Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Título del trabajo

Realizado por
Javier García Aguilar

Para la obtención del título de
Grado en Ingeniería Informática - Ingeniería del Software

Dirigido por
José Antonio Parejo Maestre
Antonio Ruiz Cortés

En el departamento de
Lenguajes y Sistemas Informáticos

Convocatoria de junio, curso 2023/24

Aquí la dedicatoria del trabajo

Agradecimientos

Quiero agradecer a X por...

También quiero agradecer a Y por...

Resumen

Incluya aquí un resumen de los aspectos generales de su trabajo, en español.

Palabras clave: Palabra clave 1, palabra clave 2, ..., palabra clave N

Abstract

This section should contain an English version of the Spanish abstract.

Keywords: Keyword 1, keyword 2, ..., keyword N

Índice general

Índice de figuras

Índice de tablas

Índice de extractos de código

1. Introducción

1.1. Presentación del problema

Las API (Application Programming Interface) es una pieza de código que permite a diferentes aplicaciones comunicarse entre sí y compartir información y funcionalidades. Actúa como un intermediario entre sistemas, permitiendo la comunicación y el intercambio de datos y acciones específicas[5]. En el contexto web, estas interfaces procesan solicitudes HTTP y suelen devolver una respuesta en formato JSON, aunque su implementación puede variar. En la figura 1.1, podemos ver un ejemplo sencillo de una reserva y como pasa por las diferentes capas hasta devolver una respuesta.

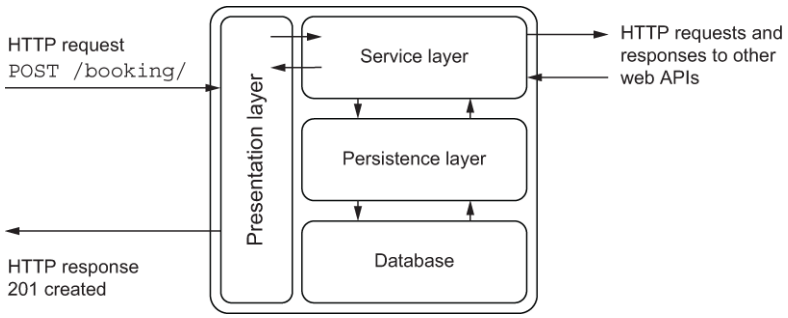

Figura 1.1: Este modelo visual representa una API web, sus componentes y su funcionamiento (Fuente: 1)

A continuación, exploraremos la evolución del mercado de las API desde el año 2000 hasta el 2019, dividiendo en tres intervalos temporales: 2000-2008, 2008-2010, y 2010-2019. Utilizaremos figuras (1.2, 1.3) visuales para respaldar nuestro análisis, detallando el crecimiento y la relevancia de las API en cada periodo.

Durante el primer intervalo, que abarca desde el año 2000 hasta el 2008, se observa un crecimiento gradual pero modesto en la adopción de API, con apenas mil interfaces registradas en el mercado.

El periodo entre los años 2008 y 2010 marca un cambio significativo, con un aumento notable en la cantidad de API, que alcanza las dos mil interfaces, demostrando un interés creciente y una mayor adopción en el ámbito web.

Finalmente, en el intervalo que abarca desde el 2010 hasta el 2019, presenciamos un aumento exponencial en el número de API registradas, llegando a la impresionante cifra de 22000 interfaces, lo que refleja su consolidación como una parte fundamental del ecosistema digital.

