Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Trabajo de Fin de Máster

Sistema de Comentarios basado en GitHub Issues

Comment System based on GitHub Issues

Pedro Miguel Lagüera Cabrera

La Laguna, 29 de junio de 2020

D. **Casiano Rodríguez León**, con N.I.F. 42.020.072-S profesor Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*Sistema de Comentarios basado en GitHub Issues*"

ha sido realizada bajo su dirección por D. **Pedro Miguel Lagüera Cabrera**, con N.I.F. 79.083.153-E.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 29 de junio de 2020

Agradecimientos

Quiero dar las gracias a todas aquellas personas que han contribuido a la realización de este Trabajo Fin de Máster.

A Casiano Rodríguez León, tutor de este proyecto, por la confianza que ha depositado en mí y por la amplia libertad que me ha dado durante todo el desarrollo para utilizar mis propias ideas y tomar mis propias decisiones. Sus conocimientos, apoyo y paciencia han sido una parte fundamental de este proyecto, y sin ellos, su desarrollo no hubiera sido posible.

A mis amigos, dentro y fuera de la universidad por ayudarme siempre a ver el lado bueno de cualquier situación y a afrontar los obstáculos que se han presentado.

A mi familia por apoyarme, ayudarme y animarme cuando las circunstancias no acompañaban y por celebrar conmigo cuando sí.

Licencia

Resumen

En los últimos años se ha producido un crecimiento explosivo en la cantidad, calidad y variedad de tecnologías, sistemas y frameworks *web. Desde pequeños superconjuntos o* supersets *de lenguajes de programación que facilitan el desarrollo hasta gestores de dependencias con millones de paquetes y* frameworks *con una complejidad y completitud inimaginable hace solo 15 años.*

Las aplicaciones web se han desplazado hacia la modularidad, prueba de ello son las arquitecturas más utilizadas actualmente: Single-Page Application*,* Microservicios *y* Serverless*. Es por ello que ha nacido un enorme segmento dentro del mundo de la programación dedicado a crear módulos, paquetes y servicios con el fin de proporcionar funcionalidades a otras aplicaciones más complejas.*

La ambición de este proyecto es crear un sistema de comentarios que pueda ser utilizado por cualquier página, explorando distintas tecnologías, frameworks y programas, además de usar las metodologías de desarrollo y los servicios más populares.

Palabras clave: Widget, Sistema de Comentarios, GitHub, Issues, Node.js, ReactJS, API REST, GraphQL, AWS, HTML, SASS, Typescript, Integración Continua, Travis CI

Abstract

There has been an incredibly rapid growth in quantity, quality and variety of web technologies, systems and frameworks in the last decade. From small supersets of programming languages that simplify development to large dependency managers with millions of packages and high level frameworks with a complexity and richness unimaginable only 15 years ago.

Web applications have shifted towards modularity, the growing popularity of Single-Page Applications*,* Microservices *and* Serverless *architectures attest to this fact. This shift is the root of a relatively new niche within the world of computer programming dedicated to the creation of modules, packages and services, with reduced size and functionality, which can be used and joined to form larger, more complex applications.*

This project's main ambition is the creation of a comment system which can be used in any webpage, exploring various new and popular technologies, frameworks and programs along the way, and using the most advanced development techniques, methodologies and deployment services.

Keywords: Widget, Comment System, GitHub, Issues, Node.js, ReactJS, REST API, GraphQL, AWS, HTML, SASS, Typescript, Continuous Integration, Travis CI

Índice general

Índice de Figuras

Índice de Fragmentos de Código

Índice de Tablas

Capítulo 1

Introducción

Una sección de comentarios es una herramienta creada con el fin de proveer a *blogs* y páginas de noticias en internet con la posibilidad de que su audiencia pueda comentar sobre el contenido publicado, siendo una continuación de la antigua práctica de publicar cartas al editor. Sin embargo, en la actualidad se utilizan secciones de comentarios en toda clase páginas y aplicaciones web para poder establecer una comunicación entre la audiencia y el contenido y entre los propios miembros de la audiencia[37].

Un servicio de *hospedaje* o *hosting* de comentarios es un servicio que almacena los comentarios hechos en una determinada página web de manera externa a dicha página, además, tales servicios suelen permitir iniciar sesión en su sistema de comentarios a través de credenciales de perfiles de redes sociales. De este modo, las tareas de almacenamiento de comentarios y administración de credenciales son gestionadas por servicios externos, ahorrando a los responsables de la página el tiempo, dinero y esfuerzo necesarios para implementarlos por su propia cuenta[35].

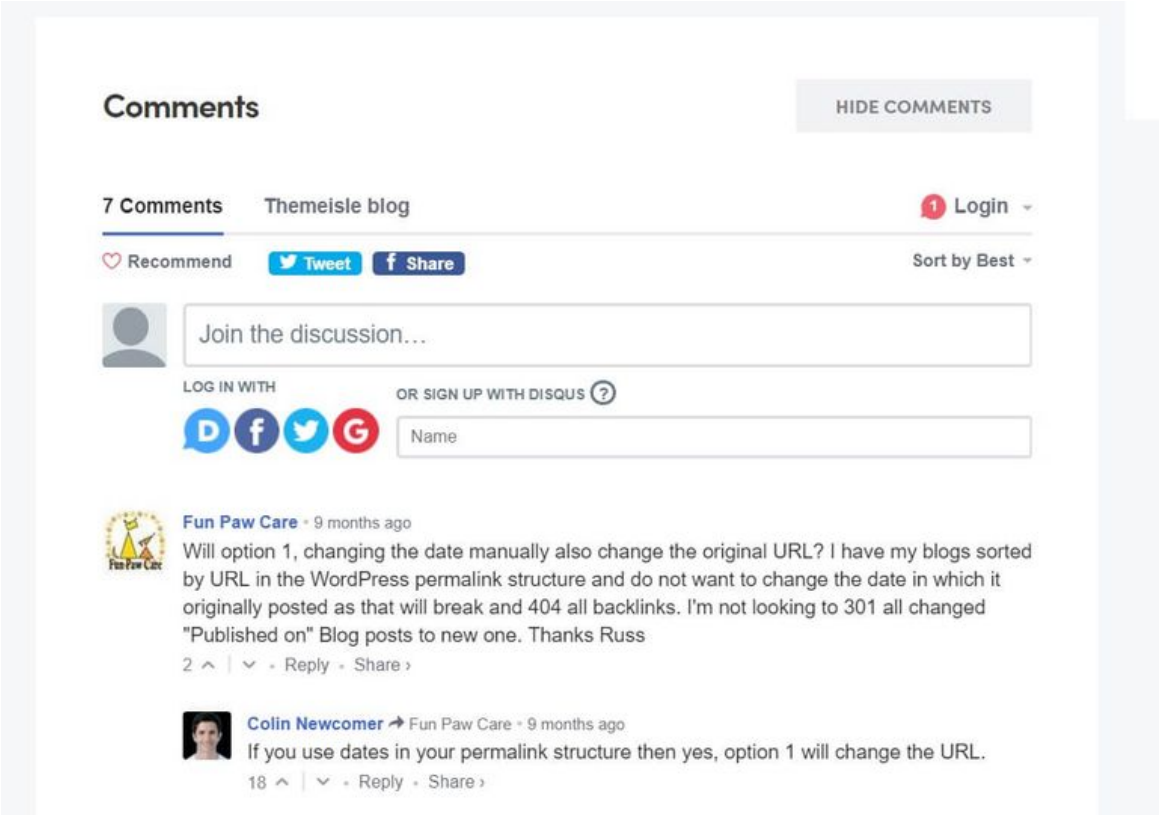


Figura 1.1: Un sección de comentarios hospedada por *Disqus*

Estos servicios de hospedaje de comentarios son implementaciones específicas para sistemas de comentarios de servicios de *computación en la nube* o *cloud computing*. *Cloud computing* es la disponibilidad bajo demanda de recursos de sistemas informáticos, especialmente almacenamiento de datos

y poder de cómputo, sin la necesidad de una administración directa y activa por parte del usuario. La computación en la nube permite que las empresas minimicen o eliminen costes por adelantado de infraestructura, además de permitir un despliegue más rápido, mejor administración, menos mantenimiento y la habilidad de ajustar los recursos para cubrir períodos de alta demanda. El modelo de pago que suelen usar los proveedores de computación en la nube es '*pay-as-you-go*', en el que se paga solo por los recursos que han sido utilizados, lo que puede ser problemático en los períodos de alta demanda si no se establecen algunos límites sobre los recursos. La disponibilidad de redes de alta capacidad y el bajo coste de ordenadores y dispositivos de almacenamiento, junto con la amplia adopción de la *virtualización de hardware* y la arquitectura orientada a servicios ha conducido a que la mayoría de empresas están optando por esta estrategia en lugar de tener que comprar, instalar, desplegar y administrar sus propias infraestructuras[36].

Una aplicación monolítica es una aplicación construida como una sola unidad que incluye todas las funcionalidades y dependencias necesarias para poder funcionar. Esta clase de arquitectura es perfectamente natural, toda la lógica es ejecutada en un proceso y el desarrollador puede utilizar las características del lenguaje en el que se programe para dividir la aplicación en clases, funciones y espacios de nombres, no obstante, dado que los procesos de prueba y despliegue deben englobar a toda la aplicación, estos pueden resultar lentos y complejos. Esta clase de aplicaciones puede ser escalada horizontalmente al ejecutar varias instancias detrás de un *balanceador de carga*[45].

Sin embargo, dado el aumento en popularidad de aplicaciones desplegadas en la nube mencionado anteriormente, muchos desarrolladores están optando por otras arquitecturas que aprovechen mejor la naturaleza distribuida de la nube. Cualquier cambio en una aplicación monolítica requiere reconstruir (*rebuild*) y volver a desplegar la aplicación entera. Mantener una estructura modular, que limite los efectos de un cambio en un módulo sobre otros módulos, puede resultar complicado a medida que la aplicación crece y su escalado requiere toda la aplicación, en lugar de las partes que requieren más recursos.

Por estos motivos, es común utilizar arquitecturas como la de *microservicios*. La idea principal sobre la que gira la *arquitectura de microservicios* es que algunas aplicaciones son más fáciles de desarrollar, mantener y escalar cuando están divididas en partes más pequeñas que trabajan conjuntamente. Cada parte es desarrollada y mantenida por separado, y por lo tanto, es más fácil de entender, probar y mantener, permitiendo un desarrollo más ágil y reduciendo el tiempo necesario para poner mejoras en producción.

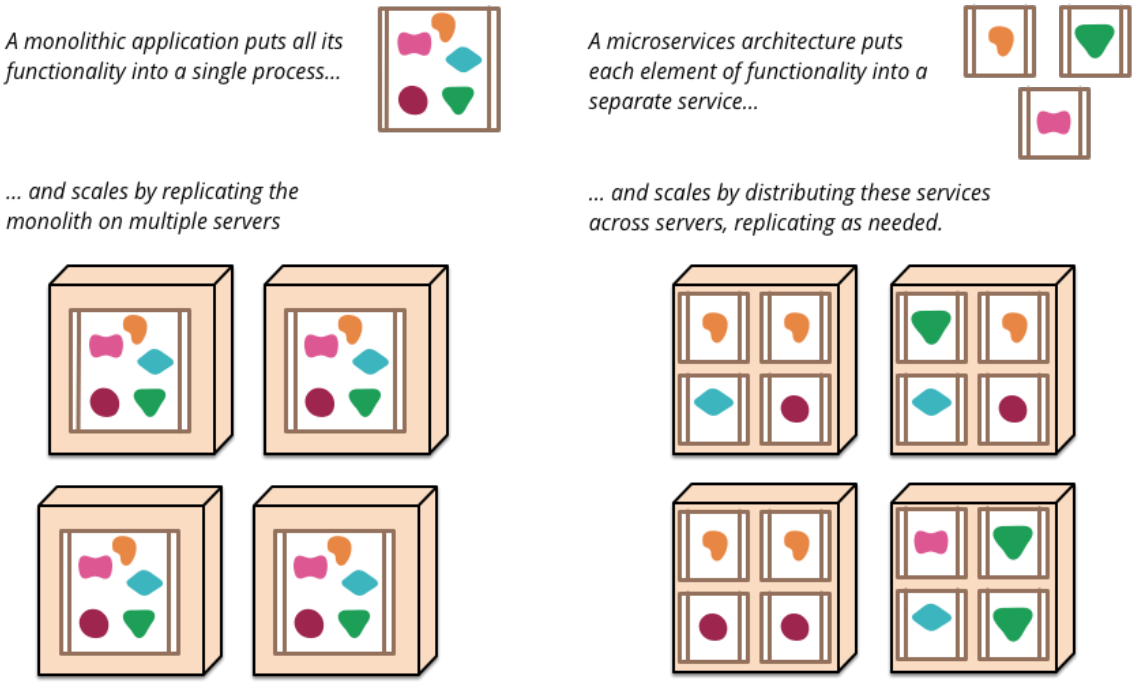


Figura 1.2: Arquitectura Monolítica y Arquitectura de Microservicios

1.1. Objetivos

1.1.1. Objetivo General

El objetivo general de este proyecto es desarrollar un servicio de comentarios que pueda ser utilizado por cualquier página o aplicación web. El servicio debe ser desplegado públicamente de manera que esté disponible para cualquier usuario independientemente del lugar desde que se conecte y el coste total del proyecto y por tanto, del despliegue debe ser cero.

1.1.2. Objetivos Específicos

El resultado de este proyecto deberá cumplir los siguientes objetivos específicos para poder considerarse como satisfactorio.

- El sistema de comentarios utilizará la existente *GitHub API*, sus *issues* y *comentarios*, de manera que no será necesario implementar un *backend* con servicios de almacenamiento y control de acceso y credenciales.
- El proyecto deberá utilizar la práctica de *integración continua* desde su comienzo, de tal modo que el desarrollo avance sobre muchos cambios pequeños e incrementales, en lugar de cambios grandes y revolucionarios que complican el proceso de pruebas.
- Cada sección de comentarios será una instancia del cliente web que debe utilizar tecnologías actuales para generar una interfaz de usuario moderna y semejante a la de *GitHub*.
- Se deberá implementar un servidor *API REST* que sirva de intermediario entre los clientes web y *GitHub API*.
- Todos los componentes del proyecto deberán estar desplegados públicamente utilizando servicios de *Infraestructura o Plataforma como Servicio* (*IaaS* o *PaaS*).

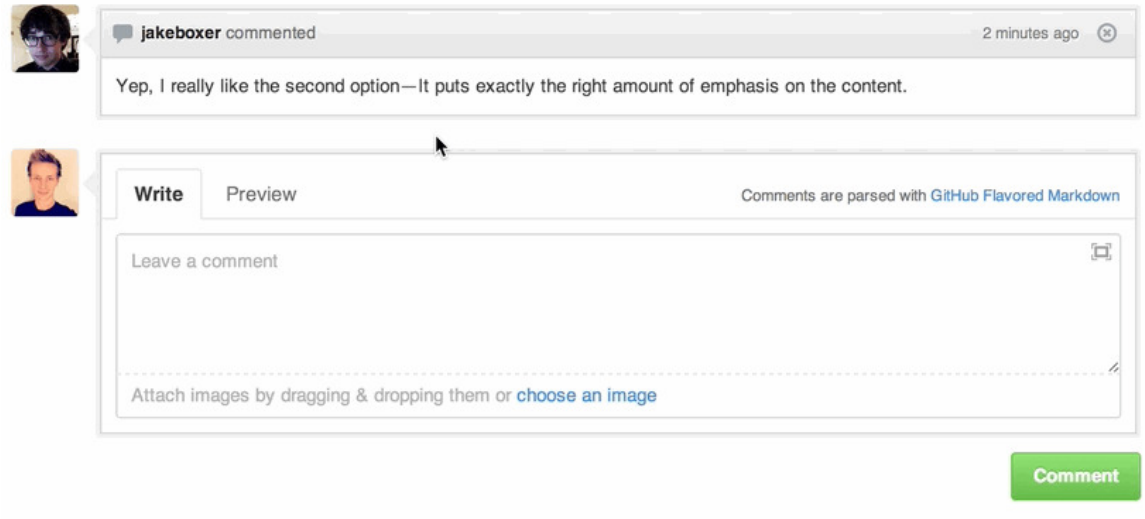


Figura 1.3: Un sección de comentarios de *GitHub*

1.2. Tareas

Para cumplir los objetivos enunciados anteriormente se llevarán a cabo las siguientes tareas:

1. Buscar, descubrir y estudiar implementaciones similares al proyecto que se va a desarrollar y establecer qué lenguajes, tecnologías y *frameworks* van a ser utilizados durante su elaboración.
2. Investigar y aprender los lenguajes, tecnologías y *frameworks* que se han elegido en el paso anterior para utilizarlos en el proyecto.
3. Aprender las estructuras de datos y los métodos de comunicación que se van a usar en *GitHub API*.
4. Establecer la estructura del proyecto, crear repositorios y directorios y establecer el flujo de pruebas e integración continua que seguirá a lo largo de su desarrollo.
5. Implementar un servidor capaz de comunicarse con *GitHub API* y obtener los datos necesarios, haciendo solo peticiones de lectura que no requieran autenticación.
6. Añadir al servidor implementado en el paso anterior, métodos de autenticación y realización de peticiones acompañadas de credenciales.
7. Crear un cliente web que se comunique con el servidor y presente la información que reciba al usuario.
8. Implementar en el cliente algún mecanismo de persistencia de identificación como sesiones o *cookies*.
9. Modificar el estilo del cliente web para que se asemeje al estilo de *GitHub* y crear un tema claro (*light*) y otro oscuro (*dark*).
10. Adquirir un dominio público y desplegar:
    - El servidor en una plataforma de *infraestructura como servicio* (*IaaS*).
    - El cliente en una red de distribución de contenidos estáticos que optimice la entrega del cliente independientemente de la posición geográfica del usuario.
    - Una página que describa y detalle el proyecto y explique los pasos a seguir para utilizar el cliente web.
11. Realizar pruebas de rendimiento y optimizar los métodos y recursos necesarios para mejorar la experiencia del usuario.
12. Comentar y documentar el código del proyecto para facilitar su comprensión.

Capítulo 2

Contexto

2.1. Antecedentes

Los sistemas y secciones de comentarios no son una invención reciente, desde que existen *blogs*, portales de noticias y redes sociales han existido muchas y diferentes aproximaciones. A finales de los años 90 aparecieron los primeros *blogs* y noticiarios que ofrecían la posibilidad de colgar comentarios públicos o privados para sus autores y aunque muchos dudaban su utilidad al principio, en el año 2008 el 75 % de los 100 periódicos más populares contaban con secciones de comentarios[38]. Sin embargo, la implementación de estas secciones ha sido tradicionalmente individual para cada página o aplicación web, es decir, cada una requiere una infraestructura particular que le permita proporcionar un sistema de comentarios.

En la actualidad, la implementación de un sistema de comentarios es más simple que nunca para un desarrollador, no obstante, la necesidad de estos sistemas se encuentra en un máximo histórico debido a que la mayoría de empresas e individuos necesitan comunicarse con otros usuarios para compartir opiniones o información, pero no todos pueden permitirse invertir tiempo en aprender a crear un sistema de comentarios o dinero en alguien que sepa hacerlo. Motivado por el reciente auge de los **servicios** web, ha surgido un nuevo paradigma llamado *comentarios como servicio*, en el que el usuario no tiene que preocuparse por los detalles de *software* o *hardware* gracias a que un proveedor se encarga de ello a cambio de una compensación monetaria, significativamente menor que los costes de las implementaciones tradicionales.

2.2. Estado Actual

Gracias a la enorme comunidad de desarrolladores activa en internet y más específicamente en *GitHub*, se han descubierto múltiples proyectos variados en complejidad y popularidad que hacen uso de algunas funcionalidades de *GitHub* para ofrecer secciones de comentarios en algunos casos. Estas secciones de comentarios usan diferentes estilos, implementaciones, arquitecturas y lenguajes, esta multitud de formas distintas de crear una sección de comentarios ha inspirado algunas de decisiones y estrategias usadas en este proyecto

- **Disqus** - Aunque no utiliza *GitHub*, es con diferencia el *widget* de comentarios más famoso y utilizado en internet, tiene herramientas avanzadas de monitorización y estadística, permite identificarse con varios tipos de redes sociales, muestra comentarios en tiempo real, adapta su estilo al sitio donde se disponga e incluso ofrece herramientas de monetización de anuncios[8].
- **Coral** - Es una plataforma de comentarios *open-source* escrita en *Typescript* que toma una estrategia alternativa sobre las funciones de moderación, presentación y conversación en un sistema de comentarios.[5].
- **Utterances** - Un *widget* de comentarios ligero basado en *GitHub Issues* para añadir comentarios en blogs, wikis y mucho más. Está escrito en *Typescript*, es *open-source*, no tiene anuncios, tiene varios temas y no utiliza *frameworks* adicionales.[32].
- **gh-comentify** - Es un simple servidor *API REST* que se comunica con *GitHub* para crear y leer comentarios. Está escrito en *Typescript* y solo consta de *backend*, la interaz de usuario debe ser creada por separado[14].

- **Remark** - Es un sistema de comentarios *autohospedado* o *self-hosted*, ligero, funcional y simple que no espía a sus usuarios. Puede ser embebido en blogs, artículos o cualquier lugar donde los usuarios añadan comentarios. Está escrito en *Go* y *Typescript* y dispone de inicio de sesión con redes sociales, correo electrónico o anónimo, moderar comentarios y usuarios, sistema de copias de seguridad y está completamente *dockerizado*[26].
- **Diskuto** - Es un sistema de comentarios *open-source*, ligero y embebible creado para ser integrado con páginas web existentes. Está escrito en lenguaje de programación *D* y soporta comentar sin registrarse, votar comentarios, tiene un sistema *antispam* e incluso puede funcionar mínimamente con *Javascript* desactivado[7].

