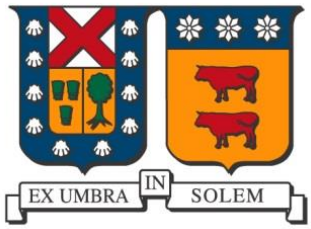
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



"OPTIMIZACIÓN DE LA ARQUITECTURA BACKEND DEL SOFTWARE VRAVA"

DIEGO EMILIO MORAGA ARAYA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Cecilia Reyes Covarrubias
Profesor Correferente: Pedro Godoy Barrera

Septiembre - 2023

DEDICATORIA

Dedico este trabajo de título a mi madre Aurora (Q.E.P.D) que dedicó su vida para criarnos y enseñarnos sus valores que me hacen la persona que soy hoy, daría todo por que estuviera aquí conmigo cerrando esta etapa de mi vida.

A mi papá que es lo más importante que tengo en esta vida y le debo todo lo que soy, a mi hermano que es el mejor que pude haber tenido y ha estado siempre para mí.

A Pepa (Q.E.P.D.), mi mascota que falleció en el transcurso de la elaboración de este proyecto luego de acompañarme 16 años desde mi infancia.

A mi profesor Pablo (Q.E.P.D.) que me enseñó e inculcó el amor por las matemáticas y la ingeniería.

AGRADECIMIENTOS

Gracias a mi familia por ser mi soporte en todo momento, en especial a mi papá y hermano quienes son las personas que más quiero en mi vida y son los que hicieron posible el que esté finalizando mi etapa universitaria en la carrera que amo y en la universidad que siempre quise.

A mi novia por estar ahí apoyándome en los momentos difíciles y en las largas noches de avance y escritura, y a mis amigos que forman parte de mi vida y han estado siempre.

A todos mis profesores que fueron parte de mi aprendizaje en la Universidad, de cada uno puedo llevarme algo que aprendí, en especial a la profesora Cecilia Reyes, quien estuvo presente desde el primer año recibiéndonos a los alumnos nuevos, hasta este momento donde es mi profesora guía.

RESUMEN

Resumen — El presente trabajo de título muestra un rediseño y desarrollo de una nueva API para la aplicación VraVa, nacida en la FESW del año 2021, con el propósito de digitalizar y gestionar los talleres de desabolladura y pintura a través de una aplicación móvil. Se desarrolló una nueva API en Ruby on Rails con el objetivo de optimizar métricas de tiempos de respuesta y consumo de recursos, guiado por la metodología de desarrollo Test Driven Development. Mediante la implementación de diversas técnicas avanzadas de diseño de software y la adopción de buenas prácticas en Rails, se lograron mejoras en cuanto a la velocidad de procesamiento, tiempos de respuesta y el uso de recursos por parte del sistema. Estos resultados son de gran relevancia, ya que no sólo mejoran la experiencia del usuario, sino que también ofrecen una base más sólida y escalable para futuras expansiones y adaptaciones del sistema, pensando además en una futura proyección comercial de VraVa. Las técnicas empleadas y los resultados obtenidos son presentados en detalle, proporcionando una visión clara de los beneficios y ventajas de la optimización propuesta.

Palabras Clave — Optimización – Test Driven Development – APIs – Ruby on Rails – VraVa

ABSTRACT

Abstract— This title work shows a redesign and development of a new API for the VraVa application, born in the FESW of the year 2021, which aimed to digitize and manage paint and dent removal workshops through a mobile application. A new API was developed in Ruby on Rails in order to optimize certain metrics such as response times and resource consumption guided by the Test Driven Development methodology. Through the implementation of several advanced software design techniques and the adoption of best practices in Rails, improvements were achieved in terms of processing speed, response times and resource usage by the system. These results are of great relevance, since they not only improve the user experience, but they also provide a more solid and scalable base for future expansions and adaptations of the system, also thinking about a future commercial projection of VraVa. The techniques employed and the results obtained are presented in detail, providing a clear view of the benefits and advantages of the proposed optimization.

Keywords— Optimization - Test Driven Development - APIs - Ruby on Rails - VraVa

GLOSARIO

API: *Application Program Interface.*

AWS: *Amazon Web Services.*

Bucket S3: Un bucket es un contenedor para objetos almacenados en Amazon S3.

CMS: *Content Management System.*

CRUD: *Create, Read, Update and Delete.* Los cuatro métodos principales para gestionar recursos y entidades en una API.