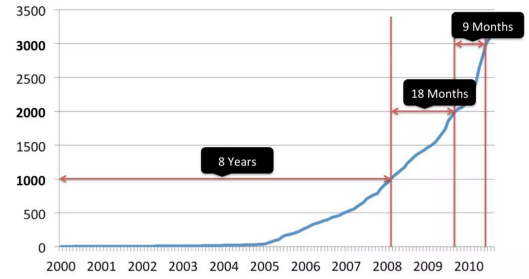

Figura 1.2: Estado del mercado de APIs hasta el 2010 (Fuente: 2)

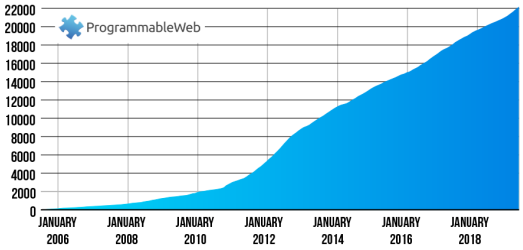

Figura 1.3: Representa el crecimiento de directorios API en ProgrammableWeb (Fuente: 3)

El notable crecimiento del mercado de las API, como se ha ilustrado en las figuras 1.2 y 1.3, refleja su creciente relevancia y adopción en el ámbito digital. Sin embargo, este aumento en la disponibilidad y variedad de API, también ha planteado nuevos desafíos, especialmente en lo que respecta a garantizar su funcionalidad y fiabilidad. A medida que las API se han vuelto más omnipresentes en el ecosistema digital, la necesidad de un riguroso proceso de testing se ha vuelto aún más crucial. En este sentido, el análisis del mercado de las API no solo nos proporciona una visión del panorama actual, sino que también nos señala la importancia de abordar la calidad y el rendimiento de estas interfaces. A continuación, exploraremos une herramienta de testing de API muy bien conocida, Postman[1].

Postman es una plataforma de construcción y uso de API[6], en la cual puedes crear colecciones donde se organizan solicitudes API agrupadas y donde se ejecutan fácilmente para probar API. Además, Postman tiene una sección de blog donde publica, entre otras cosas, estadísticas anuales sobre sus usuarios y colecciones. Las cuales vamos a ver en la figura 1.4.

La figura 1.4 muestra que hay más de 25 millones de usuarios, aproximadamente 121 millones de colecciones y aproximadamente 1.29 mil millones. Viendo estas cifras se puede apreciar fácilmente qué tan importante es el testing de API.

Ya Alberto Martín López en su artículo *RESTest: automated black-box testing of RESTful web APIs*[7], comentó la importancia de probar a fondo una API RESTful en la integración de software. Un año después realizó un estudio empírico sobre el uso de métodos automatizados de generación de casos de prueba[8], en ese estudio probó 13 API durante 15 días y analizó los errores encontrados.

Tras esos estudios realizados por Alberto Martín, surgió la duda de si sería buena idea establecer un limitador a las llamadas de las API durante el testing, de forma que se evitara el bloqueo temporal o *cooldown*[2], hablando en códigos de estado HTTP suelen ser 429. De esta forma surgió la idea de PlanifyAPI[3], una librería en Python que establece un limitador a las llamadas basándose en el *rate*

[1]https://www.postman.com/
[2]Tiempo de espera hasta que puedas realizar llamadas de nuevo
[3]https://github.com/PlanifyAPI/US-TFG-PlanifyAPI-Python