Estos proyectos variados en su implementación pero similares en su funcionamiento servirán de base durante la fase aprendizaje necesaria para poder construir un sistema de comentarios utilizando *issues* y comentarios de *GitHub*. Si bien se espera conseguir un resultado único, funcional, satisfactorio y con múltiples y atractivas características, resultará imposible implementar todas las funcionalidades que hacen especiales a las aplicaciones mencionadas anteriormente dado que su desarrollo por sí mismo, sin testeo, memoria o despliegue llevaría más tiempo que el total del desarrollo del trabajo de fin de máster.

2.3. Tecnologías y Herramientas Utilizadas

2.3.1. Javascript

JavaScript es un lenguaje de programación que se ajusta a la especificación *ECMAScript*, un patrón creado con el fin de estandarizar *Javascript* para fomentar múltiples implementaciones independientes[10]. *Javascript* es un lenguaje de alto nivel, a menudo compilado en tiempo de ejecución y permite más un tipo de paradigma de programación; su sintaxis se basa en llaves '{}', su tipado es dinámico, su orientación a objetos se basa en prototipos y sus funciones son de primera clase.

Junto con *HTML* y *CSS*, *Javascript* es una de las tecnologías fundamentales de la *World Wide Web*. *Javascript* permite crear páginas web interactivas y es una parte esencial de las aplicaciones web. La mayoría de páginas web lo utilizan para comportamientos en la lado del cliente, no obstante, en los últimos años se ha producido un crecimiento importante en tecnologías que lo usan en servidores.

Javascript soporta tres paradigmas de programación, imperativo, funcional y dirigido por eventos. Tiene *Application Programming Interface* o *API*s para trabajar con texto, fechas, estructuras de datos estándares, expresiones regulares y el *Document Object Model* o *DOM*, sin embargo, no incluye funcionalidades de *entrada/salida (E/S)* como redes, almacenamiento o gráficos, dado que el entorno que lo ejecuta suele proveer dichas funcionalidades.

2.3.2. Typescript

Typescript es un lenguaje de programación *open-source* o *fuente-abierta* desarrollado y mantenido por *Microsoft*. Es un superconjunto o superset sintáctico estricto de *Javascript* y le añade tipado estático opcional. *Typescript* está diseñado para el desarrollo de grandes aplicaciones que se compilan a *Javascript* y dado que *Typescript* es un superconjunto de *Javascript*, cualquier programa *Javascript* es también un programa *Typescript* válido. El propio compilador de *Typescript* está escrito en *Typescript* y es compilado a *Javascript*.

Entre las características de *Typescript* destacan dos, en primer lugar, deja de lado el tipado dinámico que hace a *Javascript* tan fácil y flexible, pero a veces tan confuso a la hora de desarrollar y utiliza un tipado estático que aunque puede resultar muy estricto en algunas ocasiones, da lugar a un código mucho más legible y fácil de depurar, especialmente cuando se desarrolla en equipo. Y en segundo lugar, *Typescript* soporta archivos de definición que contienen información sobre tipos de diferentes librerías, similar a los archivos *header* en *C++* que describen la estructura de archivos objeto existentes.

2.3.3. Node.js y NPM

Node.js es un entorno de ejecución de *Javascript open-source* y multiplataforma que ejecuta código *Javascript* fuera de un navegador web. Permite a los desarrolladores utilizar *Javascript* para escribir herramientas de línea de comandos y *scripts* en el *lado del servidor (server-side)* que producen contenido web dinámico antes de que sea enviado al navegador web del usuario. De esta manera, *Node.js* unifica el desarrollo de aplicaciones web mediante un único lenguaje de programación, en lugar de lenguajes distintos en el *lado del cliente (client-side)* y el *lado del servidor (server-side)*.

Node.js permite crear *servidores web (web servers)* y herramientas de red usando *Javascript* y una colección de módulos que manejan varias funcionalidades esenciales. Existen módulos para manejar E/S en el sistema de archivos, redes (*DNS, HTTP, TCP, TLS/SSL*, o *UDP*), datos binarios (*buffers*), corrientes de datos (data streams), funciones de criptografía, entre otros. *Javascript* es el único lenguaje soportado de forma nativa por *Node.js* pero como se ha visto en Typescript, existen lenguajes que son compilados a *Javascript* que pueden ser utilizados como *Typescript*, *Coffescript*, *Dart*, etcétera.

El uso más común de *Node.js* es para construir programas de red como servidores web. *PHP* es un competidor de *Node.js*, la mayor diferencia entre ellos es que muchas funciones en *PHP* bloquean el flujo de ejecución hasta que terminan de ejecutarse, mientras que las funciones *Node.js* son ejecutadas concurrentemente o incluso en paralelo y así no bloquean el flujo de ejecución.

Node Package Manager o *npm* es un *sistema de gestión de paquetes* o *package manager* para *Javascript*. Es el sistema de gestión de paquetes por defecto de *Node.js* y consiste en un cliente en línea de comandos llamado *npm* y una base de datos de paquetes públicos y privados llamada *npm registry*. Con esta herramienta es posible descargar cualquier paquete de *Node.js* como dependencia e incluso crear un nuevo paquete y subirlo al registro.

2.3.4. Yarn

Yarn es un *sistema de gestión de paquetes* o *package manager open-source* desarrollado por *Facebook* para *Javascript*. Yarn apunta a ser una alternativa más rápida, segura y fiable a *npm*, el *sistema de gestión de paquetes* por defecto de *Node.js*. Cada *base de código* o *codebase* es compartida como un paquete o módulo que contiene todos sus archivos, además de un archivo *package.json* que describe el paquete y un archivo *yarn.lock* que guarda las dependencias con otros paquetes.

2.3.5. GitHub y GitHub Pages

Las copias de seguridad de los datos de un usuario son importantes, tanto si son fotografías, documentos o código de programación. Las alternativas siempre son las mismas: la nube usando *Dropbox* o *Google Drive*, en local usando discos duros, pero en el caso del código existe una alternativa mucho mejor: los repositorios *Git*. Esta clase de repositorios son una copia local del código generado con una característica muy importante, y es que es posible controlar las versiones del código para poder utilizar una versión anterior, o para trabajar en funcionalidades nuevas sin necesidad de modificar la versión funcional y así no romper el proyecto. Sin embargo, estos repositorios por sí mismos se mantienen almacenados localmente, por este motivo nacen servicios como *GitHub*, que pretenden llenar ese vacío almacenando repositorios *Git* en sus servidores para poder acceder a todas las versiones del código desde cualquier lugar. Un repositorio posee muchas más utilidades aparte de almacenar código y controlar versiones, existen herramientas de *feedback* para permitir que autor de un repositorio reciba preguntas, problemas, soluciones y sugerencias de usuario y otros desarrolladores. *Issues* son una manera excelente de vigilar el progreso de tareas, mejoras y *bugs* en proyectos, son como el correo electrónico pero pueden ser compartidos y discutidos con cualquier otra persona que tenga acceso a ellos mediante comentarios. Cada repositorio tiene una sección de *issues* y cada *issue* contiene comentarios.

GitHub ha sido una herramienta de vital importancia en el desarrollo de este proyecto dado que ha permitido alternar dicho desarrollo entre múltiples máquinas, tener acceso a versiones anteriores del proyecto cuando se han intentado implementar cambios que finalmente han sido descartados y, más importantemente, sin GitHub API este proyecto no habría sido posible.

GitHub Pages es un servicio de *hospedaje* o *hosting* de sitios estáticos que toma archivos *HTML*, *CSS* y *Javascript* directamente de un repositorio en *GitHub*, opcionalmente ejecuta un *proceso de construcción* o *build* y lo publica en una página web. Dicha página web puede ser un subdominio de *GitHub.io* perteneciente a *GitHub*, o utilizar un dominio personal.

2.3.6. API REST

API o *Application Programming Interface* representa la capacidad de comunicación entre componentes de software, y es un conjunto de subrutinas, funciones, procedimientos y métodos que ofrece una determinada librería para ser utilizada por otro *software* como una capa de abstracción.

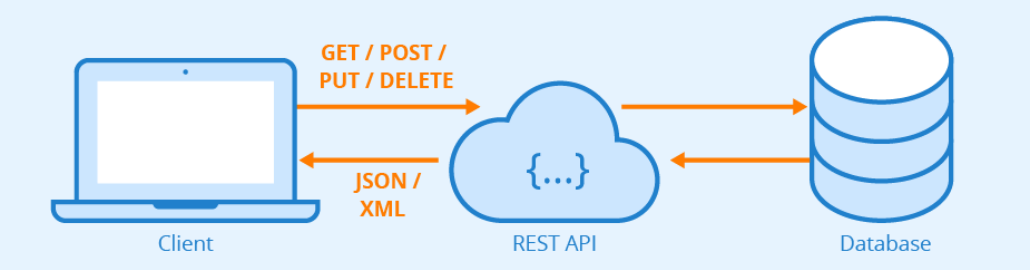


Figura 2.1: Esquema del funcionamiento de una *API REST*

REST o *REpresentational State Transfer* es un conjunto de reglas que los desarrolladores deben seguir a la hora de crear una *API*. Solo *APIs* que cumplan las siguientes 6 condiciones podrán ser consideradas *RESTful*.

1. *Cliente–Servidor* – Al separar las cuestiones de la interfaz de usuario de las del almacenamiento, aumenta la portabilidad de la interfaz de usuario a otras plataformas y mejora la escalabilidad al simplificar los componentes del servidor.
2. *Sin Estados* – Cada petición de un cliente al servidor debe contener toda la información necesaria para entender la petición y no puede aprovecharse de ningún contexto almacenado en el servidor. Por lo tanto, el estado de cada sesión es gestionado exclusivamente en el cliente.
3. *Cacheable* – Restricciones de caché requieren que los datos incluidos en una respuesta a una petición deben ser etiquetado explícita o implícitamente como *cacheable* o *no cacheable*. De este modo, si una respuesta es cacheable, una caché cliente tendrá permiso para reutilizar dicha respuesta para peticiones equivalentes subsiguientes.
4. *Interfaz Uniforme* – Define la interfaz entre clientes y servidor, simplifica y desacopla la arquitectura y permite que cada parte evoluciones independientemente.
    a) *Basado en Recursos* - Cada recurso es identificado en peticiones usando *Universal Resource Identifiers (URI)*. Los recursos almacenados en el servidor son conceptualmente diferentes de las representaciones enviadas al cliente, el servidor no envía su base de datos sino la información que almacena formateada en *XML* o *JSON*.
    b) *Manipulación de Recursos a través de Representaciones* - Cuando un cliente posee una representación de un recurso y cualquier metadato adjunto, tiene suficiente información para modificar o eliminar el recurso del servidor, siempre y cuando tenga permiso para hacerlo.
    c) *Mensajes Autodescriptivos* - Cada mensaje incluye suficiente información para describir cómo procesar el mensaje.
    d) *Hypermedia como el Motor del Estado de la Aplicación (HATEOAS)* - Al acceder a una *URI* inicial, un cliente *REST* debe ser capaz de descubrir dinámicamente todos los recursos disponibles que necesita a través enlaces proporcionados por el servidor. A medida que se produce el acceso el servidor responde con texto que incluye enlaces (*hyperlinks*) a otros recursos relacionados que estén disponibles.
5. Sistema de Capas – El sistema de capas permite que una arquitectura sea compuesta por capas jerárquicas al restringir el comportamiento de cada componente de manera que cada uno no pueda acceder más allá de la capa donde interactúa.

6. *Código bajo Demanda (opcional)* – *REST* permite que la funcionalidad del cliente sea extendida al descargar y ejecutar código en forma de *applets* o *scripts*, resultando en clientes más simples al reducir el número de funcionalidades requeridas.

La comunicación con una *API REST* se realiza mediante *peticiones HTTP*, que representan el concepto de *interfaz uniforme*. Los tipos de peticiones *HTTP* más utilizados son los métodos *CRUD* o *create*, *read*, *update* y *delete*[50].

- *GET* - Este método HTTP se utiliza para leer (*read*) una representación de un recurso, que es devuelta en forma de *XML*, *JSON* o texto.
- *POST* - Este método HTTP se utiliza para crear (*create*) un nuevo recurso, es habitual crear recursos como hijos de otros recursos.
- *PUT* - Este método HTTP se utiliza para actualizar (*update*) la representación de un recurso en el caso de que su *URI* exista, o de lo contrario, crear un nuevo recurso.
- *DELETE* - Este método HTTP se utiliza para borrar (*delete*) un recursos identificado por una *URI*.

2.3.7. GraphQL

GraphQL es un *lenguaje de consultas* o *query language* para *API*s y un sistema de ejecución para realizar dichas consultas creado por *Facebook*. GraphQL provee una descripción completa y comprensible de los datos en una *API*, permite que clientes soliciten exactamente la información que necesitan y nada más, facilita la evolución de *API*s y habilita potentes herramientas de desarrollo.

Proporciona un nuevo enfoque sobre el desarrollo de *API*s y ha sido comparado y contrastado con *REST* y otras arquitecturas de servicios web. Permite que cada cliente defina la estructura de los datos requeridos y que el servidor devuelva esa misma estructura de datos, previniendo así que excesivas cantidades de datos sean devueltas, pero afectando a la efectividad del caché web sobre los resultados de las consultas. La flexibilidad y riqueza del lenguaje de consultas también añade una complejidad que puede no valer la pena para *API*s simples. *GraphQL* soporta lectura, escritura (*mutaciones*) y suscripción a cambios en los datos en tiempo real, normalmente mediante el uso de *WebHooks*.

2.3.8. ReactJS

React o *ReactJS* es una librería *Javascript open-source* centrada en la construcción de interfaces de usuario (*UI*). Su mantenimiento corre a cargo de *Facebook* y una gran comunidad de desarrolladores individuales y empresas.[42]

1. *Componentes* - El código en *ReactJS* está compuesto por entidades llamadas componentes. Un componente puede ser renderizado en un elemento del *DOM* usando la librería *React DOM* y puede ser tan pequeño como un botón o un *input*, o estar compuesto por muchos componentes que forman un componente más complejo. Al renderizar un componente, es posible pasarle valores conocidos como *props*. Existen *componentes funcionales* que son declarados con una función que devuelve código *JSX* y *componentes basados en clases* que heredan de la clase *React.Component* y poseen estados que pueden ser modificados y heredados por clases hijas.
2. *DOM Virtual* - Es un mecanismo de caché de estructuras de datos que permite que cada vez que haya un cambio en un componente se actualice únicamente dicho componente en lugar de la página entera. Este renderizado selectivo mejora notablemente el rendimiento dado que no hay que recalcular el estilo *CSS*, el *layout* de la página y renderizar toda la página.
3. *Métodos del Ciclo de Vida* - Son *hooks* que permiten la ejecución de código en ciertos puntos durante la vida de un componente. Cada vez que un componente se crea, destruye o renderiza, se produce una llamada a sus respectivos *hooks* que a su vez ejecutan el código que el desarrollador especifique.
4. *JSX* - *JSX* o *JavaScript XML*, es una extensión de la sintaxis de *Javascript*. *JSX* establece una forma de estructurar el renderizado de componentes usando una apariencia similar a *HTML* y una sintaxis familiar para muchos desarrolladores.

5. *React Hooks* - Son funciones que permiten al desarrollador 'engancharse' a estados y ciclos de vida en *ReactJS* desde un componente y ser notificados cada vez que se produzca un cambio.

```
class App extends React.Component {
   render() {
      return (
         <div>
            <p>Header</p>
            <p>Content</p>
            <p>Footer</p>
         </div>
      );
   }
}
```

Listado 2.1: Ejemplo componente basado en *Clases React JSX*

2.3.9. DNS

Domain Name System o *DNS* es las páginas amarillas de internet. Los usuarios acceden a sitios web a través de dominios como *elpais.es* o *google.com*, sin embargo, los navegadores web interactúan a través de direcciones *IP* o *Internet Protocol*. *DNS* se ocupa de traducir dominios a direcciones *IP* para los navegadores puedas acceder correctamente a recursos en internet. Cada dispositivo conectado a internet tiene una dirección *IP* única que otros dispositivos utilizan para comunicarse. El proceso de traducción de dominio a dirección *IP* se denomina *resolución DNS* y consta de los siguientes pasos.

1. *Resolutor Recursivo DNS* - Es el bibliotecario al que se le pide encontrar un determinado libro en algún lugar de la biblioteca. El *resolutor recursivo DNS* es un servidor diseñado para recibir consultas de clientes a través de navegadores y otras herramientas, en la mayoría de casos es responsable de realizar todas las consultas adicionales necesarias para satisfacer la consulta *DNS* del cliente.
2. *Servidor de Nombres Raíz* - Es el primer paso en la resolución de dominios a direcciones *IP*. En una biblioteca sería un índice que indica la localización de distintas estanterías, normalmente sirve como una referencia hacia otras localizaciones más específicas.
3. *Servidor de Nombres TLD* - El servidor de nombre (*nameserver*) de dominios de primer nivel (*TLD*) puede ser interpretado como una estantería en particular en una biblioteca. Este servidor contiene la última parte de un nombre de dominio, por ejemplo, en '*google.com*' el servidor de nombre *TLD* es '*com*'.
4. *Servidor de Nombres Autoritativo* - Es el último servidor de nombres, se asemeja a un diccionario individual dentro de una estantería en el que un nombre de dominio específico puede ser traducido a su dirección *IP*. Si el servidor tiene acceso a dicha entrada en el 'diccionario', devolverá la dirección *IP* para el nombre de dominio consultado al *resolutor recursivo DNS* que hizo la consulta inicial.

Los servidores *DNS* crean *registros DNS* para proporcionar información importante sobre un nombre de dominio o de *host*, particularmente su dirección *IP*. Los tipos más de comunes de registro *DNS*[43] son los siguientes:

- *A* - Dirección (*address*). Este registro se usa para traducir nombres de servidores de alojamiento a direcciones *IPv4*.
- *CNAME* - Nombre canónico (*canonical name*). Se usa para crear nombres de servidores de alojamiento adicionales, o alias, para los servidores de alojamiento de un dominio. Es usado cuando se están ejecutando múltiples servicios (como *FTP* y servidor web) en un servidor con una sola dirección *IP*. Cada servicio tiene su propio registro *DNS* (como *ftp.ejemplo.com.* y *www.ejemplo.com.*).
- *NS* - Servidor de nombres (*nameserver*). Define la asociación que existe entre un nombre de dominio y los servidores de nombres que almacenan la información de dicho dominio. Cada dominio se puede asociar a una cantidad cualquiera de servidores de nombres.

- *MX* - Intercambio de correo (*mail exchange*). Asocia un nombre de dominio a una lista de servidores de intercambio de correo para ese dominio. Tiene un balanceo de carga y prioridad para el uso de uno o más servicios de correo.
- *PTR* - Indicador (*pointer*). También conocido como 'registro inverso', funciona a la inversa del registro A, traduciendo *IP*s en nombres de dominio. Se usa en el archivo de configuración de la zona *DNS* inversa.
- *SOA* - Autoridad de la zona (*start of authority*). Proporciona información sobre el servidor *DNS* primario de la zona.

2.3.10. Heroku

Heroku es una plataforma como servicio (*PaaS*) en la nube que permite crear, entregar, monitorizar y escalar aplicaciones web, y se posiciona como la manera más sencilla y rápida de desplegar una aplicación web sin tener que preocuparse por infraestructuras y configuraciones; además, soporta varios lenguajes y *frameworks* populares como *Node.js, Ruby, Java, PHP, Python* y *Go*, entre otros.

Heroku dispone de un *Free Tier* o *nivel gratuito* muy completo que permite desplegar pequeñas aplicaciones sin tener que añadir métodos de pago. Para ello solo hay que seguir sus instrucciones para instalar *heroku-cli*, autenticarse, crear una aplicación de *Heroku* y simplemente hacer un *push* a la rama *Heroku* a través de *Git*. Una vez desplegada correctamente, la aplicación será accesible desde un subdominio de *herokuapp.com* mediante *HTTPS* con un *certificado SSL* de *Heroku*, el nombre del subdominio será el nombre de la aplicación, por ello, este deberá ser único.

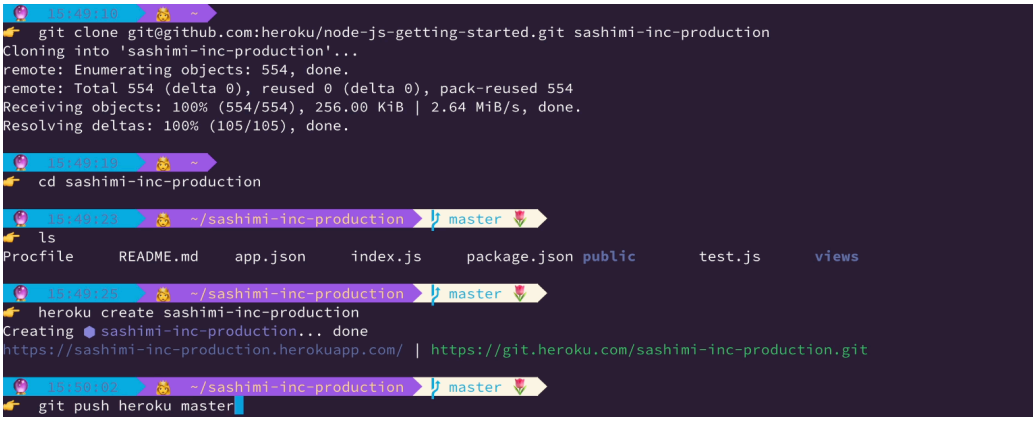


Figura 2.2: Despliegue en *Heroku* de un proyecto *Node.js*

2.3.11. Amazon Web Services

Amazon Web Services o *AWS* es una empresa subsidiaria de *Amazon* que provee plataformas (*PaaS*) e infraestructuras (*IaaS*) de computación en la nube y *API*s para interactuar con ellas bajo demanda a usuarios, empresas y gobiernos. En *AWS* cada usuario puede seleccionar la combinación específica de servicios y soluciones que necesita y únicamente pagar por lo utilizado, resultando en un gasto menor sin tener que sacrificar el rendimiento de la aplicación o la experiencia del usuario.