HTTP: *Hypertext Transfer Protocol* (o Protocolo de Transferencia de Hipertexto en español) es un protocolo de la capa de aplicación para la transmisión de documentos hipermedia, como HTML.

MVC: Modelo – Vista – Controlador.

PaaS: *Platform as a Service.*

REST: *Representational State Transfer.*

TDD: *Test Driven Development.*

UTFSM: Universidad Técnica Federico Santa María.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INTRODUCCIÓN

En el vasto mundo de la tecnología y el desarrollo de software, la optimización del backend se ha convertido en una necesidad primordial para las empresas que buscan ofrecer servicios digitales de alta calidad. El backend, siendo el pilar central de cualquier aplicación o sistema, determina en gran medida la eficiencia, velocidad y capacidad de respuesta de una plataforma. VraVa, siendo una entidad prominente en este ámbito, ha reconocido la importancia de mantener su sistema optimizado y al día con las últimas tendencias y tecnologías.

El área de trabajo se centra en el rediseño y optimización del backend de VraVa utilizando Rails, un marco de trabajo conocido por su robustez y eficiencia. El problema que se aborda es multifacético. Por un lado, se identificó la necesidad de mejorar la velocidad de procesamiento y la capacidad de respuesta del sistema. Por otro lado, se reconoció la importancia de tener un sistema escalable que pueda adaptarse a las crecientes demandas y cambios en el futuro.

Para abordar estos desafíos, se propuso una solución integral que no sólo se centra en la optimización técnica, sino también en la adopción de las mejores prácticas en Rails. La metodología empleada para este rediseño y optimización se basa en una revisión exhaustiva del sistema existente, identificando áreas de mejora y proponiendo soluciones innovadoras. Esta metodología no sólo garantiza mejoras tangibles en el rendimiento del sistema, sino que también proporciona una base sólida para futuras expansiones y adaptaciones.

Este documento se estructura de manera que guía al lector a través de las diferentes etapas del proceso de optimización. Comenzando con una revisión detallada del problema, seguida de la propuesta de solución y la metodología empleada, y culminando con los resultados obtenidos y su relevancia. Al finalizar la lectura de este documento, el lector tendrá una comprensión clara de cómo se abordó el desafío de optimizar el backend de VraVa en Rails y las soluciones propuestas para superar los obstáculos identificados.

1 CAPÍTULO I: DEFINICIÓN DEL PROBLEMA

1.1 CONTEXTO

La Feria de Software (FESW) es un evento que organiza el Departamento de Informática de la UTFSM año tras año, donde equipos compuestos por alumnos de la carrera Ingeniería Civil Informática presentan al público sus proyectos que desarrollaron a lo largo de dos semestres en las asignaturas de Gestión de Proyectos de Informática, que es donde se genera una idea a desarrollar y se determinan aspectos de gestión y planificación para pasar al semestre siguiente al Taller de Desarrollo de Proyectos de Informática, una asignatura que apunta a poner en práctica la metodología de proyectos a través de los conocimientos adquiridos a lo largo de la carrera, aplicándolos para obtener una solución informática a un problema real y multidisciplinario. Es bajo este contexto que nace la idea de la aplicación VraVa.

Esta aplicación presenta una solución a la necesidad de aquellas personas que precisan reparar sus vehículos dañados en talleres de desabolladura y pintura de una manera más rápida y cómoda, ofreciéndoles una plataforma donde a través de fotografías pueden hacer llegar la información necesaria a los talleres para que éstos les respondan con cotizaciones tentativas y, así finalmente, obtener presupuestos sin tener que salir de su casa ni mover el vehículo. La aplicación ofrece un análisis de detección de daños realizado por una red neuronal, que proporciona una asistencia al momento de revisar y estimar los daños presentes en las fotografías. Asimismo, VraVa busca dar un soporte orientado a la gestión interna de cada uno de los talleres pertenecientes a la plataforma, ofreciendo un sistema donde los encargados de estos puedan gestionar la información de manera digital y ordenada, algo que hoy en día es un proceso manual.