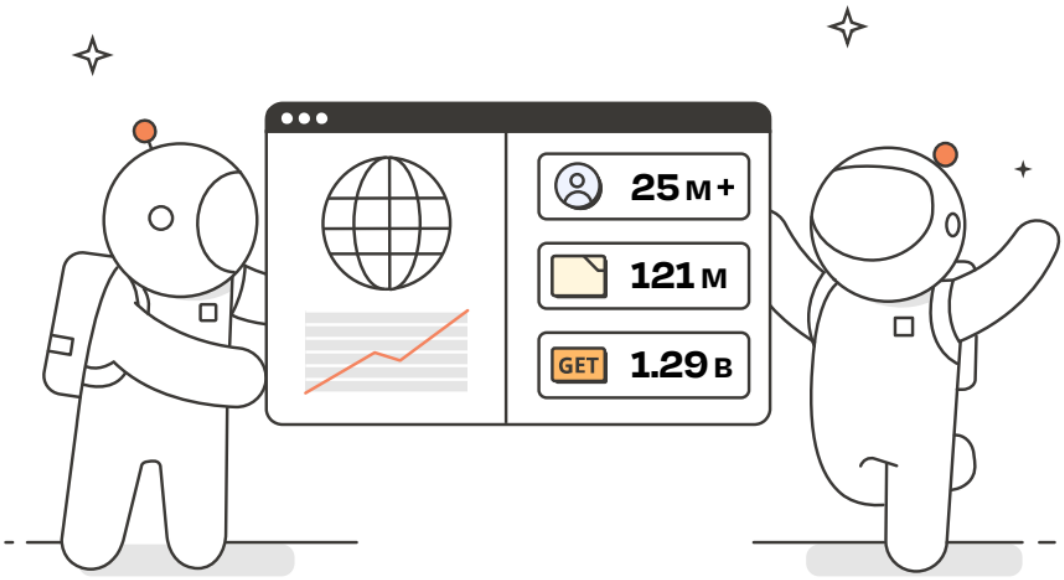

Figura 1.4: Estadísticas de Postman (Fuente: 4)

limit o las cuotas de los planes de precio de las API.

1.2. Motivación

La motivación principal surge del profundo interés del autor por el campo de la investigación, lo que le ha llevado a poner a prueba su pasión en este ámbito. Es así como surge la idea de este trabajo de fin de grado, el cual se inspira en el experimento realizado por Alberto Martín López[8].

Este trabajo de fin de grado se fundamenta en dos aspectos motivacionales: uno de carácter investigador y otro de índole académica. En lo que concierne a la investigación, se ha tratado de recrear el experimento de Alberto Martín López[8] y analizar detenidamente las implicaciones que surgirían al establecer un límite en las llamadas a las API durante las pruebas. Por otro lado, desde una perspectiva académica, el proyecto representa un desafío de considerable envergadura en comparación con las tareas habituales realizadas durante el grado de Ingeniería Informática - Ingeniería del Software. Además, supone la oportunidad de familiarizarse con herramientas utilizadas en entornos profesionales, como GitHub y Grafana, entre otras.

1.3. Objetivos del proyecto

1.3.1. Objetivos técnicos

Teniendo en cuenta la situación del testing actual, surgió la necesidad de probar una herramienta que facilite el control de velocidad de velocidad de llamadas a las API, optimizando así los recursos durante el testing. En este sentido, nuestra librería se presenta como una propuesta innovadora para ayudar a los testers.

Nuestra librería tiene como objetivo principal limitar las llamadas a las API por unidad de tiempo, de forma que no encontremos esos *cooldown* durante el testing.

1.3.2. Objetivos docentes

El proyecto busca desarrollar habilidades y competencias clave de el estudiante. Durante el proceso de desarrollo de la librería, el estudiante aplicará los conocimientos teóricos y prácticos adquiridos durante su formación académica en el grado de Ingeniería Informática - Ingeniería del Software. Además, tendrá la oportunidad de adquirir nuevos conocimientos y habilidades a través de la investigación y el uso de herramientas que no fueron cubiertas en el plan de estudios (por ejemplo, el uso de Grafana y Docker a un nivel más avanzado).

La culminación del proyecto implicará la presentación y defensa del trabajo ante un tribunal académico, lo que permitirá al estudiante mejorar sus habilidades de comunicación, síntesis y argumentación. En el supuesto de una calificación de aprobado, el estudiante podrá procedes a la obtención del título y a su graduación como ingeniero informático de pleno derecho.

En resumen, además de crear una librería útil, el proyecto buscará fomentar el crecimiento académico y profesional de los estudiantes implicados, preparándolos para enfrentar desafíos técnicos y desarrollar habilidades fundamentales para su futura trayectoria laboral.