Todos estos servicios están integrados en la infraestructura de *AWS*, por lo que están diseñados para trabajar juntos sin tener que requerir configuración adicional por parte del usuario y ser utilizados como bloques individuales a usar dentro de proyectos complejos. Aunque *AWS* tiene un modelo *pay-as-you-go* en el que solo se paga por los recursos utilizados, también dispone de un *Free Tier* o *Nivel Gratuito* que a pesar de requerir el registro de un método de pago, ofrece acceso gratuito a todos los servicios *AWS* con unas cuotas máximas de uso mensual. *AWS* pone a disposición de sus usuarios más de 200 servicios, no obstante, son 5 los que van a tener importancia a lo largo de este proyecto.

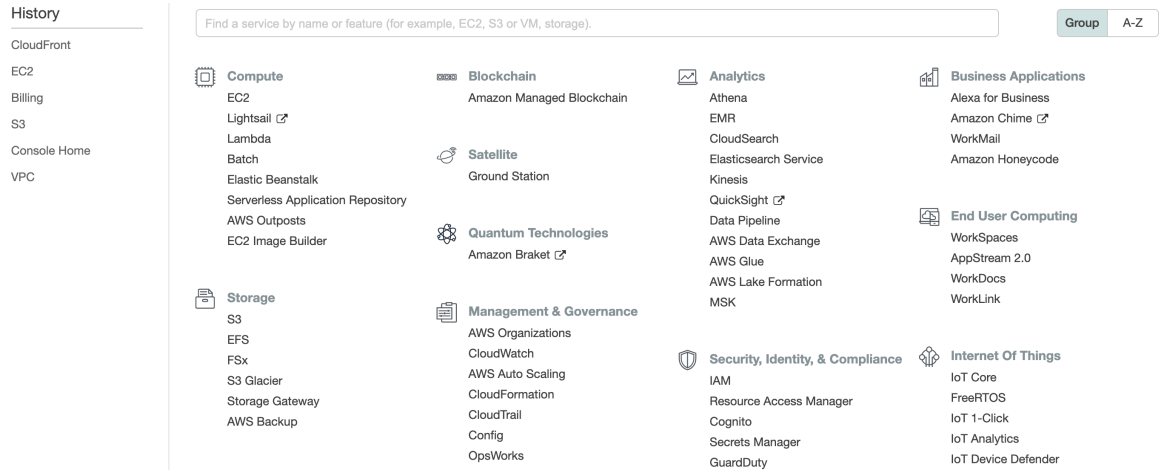


Figura 2.3: Algunos de los servicios disponibles en la consola de administración de *AWS*

- *Elastic Cloud Compute (EC2)* - Es un servicio web que proporciona capacidad de cómputo de manera segura y escalable en la nube. *EC2* permite crear máquinas virtuales con las especificaciones que cada usuario necesite y administrarlas desde una interfaz muy sencilla. El nivel gratuito de *AWS* proporciona 750 horas de cómputo mensuales al usuario en máquinas del tipo t2.micro, uno de los tipos de máquina virtual más modesto en términos de recursos. *AWS* no especifica como utilizar estas horas de cómputo, así que es posible tener una máquina encendida 24 horas al día durante un mes o tener 750 máquinas encendidas durante 1 hora.
- *Elastic IP* - Es la designación en el ecosistema *AWS* para direcciones *IP* estáticas. Cada usuario puede utilizar estas direcciones de manera gratuita siempre y cuando estén asociadas a máquinas en ejecución y no exista más de una dirección *IP* por máquina.
- *Simple Storage Service (S3)* - Es un servicio de almacenamiento de objetos que ofrece una disponibilidad, seguridad y rendimiento que pocos pueden igualar. En S3 los datos son almacenados como objetos dentro de recursos llamados *buckets*, cada objeto tiene *metadatos*, permisos y controles de acceso, análisis de *big data* y monitorización a nivel de objeto y de *bucket*. Es muy importante monitorizar el uso de almacenamiento debido a que el nivel gratuito de *AWS* garantiza 5 GB de almacenamiento estándar S3 y 20.000 peticiones *GET S3* cada mes; una crecida en popularidad de la aplicación puede agotar fácilmente las 20.000 peticiones y resultar en un corte del servicio o en cargos monetarios inesperados.
- *Cloudfront* - Una *red de distribución de contenido* o *CDN* es un grupo de servidores geográficamente distribuidos que trabajan conjuntamente para proporcionar una entrega rápida de contenido web independientemente del lugar de destino[33]. *Cloudfront* es un servicio *CDN* que entrega datos, vídeos, aplicaciones y *API*s a clientes en todo el mundo con bajas latencias, altas velocidades de transferencia y un entorno familiar para desarrolladores. *Cloudfront* está integrado con la infraestructura global de *AWS*, pudiendo así trabajar conjuntamente con otros servicios como *AWS Shield* para *mitigación DDoS*, *Amazon S3*, *Elastic Load Balancing* o *Amazon EC2*.
- *AWS Certificate Manager (ACM)* - Es un servicio que permite crear, administrar y desplegar certificados *Secure Sockets Layer/Transport Layer Security* (*SSL/TLS*) fácilmente para ser utilizados por otros servicios *AWS* y recursos internos asociados. Los certificados *SSL/TLS* son utilizados para asegurar comunicaciones de red y establecer la identidad de sitios web a través de internet y de recursos a través de redes privadas. ACM elimina el proceso manual de comprar, subir y renovar certificados *SSL/TLS* que, sean públicos o privados, si son generados mediante ACM y usados por servicios con integración ACM son gratuitos[41].

Capítulo 3

Desarrollo e Implementación

Tras analizar otros proyectos similares, se llegó a la conclusión de que *Node.js* sería la plataforma más apropiada debido a su amplio soporte, gran número de paquetes y dependencias disponibles y su fácil despliegue y configuración. Del mismo modo, se optó por utilizar *Typescript* como lenguaje de programación para agilizar la depuración y comprensión del código, como se ha explicado en Typescript. Por último, se escogió *Yarn* como sistema de gestión de paquetes por una mejor estabilidad y facilidad de uso, sin embargo, *npm* no es una mala elección en ningún caso; preferencia y experiencia personal suelen decantar a cada desarrollador por uno u otro.

La totalidad del desarrollo se ha realizado utilizando *Microsoft Visual Studio Code*[9], un *Integrated Development Environment* o *IDE* muy completo que cuenta con miles de extensiones, soporte para cualquier lenguaje y un sistema de terminales integrado. Es importante mencionar que la extensión *Remote - SSH*[27] ha permitido continuar el desarrollo en *Microsoft Visual Studio Code* a través de *SSH* incluso en máquinas virtuales sin interfaz gráfica.

Una vez instalado todo el *software* necesario, el proyecto comienza por la creación de su estructura. En primer lugar, se crea un repositorio en *GitHub*, se clona en local y se abre el directorio vacío con *Visual Studio Code*, donde será posible abrir una nueva terminal. Con '*yarn init*' se crear un archivo *package.json* que contendrá los detalles, *scripts* y dependencias de la aplicación, estas últimas serán instaladas en el directorio '*node_modules*', cuyo tamaño y cantidad de ficheros y subdirectorios puede abultarse significativamente conforme se añaden dependencias, por tanto, es muy importante excluir '*node_modules*' de control de versiones, compilación, monitorización, etc.

Typescript es una dependencia como otra cualquiera y para añadirla a la aplicación se utiliza '*yarn add typescript*'. *Node.js* hace una distinción entre *dependencias* y *dependencias de desarrollo*, por tanto, es importante definir qué dependencias serán necesarias en la versión de producción de la aplicación y cuáles en la de desarrollo. *Typescript* requiere un archivo de configuración, '*tsc –init*' genera un archivo *tsconfig.json* que contiene su configuración, es necesario cambiar ciertos parámetros como los directorios de entrada y salida, directorios excluidos, librerías a incluir y opciones de sintaxis, entre muchos otros.

Para agilizar el proceso, se usan *scripts* en *package.json*, son atajos para ejecutar comandos posiblemente largos y que se repiten mucho durante el desarrollo. Sin embargo, durante el desarrollo no es eficiente ejecutar los mismos comandos de compilación y ejecución, por muy cortos que sean, cada vez que se realice un cambio en el código de la aplicación.

```
1   "scripts": {
2      "build": "tsc --listFiles",
3      "bundle:dev": "cd frontend && npx webpack --mode development",
4      "bundle:prod": "cd frontend && npx webpack --mode production",
5      "cdn": "ts-node src/uploadStaticBucket.ts",
6      "coverage": "codecov",
7      "start": "NODE_ENV=PRODUCTION node dist/index.js",
8      "test": "nyc --reporter=lcov --reporter=text mocha -r ts-node/register src/test/**/*.test.ts",
9      "watch": "concurrently \"yarn watch:front\" \"yarn watch:back\"",
10     "watch:front": "cd frontend && npx webpack --mode development --watch",
11     "watch:back": "NODE_ENV=DEVELOPMENT nodemon src/index.ts"
12  }
```

Listado 3.1: *Scripts* resultantes del proyecto

La solución es utilizar un *watcher*, un programa que observe la base de código y compile y ejecute la aplicación automáticamente cada vez que observa un cambio. *Node.js* dispone de *nodemon* y para observar archivos *Typescript* (*.ts*) existe *ts-node*, ambos son añadidos como dependencias de desarrollo con *'yarn add -D nodemon ts-node'*.

Por otro lado, el Cliente Web utiliza *ReactJS* y *Typescript*, dado que va a convivir en el repositorio con el Servidor API REST, se crea un directorio *frontend* donde se ejecuta el comando *'yarn create react-app widget –template typescript'*, que produce un proyecto *ReactJS + Typescript* genérico.

Por último, tanto el *servidor API REST* como el *cliente web* van a utilizar algunos datos secretos que no pueden ser compartidos, como contraseñas, claves y secretos. La solución es utilizar *variables de entorno*, creando un archivo *.env* e instalando el paquete *dotenv* usando *'yarn add dotenv'*. Esto permite escribir todas las variables que no se desea compartir en el archivo *.env* y cargarlas al cuando comienza la ejecución de la aplicación. Es muy importante incluir *.env* en el archivo *.gitignore* para excluirlo del control de versiones, de lo contrario sería añadido al repositorio y compartido públicamente.

3.1. Arquitectura

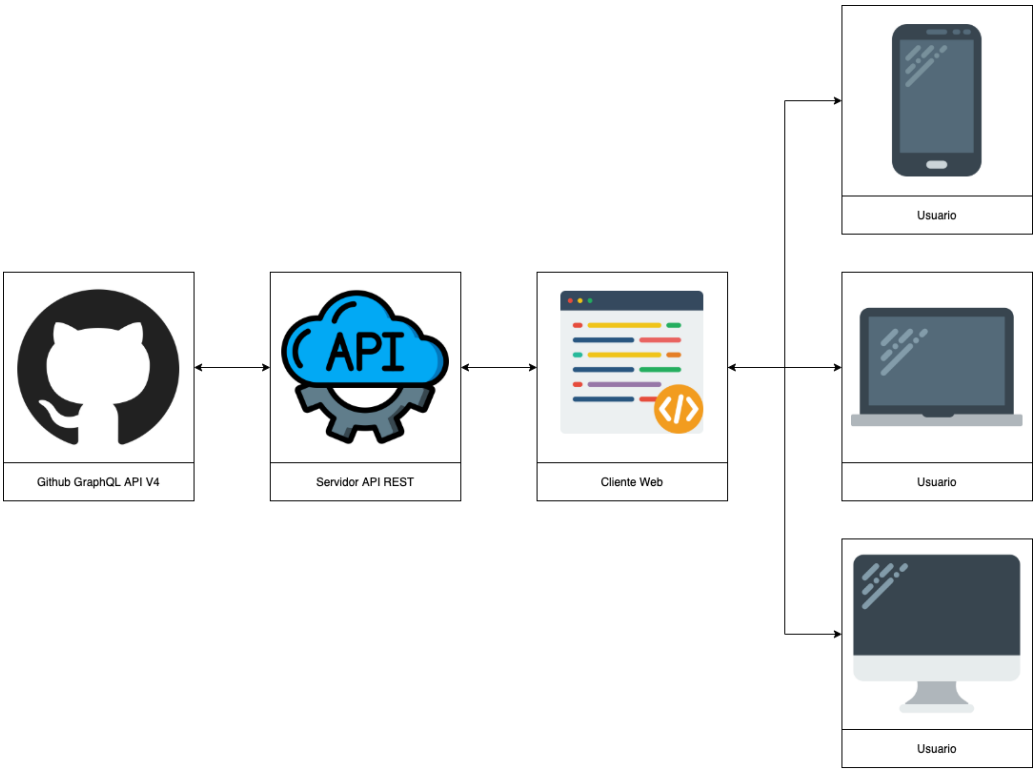


Figura 3.1: Arquitectura del proyecto

Para la implementación del proyecto se ha optado por una *arquitectura de microservicios*, que consiste en una colección de servicios pequeños y autónomos donde cada servicio es autocontenido e independiente y solo implementa una única funcionalidad[51]. Sin embargo, no existe una definición Cada servicio :

- Es pequeño, independiente y débilmente acoplado[49] de manera que un equipo reducido de desarrolladores puede trabajar y mantener un servicio.
- Tiene una base de código separada, que puede ser manejada por un equipo de desarrollo reducido.
- Puede ser desplegado independientemente, las actualizaciones no requieren volver a construir y desplegar toda la aplicación.
- Es responsable de que sus datos y estados persistan.
- Se comunica con otros servicios usando *APIs* bien definidas, de manera que los detalles internos de cada servicio no son relevantes para el resto.

- No necesita utilizar las mismas tecnologías, librerías o *frameworks* que los servicios con los que se comunica.

3.1.1. GitHub API

El eje central de este proyecto es *GitHub API*, la interfaz que ha construido *GitHub* para poder acceder a todos sus datos desde cualquier aplicación independientemente del lenguaje, *framework* o librería que utilice. La comunicación se realiza a través de peticiones que permiten ver, modificar, crear o eliminar *usuarios*, *organizaciones*, *repositorios*, *issues*, *comentarios*, *pull requests*, etcétera. La autenticación no es necesaria para ver información pública, pero si se desea realizar modificaciones será necesario autenticarse como usuario de *GitHub*. *GitHub* utiliza el estándar OAuth[20] para autenticar usuarios a través de su *API*, *OAuth* es un protocolo de autenticación que permite que una aplicación o servicio interactúe con otro en nombre del usuario sin tener que compartir su contraseña, en su lugar se comparten *authorization tokens* o *access tokens* para demostrar una identidad entre usuario y proveedor de servicios.

```
1  $ curl https://api.github.com/users/plaguera
```

Listado 3.2: Ejemplo *bash GitHub API* sin autenticación

Si se desea realizar una petición que requiere autenticación será necesario crear un *access token* en la sección de ajustes de desarrollador de *GitHub*. Por ejemplo, si se desea crear un comentario en un *issue*, habrá que añadir *encabezados* o *headers* a la petición para comunicar qué tipo de datos se están enviando, qué tipo se desea recibir y en nombre de quién se realiza, además de los datos que se van a enviar.

```
1    $ curl -X POST \
2           -H 'Authorization: token OAUTH-TOKEN' \
3           -H 'Accept: application/json\' \
4           -H 'Content-type: application/json' \
5           -d '{"body": "Este es un nuevo comentario"}'
6           https://api.github.com/repos/:owner/:repo/issues/:issue_number/comments
```

Listado 3.3: Ejemplo *bash GitHub API* con autenticación

No obstante, los ejemplos anteriores utilizan *GitHub REST API v3*, cuyas implicaciones serán explicadas en el Servidor API REST pero nótese como el usuario que hace la petición no tiene palabra sobre qué datos está pidiendo específicamente, es decir, tanto si el usuario esté interesado en un campo como en 10, *GitHub REST API v3* va a enviar la misma cantidad de información.

Por otro lado, *GitHub GraphQL API v4* ofrece significativamente más flexibilidad que su homólogo *REST* ya que posee la habilidad de definir exactamente los datos que el usuario desea, pudiendo así reemplazar múltiples peticiones *REST* con una sola petición *GraphQL* para recibir los datos que el usuario especifique.

3.1.2. Servidor API REST

El servidor que actúa como intermediario entre *GitHub API* y el cliente web, utiliza *GraphQL* y *API REST*, respectivamente, para comunicarse con ellos. Esta comunicación se realiza a través de *Fetch API*, que permite realizar peticiones *HTTP* a través de la red usando *Javascript*, y se basa en el uso de objetos *Request*, *Response* y *Promise*[13]. *Promise* es un *proxy* para un valor posiblemente desconocido en el momento en el que se creó, es decir, permite que métodos asíncronos devuelvan valores como si fueran síncronos en forma de un *Promise* del valor en vez tener que devolver el valor final en el momento en el que se ejecuta[25]. El cliente web va a hacer 4 tipos de peticiones al Servidor API REST que a su vez se las transmitirá a *GitHub API*.

- *GET Usuario Actual* - En el caso de que el usuario se haya autenticado con *GitHub* se devolverán sus datos. De lo contrario no se devuelve nada.
- *GET Usuario Específico* - Buscar los datos de un usuario específico.

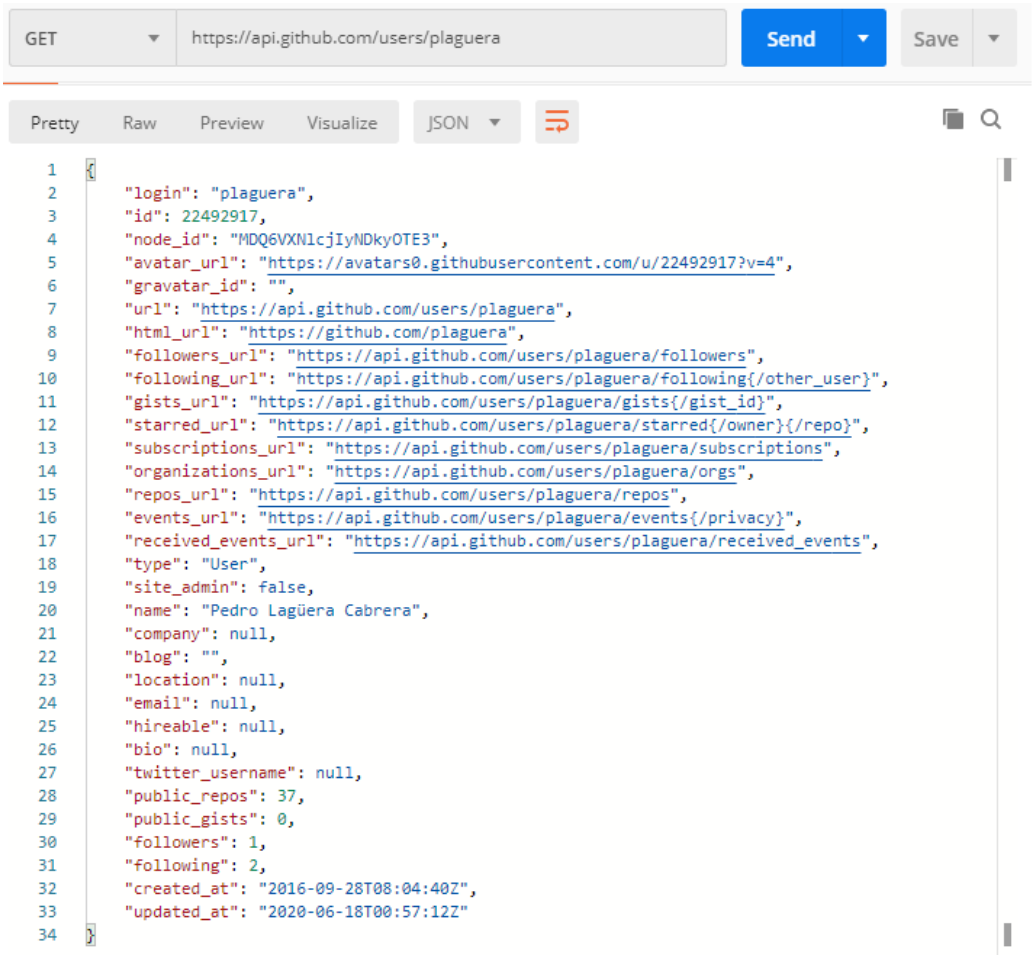


Figura 3.2: Ejemplo de petición y respuesta a *GitHub REST API* utilizando *Postman*[3]

- *GET Número de un Issue* - Buscar el número identificador de un *issue* específico a partir de su nombre.
- *GET Comentarios de un Issue* - Buscar los comentarios de un *issue* específico.
- *POST Comentarios en un Issue* - Crear un nuevo comentario en un *issue* específico.