Existen tres actores en la aplicación que interactúan entre sí, en primer lugar, se encuentra el cliente del taller y dueño de un vehículo que necesite reparación. Este actor es quien tiene la posibilidad de adjuntar las fotografías de su vehículo para que con el previo análisis de la inteligencia artificial haga llegar este reporte a los talleres que él estime conveniente en su zona, quedando a la espera de una respuesta por parte de ellos. En segundo lugar, se encuentran los jefes de los talleres de desabolladura y pintura, quienes con la aplicación tienen la posibilidad de mantener un listado de vehículos y órdenes de trabajo que mantiene el taller, además de poder recibir los reportes antes mencionados generados por los clientes y responderles con presupuestos tentativos, para posteriormente coordinar o discutir un posible arreglo o acuerdo entre ambas partes. Finalmente están los maestros

del taller, quienes tienen la facultad de visualizar y modificar datos e información de las órdenes de trabajo que tienen a cargo.

1.2 SITUACIÓN ACTUAL

Actualmente, VraVa funciona como una aplicación móvil soportando tanto iOS como Android, dado que está desarrollada en React Native aprovechando el builder Expo. Mientras que para el caso del backend, se montó una API REST a través del CMS (Content Management System) Strapi, el cual ofrece un completo servicio de levantamiento de API y endpoints para las distintas colecciones desarrollado completamente en JavaScript, además de ser completamente personalizable y con un enfoque de tipo developer-first. Esta API es la encargada de realizar las consultas a las colecciones y tablas que están montadas en PostgreSQL.

En adición a esto, la red neuronal montada para la detección de objetos está desarrollada con el backend de TensorFlow y Keras, exportado para ser utilizado en JavaScript con la librería TensorFlowJS. Ésta se encuentra integrada en un endpoint personalizado dentro de la API montada en Strapi, funcionando de forma monolítica respecto al resto de funciones y métodos del backend.

1.2.1 TALLERES Y VRAVA

Los talleres de desabolladura y pintura cumplen el importantísimo rol de reparar aquellos defectos que sufren los vehículos en lo que respecta a las piezas externas, quitando abolladuras, rayones, reparando roturas o simplemente cambiando piezas por unas nuevas. En Chile, por norma general la mayoría de los talleres opera de la misma manera, el cliente lleva el vehículo al taller para que el jefe o encargado de éste lo revise e inspeccione y genere un presupuesto, detallando una serie de arreglos o cambios que deben hacerle al vehículo para que quede en óptimas condiciones. Llegados a un acuerdo, el taller se compromete con una fecha de entrega donde el cliente debe retirar su vehículo ya reparado del taller. Todo este proceso queda sin más documentación que el presupuesto mismo y la posterior orden de trabajo que genera el taller cuando comienza a trabajar en el vehículo, ambos documentos escritos a mano, salvo en ocasiones donde el presupuesto puede ser de forma digital.

Para efectos del proceso explicado en el párrafo anterior, es que VraVa aporta un gran valor para las partes involucradas. En primer lugar, cabe destacar que para obtener una cifra – aunque sea aproximada – del precio del posible arreglo no hace falta siquiera de ir al taller con el vehículo, incluso teniendo la ventaja de enviar las fotografías con el análisis de la IA a múltiples talleres al mismo tiempo. Por otro lado, VraVa cumple un rol de gestión y

documentación desde que el jefe de taller responde al cliente con un presupuesto aproximado, hasta, en el mejor de los casos, la finalización del arreglo y la entrega del vehículo al cliente. Otro valor importante que ofrece VraVa es la gestión interna para el taller, pasan de los presupuestos y órdenes de trabajo escritas a mano en papel a data estructurada y almacenada de forma digital.

1.2.2 DIFERENCIAS CON LA COMPETENCIA

VraVa se diferencia de la competencia por ser un completo conjunto de herramientas que trabajan de forma unificada e íntegra. Por un lado, se diferencia de aquellas plataformas que se especializan en gestionar y manejar el orden interno de los talleres, ya que éstas no poseen la capacidad de conectar estos talleres con potenciales clientes, ni le ofrecen a estos últimos la posibilidad de obtener presupuestos y comunicación con talleres.

En el mercado actual, en Chile no se conocen aplicaciones similares de cara a los clientes dueños de vehículos. Lo más cercano son aquellas plataformas desarrolladas por aseguradoras donde los clientes de las compañías pueden realizar solicitudes de inspección a sus vehículos, o bien registrar siniestros para cobros de pólizas.

1.3 EL PROBLEMA

1.3.1 ÁRBOL DEL PROBLEMA

Ilustración 1: Árbol del problema
Fuente: Elaboración propia.