1.4. Estructura de la memoria

2. Metodología y planificación

2.1. Metodología

2.1.1. Metodología del proyecto

Introducción

La motivación de este documento es detallar, en la medida de lo posible, los procedimientos de trabajo a llevar a cabo a lo largo del desarrollo de este proyecto. La metodología que seguiremos principalmente es *Scrum*, no obstante, se realizan varias modificaciones para adaptarse correctamente al proyecto, además utilizaremos un tablero en Github Project para la organización de tareas.

Eventos

Los eventos son una parte fundamental para el éxito del proyecto, permite a todos los miembros del equipo comunicarse y colaborar para crear regularidad y consistencia. La primera modificación de *Scrum* que encontraremos es en los eventos, ya que solo es un alumno se eliminará las *daily* al no desarrollar solo. Otra modificación es que fusionaremos las reuniones de planificación, revisión y retrospectiva en una sola, realizada el último día del *sprint* y da comienzo al siguiente. Los eventos que llevaremos a cabo a lo largo del *sprint* son:

- **Sprint:** Son las iteraciones que va a tener nuestro proyecto en su desarrollo, que nos va a permitir tener un ritmo de trabajo predefinido. Usualmente tendrán una duración de 3 semanas.
- **Sprint Planning / Review:** Es la primera y última reunión de planificación/revisión de cada sprint, se lleva a cabo al final del sprint y da inicio al siguiente. Durante su desarrollo, se revisan las tareas realizadas y tareas no completadas, posteriormente se establecen cuáles son los objetivos que se deben completar para el siguiente y se replanifican las tareas no acabadas.

Artefactos

Los artefactos son elementos cruciales en *Scrum*, pues se utilizan para planificar, monitorear y contolar el trabajo durante el desarrollo de un proyecto. En nuestra adaptación encontramos los siguientes:

- **Product Backlog:** Es una lista compuesta con todas las tareas a realizar, sean o no de código.

- **Sprint Backlog:** Es una lista con las tareas a realizar durante el *sprint*, es definida durante la reunión inicial del *sprint*.

Equipo de trabajo

El equipo de trabajo se compone por un solo miembro, el autor de esta memoria, y se tratará de autogestionar durante el proceso. Por supuesto, siempre puede contar con cualquiera de los dos tutores, él que esté libre en ese momento, para ayudar a la hora de declinar la balanza con el rumbo de alguna tarea o decisión. Además, con los tutores se realizan las reunión de inicio/fin del *sprint*.

Reuniones

Debido a la dificultad para coordinar las reuniones se decidió eliminar las *daily*, y se unificaron todas las reuniones a una en cada sprint, a menos que se requiera alguna reunión extra requerida por cualquiera de las dos partes. Las reuniones en su defecto serán de forma online, aunque dependiendo de la disponibilidad podrán ser presenciales. En el caso de ser online serán usadas las herramientas que se comentarán a continuación (véase 3.1.3).

2.1.2. Metodología de desarrollo

Introducción

En este apartado se detallarán las diferentes políticas y acciones a aplicar por parte del equipo durante la fase de realización de las tareas-

Definición de hecho

Vamos a contemplar las siguientes definiciones de hecho en función del tipo de tarea:

- **Código:** Una tarea estará finalizada cuando se haya comprobado que funciona correctamente y se haya revisado por alguno de los tutores. Si es necesario y lo indica el revisor, se deberá refactorizar antes de poder marcar la tarea como completada. Además, en caso de aparecer algún conflicto, deberá solucionarlo, volviéndose a revisar que todo lo anterior funciona correctamente. No se debe subir a la rama sin haber comprobado que la modificación no rompa otra funcionalidad.
- **Documentación:** Este tipo de tarea se considerará finalizada cuando se haya validado por uno de los tutores. En caso de que se detecte algún error se exigirá una corrección antes de poder darlla como finalizada.