Sin embargo, al principio del desarrollo del servidor se utilizó *GitHub REST API v3* para comunicarse con *GitHub API* por ser sencilla e intuitiva y agilizar el desarrollo. Como se ha mencionado anteriormente, *API REST* no permite que el usuario establezca la estructura de datos que desea recibir y por este motivo, cada vez que se carga una página web que incluya el cliente web con *N* comentarios se van a producir las siguientes peticiones.

- Una petición obligatoria para saber si el **usuario está identificado o no**, y en caso afirmativo obtener sus datos.
- Una petición que **devuelva el repositorio** que contiene el *issue* especificado.
- Una petición que **devuelva el issue** específico, incluyendo sus comentarios.
- N peticiones, una para **cada comentario** será necesaria para obtener la información de su autor.

```
1 GET https://api.github.com/user
2 GET https://api.github.com/users/:username
3 GET https://api.github.com/repos/:owner/:repo
4 GET https://api.github.com/repos/:owner/:repo/issues/:issue_number
5 GET https://api.github.com/users/:username X N Comentarios
```

Listado 3.4: Peticiones *REST* necesarias para cargar los comentarios de un repositorio

A pesar de que el rendimiento obtenido fue aceptable en una página web reducida en tamaño y componentes, surgieron dudas sobre el posible impacto de los tiempos de carga sobre una página que tuviese muchos componentes, servicios y peticiones. Por este motivo, y por el exceso de peticiones y de información desperdiciada, se tomó la decisión de aprender *GraphQL* y aprender a utilizar *GitHub GraphQL API v4* con el fin de mejorar la eficiencia y el rendimiento del cliente. *GraphQL* es completamente flexible y personalizable, solo se reciben los datos necesarios en una sola petición, de tal manera que se pasó de realizar 3 + N peticiones por carga de página a 2 peticiones.

```
{
  viewer {
    login
    avatarUrl(size: 40)
    url
  }
  repository(name: ":repo_name", owner: ":user_name") {
    createdAt
    issue(number: :issue_number) {
      comments(last: :page_size before :cursor) {
        pageInfo { startCursor }
        totalCount
        nodes {
          bodyHTML
          id
          authorAssociation
          author {
            avatarUrl
            login
            url
          }
          createdAt
          url
          viewerDidAuthor
        }
      }
    }
  }
}
```

Listado 3.5: Petición *GraphQL* necesaria para cargar los comentarios de un repositorio

- ***viewer*** - Información del usuario que realiza la petición, depende de las credenciales con las que se realice.
  - ***login*** - Nombre de usuario.
  - ***avatarUrl(size: 40)*** - Dirección web de la imagen de perfil del usuario en tamaño 40x40 píxeles.
  - ***url*** - Dirección web del perfil del usuario.
- ***repository*** - Repositorio que contiene los *issues*, identificado por su nombre y el nombre de su dueño.
  - ***createdAt*** - Fecha de creación del repositorio.
  - ***issue*** - *Issue* que contiene los comentarios, identificado por su *issue number*.
    - ***comments*** - Listado de comentarios del *issue*, para usar paginación es necesario especificar el tamaño de paginación y el cursor de inicio. El tamaño de paginación puede ir precedido de *last* o *first*, de manera que se eligen los primeros o últimos comentarios en el orden en el que han sido creados. El cursor de inicio representa el identificador de un comentario específico, es opcional y va precedido de *before* o *after*. Si no se especifica *before* o *after* y un cursor, se devuelven los últimos comentarios del *issue*, de lo contrario se devuelven los últimos comentarios del *issue* que preceden al comentario que indica el cursor.

- ***pageInfo*** - Contiene los cursores de inicio y fin, además de otros datos de paginación.
  - ***startCursor*** - Es el cursor del primer comentario que se va a devolver.
- ***totalCount*** - Número total de comentarios.
- ***nodes*** - Contiene todos los atributos necesarios de cada comentario, siempre es devuelto como una lista.
  - ***bodyHTML*** - Contenido del comentario, traducido de *GitHub Flavored Markdown*[19] a *HTML*.
  - ***id*** - Identificador del comentario, diferente al cursor.
  - ***authorAssociation*** - Rol del autor del comentario en el repositorio que contiene el *issue* que contiene su comentario.
  - ***author*** - Contiene todos los atributos necesarios de cada autor de cada comentario.
    - ***avatarUrl*** - Dirección de la imagen de perfil del usuario.
    - ***login*** - Nombre de usuario (*nickname*).
    - ***url*** - Dirección del perfil en *GitHub* del usuario.
  - ***createdAt*** - Fecha de creación del comentario
  - ***url*** - Dirección en *GitHub* del comentario.
  - ***viewerDidAuthor*** - Valor *booleano* que indica si el usuario que está solicitando esta información es el autor del comentario.

Una vez comprendido cómo se realizará la comunicación será necesario implementar el servidor web que utilice. *Express* es el *web framework* más popular de *Node.js* y es la librería subyacente de muchos otros web *frameworks* populares en *Node.js*. *Express* es la librería por defecto por demanda popular para la creación de servidores web y *APIs* en *Node.js*. Si bien, *Express* por sí mismo es bastante minimalista, gracias al uso de *middleware* se han creado paquetes para añadir todas las funcionalidades que un desarrollador podría necesitar en un servidor web. Los *middleware* más populares trabajan con *cookies*, sesiones, autenticación de usuario, parámetros *URL*, datos POST y encabezados de seguridad, entre muchos otros[12]. En términos simples, un *middleware* es un fragmento de código que puede ser acoplado en el ciclo de respuesta de un servidor web y tiene acceso a la petición recibida, la respuesta que se va a enviar y a la siguiente función que se ejecutará en el ciclo de respuesta. Este *servidor API REST* utilizará varios *middleware* para adquirir las funcionalidades necesarias para funcionar adecuadamente.

3.1.2.1. CORS

A pesar de que el usuario no lo nota, cada página web que visita hace muchas peticiones frecuentes para cargar recursos como imágenes, fuentes o estilos procedentes de muchos y variados lugares de internet. Si no se vigilan estas peticiones, la seguridad del navegador y de los datos que maneja puede verse comprometida. Estas preocupaciones de seguridad son el motivo por el que los navegadores web restringen las *peticiones HTTP de origen cruzado* realizadas desde *scripts*, y *XMLHttpRequest* y *Fetch API* utilizan la política *same-origin*, que solo permite peticiones de recursos alojados en el mismo origen.

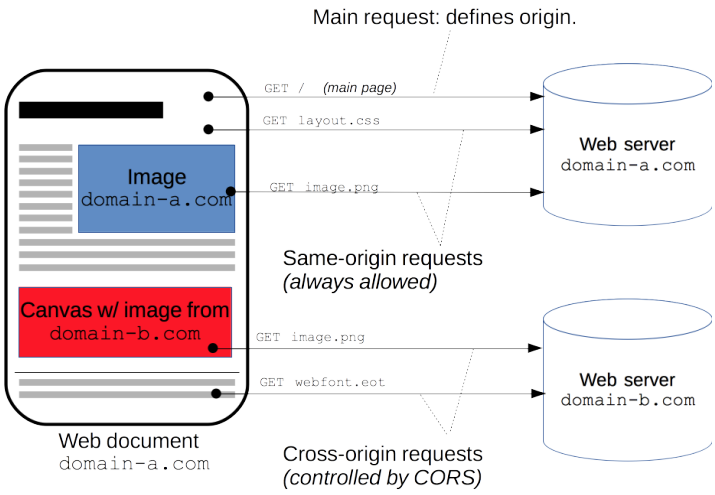


Figura 3.3: Ejemplo de funcionamiento de *CORS*[6]

Cross-Origin Resource Sharing o *CORS* es un mecanismo que utiliza encabezados *HTTP* adicionales para comunicar a un navegador web puede permitir que una aplicación web alojada en un determinado origen acceda a recursos en un origen diferente. Una aplicación web utiliza peticiones *HTTP cross-origin* o de *origen cruzado* cuando solicita un recurso que tiene un origen diferente al suyo (dominio, protocolo o puerto)[6]. CORS es un término medio que proporciona mayor libertad que las peticiones *same-origin* pero es más seguro que permitir todas las peticiones de origen cruzado.

En el caso de este proyecto, el cliente web puede ser utilizado por cualquier persona y como consecuencia, será integrado en páginas web alojadas en diferentes dominios, por ello, será necesario que el servidor utilice *CORS* para poder responder a las peticiones de todos los clientes independientemente del dominio donde se alojen.

3.1.2.2. Cookies

Más adelante se explicará el proceso de autenticación pero para que cada usuario que inicie sesión en el *cliente web* no tenga que volver a hacerlo cada vez que cierre la página, se utilizarán *sesiones* y *cookies*. Una *sesión* es el almacenamiento de información que debe persistir a lo largo de la interacción de un usuario (cliente) con una aplicación o página web (servidor)[31]. Dicho almacenamiento puede realizarse en una base de datos del servidor web o en el cliente gracias al uso de *cookies*.

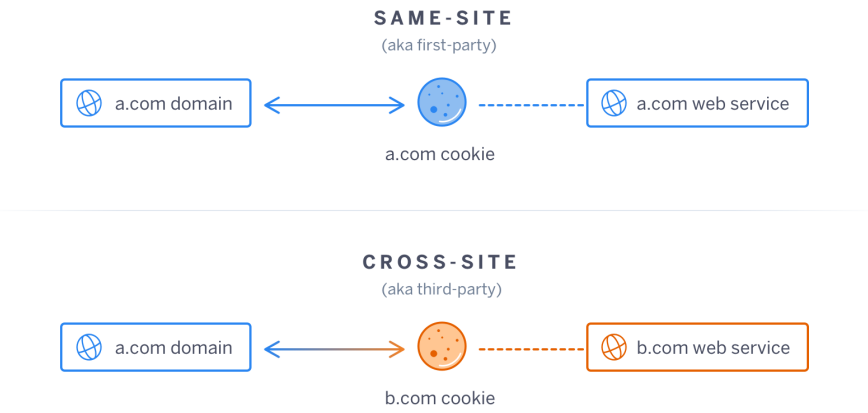


Figura 3.4: Comparación de tipos de *cookies*

Una *cookie* o *HTTP cookie* es una pequeña cantidad de información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del navegador. La actividad del usuario en una página web, los datos introducidos en un formulario y las credenciales de autenticación son los tipos de información más habituales en *cookies*[40]. Existen dos tipos de *cookies*, de *origen* (*first-party*) o de *terceros* (*third-party*), y aunque ambos tipos guardan la misma información, son leídos y creados de maneras diferentes. En este proyecto se utilizarán *cookies de autenticación* para que la autenticación de usuarios sea persistente, y dichas *cookies* serán de *terceros* debido a que el servidor web siempre estará en un dominio diferente al dominio de cada cliente web, el mismo motivo por el que se necesita utilizar *CORS* en el servidor.

Sin embargo, el uso de *cookies* entre dominios diferentes conlleva un riesgo muy importante llamado *CSRF*. *Cross-site request forgery* es una vulnerabilidad de seguridad web, si un usuario que haya iniciado sesión y tenga sus credenciales almacenadas en *cookies* hace click en un enlace aparentemente fiable, un desconocido puede aprovecharse de la sesión previamente autenticada y hacer peticiones fraudulentas a otras páginas en su nombre, sin poder distinguir una petición falsificada de una verdadera. Para regular el envío de *cookies* con terceros se utiliza el siguiente atributo a la hora de crear una *cookie*:

- *SameSite* - Permite al servidor decidir si una *cookie* puede ser enviada a través de *peticiones de origen cruzado* y puede tener los siguientes valores.
  - *Strict* - Prohíbe completamente el envío de *cookies* entre dominios diferentes.
  - *Lax* - Es igual que *strict* pero permite enviar *cookies* cuando se navega a una *URL* externa mediante un enlace y se utilizan métodos HTTP seguros como *GET*.

- *None* - Permite el envío de *cookies* en peticiones de origen cruzado.

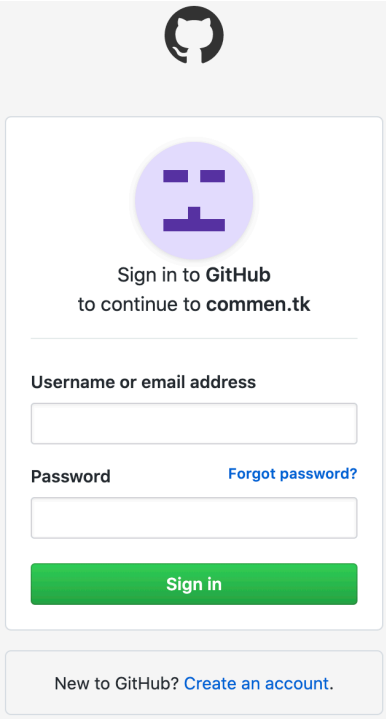
None fue el valor por defecto de *SameSite* hasta febrero de 2020, cuando *Chrome* decidió cambiarlo por *Lax* en su *versión 80*[34] y empezó a requerir que cualquier *cookie* que sea creada con el atributo '*SameSite: None*' debe incluir el atributo *Secure*. Este atributo solo permite que una *cookie* sea enviada cuando su petición se realiza mediante *HTTPS* y, al igual que el atributo *HTTPOnly*, impide el acceso a ella a través de *Javascript*[29].

3.1.2.3. Proceso de Autenticación

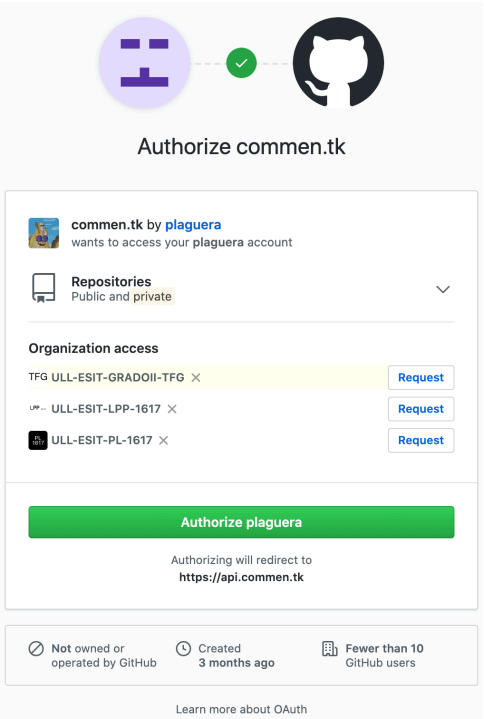
El proceso de autenticación, es el proceso que sigue el usuario en el *cliente web* para iniciar sesión como usuario de *GitHub*. Esta autenticación autoriza al servidor web a hacer peticiones a *GitHub API* en su nombre, siguiendo los pasos que indica *GitHub* para las aplicaciones que utilizan su *API* y *OAuth*[4].

1. El cliente hace una petición a la ruta '*/authorize*' del servidor. Este genera 64 bytes al azar y los convierte en un *string* que se denominará *estado* o *state*, que es usado como clave en un diccionario donde el valor es la dirección desde la que se realizó la petición. Por último, se redirige al cliente a la página de autorización de aplicaciones *oauth* en *GitHub* con los siguientes parámetros *query string*.
    - *client_id* - El identificador único de la aplicación *oauth* que utiliza el servidor.
    - *scope* - El contexto sobre el que el servidor podrá actuar en nombre del usuario, en este caso es *repo*, permitiendo lectura y escritura sobre repositorios, *issues* y comentarios.
    - *state* - El *string* aleatorio generado anteriormente usado para proteger contra ataques de *cross-site request forgery*.
2. El usuario introduce sus credenciales en la página de *GitHub*, autoriza al servidor a ver y crear comentarios en su nombre y es redirigido a la ruta '*/oauth/redirect*' en el servidor web. Esta redirección va acompañada por dos parámetros, el estado y un *código*. Si el estado recibido no coincide con el generado, la petición pertenece a un tercero. El *código* es un parámetro necesario para crear un *token de acceso* o *access token* que expira a los 10 minutos de crearse.
3. A continuación, se realiza una petición *POST* a la ruta de *GitHub* encargada de generar *tokens de acceso* con los siguientes parámetros *query string* y devuelve un *token de acceso*.
    - *client_id* - El identificador único de la aplicación *oauth* que utiliza el servidor.
    - *client_secret* - El secreto de la aplicación *oauth* que utiliza el servidor.
    - *code* - El código recibido.
    - *state* - El *string* aleatorio recibido que se usa para proteger contra ataques de *cross-site request forgery*.
4. Una vez obtenido el *token de acceso*, este no será guardado en el servidor sino en una *cookie* en el navegador del usuario. Aunque se utilizarán los atributos *SameSite* y *Secure* vistos en Cookies, no es recomendable utilizar *cookies* para guardar credenciales textualmente, por ello, cada *cookie* va a ser firmada (cifrada) en el servidor, es decir, el *token de acceso* contenido en la *cookie* es convertido en un *hash* utilizando un *secreto*. El *secreto* es un *string* muy grande generado aleatoriamente que es conservado entre ejecuciones del servidor, ya que, de lo contrario, cada vez que se reiniciase el servidor se invalidarían todas las *cookies* firmadas anteriormente.
5. Una vez creada y firmada la *cookie* con los atributos adecuados se busca la dirección que hizo la primera petición en el primer paso, usando el *estado* para identificarla y se redirige al usuario a su dirección original con una *cookie de acceso* que será válida durante 24 horas.

De este modo, el servidor web no almacena ninguna información, los datos de comentarios y usuarios son almacenados por *GitHub* y las credenciales de acceso son almacenadas por los navegadores de los usuarios, minimizando los requerimientos de almacenamiento del servidor web. Además, el navegador del usuario dispone de la *cookie de acceso*, pero no puede hacer nada con ella dado que está cifrada y no puede acceder a ella a través de *Javascript*, la *cookie* solo será usada al hacer una petición al servidor web usando el atributo '*Credentials: Include*'. De esta manera, el servidor web recibe la petición junto con la *cookie de acceso*, que después de ser descifrada, produce el *token de acceso* y lo utiliza para realizar sus propias peticiones a *GitHub* en nombre del usuario. Este proceso mantiene la información sensible (*token de acceso*) segura durante su transferencia y su almacenamiento.



(a) Inicio de sesión en *GitHub*



(b) Autorización del servidor para acceder a *GitHub*

Figura 3.5: Proceso de autenticación del usuario en *GitHub*

3.1.2.4. Proceso de Autenticación como Instalación de GitHub App

A diferencia de *GitHub REST API v3*, que permite realizar peticiones anónimas de lectura,*GitHub GraphQL API v4* requiere autenticación para realizar cualquier consulta, independientemente de si se trata de lectura o escritura. Ello supone un problema dado que cada vez que un usuario cargue el *cliente web* sin haber iniciado sesión, el servidor no tendrá credenciales con las que pedir los datos que necesita. La solución utilizada al comienzo del desarrollo fue la más simple, se creó un *personal access token* para ser usado como credenciales por defecto en el caso de que el cliente no las proporcione. Esta estrategia funcionó durante el desarrollo, pero no es buena práctica utilizarla en la fase de producción ya que limita el número de accesos al *cliente web* a 6,000 por hora y existe una alternativa mejor.

Una *aplicación GitHub* o *GitHub App* es un programa que puede ser instalado y utilizado en usuarios, organizaciones y repositorios de *GitHub*. Una *GitHub App* actúa usando su propio nombre, interactuando con *GitHub API* usando su propia identidad. Cada *GitHub App* tiene su propio *OAuth*, *webhooks* y permisos precisos para poder elegir únicamente los tipos de datos que se van a manipular.

Por este motivo se ha creado una *GitHub App* denominada **commentk** que tiene permisos de lectura y escritura sobre *issues* y de lectura de *metadatos*[2]. Así, cuando un usuario instale la *GitHub App commentk* en un repositorio, esta podrá manipular sus *issues* y leer *metadatos* tomando el rol de *instalación*. Al igual que el servidor web usa *OAuth* para realizar acciones en nombre de un usuario que se haya autenticado en *GitHub*, el servidor web puede utilizar la instalación de la *GitHub App* para generar credenciales temporales en nombre de la propia *GitHub App*. Su generación se realiza en los siguientes pasos.

1. El *cliente web* hace una petición al *servidor API REST* para que devuelva información de un determinado *issue* mediante *GitHub API*, el repositorio que contiene este *issue* deberá haber instalado la *GitHub App commentk*. Dado que se necesitan credenciales para realizar peticiones, se comprobará la existencia de *cookies de acceso*, si no existe una *cookie de acceso personal* creada mediante el Proceso de Autenticación y no existe una *cookie de acceso de instalación*, se procederá a crear una nueva *cookie de acceso de instalación* a partir de un *token de acceso de instalación*.

2. El paquete *npm '@ocktokit/app'* permite recibir *tokens* para *GitHub Apps* y sus instalaciones[21]. A partir del identificador y la clave privada de la *GitHub App commentk* se crea un objeto *App* que va a generar los *tokens* necesarios. En primer lugar, se crea un *JSON Web Token* o *JWT* para solicitar

el *installation id* o *identificador de instalación* del repositorio que contiene el *issue* en cuestión. Usando el *installation id* y el objeto *App* se solicita un *installation access token* o *token de acceso de instalación* que tendrá una validez de 10 minutos.

3. Una vez obtenido el *token*, se realiza el mismo proceso que se ha usado en Proceso de Autenticación para crear una *cookie de acceso*. Esta *cookie* se denominará *cookie de acceso de instalación* y tendrá los mismos parámetros y garantías de seguridad y cifrado que la *cookie de acceso personal*, excepto su validez de 10 minutos.