De la Ilustración 1 se desprende una serie de causas que en conjunto generan un problema central y, provocando por consecuencia, una serie de inconvenientes y efectos negativos tanto para los usuarios como para los desarrolladores y mantenedores de VraVa.

En primer lugar, se menciona el hecho de que existe una distribución centralizada en cuanto a los distintos componentes que interactúan en el funcionamiento de la aplicación, es el caso que ocurre con la IA, la red neuronal se encuentra montada dentro del mismo backend de Strapi funcionando y haciendo predicciones con los mismos recursos, provocando en ocasiones un alto consumo y tiempos de respuesta. Esto también va en la línea de la causa descrita como estructura de monolito, al pertenecer todos estos componentes a un mismo entorno montado en la misma máquina, es un gran conjunto de funcionalidades operando en el mismo lugar, generando colapsos y altos consumos de recursos.

Por otro lado, se tiene que para el proceso de diseño de tablas y colecciones se utilizaron técnicas genéricas de normalización hasta 3FN, un proceso estándar que en la mayoría de los casos funciona de forma correcta, sin embargo, para casos como VraVa puede ser contraproducente e incluso un cierto grado de redundancia en las tablas mejoraría el rendimiento y tiempos de respuesta para algunas consultas, además de ahorrar el número de consultas necesarias para obtener o chequear ciertos datos. De aquí mismo nace la siguiente causa que relata sobre el flujo de las llamadas que en ocasiones se torna complejo y redundante, sobre todo en casos donde se deben chequear datos con grandes grados de herencia o relaciones entre las distintas tablas. Por lo general, las consultas tipo GET se hacen de forma normal y luego se filtran de forma manual en el front, generando procesos lógicos innecesarios y más retraso en cada uno de estos procesos.

Por último, se tiene además que el contexto en el que se desarrolla VraVa de tres sprints iterativos e incrementales es un contexto universitario y realizado por un equipo formado en el primer semestre de los dos que comprenden todo el desarrollo de la FESW. Es un factor para considerar dada la inexperiencia laboral que se posee, si bien a esta altura de la carrera los conocimientos son relativamente avanzados, no existe mayor experiencia en trabajos en equipo de este calibre más allá de posibles prácticas que pueda haber realizado cada estudiante de forma individual. No se tienen métricas ni comportamientos pasados del equipo en fases de desarrollo, por lo que se hace tremendamente difícil el proceso de estimación de tiempos y esfuerzos.

Todas estas causas mencionadas provocan el problema principal que apunta básicamente a un déficit en cuanto al rendimiento y desempeño de la aplicación, sobre todo en lo que

respecta a recursos utilizados, tiempos de respuesta, costos asociados al uso y mantenimiento y complejidad de código y distribución de componentes.

1.3.2 INDICADORES ACTUALES

1.3.2.1 TIEMPOS DE RESPUESTA

La IA que se encarga de detectar los daños presentes en las fotografías requiere una gran cantidad de esfuerzo computacional, por lo que dependiendo de cómo esté configurada o qué tipo de plataforma sea la instancia que aloje la red neuronal, esta demorará más o menos tiempo en responder con las predicciones.

VraVa a lo largo de los tres sprints de desarrollo, presentó dos versiones de IA, la primera y más básica, trataba de una simple clasificación de imágenes etiquetándolas en base al daño que identificara la red neuronal en ellas. La segunda versión y final, consta de una detección de objetos, marcando y encerrando en cuadrados los daños encontrados, pudiendo ser múltiples y de distinto tipo para una sola imagen.

Tiempos de respuesta IA inicial (básica):

- Caso particular para la primera predicción, en promedio 5000 [ms]
- Tiempo promedio por imagen: 1000 [ms]
- Rango promedio: 800 – 1200 [ms]

Tiempos de respuesta IA final (detección de objetos):

- Caso particular para la primera predicción, en promedio 40000 [ms]
- Tiempo promedio por imagen: 4000 [ms]
- Rango promedio: 3800 – 5200 [ms]

Data obtenida de una serie de pruebas unitarias a la IA, en específico 10 solicitudes independientes y con imágenes distintas.

En adición a esto, de la experiencia con el usuario se tuvo que, en una de las tareas más importantes para estos, el tiempo de espera de la IA (básica) representa el 12,5% del total de tiempo que le toma al usuario completar la tarea. Esta tarea consistía en agregar un vehículo a la lista del taller, al usuario le tomó 128 [s] completar la tarea.