2.1.3. Estándares de código

Seguiremos el siguiente estándar a la hora de programar:

- Para nombrar variables, atributos, nombre de módulos y métodos o funciones se empleará el estilo *snake_case*
- Se usará el estilo *PascalCase* para nombrar clases.
- Se usará el estilo *SCREAMING_SNAKE_CASE* para las constantes.
- Debido a que Python no puede declarar funciones privadas, se utilizará un carácter _ al principio para indicarlo.
- Usaremos Git como sistema de control de versiones y Github para alojar el repositorio.
- Se utilizarán tabulaciones de 4 espacios.
- Las variables no utilizadas deberán declararse como _.
- Todas las funciones, variables, etc, se realizarán en inglés.

Política de mensajes de commit

Para la política de mensajes de *commit* utilizaremos *conventional commits*[1], sin scope. A continuación se explicarán las reglas a seguir:

1. Los mensajes deben se precedidos por su tipo, dos puntos y un espacio. Los tipos son los siguientes:
   - **feat:** Este tipo indica un incremento o funcionalidad en el repositorio.
   - **docs:** Este tipo indica una tarea referente a la documentación del repositorio.
   - **refactor:** Este tipo indica una tarea de refactorización o mejora de la calidad del software.
   - **test:** Este tipo indica una tarea de pruebas de una funcionalidad.
   - **fix:** Este tipo indica un arreglo de algún error en el código.
   - **conf:** Este tipo indica una tarea de configuración del repositorio.
2. Los mensajes tendrán un título descriptivo en inglés.
3. Opcionalmente y si fuera necesario explicar más información, se utilizará un cuerpo del mensaje. Recomendable utilizar menos de 72 caracteres.
4. No poner punto al final del título,
5. Se recomienda limitar el título a 50 caracteres.

[1] https://www.conventionalcommits.org/en/v1.0.0/

El *commit* resultante debería cumplir la siguiente plantilla:

1. `<tipo>: <título>`
2.
3. `(<cuerpo>)`

Para fusionar las *pull requests* puede utilizarse el mensaje que aparece por defecto (aunque no cumpla con las reglas definidas arriba) puesto que este ya contiene toda la información necesaria. Los *commits* que sirvan para generar una nueva versión pública de la librería debe seguir la estructura vX.Y.Z, siendo X.Y.Z los valores de la versión específica (más información véase 2.1.3).

Estrategia de ramas

La estrategia de ramas que seguiremos para la gestión del código fuente del proyecto consta de una rama principal *main*, y después se crea ramas por *issue*, pudiendo haber una *issue* con varias ramas asociadas. Cuando se considera que la tarea está finalizada y tras ser revisada por uno de los tutores, el autor realizará la fusión de la rama con la principal. Si hubiera que corregir algún error en la rama principal se crearía una *issue* y una rama asociada donde se realizará la corrección.

Hay que señalar que los documentos se realizan y suben en una rama con un título indicando la tarea de documentación que representa.

Ciclo de vida de las issues

Una *issue* podrá ser creada con el motivo de realizar una tarea cualquiera dentro del proyecto. El nombre de la *issue* será determinado por el creador de la misma y no tendrá ningún patrón concreto. Algunas *issues* indican tareas "grandes", estas se llaman épicas, la cual en el primer comentario tendrá todas las tareas desglosadas.

El ciclo de vida de las issues será el siguiente:

1. Se crea la issue en la columna *Backlog* del tablero del proyecto.
2. Se mueve la issue a la columna *Ready* si se va a trabajar en ella durante el *sprint* actual y está lista para llevar a cabo.
3. Una vez se va a comenzar a realizar la tarea se mueve a la columna *In Progress* y se creará una rama asociada a dicha tarea.
4. Una vez completada la tarea, se crea la *pull requests* y se pasa a la columna *In Review*.
5. Finalmente, cuando la revisión es aceptada, pasa a la columna *Done* y se cierra la *issue*.
6. Opcionalmente, una *issue*, puede decidirse que ya no es necesaria y dejarla a un lado, para ello tenemos una columna *Drop*.