De esta manera, cuando un *cliente web* se comunica por primera vez con el *servidor API REST*, se generará una *cookie de acceso de instalación* que le permitirá comunicarse con *GitHub API* aunque no haya iniciado sesión como usuario de *GitHub*.

3.1.2.5. Rutas API REST

Gracias al uso de *GraphQL*, se ha podido simplificar la *API REST* y solo son necesarias 6 rutas en el servidor, 4 para atender peticiones de datos y 2 que pertenecer al proceso de autenticación. Los elementos que estén contenidos entre los caracteres '{' y '}' serán parámetros que deberán ser sustituidas por los valores deseados.

- ***/user*** - No tiene argumentos, comprueba la *cookie de acceso* del usuario que realiza la petición y devuelve sus datos. Esta *cookie* acompaña a la petición automáticamente, como se ha explicado en el Proceso de Autenticación.
- ***/users/{login}*** - Devuelve los datos un usuario cuyo *login* coincide con el especificado.
- ***/issuenumber/{owner}/{repo}/{name}*** - A partir del nombre, repositorio y dueño de un *issue* devuelve el número del *issue*.
- ***/comments/{owner}/{repo}/{number}*** - Si recibe una petición *GET*, devuelve la lista de comentarios de un *issue* especificando el número del *issue*, el nombre del repositorio que lo contiene y el nombre de su dueño. El objeto devuelto tiene la estructura *GraphQL* explicada en Servidor API REST. Si recibe una petición *POST*, crear un nuevo comentario en el *issue* especificado utilizando las credenciales que acompañen a la petición. Cuando el tipo de petición es *GET* se aceptan los siguientes parámetros opcionales *query*, es decir, presentes en la propia dirección *URL* de la petición, para permitir la paginación de los comentarios.
  - *pagesize* - Permite especificar el número de comentarios que se desea mostrar.
  - *cursor* - Permite especificar el comentario que sucede al resto de comentarios que se desea mostrar, es decir, los *pagesize* comentarios que preceden al comentario al que apunta el cursor.
- ***/authorize*** - Ruta a la que accede el usuario al hacer click sobre el botón de inicio de sesión y que le redirige a la página de inicio de sesión de *GitHub*.
- ***/oauth/redirect*** - Ruta a la que se redirige al usuario una vez completado el inicio de sesión en *GitHub*. Se ocupa de crear la *cookie* del usuario y de redirigirle a su página web original.

3.1.3. Cliente Web

El *cliente web* es una herramienta que muestra los datos que recibe del servidor Servidor API REST de forma similar a los comentarios de *issues* en *GitHub*. Este cliente es un *embeddable widget*, un componente que permite al usuario utilizar una funcionalidad adicional sin nuevas dependencias ni procesos complicados.

Una página web es un documento que puede ser mostrado en un navegador o como código fuente *HTML*, pero es el mismo documento en ambos casos. *Document Object Model* o *DOM* es una interfaz de programación para documentos *HTML* y *XML* que representa una página de tal manera que otros programas puedan manipular su estructura, estilo y contenido[15]. Así, lenguajes como *Javascript* pueden modificar elementos o nodos dentro de una página web. La manera de ejecutar código *Javascript* externo desde código *HTML*, que es un *markup language* o *lenguaje basado en etiquetas*, es usando la

etiqueta *<script>*. En términos de implementación, un *widget embebible* es archivo de código externo que al ser ejecutado en un archivo *HTML*, añade componentes y funcionalidades al documento.

Al principio del desarrollo del cliente web se optó por utilizar *Vanilla Javascript*, es decir, solo *Javascript* sin librerías, *frameworks* ni dependencias. El razonamiento detrás de esta decisión fue que evitando el uso de software adicional externo se mantendrían al mínimo posible tanto el tamaño como el coste computacional del programa resultante. Sin embargo, a medida que se añadían funcionalidades y componentes, la claridad del código iba degradándose, dificultando la búsqueda y resolución de errores, la expansión del código y la adición de nuevos componentes.

3.1.3.1. ReactJS

Por estos motivos se decidió cambiar de ruta y utilizar un *framework* y un lenguaje de programación distinto, decidiendo que el *overhead* adicional de computación y tamaño que supone usar un *framework* es aceptable dadas las ventajas en facilidad de depuración, mantenimiento y expansión que proporciona. El *framework* escogido es ReactJS, una librería diseñada para crear interfaces gráficas, específicamente para *Single-Page Applications*. *ReactJS* permite subdividir una página en componentes, el uso de componentes aumenta la reusabilidad del código, mejora su claridad y permite encontrar errores con mayor facilidad. Asimismo, dado que Typescript ya fue utilizado para implementar el Servidor API REST y puestos a mejorar la claridad del código, se determinó que también se usaría para escribir el código fuente de esta parte del proyecto.

ReactJS cuenta con dos tipos de modelos de datos, *estados* y *props*. *Props* son argumentos que pueden pasarse a componentes en el momento que se crean y por tanto son una manera de transmitir información de un componente padre a sus hijos. *Estados* son datos pertenecientes a un componente cuyo valor podrá cambiar a lo largo de su vida y en el momento que lo haga, provocará que se vuelva a renderizar el componente que los muestra para mostrar el valor actualizado. Por último, es necesario saber cuándo un componente entra en determinadas fases de su *ciclo de vida*, el momento que se crea, renderiza, actualiza o borra es significativo y por ello existen métodos son llamados cuando ocurren y ejecutan código del usuario, por ejemplo, es muy común pedir los primeros valores de los estados cuando un componente se crea a través de la función *ComponentDidMount*.

3.1.3.2. Webpack

Ambas implementaciones, *Javascript* y *Typescript + ReactJS*, comparten el hecho de que están compuestas por múltiples ficheros, clases, funciones y variables. Sin embargo, no es factible enviar cada archivo individualmente cada vez que se muestre el cliente web ni es posible que se ejecuten en orden, aunque sean recibidos por el usuario. Por lo tanto, será necesario combinar todos los archivos en un solo ejecutable *Javascript* que sí pueda ser compartido y ejecutado. *Webpack* es un empaquetador de módulos estáticos para aplicaciones *Javascript*, cuando *Webpack* procesa una aplicación, primero calcula un grafo interno de dependencias para mapear todos los módulos necesarios y después genera un *bundle*, un único archivo equivalente a la combinación de todos los que lo componen.

En Configuración *WebpackJS*, se muestra la configuración de *Webpack* para empaquetar el *cliente web*. *Loaders* son paquetes que permiten a *Webpack* cargar archivos que no sean *Javascript*, en este caso se han utilizado *loaders* para cargar archivos *Typescript, CSS y SASS*. Sin embargo, *Webpack* no soporta generar *bundles CSS*, solo *Javascript*, ello representa un problema que es explicado en Temas, que se soluciona utilizando dos *loaders* para generar cada tema en un *bundle CSS* separado y *minificarlo*.

Preact es una librería *Javascript* que se presenta como una alternativa más rápida a *ReactJS*. *Preact* tiene exactamente las mismas funcionalidades que *ReactJS*, pero lo consigue con un peso de 3.3 *kB* después de ser comprimido con *gzip* frente a 45 *kB* de *ReactJS*. Su reducido tamaño y su simple y predecible implementación han establecido a *Preact* como una de las librerías *Virtual DOM* más rápidas en la actualidad[23].

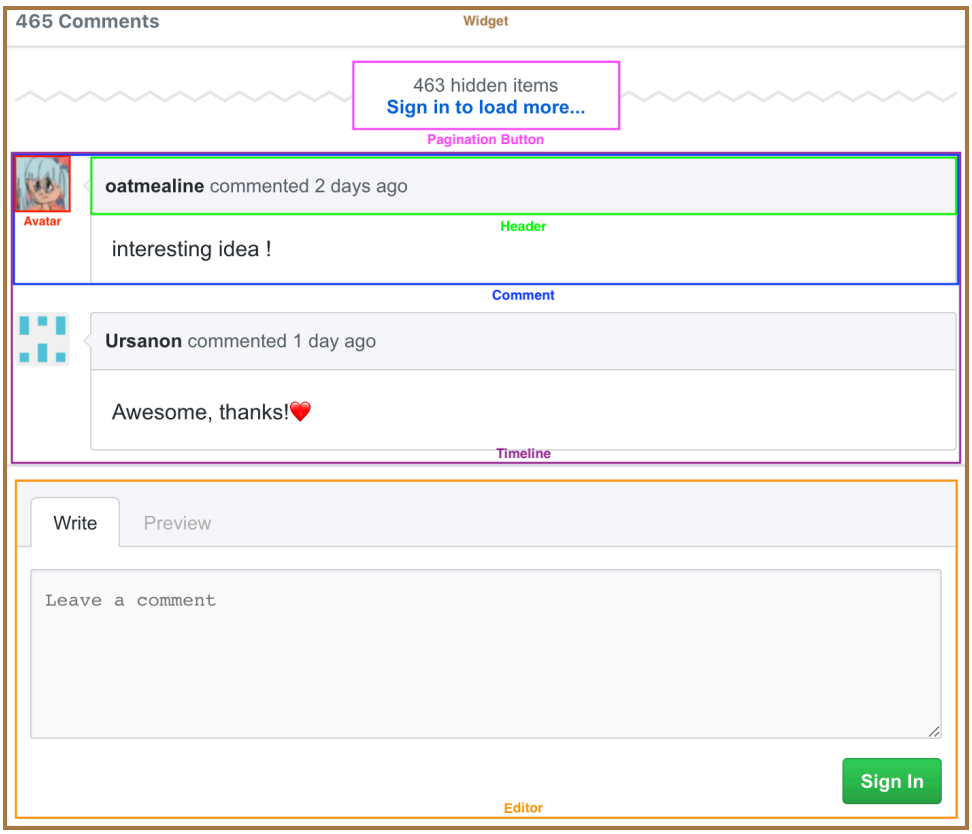


Figura 3.6: Esquema de componentes del *cliente web*

3.1.3.3. Componentes

1. ***Avatar*** - Es un pequeño componente que muestra la imagen de perfil del autor de un comentario o del usuario que haya iniciado sesión. También contiene un enlace al perfil en *GitHub* del autor.
2. ***Comment Header Label*** - *GitHub* especifica en cada comentario el rol que tiene el autor en el repositorio (si lo tiene), por ejemplo, colaborador, miembro o dueño.
3. ***Header*** - Es el encabezado de cada comentario, contiene el nombre de su autor, la fecha en la que se creó y su color cambia si el usuario que lo está viendo es su autor. El nombre del autor también es un enlace a su perfil en *GitHub* y la fecha de creación, que se muestra como tiempo transcurrido, es un enlace al comentario dentro de *issue* en *GitHub*.
4. ***Comment*** - Es una componente que usa los 3 anteriores y añade un cuerpo o *body* debajo del encabezado donde se muestra el contenido del comentario, dicho contenido puede estar escrito en *Markdown* y será mostrado con los estilos adecuados incluyendo enlaces e imágenes.
5. ***Editor*** - Permite escribir comentarios en *Markdown*, previsualizarlos y crearlos. En caso de no haber iniciado sesión, el área de escritura y la pestaña de previsualización estarán bloqueadas, de lo contrario están disponibles y se mostrará la imagen de perfil de usuario que ha sido autenticado.
6. ***Pagination Button*** - Se muestra solo si no se han cargado todos los comentarios disponibles, y permite cargar un número de los siguientes comentarios más antiguos que depende del tamaño de página que haya elegido el usuario.
7. ***Timeline*** - Contiene una lista de todos los comentarios.
8. ***Widget*** - Es el componente principal que se encarga de pedir la información del usuario que se encuentra autenticado (si lo hay) y de los comentarios del repositorio que debe mostrar. Contiene *Pagination Button*, *Timeline* y *Editor*.

Figura 3.7: Esquema de componentes de un comentario del *cliente web*

3.1.3.4. Paginación

No es eficiente ni necesario mostrar todos los comentarios cada vez que se cargue el cliente web, por eso se ha implementado un mecanismo de *paginación*. Como se ha explicado en Servidor API REST, al cargar el cliente web, este devuelve los *page_size* últimos comentarios de un *issue*, además de un cursor que apunta al primero. Si el usuario quiere ver comentarios más antiguos se realizará una petición que pida los *page_size* comentarios más recientes que precedan al comentario que indica el cursor. Gracias al uso de *ReactJS* no es necesario recargar la página cada vez que se cargan más comentarios, sin embargo, sí es necesario iniciar sesión para poder hacerlo y existen limitaciones en el tamaño de página siendo 100 el máximo, 1 el mínimo y 10 el valor por defecto.

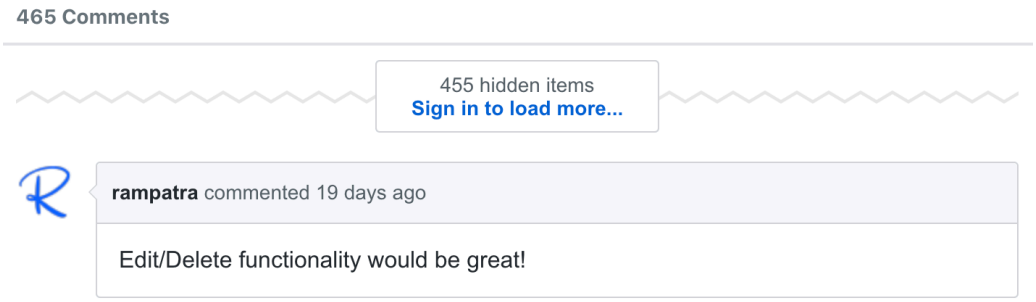


Figura 3.8: Botón *Cargar más comentarios*

3.1.3.5. Temas

Para preservar la apariencia de los comentarios en el cliente web se ha emulado el estilo de *GitHub*. *Primer* es el sistema de diseño de *GitHub*, es completamente *open-source* y proporciona estilos *CSS*, componentes *ReactJS*, iconos y diapositivas para añadir el estilo de *GitHub* en cualquier aplicación o página web[24]. Sin embargo, se ha optado crear el estilo del cliente web desde cero en lugar de utilizar *@primer/css* con el fin de tener un mejor control del funcionamiento del estilo y de reducir el tamaño de las hojas de estilo al mínimo necesario.

Todas las hojas de estilo han sido escritas en *SASS*, un lenguaje que extiende *CSS* y que utiliza un preprocesador para convertir el código *SASS* en código *CSS*[28]. Sus características más atractivas son las siguientes.

- ***Soporte de anidación de* selectores** - *HTML* permite anidar etiquetas dentro de otras etiquetas, reduciendo sustancialmente la cantidad de código repetido que sería necesario si fuera como *CSS*, que no lo permite. *SASS* ofrece la posibilidad de anidar *selectores*, mejorando la calidad del código y facilitando su comprensión.
- ***Soporte de variables*** - Aunque *CSS* usa variables, estas son evaluadas en el tiempo de ejecución en el navegador y no son soportadas por algunos navegadores más antiguos. Las variables en *SASS* son reemplazadas por sus valores cuando el preprocesador *SASS* genera los ficheros *CSS*, eliminando así cualquier duda sobre posibles incompatibilidades[47].
- ***Soporte de módulos e importaciones*** - *SASS* permite repartir el código en varios archivos e importarlos, reduciendo la posibilidad de código duplicado o redundante.

La posibilidad no solo de elegir el tema del cliente web, sino de facilitar la creación de nuevos temas ha sido el objetivo final de la parte de estilos del proyecto. Gracias a los módulos e importaciones de *SASS* se ha creado un estilo base para cada uno de los componentes donde el valor de cada atributo de color ha sido asociado con una variable. Cada archivo base contiene el estilo de un componente y cada uno de ellos es importado en un SASS - Archivo Principal que los combina. Para crear un tema solo es necesario crear un archivo que importe el archivo principal y asigne valores a las variables de los colores, usando esta estructura se ha creado un tema *light* y otro *dark* para el cliente web, cuyo código se puede ver en SASS - Tema Oscuro.

Como se ha explicado en Webpack, todos los archivos del cliente web son combinados en uno único archivo. Sin embargo, debido a la posibilidad de elegir un tema, ha sido necesario excluir al estilo del *bundle webpack*, ya que no es eficiente empaquetar todos los temas en un solo archivo si sólo se podrá usar uno en un determinado momento, por ello, los estilos son procesados y guardados como ficheros *CSS* separados del resto.

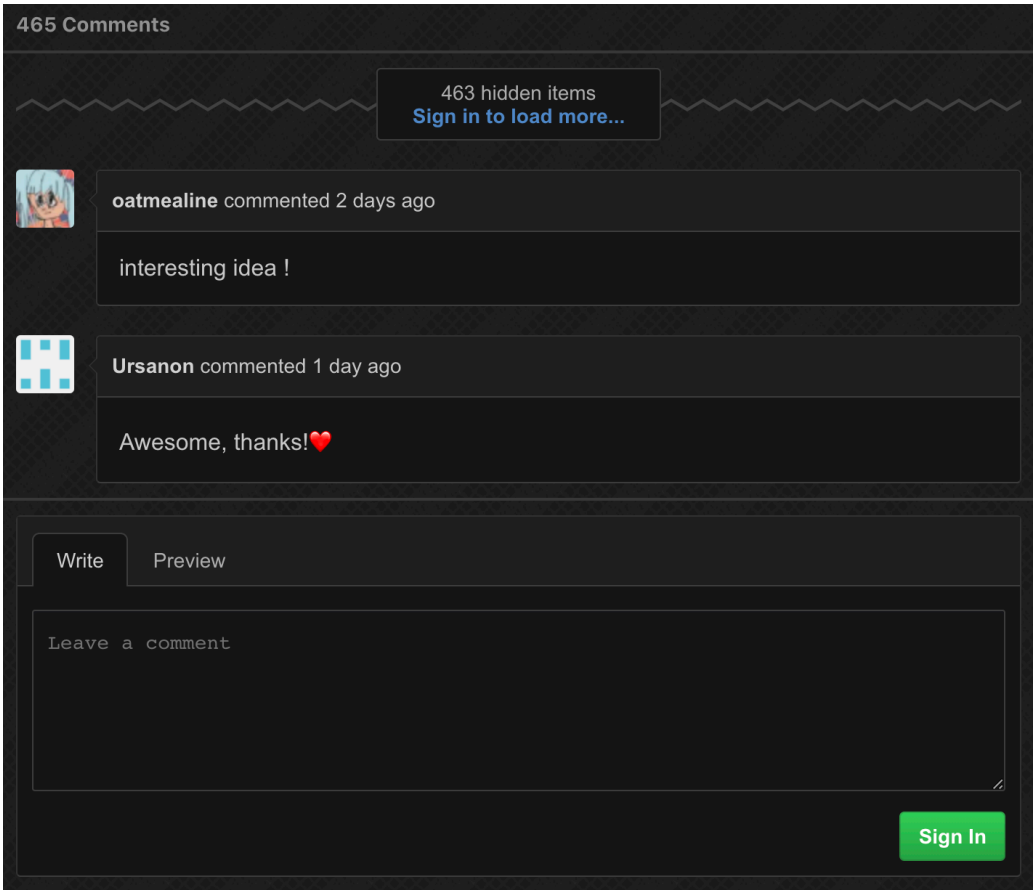


Figura 3.9: Tema oscuro

3.1.3.6. Uso

Para utilizar el cliente web en una página, es necesario instalar **commentk** en el repositorio deseado y añadir la siguiente etiqueta en el lugar específico donde se quiera colocar la sección de comentarios.

```
<script type='text/javascript'
    src='https://cdn.commen.tk/client.js'
    repo='[owner]/[repo]'
    issue-param='title'
    theme='light'
    page-size='10'
    async
></script>
```

Listado 3.6: Etiqueta *HTML* que genera una sección de comentarios

- **type** - *Obligatorio*. Especifica el tipo de contenido que se va a importar.
- **src** - *Obligatorio*. Especifica el archivo que se va a importar.
- **repo** - *Obligatorio*. Especifica el repositorio que se desea utilizar, siguiendo el esquema '*dueño/repositorio*'.
- **issue-param** - *Obligatorio*. No se puede escribir el nombre literal que tiene el *issue*, se debe especificar el parámetro que se utilizará como nombre. Puede ser:
  - *title* - Utiliza el título de la página donde se encuentra el *cliente web* como nombre del *issue*.
  - *hostname* - Utiliza el *hostname* o *nombre de host* de la *URL* de la página donde se encuentra el *cliente web* como nombre del *issue*.
  - *pathname* - Utiliza la ruta de la *URL* de la página donde se encuentra el *cliente web* como nombre del *issue*.
- **issue-number** - *Obligatorio*. Especifica el *número* del *issue* que se desea mostrar. Es mutuamente excluyente con *issue-param*, es decir, si se usa uno no se podrá usar el otro.
- **theme** - *Opcional*. Permite elegir el tema del cliente web, su valor por defecto es *light*.
- **page-size** - *Opcional*. Permite elegir el tamaño de página del cliente web, su valor por defecto es 10.
- **async** - *Opcional*. Es recomendable usarlo dado que, sin él, el navegador bloqueará el renderizado de los contenidos que suceden al cliente web hasta que este sea cargado por completo.

