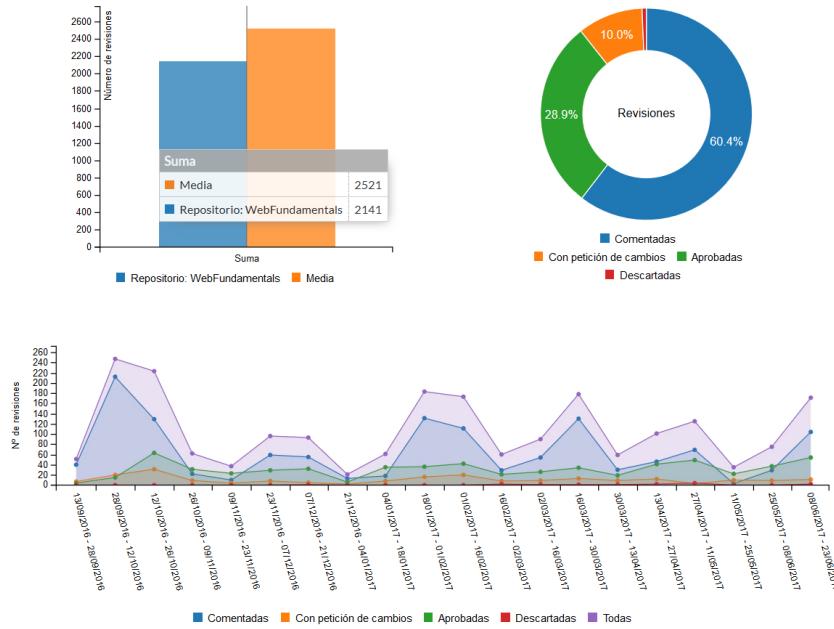


Figura 5.25: Ejemplos de los tres tipos de gráfico.

5.5. Otros aspectos

A continuación se describen otros aspectos relevantes como la gestión de memoria o las pruebas.

Gestión de memoria: paginando resultados

Durante el desarrollo del proyecto hemos tenido problemas de rendimiento relacionados con la memoria y el uso de la red debido al elevado número de entidades que se retornaban al solicitar un listado.

Por ejemplo, al solicitar todas las *pull requests* del repositorio Elastic-search, la lista resultante contiene cerca de 12.000 entidades, las cuales tienen un tamaño aproximado de 35 MB. Mantener *arrays* con esa cantidad de datos en memoria supone un problema en entornos donde los recursos son limitados (Heroku ofrece 500 MB). Asimismo, enviar esa cantidad de datos al cliente en una sola petición penaliza gravemente los tiempos de carga.

Basándonos en el funcionamiento de la API de GitHub, se implementaron consultas a la base de datos con paginación de resultados, obteniendo páginas de un máximo 100 entidades.

Testing y la inyección de dependencias

Las diferentes clases de la aplicación hacen uso del patrón de inyección de dependencias. Gracias al uso de este patrón y al uso de interfaces se reduce el acoplamiento, lo cual resulta muy útil para realizar pruebas unitarias con *stubs* y *mocks* (dependencias simuladas).

Esta estrategia permite probar el funcionamiento de un subsistema independientemente de sus dependencias externas.

Trabajos relacionados

En este apartado se exponen tres herramientas relacionadas con la desarrollada en este proyecto, una dedicada a la obtención de datos y dos al análisis.

6.1. GHTorrent

GHTorrent [20] es una herramienta que obtiene datos desde la API de GitHub y los almacena en dos bases de datos:

- MongoDB no relacional, guarda los datos en bruto.
- Base de datos relacional con entidades enlazadas.

GHTorrent es utilizado por investigadores y empresas, como fuente procesos software y analíticas de productos.

A continuación se muestra una tabla comparativa entre nuestra herramienta y GHTorrent:

Característica	Anvireco	GHTorrent
Obtención de entidades de GitHub	✓	✓
API REST	✓	✗
Cliente web	✓	✗
Ánalisis visual	✓	✗
Base de datos NoSQL	✓	✓
Base de datos SQL	✗	✓
Ejecución distribuida	✗	✓

Tabla 6.3: Comparativa de las características de Anvireco y GHTorrent.

6.2. ReDA - Review Data Analyzer

ReDA [27] es una aplicación web para la visualización del proceso de revisiones en software de código libre. Para la creación de gráficos hace uso de la librería JavaScript D3.js.

Actualmente permite visualizar el histórico de revisiones del repositorio Android Open Source Project basado en Gerrit.