Política de etiquetado

Las *issues* es muy recomendable que sean etiquetadas ya que es la forma que tenemos de ver el tipo de tarea a realizar. Los tipos son:

- **bugs:** Representa una tarea de solucionar algún error.
- **test:** Representa una tarea de *testing* o pruebas.
- **enhancement:** Representa una tarea que aporta un incremento.
- **docs&research:** Representa una tarea de investigación o lectura para obtener información necesaria para la realización de otras tareas.
- **good first issue:** Representa una tarea que es buena para comenzar.
- **documentation:** Representa tareas de documentación.

Adicionalmente, en el *Github Project* se crearán 4 campos que son los siguientes:

- *Priority*: Indica la prioridad de realización de las tareas, yendo desde P0 a P5, siendo P0 la de mayor prioridad y P5 la de menor.
- *Start date*: Indica la fecha de inicio de la tarea.
- *End date*: Indica la fecha de fin de la tarea.

Política de versionado

Seguiremos la política de versionado semántico, siguiendo el patrón X.Y.Z, donde X, Y, Z tomarán los siguientes valores:

- X será usada para cambios mayores y supondrá romper la compatibilidad hacia versiones anteriores. Para el desarrollo inicial, se empezará en la 0.Y.Z, siendo la versión 1.0.0 la que defina la primera versión pública.
- Y será usada para los cambios menores. No romperá la compatibilidad del proyecto, e incluirá nuevas funcionalidad sustanciales o mejoras. También pueden incluir parches.
- Z será usada para corregir errores. No romperá la compatibilidad del proyecto y solo se realizarán cambios internos para corregir comportamientos indebidos.

Es importante señalar que cuando una versión mayor se incrementa (X), el resto se pone a 0. Cuando una versión menor se incrementa (Y), la versión de parches (Z) se pone a 0. La versión se indicará creando una etiqueta de tipo *tag* con el número de la versión cada vez que se realice un incremento de suficiente magnitud.

2.2. Planificación

2.2.1. Product Backlog del proyecto

2.2.2. Sprints

Sprint 1

Sprint 2

Sprint 3

Sprint 4

Sprint 5

3. Análisis tecnológico

3.1. Lenguajes, librerías y herramientas

3.1.1. Lenguajes

3.1.2. Librerías

3.1.3. Herramientas

4. Diseño de la solución

4.1. Diagramas

5. Estudio previo

5.1. Lista de APIs

5.2. Modelado SLA4OAI

5.2.1. Problemas encontrados al modelar

5.3. Dashboards en Grafana

5.3.1. Problemas encontrados

6. Recreación

6.1. Clúster HPC

6.2. Librería

6.3. Conclusión de resultados

7. Conclusiones

Bibliografía

[1] Mark Winteringham. Testing web apis, 2022. 1st edition. Shelter Island, New York: Manning Publications Co. ISBN 1-63835-153-8.

[2] Open APIs - State of the Market 2011, May 2011. URL https://es.slideshare.net/jmusser/open-apis-state-of-the-market-2011. [Online; accessed 22. Apr. 2024].

[3] APIs show Faster Growth Rate in 2019 than Previous Years, July 2019. URL https://web.archive.org/web/20191212210454/https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17. [Online; accessed 22. Apr. 2024].

[4] 2023 State of the API Report | Brought to You by Postman, March 2024. URL https://www.postman.com/state-of-api/api-global-growth/#api-global-growth. [Online; accessed 21. Apr. 2024].

[5] Colaboradores de los proyectos Wikimedia. API - Wikipedia, la enciclopedia libre, February 2024. URL https://es.wikipedia.org/w/index.php?title=API&oldid=158385052. [Online; accessed 22. Apr. 2024].

[6] Postman api platform — sign up for free. https://www.postman.com/. (Accessed on 04/24/2024).

[7] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 682–685, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3469082. URL https://doi.org/10.1145/3460319.3469082.

[8] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of restful apis: promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 408–420, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549144. URL https://doi.org/10.1145/3540250.3549144.