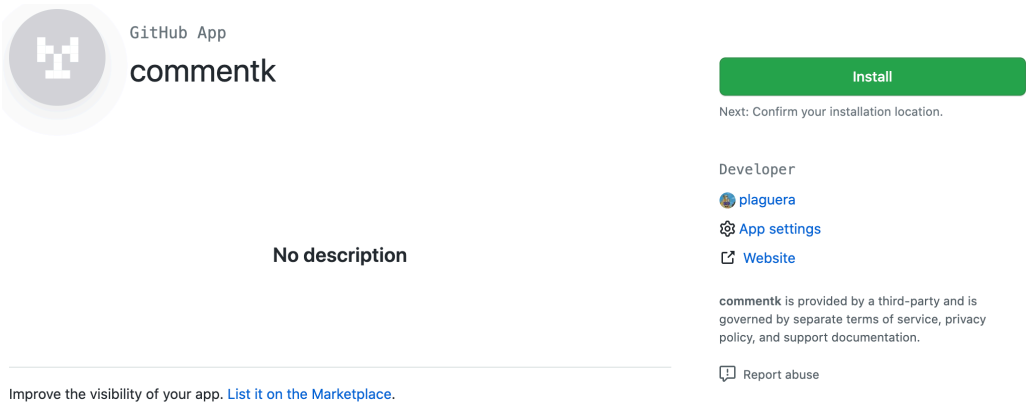


Figura 3.10: Página de instalación de *commentk*

3.1.3.7. Especificación de *Issue*

Para facilitar el uso del *cliente web*, no es necesario introducir el *número* del *issue* que se desea mostrar. Se puede indicar al *cliente web* que utilice el *title*, *hostname* o *pathname* de la página para identificar el *issue*. Este *issue* no necesita existir, en este caso el *servidor API REST* lo crea automáticamente utilizando la identidad de la *GitHub App* y se muestra la etiqueta *bot* para indicar que se ha sido creado por una *GitHub App*.

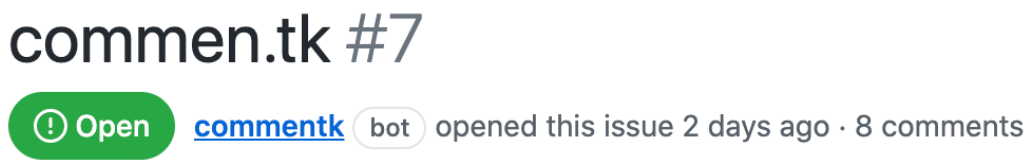


Figura 3.11: *Issue* creado automáticamente por *GitHub App commentk*

Sin embargo, debido a que los *issues* están indexados por su *número* en *GitHub API*, siempre que se especifique un *issue* en el *cliente web* mediante su nombre, será necesario hacer una búsqueda en *GitHub API* para conocer su *número* cada vez que se cargue el *cliente web*. Por este motivo siempre que se utilice el atributo *issue-number*, el *cliente web* cargará más rápido que usando *issue-param*.

3.2. Despliegue

La sencillez mencionada en Heroku es el motivo por el que se ha usado *Heroku* para el despliegue de las versiones de producción del servicio durante la mayor parte de su desarrollo. A pesar de no ser la estrategia ideal pero con el fin de agilizar el proceso de desarrollo, todos los despliegues en *Heroku* han constado de una sola instancia que escucha peticiones para la *API* y sirve archivos estáticos simultáneamente.

Una vez conseguidos todos los objetivos del desarrollo del proyecto, llegó la hora de pensar en el despliegue definitivo del proyecto. Desde el principio, se partió de la idea de tener un dominio propio donde alojar una página web que explique el proyecto, y subdominios que alojen los servicios desarrollados.

En la mayoría de *Top-Level Domains* o *TLDs* más utilizados, los dominios no son particularmente caros (12€ - 50€ / año) siempre y cuando no sean palabras especiales. Sin embargo, uno de los objetivos que se han añadido durante el desarrollo ha sido mantener el coste del despliegue a 0. Para conseguir un dominio gratis se ha utilizado el servicio *Freenom*, que ofrece dominios gratuitos en:

- *.tk – Tokelau*
- *.ml – Mali*
- *.ga – Gabón*
- *.cf – República Centroafricana*
- *.gq – Guinea Ecuatorial*

Dado que cualquier palabra del diccionario se considera especial y por tanto de pago, se ha intentado conseguir un dominio corto, significativo y gratuito, optando finalmente por *commen.tk*. Una vez adquirido el dominio es posible administrar sus *Nameservers*, *URL Forwarding* y *Registros DNS*[16].

El siguiente paso es obtener tres máquinas con direcciones *IP* públicas y estáticas, para servir los archivos del *cliente web*, atender las peticiones *API* y servir la página web del proyecto, respectivamente. Aunque no existen muchas opciones gratuitas que ofrezcan estos servicios, la opción que destaca sobre el resto es *Amazon Web Services* (*AWS*), ya que dispone de un *Free Tier* o *nivel gratuito* que a pesar de que requiere un método de pago, ofrece una cantidad y calidad de servicios tan extensa que lo compensa con creces.

El primer servicio de *Amazon Web Services* que se ha utilizado es *Elastic Compute Cloud*; mejor conocido como *EC2*, es un servicio web que provee poder de computación en la nube de manera segura y flexible, permitiendo crear máquinas virtuales con los recursos que el usuario desee. Cada cuenta de usuario gratuita posee 750 horas de cómputo al mes libres de coste en instancias *Linux t2.micro* y cada usuario elige como gastarlas, se puede tener una máquina virtual encendida todo el mes o 750 máquinas encendidas simultáneamente durante una hora. Ello supone un problema dado que el despliegue del proyecto requiere tres máquinas independientes y *AWS* solo permitiría usar tres máquinas simultáneamente durante 250 horas gratuitamente. Afortunadamente, tanto la página web como los archivos del *cliente web* son estáticos, lo que amplía considerablemente las opciones disponibles para servir dichos archivos en servicios gratuitos.

3.2.1. GitHub Pages

GitHub Pages es un servicio gratuito que permite crear páginas web desde cero o generar una para un usuario, organización o proyecto en *GitHub*. Para crear la página web del proyecto se ha utilizado *Node.js* para usar *SASS* y poder automatizar la generación de los archivos estáticos con *Parcel-Bundler*[1]. De esta manera los archivos fuente están en la raíz del repositorio y los archivos estáticos que se sirven como página web están dentro del directorio '*docs*'.

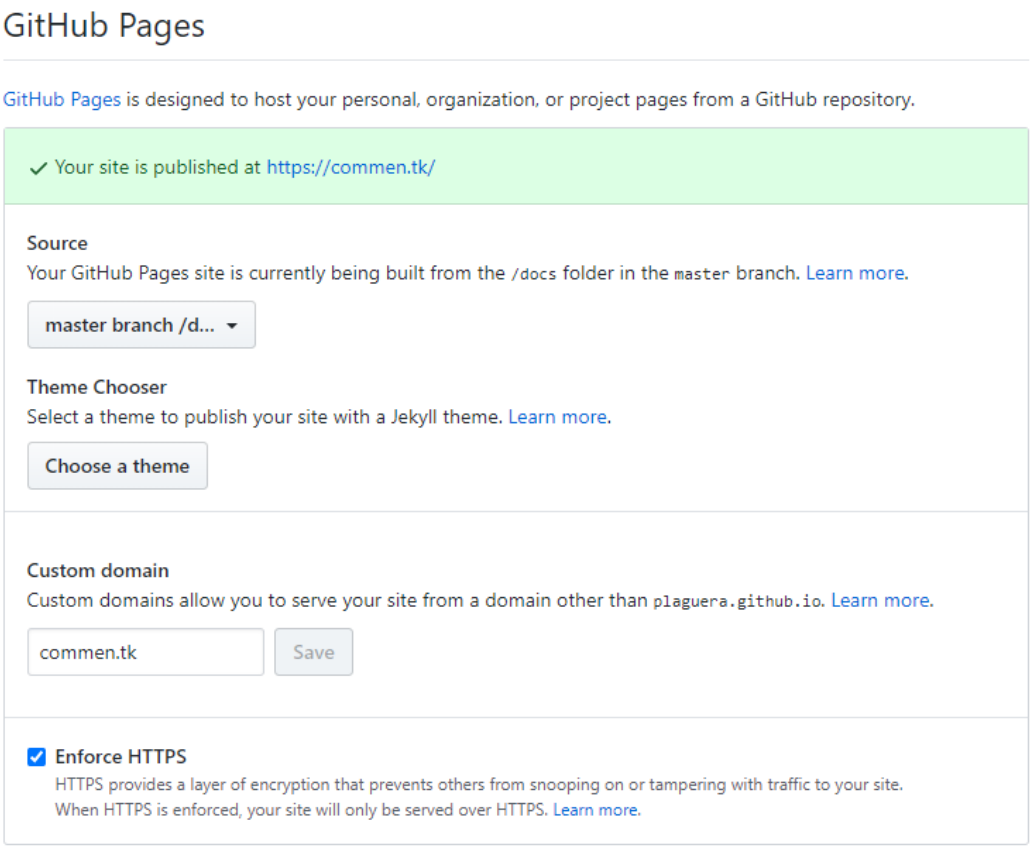


Figura 3.12: Configuración de *GitHub Pages*

La configuración de *GitHub Pages* es muy simple, '*Source*' es '*master branch /docs folder*' y en este caso '*Custom Domain*' es '*commen.tk*', sin embargo, antes de poder usar un dominio propio es necesario seguir la guía de *GitHub* sobre cómo utilizar dominios propios, más concretamente, un dominio *ápice* o *apex*[17]. Solo es necesario crear 4 registros *DNS* de tipo *A* que apunten la raíz del dominio hacia los servidores de *GitHub Pages* y apuntar el subdominio '*www*' a la raíz del dominio con un registro *DNS* de tipo *CNAME*. Después de que se propaguen los cambios en *DNS*, *GitHub* genera el *certificado SSL* y se podrá acceder a la página web mediante *HTTPS*.

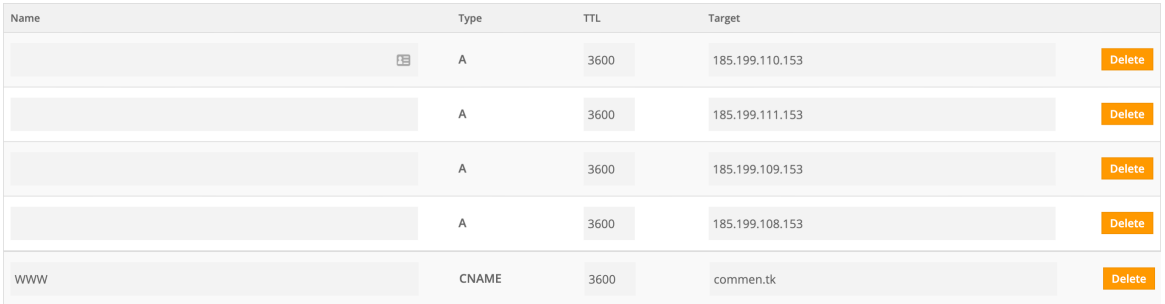


Figura 3.13: Registros *DNS* para que *commen.tk* y *www.commen.tk* lleven a *GitHub Pages*

La página web es sencilla pero funcional, proporciona las instrucciones de cómo utilizar el *cliente web*, además de algunos enlaces relevantes para el usuario. A diferencia del *cliente web*, sí que se ha utilizado el paquete de estilos *@primer/css* para dar la apariencia de *GitHub* a la página y se utiliza *Javascript* para añadir el comportamiento de los campos y botones. Los campos son utilizados para que el usuario pueda elegir los atributos de la sección de comentarios.

- *Owner* - Permite introducir el nombre de un usuario.
- *Repository* - Una vez introducido el usuario, se cargan en este campo todos sus repositorios públicos. Es posible usar repositorios privados cambiando manualmente la etiqueta.

- *Issue Parameter* - El parámetro que se usará como nombre del *issue* que mostrará el *cliente web*.
- *Theme* - Tema del *cliente web*.
- *Page Size* - Tamaño de página, un número entero entre 1 y 100.

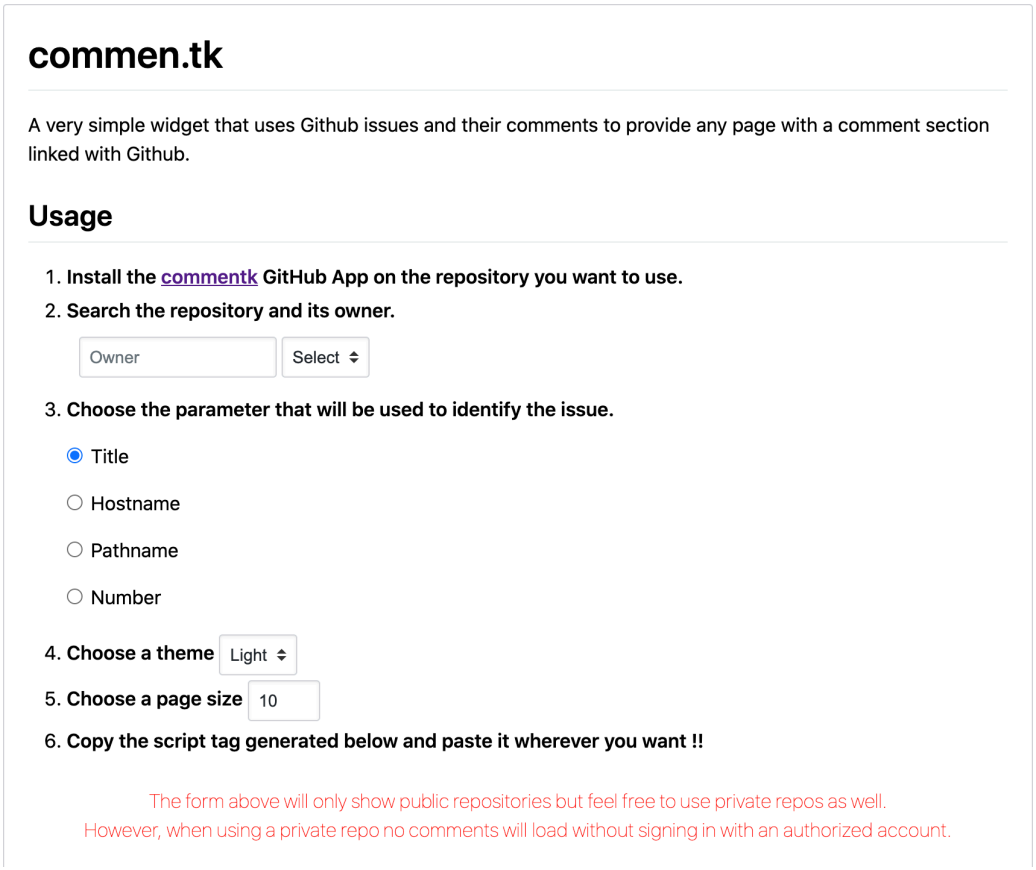


Figura 3.14: *commen.tk*

3.2.2. Content Delivery Network (CDN)

Por otro lado, servir los archivos del *cliente web* requiere un enfoque distinto dado que se trata de archivos mayores en tamaño y potencialmente en número de peticiones que la página web. El *cliente web* debe afectar mínimamente al rendimiento y tiempos de carga de la página donde se ejecute, y utilizar un único servidor que responda a todas las peticiones independientemente de su localización, conocido como *Single Server Distribution*, perjudicará severamente a los usuarios más alejados geográficamente. En 2020 es estándar utilizar una *Content Delivery Network* o *CDN*, un grupo de servidores geográficamente distribuidos que trabajan conjuntamente para proveer una rápida distribución de contenido digital. Utilizar una *CDN* proporciona unas ventajas que la han establecido como un pilar fundamental de las aplicaciones web:

- *Mejorar tiempos de carga* - Al distribuir el contenido más cerca de cada usuario usando un servidor *CDN* cercano, los tiempos de carga disminuyen.
- *Reducir costes de ancho de banda* - El consumo de ancho de banda en *hosting* es el principal gasto para aplicaciones web. Gracias al uso de caché y otras optimizaciones, una *CDN* permite reducir la cantidad de datos que un servidor de origen debe enviar, así reduciendo los gastos de *hosting*.
- *Aumentar la disponibilidad y redundancia del contenido* - Grandes cantidades de tráfico o fallos de hardware pueden interrumpir la disponibilidad de una aplicación web. Gracias a su naturaleza distribuida, una *CDN* puede manejar más tráfico y resistir fallos de hardware.
- *Mejorar seguridad en aplicaciones web* - Mitigación *DDoS*, mejoras en certificados de seguridad y otras optimizaciones.

Para continuar con el ecosistema *AWS* se ha elegido *Amazon CloudFront*, un servicio *CDN* que distribuye datos, vídeos, aplicaciones y *API*s de manera segura a usuarios en todo el planeta con baja latencia y altas velocidades de transferencia. *Cloudfront* está integrado con la infraestructura global de *AWS*, pudiendo así trabajar conjuntamente con otros servicios como *AWS Shield* para mitigación *DDoS*, *Amazon S3*, *Elastic Load Balancing* o *Amazon EC2*.

Cloudfront es una red de distribución de contenido y no se encarga de almacenar los datos que distribuye. Para almacenar los datos que se desea distribuir se ha utilizado *Amazon Simple Storage Service* o *S3*, un servicio de almacenamiento de objetos que ofrece muchas ventajas, sin embargo, el motivo por el que es difícil considerar otro servicio de almacenamiento es que todas las transferencias de datos entre *S3*, *EC2* o *Elastic Load Balancing* y *Cloudfront* son completamente gratuitas. En *Amazon S3* los datos son almacenados como objetos dentro de recursos llamados *buckets*, cada objeto tiene *metadatos*, permisos y controles de acceso, análisis de *big data* y monitorización a nivel de objeto y de *bucket*.

El primer paso para ofrecer los archivos del *cliente web* a través de una *CDN* es crear un *bucket* en *Amazon S3* introduciendo un nombre único y una región donde alojarlo. También es posible activar funciones de control de versiones, *logging*, etiquetado, cifrado, monitorización y permisos, en este caso lo más importante es bloquear todos los accesos públicos dentro de los permisos, ya que todos los accesos deberán realizarse a través de la *CDN* y nunca directamente desde el *bucket*.
Una vez creado el *bucket* es posible subir archivos, pero no acceder a ellos. Utilizando *Cloudfront* es posible crear una *distribución*, específicamente una distribución web, la configuración inicial puede parecer compleja pero la mayoría de campos no deben ser modificados.

- *Origin Domain Name - Bucket S3* creado anteriormente.
- *Viewer Protocol Policy - Redirect HTTP to HTTPS*
- *Alternate Domain Names (CNAMEs) - cdn.commen.tk*
- *Cache Based on Selected Request Headers - Whitelist*
- *Whitelist Headers - Origin*

Debido al uso de *Cross-Origin Resource Sharing (CORS)* por parte del servidor, es fundamental permitir el *header Origin* en la distribución, de lo contrario al intentar cargar el *cliente web* desde un origen que no sea *cloudfront.net* se producirá un error de *CORS*.

A continuación, es necesario utilizar *Amazon Certificate Manager* o *ACM* para crear un *certificado SSL* para el subdominio *cdn.commen.tk*, para ello solo ha de crearse un registro *DNS* de tipo *CNAME* con unos valores generados por *Amazon* para verificar que el usuario tiene acceso al dominio. Por otro lado, hay que crear otro registro *DNS* de tipo *CNAME* que apunte el subdominio *cdn.commen.tk* a la *CDN* en *cloudfront.net*.

Figura 3.15: Registros *DNS* del *certificado SSL* y de *Cloudfront* en *cdn.commen.tk*

Una vez desplegada la distribución, aceptado el certificado y propagados los cambios en el *DNS* será posible acceder todos los archivos existentes dentro del *bucket* a través de la *CDN* y el dominio personalizado *cdn.commen.tk* usando *HTTPS*. Sin embargo, subir y sobrescribir archivos manualmente cada vez que se desea hacer cambios es un proceso ineficiente y repetitivo que se puede automatizar, y para ello se ha creado un *script* en *Node.js* que permite subir todos los archivos presentes dentro de un directorio al *bucket* que utiliza la *CDN*. Estos archivos subidos a S3 van a tener como *Content-Type* por defecto *binary/octet-stream* en sus metadatos, lo que va a ocasionar problemas a la hora de servir archivos HTML, CSS y *Javascript*, por tanto, es muy importante que el *script Node.js* especifique el *Content-Type* adecuado de cada archivo para que no ocurran errores de interpretación en los navegadores. El uso de *cachés* en una red de distribución implica que cada archivo será guardado en caché durante un tiempo determinado y no se consultará el *bucket* hasta que la copia caduque, de esta manera, si se quiere realizar un cambio en uno de los archivos será necesario esperar a que su copia caduque o generar una invalidación.

Una invalidación es una manera de comunicarle a *Cloudfront* que la versión del archivo que posee ya no es válida y que debe actualizarse inmediatamente, *AWS Free Tier* proporciona 1.000 invalidaciones mensuales libres de coste.

3.2.3. Servidor API REST

En último lugar, solo queda por desplegar el Servidor API REST, para ello se ha utilizado la plataforma *EC2* de *AWS*, que permite crear máquinas virtuales *t2.micro* gratuitas con 1 vCPU, 1 GB de memoria y un máximo de 30 GB de almacenamiento *SSD*. *Elastic IP* es la designación de *AWS* para direcciones *IP* estáticas asociadas a máquinas de *AWS*, estas direcciones son gratuitas siempre y cuando estén asociadas a una máquina en ejecución y que no tenga más de una dirección *IP* asociada. Una vez creada la máquina y asignada una dirección *IP* estática será posible conectarse a ella de manera remota usando usuario y contraseña o un archivo clave. Solo 5 pasos son necesarios para desplegar el Servidor API REST:

- *Clonar el repositorio* - Descargar el repositorio que contiene el Servidor API REST.
- *Clave privada GitHub App* - Obtener el archivo *.pem* que contiene la clave privada de la *GitHub App commentk* para que el servidor pueda solicitar *tokens de acceso de instalación*.
- *Crear archivo .env* - Crear un archivo que contenga todas la variables de entorno que por privacidad no pueden formar parte del repositorio.
- *Crear registro DNS* - Crear un registro *DNS* de tipo *A* en la que el subdominio '*api*' apunte a la dirección *IP* pública de la máquina donde se ejecutará el servidor.
- *Crear certificado SSL* - Los *certificados SSL* generados por una *autoridad de certificación* o *CA* no son especialmente caros pero existe una autoridad de certificación sin ánimo de lucro llamada *Let's Encrypt* que permite crear certificados gratuitamente a cualquier persona que posea un dominio. Para ello es necesario descargar una herramienta llamada *Certbot* que en pocos pasos permite generar un certificado y una clave privada con una validez de 3 meses.