1.3.2.2 INDICADOR DE RECURSOS UTILIZADOS EN PROMEDIO POR USO

Actualmente, el recurso que más se utiliza por parte del backend es la memoria RAM. La Ilustración 2, muestra la utilización de este recurso a lo largo de 7 días donde no hubo mayor tráfico ni uso constante, se puede notar que luego de un re deploy (línea vertical punteada), el sistema liberó considerablemente la memoria ocupada.

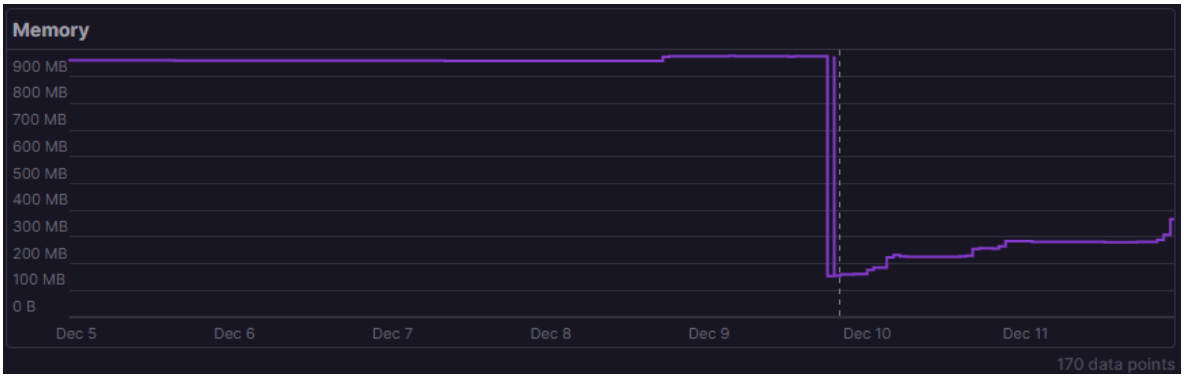


Ilustración 2: Gráfica de Memoria
Fuente: Elaboración propia. Obtenida desde el panel de Railway.

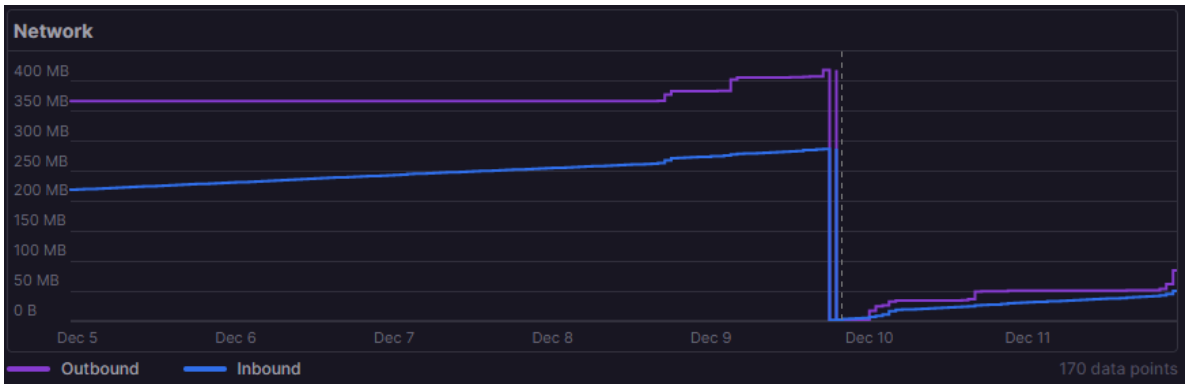


Ilustración 3: Gráfica de Trafico de Red
Fuente: Elaboración propia. Obtenida desde el panel de Railway.

La Ilustración 3, destaca el alto tráfico de red acumulado por la aplicación, producto de la cantidad de consultas HTTP que se realizan en cada uno de los procesos de la aplicación.

1.3.2.3 CANTIDAD DE CONSULTAS A LA API

Para el caso de uno de los procesos más importantes de la aplicación, el subir un auto a los registros del taller, la aplicación realiza de forma fija 5 consultas y 2 por cada imagen que se suba del vehículo, es decir, para subir un vehículo con 4 imágenes analizadas, se requiere un total de 13 consultas HTTP. Además de eso, algunas de esas consultas que

requieren un post procesado para filtrar o iterar la data obtenida, implicando mayor tiempo de espera y cómputo.