A continuación se muestra una tabla comparativa entre nuestra herramienta y ReDA:

Característica	Anvireco	ReDA
Trabaja con datos de GitHub	✓	✗
Trabaja con datos de Gerrit	✗	✓
Múltiples repositorios	✓	✗
Permite solicitar repositorios	✓	✗
API REST	✓	✗
Descarga de datos en CSV	✓	✓
Gráficos sobre revisiones	✓	✓
Gráficos sobre usuarios	✓	✓

Tabla 6.4: Comparativa de las características de Anvireco y ReDA.

6.3. GiLA - GitHub Label Analyzer

GiLA [21] es una herramienta de visualización que permite el análisis del uso de etiquetas (frecuencia, relaciones, etc) en un proyecto de GitHub.

También permite analizar el modo en que los usuarios utilizan las etiquetas.

A continuación se muestra una tabla comparativa entre nuestra herramienta y GiLA:

Característica	Anvireco	GiLA
Trabaja con datos de GitHub	✓	✓
Trabaja con datos de Gerrit	✗	✗
Múltiples repositorios	✓	✓
Permite solicitar repositorios	✓	✗
API REST	✓	✗
Descarga de datos en CSV	✓	✗
Gráficos sobre revisiones	✓	✗
Gráficos sobre usuarios	✓	✓

Tabla 6.5: Comparativa de las características de Anvireco y GiLA.

Conclusiones y Líneas de trabajo futuras

En esta parte de la memoria se exponen las conclusiones extraídas del desarrollo del proyecto y de las posibles líneas de trabajo futuras.

7.1. Conclusiones

Tras el desarrollo del proyecto se han obtenido las siguientes conclusiones:

- El estudio del proceso de revisión en el desarrollo software nos ha permitido observar que se trata de una técnica cada vez más extendida, sobre todo en grandes proyectos, lo cual es indicativo de su positivo impacto en aspectos como la calidad del código.
- Al inicio del desarrollo de la herramienta, el sistema de revisiones de código en GitHub se encontraba en fase de pruebas. Durante el desarrollo hemos podido ver cómo ha evolucionado y madurado añadiendo nuevas características. Este entorno cambiante ha supuesto una adaptación continua de nuestra herramienta a las funcionalidades de GitHub, pero también nos ha permitido ser uno de los primeros proyectos en trabajar con ellas.
- La decisión de utilizar TypeScript tiene una repercusión positiva en el código desarrollado. Los lenguajes tipados permiten detectar errores rápidamente, y mejorar la legibilidad y mantenibilidad. La principal desventaja es que requiere ser compilado a JavaScript para su ejecución, lo cual se traduce en un mayor esfuerzo en la configuración inicial, y una menor flexibilidad en tareas como la depuración de código.
- El uso de un tipo de base de datos NoSQL como MongoDB que almacena documentos JSON ha resultado muy positivo debido a que los datos son almacenados de forma persistente tal como los obtenemos desde la API de GitHub, sin la necesidad de hacer grandes transformaciones que serían necesarias para guardarlos en una base de datos relacional.

- La práctica de técnicas de integración continua, aunque precisan de una configuración inicial que puede resultar algo compleja, permiten automatizar procesos repetitivos como la ejecución de pruebas o el despliegue de la aplicación, lo que supone un ahorro de tiempo a medio y largo plazo. También asegura que la versión desplegada pasa todas las pruebas.
- Este proyecto tiene un fuerte componente tecnológico. Varias de las tecnologías utilizadas han sido nuevas para nosotros. Gracias a los conocimientos generales obtenidos durante el grado y el máster, el esfuerzo necesario para el aprendizaje de las mismas, no ha sido muy elevado, y nos ha permitido adaptarnos con cierta facilidad.

7.2. Líneas de trabajo futuras

Actualmente, nuestra herramienta permite obtener, visualizar y analizar datos relacionados con revisiones de código realizadas en GitHub. La herramienta tiene diversas posibilidades de ser mejorada en varias líneas:

- En la fase final del desarrollo, GitHub liberó la cuarta versión de su API basada en GraphQL [15] en lugar de REST. Esta API es más flexible y puede ser interesante su uso en futuras versiones de la herramienta.
- La herramienta GHTorrent permite obtener datos de GitHub de forma distribuida. Una posible mejora para nuestra herramienta pasa por adaptar nuestro algoritmo de gestión de tareas para trabajar de forma distribuida para disminuir el tiempo de obtención de datos.
- Nuestro algoritmo de gestión de tareas hace uso de una política FIFO, se podría valorar el uso de otro tipo de políticas como por ejemplo Round Robin.
- Actualmente trabajamos únicamente con datos de revisiones realizadas en GitHub, pero existen alternativas como Gerrit Code Review que también son ampliamente utilizadas. Una mejora de la herramienta pasa por implementar funcionalidades que permitan obtener y visualizar datos de Gerrit Code Review.
- Aunque se han desarrollado pruebas unitarias, el conjunto de las mismas no cubre todo el código desarrollado. Se puede continuar trabajando en el desarrollo de nuevas pruebas unitarias, así como en pruebas automáticas de interfaz o de estrés.
- Nuestra herramienta expone una API REST para acceder a los datos obtenidos, actualmente contamos con un cliente web. En el futuro se podrían implementar nuevos clientes, por ejemplo aplicaciones móviles.

- Actualmente el cliente web únicamente está disponible en castellano. Sería interesante la implementación de soporte multi-lenguaje.
- Existen técnicas de representación de datos como por ejemplo las nubes de palabras cuya implementación en el cliente puede ser interesante para visualizar los términos más utilizados en revisiones y comentarios de revisión.

Bibliografía

- [1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, May 1989.
- [2] Agile Alliance. Agile glossary and terminology. <https://www.agilealliance.org/agile101/agile-glossary/>. [Internet; accedido 2-abril-2017].
- [3] Agile Alliance. Agile manifesto. <https://www.agilealliance.org/agile101/the-agile-manifesto/>. [Internet; accedido 4-abril-2017].
- [4] Android. Home page. https://www.android.com/intl/es_es/. [Internet; accedido 20-junio-2017].
- [5] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.
- [6] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.
- [7] Charles Duan. Understanding Git: merging. <https://www.sbf5.com/~cduan/technical/git/git-3.shtml>. [Internet; accedido 14-marzo-2017].
- [8] Eclipse. Home page. <https://eclipse.org/>. [Internet; accedido 20-junio-2017].
- [9] Elastic. Elasticsearch. <https://www.elastic.co/products/elasticsearch>. [Internet; accedido 20-junio-2017].
- [10] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [11] Michael E. Fagan. Advances in software inspections. *IEEE Trans. Softw. Eng.*, 12(1):744–751, January 1986.

- [12] Roy T Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.
- [13] Linux Foundation. Home page. <https://www.linuxfoundation.org/>. [Internet; accedido 20-junio-2017].
- [14] Martin Fowler and Matthew Foemmel. Continuous integration. *ThoughtWorks* [http://www.thoughtworks.com/Continuous Integration.pdf](http://www.thoughtworks.com/Continuous%20Integration.pdf), page 122, 2006.
- [15] GitHub. Ggithub graphql api. <https://developer.github.com/v4/>. [Internet; accedido 26-junio-2017].
- [16] Github. Github features. <https://github.com/features>. [Internet; accedido 15-marzo-2017].
- [17] Github. Home page. <https://github.com/>. [Internet; accedido 20-junio-2017].
- [18] Github. Mastering issues. <https://guides.github.com/features/issues>. [Internet; accedido 2-abril-2017].
- [19] Google. Webfundamentals. <https://developers.google.com/web/fundamentals/?hl=es>. [Internet; accedido 20-junio-2017].
- [20] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.
- [21] SOM Research Lab. Gila - github label analyzer. <http://som-research.github.io/gila/>. [Internet; accedido 25-junio-2017].
- [22] Asier Marques. Conceptos sobre apis rest. <http://asiermarques.com/2013/conceptos-sobre-apis-rest/>. [Internet; accedido 21-junio-2017].
- [23] J. Palacio and C. Ruata. Scrum manager. <https://http://www.scrummanager.net/>. [Internet; accedido 4-abril-2017].
- [24] C Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2 edition, 2008.
- [25] Gerrit Code Review. Home page. <https://www.gerritcodereview.com/>. [Internet; accedido 20-junio-2017].
- [26] Code School. Single-page applications. <https://www.codeschool.com/beginners-guide-to-web-development/single-page-applications>. [Internet; accedido 22-junio-2017].

- [27] Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. Reda: A web-based visualization tool for analyzing modern code review dataset. In *The Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014.
- [28] Mike Wasson. Asp.net - single-page applications: Build modern, responsive web apps with asp.net. <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>. [Internet; accedido 23-junio-2017].
- [29] Wikipedia. Revisión automática de código — wikipedia, la enciclopedia libre, 2014. [Internet; accedido 9-marzo-2017].
- [30] Wikipedia. Control de versiones — wikipedia, la enciclopedia libre, 2017. [Internet; accedido 10-marzo-2017].