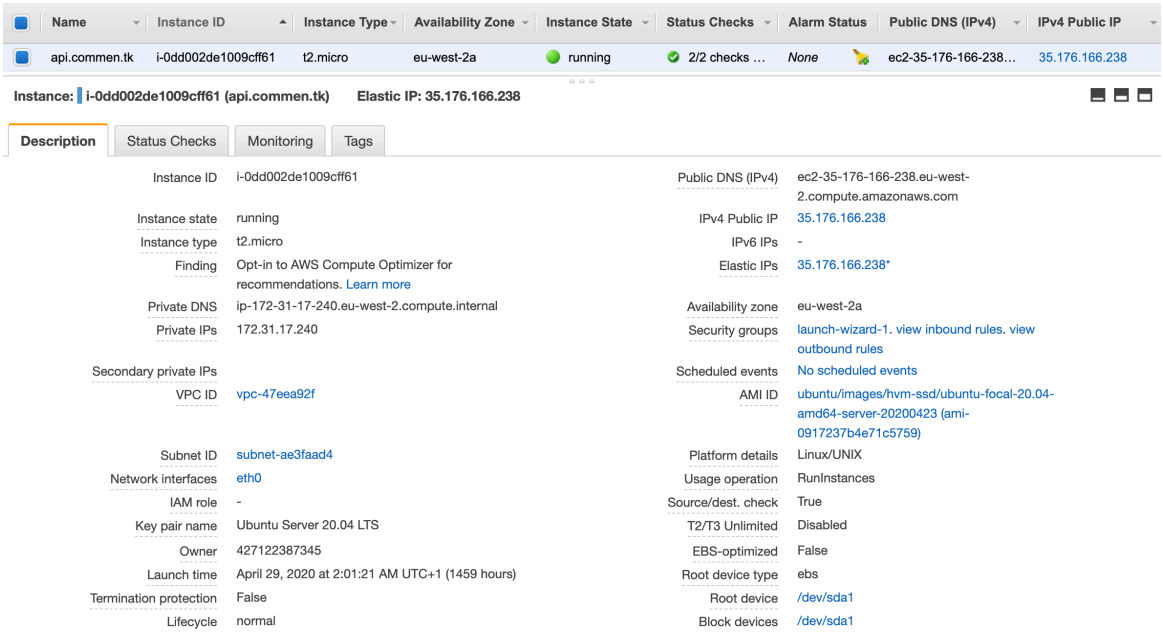


Figura 3.16: Instancia *EC2* utilizada en el despliegue

Esta simplicidad agilizó mucho un nuevo despliegue cuando se adquirió acceso a una máquina virtual con dirección *IP* pública estática en el *IaaS de la Universidad de La Laguna*, que resultó ser muy beneficioso dado que los limitados recursos de la máquina de *AWS* impedían un desarrollo estable a través de *SSH*, particularmente a la hora de crear *builds* o ejecutar el servicio.

Capítulo 4

Resultados

4.1. Integración Continua

Desde el comienzo del proyecto se ha asignado gran importancia a las pruebas del código, la integración continua y cobertura del código, con el fin de minimizar el número de *bugs* y consecuentemente la cantidad de tiempo perdido buscándoles solución.

Existen muchos *frameworks* para hacer pruebas en *Javascript*, cada uno con sus ventajas, desventajas, fortalezas y debilidades, los más populares son *Jest*, *Mocha*, *Jasmine*, *Chai* y *Puppeteer*, entre otros. Finalmente se ha escogido *Mocha* debido a su compatibilidad con *Node.js* y su claridad y sencillez a la hora de realizar pruebas con servidores web *express*.

Integración Continua o *CI* es una práctica de desarrollo en la que los desarrolladores integran código en un repositorio compartido frecuentemente, preferiblemente varias veces al día. Cada integración puede ser verificada por un proceso automatizado de construcción o *build* y pruebas automáticas. La filosofía detrás de la integración continua es hacer muchas integraciones pequeñas e incrementales para así reducir el riesgo de producir un error y poder identificarlo y solucionarlo rápidamente en el caso de que se produzca[46].

Para realizar la integración continua en este proyecto se ha utilizado *Travis CI*, un servicio de integración continua para proyectos hospedados en *GitHub* y *Bitbucket*[30]. Una vez habilitado Travis CI en el repositorio en cuestión, *Travis CI* es configurado añadiendo un archivo *.travis.yml*, un archivo de texto en formato *YAML*, en el directorio raíz del repositorio. En él se debe especificar el lenguaje de programación utilizado, los entornos, parámetros y dependencias de *build* y *test*. Con esta configuración *GitHub* notificará a *Travis CI* cada vez que se empuje un *commit* al repositorio o se realice un *pull request* que a su vez notificará a los desarrolladores del resultado de las prueba sobre de la nueva integración mediante correo electrónico.

```
1  language: node_js
2  node_js:
3      - node
4      - lts/*
5  after_success:
6      - yarn coverage
```

Listado 4.1: Archivo *.travis.yml* utilizado en el proyecto

Travis CI puede ser configurado para ejecutar pruebas en una gran variedad de máquinas con diferente software instalado, como versiones antiguas de lenguajes y entornos para probar compatibilidad, también soporta integración con herramientas externas como analizadores de cobertura del código que generan informes cuando las pruebas tienen éxito. Además funciona con una gran cantidad de lenguajes e incluso es utilizado por proyectos tan complejos como *Ruby on Rails* y *Ruby*.

Con el fin de reducir el número de *bugs* en el proyecto se ha utilizado una herramienta de cobertura de código. La *cobertura del código* es una medida que describe la cantidad de código fuente ejecutado al realizar una serie de pruebas. Una aplicación con una cobertura alta en sus pruebas, siendo la medida

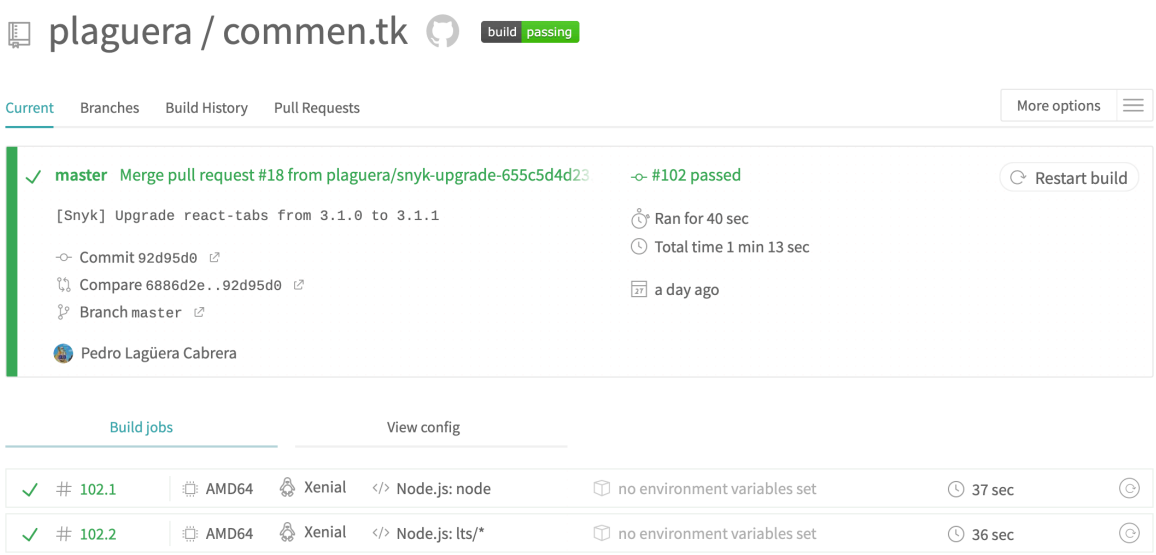


Figura 4.1: *Build* en *Travis CI* de un *pull request*

más común el porcentaje de líneas de código ejecutadas, tiene una menor probabilidad de contener errores sin detectar que otra aplicación con una cobertura baja.

Para medir la cobertura del código en este proyecto se ha utilizado *IstanbulJS*, más recientemente conocido como *nyc*. El único problema que ha impedido lograr una mayor cobertura es el hecho de que para poder utilizar las funciones de autorización del Servidor API REST es necesario haber iniciado sesión en *GitHub* en un navegador, por tanto no es posible hacer pruebas sobre el controlador de autorización, y las pruebas de creación y eliminación de comentarios se realizan utilizando un *access token* generado manualmente.

```
➜  commen.tk git:(master) yarn test
yarn run v1.22.0
$ nyc --reporter=lcov --reporter=text mocha -r ts-node/register src/test/**/*.test.t


  GET /comments/user/repo/issue
    ✓ should get any public issue (757ms)

  GET /users/user
    ✓ should get any user (527ms)


  2 passing (1s)

----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |   70.15 |    32.35 |   65.22 |   71.19 |
 src                  |   78.57 |        0 |   85.71 |   86.11 |
  request.ts          |   52.63 |        0 |      50 |   61.54 | 6-17
  server.ts           |     100 |      100 |     100 |     100 |
 src/controllers      |   61.54 |    45.83 |      60 |   57.97 |
  auth-controller.ts  |   39.47 |        0 |   33.33 |   37.14 | 16-25,29-66
  controller.ts       |     100 |      100 |     100 |     100 |
  issue-controller.ts |   71.43 |     62.5 |   42.86 |   68.42 | 42-49,53-62
  user-controller.ts  |   92.86 |       60 |     100 |   90.91 | 16
 src/routes           |   92.86 |      100 |       0 |     100 |
  index.ts            |   92.86 |      100 |       0 |     100 |
----------------------|---------|----------|---------|---------|-------------------
✨  Done in 4.71s.
```

Figura 4.2: Ejecución de pruebas y cálculo de la cobertura del código

Repositorios de paquetes como *npm* contienen cientos de miles o incluso millones de paquetes, cada uno de ellos, independientemente de su tamaño, provee una funcionalidad menos a implementar por

otros desarrolladores. Sin embargo, esta funcionalidad obtenida sin esfuerzo viene acompañada de fallos de seguridad, en *npm* 1 de cada 7 paquetes está afectado por una vulnerabilidad conocida.

SNYK es una plataforma de seguridad de código abierto que tiene como objetivo encontrar las vulnerabilidades presentes en el código de una aplicación y en el de sus dependencias[48]. *SNYK* ha compilado una base de datos que crece constantemente de todas las vulnerabilidades conocidas, cada entrada en la base de datos incluye información de la severidad, una descripción y posibles soluciones para una vulnerabilidad determinada. Su uso es muy sencillo dado que es posible integrar *SNYK* en un *pipeline* de integración continua o despliegue continuo, en este proyecto, *SNYK* ha sido integrado con *GitHub* de tal manera que cada vez que se produce un cambio en el repositorio se hace un análisis de vulnerabilidades, no obstante, en el caso de que encuentren nuevas vulnerabilidades o alguna dependencia se declare obsoleta, *SNYK* ha implementado un *bot* que abre un *pull request* en el repositorio y envía un correo electrónico al desarrollador, informándole de los detalles de la vulnerabilidad.

4.2. Pruebas de Rendimiento

En esta parte del proyecto se tuvo que descartar la idea de utilizar herramientas de *load testing* debido a que el número de peticiones necesarias para que el *load testing* sea significativo agotaría muy rápidamente los límites de peticiones mensuales de *AWS* (20.000) y por hora de *GitHub* (5.000). Por este motivo se ha optado por centrar las pruebas en el rendimiento de la página web y del sistema de comentarios, utilizado varias aplicaciones web para probar su rendimiento y comparar los aspectos que cada una considera positivos y negativos y así poder mejorar el rendimiento de todo el proyecto.

Herramienta	**Puntuación(es)**
Google PageSpeed Insights	100 %
Pingdom	95 %
WebPageTest	[F][A][A][A][A][F][OK]
Dareboost	84 %
GTmetrix	94 %

Tabla 4.1: Puntuaciones logradas en 5 pruebas de rendimiento web.

Para lograr estas puntuaciones han sido necesarios una serie de cambios en varios aspectos para poder mejorar el rendimiento todo lo posible.

- La *compresión* de las peticiones es muy importante, en todos los navegadores modernos se admite y negocia automáticamente la compresión *gzip* en todas las solicitudes *HTTP*. Al habilitar la compresión *gzip*, se puede reducir el tamaño de las respuestas transferidas hasta en un 90 %, lo que puede, a su vez, disminuir significativamente el tiempo necesario para descargar el recurso correspondiente, así como reducir el uso de datos del cliente y el tiempo que se tarda en renderizar las páginas por primera vez[11]. Para comprimir las peticiones en *express* se utiliza el *middleware* llamado *compression* que automáticamente comprime todas las peticiones que envíe el servidor *express* en codificación *deflate* o *gzip*.
- Todos los recursos estáticos están *minificados*, es decir, se han eliminado todos los caracteres innecesarios del código fuente sin alterar su funcionalidad; espacios en blanco, comentarios y otros caracteres dependiendo del lenguaje son eliminados, también se suelen cambiar los nombres de variables y funciones por otros más cortos. Menos caracteres suponen archivos más pequeños a enviar y por lo tanto, mejores velocidades y mejor experiencia de usuario.
- Durante la mayor parte del desarrollo del proyecto se han cargado las imágenes de perfil de cada usuario tal y como están almacenadas en *GitHub*, que solo acepta imágenes menores de 1 MB y recomienda una resolución mínima de 500x500 píxeles[22]. Dado que la aplicación siempre va a mostrar las imágenes de perfil en 40x40 píxeles, no solo se están transfiriendo muchos datos que no se están utilizando, sino que además ha de cambiarse el tamaño de cada imagen (*downsampling*) en el navegador mediante interpolación, añadiendo una carga de procesamiento completamente evitable. La solución es simple, para cada imagen de perfil de usuario, su dirección irá acompañada del tamaño que se precisa (*&size=40*). De esta manera el servidor encargado de enviar la imagen se ocupa de escalarla y la transferencia resultado a través de internet es mucho más corta.

- Aunque forma parte del diseño y no se tuvo que realizar ningún cambio, es necesario mencionar que todas las pruebas de rendimiento web exigen el uso de *CDN*s para la distribución y cacheo de archivos estáticos. El sistema de comentarios es distribuido por *Amazon Cloudfront* y la página web por *Fastly* gracias a *GitHub Pages*.
- Dado que el sistema de comentarios está formado por archivos estáticos, hacer que se almacenen en caché del cliente solo requiere el uso de un encabezado *Expires* que comunique al navegador hasta cuándo es válido cada archivo. Cachear las peticiones se hace igual pero como no son resultados estáticos y dado que *GitHub* le da 5 minutos de vida en caché a sus respuestas e imágenes, se ha optado por darle 5 minutos de vida en caché a cada respuesta del Servidor API REST. En cuanto a la página web, *GitHub Pages* controla todas las configuraciones y cada archivo estático que sirve expira a los 10 minutos de ser recibido.
- La paginación de los comentarios es necesaria para mantener una buena apariencia y porque a medida que crece el número de comentarios en un *issue* llega un punto en el que cargarlos todos de una vez para solo ver los más recientes resulta ineficiente y aparatoso.

Capítulo 5

Conclusiones y Líneas Futuras

5.1. Conclusiones

Como bien se ha planteado en los Objetivos, el fin de este proyecto es la composición de un sistema de comentarios que pueda ser utilizado por cualquier página web sin ningún software adicional, utilizando *GitHub API*, conservando la estética de comentarios de *GitHub* y realizando un despliegue para el público que mantenga los costes a 0.

El resultado de este proyecto ha sido un sistema con las características y funcionalidades descritas en el apartado de Objetivos Específicos, es decir, un *cliente web* que se comunica con *GitHub* a través de un *servidor API REST* de manera segura, protegiendo los datos sensibles de los usuarios y utilizando el estilo de *GitHub* en el *cliente web*. Tanto la base de código del proyecto como las infraestructuras sobre las que ha sido desplegada han sido optimizadas con el fin de mejorar su rendimiento y sus tiempos y velocidades de transferencia. Además, la base código está completamente comentada y documentada para facilitar su lectura y comprensión.

Se han aprendido y utilizado varias tecnologías y lenguajes en la interfaz de usuario, comunicación entre servicios, seguridad, distribución y almacenamiento de contenidos y la elaboración de pruebas. Produciendo un resultado que utiliza las tecnologías y paradigmas más populares en desarrollo web en la actualidad.

En conclusión, se ha desarrollado un sistema de comentarios libre de errores, dotado de múltiples funcionalidades, fácilmente desplegable y documentado en profundidad que puede ser expandido o modificado por cualquier desarrollador que esté interesado. El desarrollo ha sido ágil, realizando siempre pruebas sobre la base de código y el rendimiento del sistema, ciñéndose siempre a los plazos previstos e incluso superando las expectativas en algunas ocasiones, pudiendo así considerarse este proyecto como un éxito rotundo.

5.2. Líneas Futuras

Debido a la extensa cantidad de tecnologías disponibles en la actualidad para el desarrollo web y el despliegue de aplicaciones, el número de actualizaciones y mejoras que se pueden realizar sobre este proyecto es tan grande que se puede decir que solo depende de la imaginación y el tiempo que quiera dedicarle un desarrollador.

- *Docker* es una herramienta basada en *virtualización* a nivel del sistema operativo que facilita la creación, el despliegue y la ejecución de aplicaciones gracias al uso de *contenedores*[39]. Un contenedor permite al desarrollador empaquetar una aplicación con todas las dependencias y librerías que necesita para funcionar en un solo paquete, de tal manera que la aplicación pueda ser ejecutada rápida y fiablemente independientemente del entorno donde se encuentre. Cada contenedor está aislado del resto y posee su propio software, librerías y archivos de configuración, solo pudiendo comunicarse con otros contenedores a través de canales bien definidos. Debido a que todos los contenedores son ejecutados por un único *kernel* de sistema operativo, su uso de recursos es menor que el de máquinas virtuales. La *orquestación de contenedores* automatiza el proceso de despliegue, administración, escalado y redes de contenedores. Teniendo en cuenta las ventajas que proporcionan estas tecnologías resulta muy atractivo crear un sistema automático que cree, borre y monitorice contenedores del Servidor API REST para escalar el servicio, aportar redundancia o balancear carga.

- El proyecto utiliza integración continua pero sería recomendable ir un paso más allá y usar *entrega continua* y *despliegue continuo*. Igual que la integración continua garantiza que no hay errores en los nuevos cambios de una aplicación, la entrega continua garantiza que dicha aplicación puede ser puesta en producción y desplegada sin errores, no obstante, el despliegue no se lleva a cabo hasta que alguien lo autorice. El despliegue continuo es igual que la entrega continua pero automatiza el despliegue siempre que se produzcan cambios, sin tener que esperar por autorización.
- La implementación de un tema que se adapte al estilo (colores y fuentes) de la página donde se encuentre aportaría una aparente integración con el resto de la página, en vez de provocar un choque de estilos con otros elementos.
- Dado que *GitHub* utiliza *Markdown* para redactar comentarios, el editor actual de comentarios del proyecto admite sintaxis *Markdown*[18] pero en el caso de usuarios menos experimentados sería útil añadir botones en el editor para cambiar el estilo del texto, como títulos, negrita, cursiva, enumeraciones, listas, bloques de código, imágenes y enlaces.
- Aunque su implementación siguiendo el estilo de *GitHub* puede resultar algo tediosa, es necesario añadir la opción de editar y eliminar comentarios.
- *GitHub* dispone de un sistema de valoración de comentarios llamados *reacciones* que sería conveniente implementar, tanto para ver dichas las reacciones de un comentario como para valorarlo.
- La implementación actual no requiere recargar la página para cargar comentarios más antiguos, pero en el caso de que otro usuario cree un nuevo comentario sí que será necesario recargar la página para poder verlo. Mostrar la *sección de comentarios en tiempo real*, sin que sea necesario recargar la página cuando se cree, borre o modifique un comentario supondría un salto significativo en la calidad de la experiencia del usuario.

Capítulo 6

Summary and Conclusions

Just like it has been posed in Objetivos, the goal of this project is the creation of a comment system which can be used by any webpage without the need for any additional software, using *GitHub API*, keeping the *GitHub* aesthetic throughout the user interface and which can be deployed for everyone to see and use, completely free of charge.

The final result of this project is a system with the features and functionalities enumerated in the Objetivos Específicos chapter, that is, a *web client* which communicates securely with *GitHub* through a *REST API webserver*, protecting all sensitive user data and using the classic *GitHub* look and style on its user interface. Both the project's codebase and the infrastructure on which it has been deployed have been optimized to improve performance, transfer times and speed, and security. Moreover, the codebase is fully commented and documented to improve readability and clarity.

Multiple technologies and programming languages have been studied, learnt and used through the development of the user interface, communication pipeline between services, security designs and methods, content distribution and storage and testing. Therefore, the resulting project uses all the latest and most advanced technologies and paradigms used in web development today.