Procesos más simples como el de inicio de sesión, requieren aparte de las consultas de autenticación, dos consultas adicionales, para chequear casos como: si el usuario es un jefe de taller, maestro o cliente.

1.3.2.4 COSTO MENSUAL DE MANTENCIÓN

VraVa está funcionando alojada en la plataforma Railway, bajo el plan “Developer”. Este tiene un costo mensual de USD 10. Para el caso de la IA desarrollada con detección de objetos, se encuentra alojada en Google Cloud con un plan estudiantil, por lo que no genera costos asociados, sin embargo, en un entorno de producción requeriría un plan más avanzado.

1.3.3 DIAGRAMA DE CLASES

En la Ilustración 4, se presenta el diagrama de clases dentro del actual backend desarrollado en Strapi y cómo interactúan entre sí las clases u objetos para dar contexto a la solución que se desarrolló en su momento, en respuesta a la necesidad de un backend o API que gestionara los datos de la aplicación.

Analizando la Ilustración 4 es posible notar varios detalles que dan pie a posibles mejoras y oportunidades de optimización. En primer lugar, entidades como JefeTaller, Cliente y Maestro son claramente optimizables, ya que tienen datos repetitivos que aumentan la redundancia de estos, incluso las entidades JefeTaller y Maestro podrían ser eliminadas directamente, pues su único rol es distinguir de un usuario jefe o administrador de taller de otro usuario como el cliente mismo.

Por otro lado, hay una clara redundancia de datos en la entidad solicitud, orden de trabajo y auto, como se puede apreciar datos inherentes del auto se ven repetidos en solicitud (marca, modelo, año) y otros datos asociados al análisis de la inteligencia artificial se ven replicados tanto en la entidad orden de trabajo como en solicitud.