To sum up, this project has produced a bug-free comment system full of functionality, easily deployable and thoroughly documented, which can be expanded upon and modified by any interested developer. Development has been agile, using continuous integration methodologies and frequently testing the codebase for bugs and errors and optimizing overall performance, all while fulfilling every deadline and time limit, and even beating expectations in some situations. Thus, this project can be labeled as a complete success.

Capítulo 7

Presupuesto

7.1. Desarrollo

En 2019, un ingeniero informático recién graduado cobra entre 1.000 y 1.200 euros netos mensualmente[44], suponiendo que cada semana conlleva 40 horas trabajo, se utilizará un coste por hora de trabajo de 11 euros.

Actividad	Horas	Subtotal (€)
Investigación y Aprendizaje	25	275
Diseño de la Arquitectura	3	33
Diseño API	2	22
Implementación API	57	627
Implementación Aplicación	180	1,980
Estilo Aplicación	36	396
Desarrollo Página Web	10	110
Testeo y Validación	5	55
Despliegue	82	902
Documentación	10	110
Desarrollo de la Memoria	75	825
Total	485	5,335

Tabla 7.1: Presupuesto del Desarrollo

7.2. Despliegue

Si bien se ha cumplido el objetivo de conseguir el despliegue del proyecto sin ningún coste, se van a detallar los servicios utilizados para realizar el despliegue final y sus límites de uso.

Servicio	Límite
Dominio *commen.tk*	1 año
Amazon EC2 - Computación	750 horas / mes
Amazon EC2 - Almacenamiento	30 GB / mes
Amazon S3 - Almacenamiento	5 GB / mes
Amazon S3 - Peticiones GET	20,000 / mes
Amazon S3 - Peticiones PUT, POST	2,000 / mes
Amazon Cloudfront - Peticiones	2,000,000 / mes
Amazon Cloudfront - Transferencias	50 GB / mes
IaaS ULL	Ninguno

Tabla 7.2: Presupuesto del Despliegue

Apéndice A

Código del *servidor API REST*

El código del proyecto está almacenado en dos repositorios de *GitHub*, no obstante, estos repositorios son *mirrors*, es decir, contienen exactamente el mismo código e historial de *commits*. Esto se debe a que el proyecto fue creado originalmente en un repositorio privado de la organización ULL-ESIT-GRADOII-TFG y para permitir el uso de *Travis CI* y *SNYK* y publicar el proyecto, se creó un repositorio *mirror* público. A continuación, se enumeran los enlaces relevantes del proyecto.

- **Repositorio Privado**
- **Repositorio Público**
- **Repositorio Página Web**
- **Página Web del Proyecto**
- **CDN del Proyecto**
- **API del Proyecto**
- **GitHub App commentk**

A.1. Subida Archivos a *AWS S3* en *Node.js*

```
require('dotenv').config();
import AWS from 'aws-sdk';
import fs from 'fs';
import path from 'path';

const DIRECTORY_TO_UPLOAD = 'public';
const s3 = new AWS.S3({
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
});

const isDirectory = (file: string) => fs.statSync(file).isDirectory();
const getDirectories = (file: string) => fs.readdirSync(file).map(name => path.join(file, name)).filter(isDirectory);

const isFile = (file: string) => fs.statSync(file).isFile();
const getFiles = (file: string) => fs.readdirSync(file).map(name => path.join(file, name)).filter(isFile);

const getFilesRecursively = (file: string) => {
    let dirs = getDirectories(file);
    let files = dirs
        .map(dir => getFilesRecursively(dir)) // go through each directory
        .reduce((a, b) => a.concat(b), []);    // map returns a 2d array (array of file arrays) so flatten
    return files.concat(getFiles(file));
};

console.log(process.cwd());
var files = getFilesRecursively(DIRECTORY_TO_UPLOAD);
files.forEach((file: string) => {
    uploadFileToS3(file, file.replace(DIRECTORY_TO_UPLOAD + '/', ''))
});


function uploadFileToS3(fileName: string, key: string) {
    const fileContent = fs.readFileSync(fileName);
```

```
    var today = new Date();
    var onehourfromnow = new Date(today.getTime() + (60 * 60 * 1000));
    const params = {
        Bucket: process.env.AWS_BUCKET_NAME || '',
        Expires: onehourfromnow,
        Key: key,
        Body: fileContent
    };

    const re = /(?:\.([^.]+))?$/;
    var contentType = re.exec(fileName)![1];
    if (contentType)
        switch(contentType) {
            case 'js': params['ContentType'] = 'application/javascript'; break;
            case 'css': params['ContentType'] = 'text/css'; break;
            default: break;
        }

    s3.upload(params, (err: Error, data: any) => {
        if (err) throw err;
        console.log(` File uploaded successfully. ${data.Location}`);
    });
};
```

A.2. Rutas Servidor API REST

```
import { Router } from 'express';
import { UserController } from '../controllers/user-controller';
import { IssueController } from '../controllers/issue-controller';
import { AuthController } from '../controllers/auth-controller';
import { CommentController } from '../controllers/comment-controller';

const routes = Router();

routes.route('/').get((req, res) => res.status(200).send('<h1>API Server is <strong style="color: green;">RUNNING</strong></h1>
routes.route('/user').get(UserController.get)
routes.route('/users/:id').get(UserController.get);
routes.route('/issuenumber/:owner/:repo/:name').get(IssueController.processIssueName);
routes.route('/comments/:owner/:repo/:number').get(CommentController.get);
routes.route('/comments/:owner/:repo/:number').post(CommentController.post);
routes.route('/authorize').get(AuthController.authorize);
routes.route('/oauth/redirect').get(AuthController.access_token);

export default routes;
```

A.3. Función *Node.js* para consultas *GitHub GraphQL API v4*

```
export async function query(data: string, token: string) {
    const options = {
        headers: {
            authorization: 'token ' + token
        },
        method: 'POST',
        body: JSON.stringify({
            query: data
        })
    };

    let res = await fetch('https://api.github.com/graphql', options).catch(console.error);
    let json = undefined;
    if (res && res.headers.get('content-type')?.includes('application/json')) json = await res.json();
    return new ServerResponse(res, json);
}
```

A.4. Clase *Server* basada en *ExpressJS*

```
import compression from 'compression';
import cookieParser from 'cookie-parser';
import cors from 'cors';
import express from 'express';
import fs from 'fs';
import { InstallationController } from './controllers/installation-controller';
import routes from './routes';
class Server {
    public express: express.Application;
```

```
    constructor() {
        this.express = express();
        this.middleware();
        this.enableCors();
        this.routes();
        this.installationController();
    }

    enableCors() {
        const options: cors.CorsOptions = {
            allowedHeaders: [
                'Origin',
                'X-Requested-With',
                'Content-Type',
                'Accept',
                'X-Access-Token',
                'Authorization'
            ],
            credentials: true,
            methods: 'GET,POST',
            origin: true,
            preflightContinue: false
        };
        this.express.use(cors(options));
    }

    middleware() {
        this.express.use(cookieParser(process.env.COOKIE_SECRET));
        this.express.use(express.json());
        this.express.use(express.urlencoded({ extended: true }));
        this.express.use(compression());
        this.express.set('trust proxy', 1);
    }

    routes() {
        this.express.use('/', routes);
    }

    installationController() {
        if (process.env.NODE_ENV === 'DEVELOPMENT') return;
        InstallationController.init(
            parseInt(process.env.GITHUB_APP_IDENTIFIER || ''),
            fs.readFileSync(process.env.GITHUB_APP_PRIVATE_KEY_PATH || '').toString()
        );
    }
}

export default new Server().express;
```

Apéndice B

Código del *cliente web*

B.1. Configuración *WebpackJS*

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const FixStyleOnlyEntriesPlugin = require("webpack-fix-style-only-entries");
var path = require('path');
var glob = require('glob');

// Get all themes and save them as entry points
var entries = {
    'client': './src/index.tsx'
};
glob.sync('./src/stylesheets/themes/**/app.scss').forEach((theme) => entries[theme.split('/')[4]] = theme);

module.exports = {
    devtool: 'source-map',
    entry: entries,
    module: {
        rules: [
            {
                test: /\.ts[x]?$/i,
                use: 'ts-loader',
                exclude: /node_modules/,
            },
            {
                test: /\.s[ac]ss$/i,
                use: [
                    {
                        'loader': MiniCssExtractPlugin.loader,
                    },
                    {
                        loader: 'css-loader',
                        options: {
                            sourceMap: true,
                        },
                    },
                    {
                        loader: 'sass-loader',
                        options: {
                            sourceMap: true,
                        },
                    },
                ],
            },
        ],
    },
    resolve: {
        extensions: ['.tsx', '.ts', '.js', '.css', '.scss'],
        alias: {
            'react': 'preact/compat',
            'react-dom': 'preact/compat',
        },
    },
    output: {
        filename: '[name].js',
        path: path.resolve(__dirname, '../public'),
    },
    plugins: [
        new FixStyleOnlyEntriesPlugin(),
```

```
        new MiniCssExtractPlugin({
            // Options similar to the same options in webpackOptions.output
            // both options are optional
            filename: 'themes/[name].css',
            chunkFilename: '[id].css',
        }),
    ],
};
```

B.2. Componente Principal *Widget* en *ReactJS*

```
import React from 'react';
import * as request from './request';
import Editor from './components/Editor';
import Header from './components/Header';
import Timeline from './components/Timeline';
import { WidgetProps, WidgetState } from './props';
import PaginationButton from './components/PaginationButton';

class Widget extends React.Component<WidgetProps, WidgetState> {

    constructor(props: WidgetProps) {
        super(props);
        this.state = {
            comments: [],
            issue: this.props.issuenumber,
            user: undefined,
            totalCount: 0,
            hiddenItems: 0,
            cursor: ''
        }
    }

    async issue() {
        if (this.state.issue == -1) {
            let name = '';
            switch(this.props.issuename) {
                case 'url': name = window.location.href; break;
                case 'title': name = document.title; break;
                default: break;
            }
            let res = await request.get(`issuenumber/${this.props.owner}/${this.props.repo}/${name}`).catch(console.error);
            this.setState({ issue: res.search.nodes[0].number })
            return res;
        }
    }

    issueUrl() {
        return `https://github.com/${this.props.owner}/${this.props.repo}/issues/${this.state.issue}`;
    }

    commentsRequestUri(cursor?: string) {
        if (cursor)
            return `comments/${this.props.owner}/${this.props.repo}/${this.state.issue}?pagesize=${this.props.pageSize}&cursor=$
        return `comments/${this.props.owner}/${this.props.repo}/${this.state.issue}?pagesize=${this.props.pageSize}`;
    }

    comment(text: string) {
        request.post(this.commentsRequestUri(), { body: text })
            .then(() => this.comments())
            .catch(console.error);
    }

    comments() {
        request.get(this.commentsRequestUri())
            .then(data => {
                this.setState({ comments: data.repository.issue.comments.nodes });
                this.setState({ totalCount: data.repository.issue.comments.totalCount });
                this.setState({ hiddenItems: this.state.totalCount - this.state.comments.length });
                this.setState({ cursor: data.repository.issue.comments.pageInfo.startCursor });
            })
            // Required for Preact to set state properly
            .finally(() => { this.setState({ hiddenItems: this.state.totalCount - this.state.comments.length }); })
            .catch(console.error);
    }
```

```
    nextComments() {
        request.get(this.commentsRequestUri(this.state.cursor))
            .then(data => {
                this.setState({ comments: data.repository.issue.comments.nodes.concat(this.state.comments) });
                this.setState({ hiddenItems: this.state.totalCount - this.state.comments.length });
                this.setState({ cursor: data.repository.issue.comments.pageInfo.startCursor });
            })
            // Required for Preact to set state properly
            .finally(() => { this.setState({ hiddenItems: this.state.totalCount - this.state.comments.length }); })
            .catch(console.error);
    }

    user() {
        request.get('user')
            .then(data => this.setState({ user: data.viewer }))
            .catch(console.error);
    }

    componentDidMount() {
        this.issue().then(() => {
            this.user();
            this.comments();
        });
    }

    render() {
        return (
            <div className='widget-wrapper'>
                <Header commentCount={this.state.totalCount} url={this.issueUrl()} />
                <div className='timeline-wrapper'>
                    <PaginationButton hiddenItems={this.state.hiddenItems} onClick={this.nextComments.bind(this)} user={this.sta
                    <Timeline {...this.state.comments} />
                </div>
                <Editor user={this.state.user!} onComment={this.comment.bind(this)} />
            </div>
        );
    }
}

export default Widget;
```

B.3. SASS - Archivo Principal

```
@import './avatar';
@import './comment';
@import './editor';
@import './header';
@import './pagination';

.commen-tk {
    margin: 0;
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',
        'Oxygen', 'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans',
        'Helvetica Neue', sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    code {
        font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New', monospace;
    }
}
```

B.4. SASS - Tema Oscuro

```
@import "./variables";
@import "../../App";

$comment_background-color: #181818;
$comment_border-color: #343434;
$comment_border-bottom-color: #404040;
$comment_color: #fff;
//
$isauthor--border-color: rgb(34, 68, 102);
$isauthor--comment-header_color: rgb(157, 172, 204);
$isauthor--comment-header_background-color: rgb(24, 32, 48);
$isauthor--comment_color: rgb(210, 210, 210);
//
```

```
$comment-body_color: #ccc;
$comment-body-anchor_color: rgb(79, 140, 201);
//
$comment-header_color: #ccc;
$comment-header_background-color: #202020;
$comment-header_border-bottom-color: #404040;
//
$author_color: #d2d2d2;
//
$time-ago_color: #afafaf;
//
$editor_background-color: #181818;
$editor_border-color: #343434;
$editor_border-bottom-color: #404040;
$editor_color: #fff;
//
$react-tabs-tab-list_background-color: #202020;
$react-tabs-tab-list_border-bottom-color: #404040;
$react-tabs-tab-list_color: #ccc;
//
$react-tabs-tab-list-tab_color: #afafaf;
//
$react-tabs-tab-list-tab-selected_background-color: #181818;
$react-tabs-tab-list-tab-selected_border-color: #404040;
$react-tabs-tab-list-tab-selected_color: #d2d2d2;
//
$editor-textarea_background-color: #181818;
$editor-textarea_border-color: #404040;
$editor-textarea_color: #d2d2d2;
//
$markdown-render_background-color: #181818;
$markdown-render_border-color: #404040;
$markdown-render_color: #d2d2d2;
//
$separator_color: #383838;
//
$header_color: rgb(148, 148, 148);
//
$pagination-wrapper_background-url: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.org/2000/svg" width="44" height="34"
$pagination-button_background-color: rgb(24, 24, 24);
$pagination-button_border-color: rgb(64, 64, 64);
$pagination-button_color: rgb(175, 175, 175);
$pagination-button-load_color: rgb(79, 140, 201);
```

Bibliografía

[1] https://parceljs.org/. Accessed: 2020-06-21.

[2] About apps. =https://developer.github.com/apps/about-apps/. Accessed: 2020-06-26.

[3] Api platform: Api tools and solutions from postman. https://www.postman.com/api-platform/?utm_source=www&utm_medium=home_hero&utm_campaign=button. Accessed: 2020-06-21.

[4] Authorizing oauth apps. https://developer.github.com/apps/building-oauth-apps/authorizing-oauth-apps/. Accessed: 2020-06-16.

[5] Coral | a better commenting experience from vox media. https://docs.coralproject.net/talk/. Accessed: 2020-06-14.

[6] Cross-origin resource sharing (cors). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed: 2020-06-16.

[7] Diskuto. https://github.com/rejectedsoftware/diskuto. Accessed: 2020-06-14.

[8] Disqus | engage your audience. https://disqus.com/features/engage/. Accessed: 2020-06-14.

[9] Documentation for visual studio code. https://code.visualstudio.com/docs. Accessed: 2020-06-20.

[10] Ecmascript. https://en.wikipedia.org/wiki/ECMAScript#:~:text=ECMAScript(orES)isa,helpfostermultipleindependentimplementations. Accessed: 2020-06-14.

[11] Enable compression | pagespeed insights | google developers. https://developers.google.com/speed/docs/insights/EnableCompression. Accessed: 2020-06-14.

[12] Express/node introduction. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. Accessed: 2020-06-16.

[13] Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. Accessed: 2020-06-16.

[14] gh-commentify. https://github.com/orta/gh-commentify. Accessed: 2020-06-14.

[15] Introduction to the dom. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. Accessed: 2020-06-15.

[16] Knowledgebase. https://my.freenom.com/knowledgebase.php?language=english. Accessed: 2020-06-21.

[17] Managing a custom domain for your github pages site. https://help.github.com/en/github/working-with-github-pages/managing-a-custom-domain-for-your-github-pages-site#configuring-an-apex-domain. Accessed: 2020-06-21.

[18] Markdown: Syntax. https://daringfireball.net/projects/markdown/syntax. Accessed: 2020-06-21.

[19] Mastering markdown · github flavored markdown. https://guides.github.com/features/mastering-markdown/#GitHub-flavored-markdown. Accessed: 2020-06-21.

[20] Oauth community site. https://oauth.net/. Accessed: 2020-06-10.

[21] @octokit/app. =https://www.npmjs.com/package/@octokit/app. Accessed: 2020-06-26.

[22] Personalizing your profile. https://help.github.com/en/enterprise/2.13/user/articles/personalizing-your-profile#:~:text=Tip:Yourprofilepictureshould,about500by500pixels. Accessed: 2020-06-14.

[23] Preact. https://preactjs.com/. Accessed: 2020-06-26.

[24] Primer design system. https://primer.style/. Accessed: 2020-06-20.

[25] Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed: 2020-06-16.

[26] Remark42. https://remark42.com/. Accessed: 2020-06-14.

[27] Remote - ssh - visual studio marketplace. https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh. Accessed: 2020-06-20.

[28] Sass basics. https://sass-lang.com/guide. Accessed: 2020-06-20.

[29] Set-cookie. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie. Accessed: 2020-06-16.

[30] Travis ci documentation. https://docs.travis-ci.com/?utm_source=help-page&utm_medium=travisweb. Accessed: 2020-06-21.

[31] Tutorial: Understanding cookies and sessions. http://www.lassosoft.com/Tutorial-Understanding-Cookies-and-Sessions. Accessed: 2020-06-16.

[32] Utterances. https://utteranc.es/. Accessed: 2020-06-14.

[33] What is a cdn? | how do cdns work? https://www.cloudflare.com/learning/cdn/what-is-a-cdn/. Accessed: 2020-06-14.

[34] Developers: Get ready for new samesite=none; secure cookie settings. https://blog.chromium.org/2019/10/developers-get-ready-for-new.html, Oct 2019. Accessed: 2020-06-16.

[35] Blog comment hosting service. https://en.wikipedia.org/wiki/Blog_comment_hosting_service, Jun 2020. Accessed: 2020-06-15.

[36] Cloud computing. https://en.wikipedia.org/wiki/Cloud_computing, Jun 2020. Accessed: 2020-06-15.

[37] Comments section. https://en.wikipedia.org/wiki/Comments_section, Apr 2020. Accessed: 2020-06-14.

[38] Comments section. https://en.wikipedia.org/wiki/Comments_section, Apr 2020. Accessed: 2020-06-20.

[39] Docker (software). https://en.wikipedia.org/wiki/Docker_(software), Jun 2020. Accessed: 2020-06-21.

[40] Http cookie. https://en.wikipedia.org/wiki/HTTP_cookie, Jun 2020. Accessed: 2020-06-16.

[41] Overview of amazon web services. https://d1.awsstatic.com/whitepapers/aws-overview.pdf, Jan 2020. Accessed: 2020-06-14.

[42] React (web framework). https://en.wikipedia.org/wiki/React_(web_framework), May 2020. Accessed: 2020-06-14.

[43] Sistema de nombres de dominio. https://es.wikipedia.org/wiki/Sistema_de_nombres_de_dominio#Tipos_de_registros_DNS, Jun 2020. Accessed: 2020-06-14.

[44] Blog Universidad Europea. Cuánto cobra un ingeniero informático: Blog ue. https://universidadeuropea.es/blog/cuanto-gana-un-ingeniero-informatico#:~:text=El%20salario%20de%20un%20ingeniero,1.000%2D1.200%20euros%20netos%20mensuales. Accessed: 2020-06-09.

[45] Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, Mar 2014. Accessed: 2020-06-15.

[46] CloudBees Inc. Continuous integration: What is ci? testing, software & process tutorial. https://codeship.com/continuous-integration-essentials#:~:text=ContinuousIntegration(CI)isa,automatedbuildandautomatedtests. Accessed: 2020-06-14.

[47] Intuio @intuio. Sass variables vs. css custom properties. https://intu.io/blog/sass-css-custom-properties/, Sep 2016. Accessed: 2020-06-20.

[48] Steve O'Hear. Snyk aims to help developers secure use of open source code. https://techcrunch.com/2016/06/23/snyk/, Jun 2016. Accessed: 2020-06-15.

[49] Marcio Sete. Loosely coupled architecture. https://medium.com/@marciosete/loosely-coupled-architecture-6a2b06082316#:~:text=Writing%20software%20is%20like%20building%20Lego.&text=Loosely%20coupled%20architectures%20(aka%20Microservices,scale%20independent%2C%20increasing%20business%20responsiveness., May 2018. Accessed: 2020-06-10.

[50] Pearson eCollege Todd Fredrich. Using http methods for restful services. https://www.restapitutorial.com/lessons/httpmethods.html. Accessed: 2020-06-16.

[51] Mike Wasson. Microservices architecture style - azure application architecture guide. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices. Accessed: 2020-06-10.