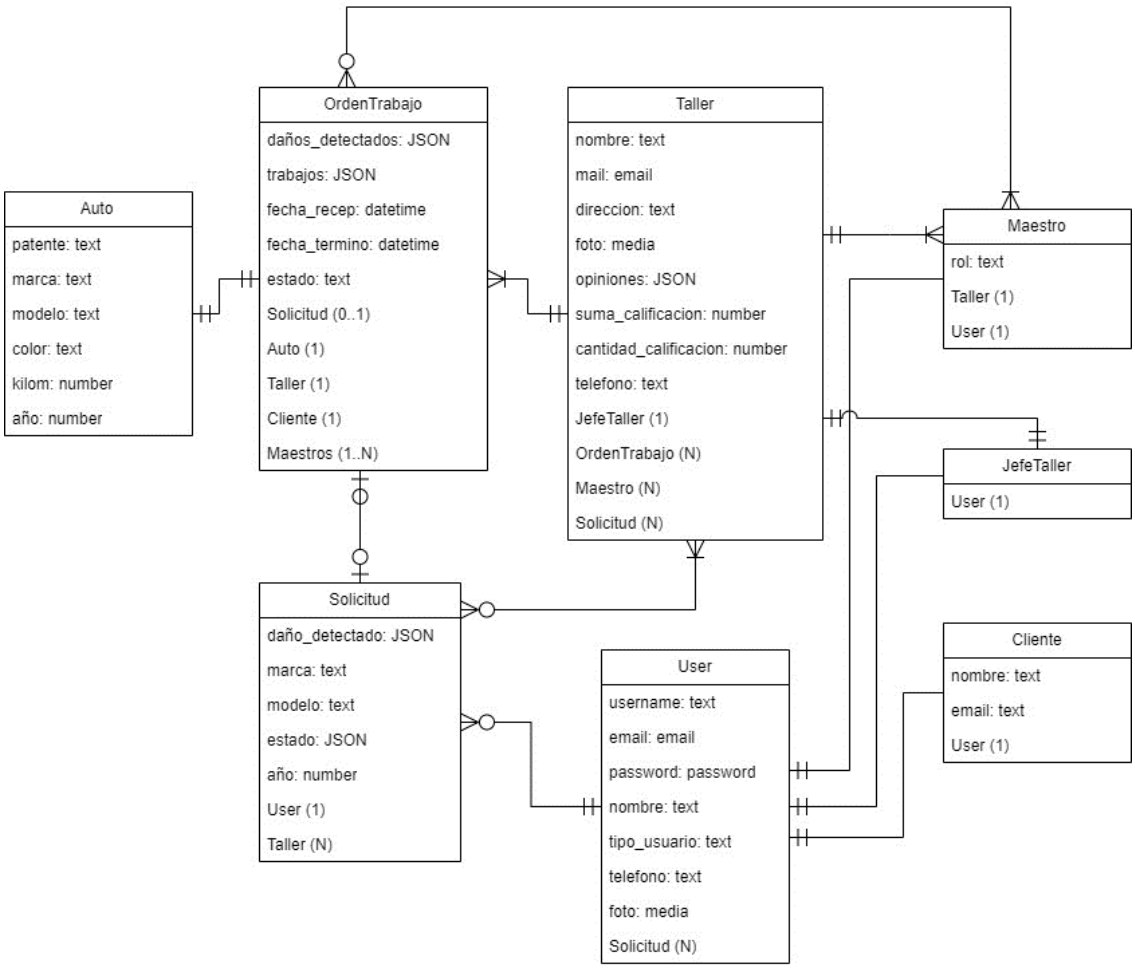


Ilustración 4 Diagrama de clases de Strapi.
Fuente: Elaboración propia

En cuanto a problemas más enfocados en la lógica de negocio, se hace cuestionable la decisión de involucrar a los maestros dentro del flujo de VraVa, el único rol que cumplen es el de asociarles un trabajo en una orden en específico, lo cual no aporta de manera considerable a los objetivos que persigue la aplicación y la propuesta de valor de esta. De hecho, el perfil maestro sería el equivalente a un perfil como el de jefe o administrador de taller, pero con permisos y acciones más limitadas.

1.4 SOLUCIÓN PROPUESTA

1.4.1 OBJETIVO GENERAL

Rediseñar el backend del software VraVa, de tal forma de optimizar el rendimiento actual y buscando disminuir los tiempos de respuesta, recursos utilizados y costos asociados al uso y mantenimiento de la aplicación.

1.4.2 OBJETIVOS ESPECÍFICOS

1. Medir la cantidad de consultas a la API REST de los flujos de información de las funcionalidades importantes para definir y probar una estrategia de optimización del flujo lógico que permita disminuir la cantidad de llamadas a la API, mejorando el uso y mantenimiento de la aplicación.

2. Desarrollar una API en Ruby on Rails con la metodología Test Driven Development (TDD) para mejorar la calidad del código y su mantenimiento.

3. Evaluar opciones de tecnologías para levantar la instancia del backend comparando sus costos, de tal manera de encontrar la alternativa más rentable para el desarrollador.

2 CAPÍTULO II: MARCO CONCEPTUAL

2.1 OPTIMIZACIÓN DE CONSULTAS

Para optimizar las secuencias de llamadas o las iteraciones para filtrar los mismos resultados de las llamadas resulta muy útil la aplicación de los *custom endpoints*, una posibilidad de personalizar los controladores del sistema MVC que provee el mismo CMS Strapi.

Los controladores son precisamente archivos JavaScript que contienen un conjunto de métodos, llamados acciones, que son alcanzados por el cliente dada una ruta solicitada. Siempre que un cliente solicita una ruta, la acción ejecuta el código de la lógica desarrollada en el método y retorna la respuesta. Los controladores por defecto son creados para cada tipo de contenido por Strapi, que son los que se usan de forma estándar en cada una de las llamadas que llegan a la API, es aquí donde resalta una gran cualidad de Strapi, ya que permite personalizar estos controladores predeterminados para implementar una lógica propia y a medida del desarrollador, es decir, cada uno de los controladores se puede ajustar o modificar como el desarrollador estime conveniente. (Strapi Inc., 2021)

En la mayoría de los casos, los controladores contendrán la mayor parte de la lógica interna de un proyecto. Sin embargo, a medida que esta lógica se torna más compleja y va escalando, es buena práctica utilizar servicios para organizar el código en partes reutilizables. Este es otro factor interesante, ya que al igual que los controladores, Strapi permite crear servicios y agregarlos al sistema completo como si fueran componentes, permitiendo que estas partes de código reutilizables interactúen o sean utilizadas por cualquier controlador, proporcionando un mayor grado de encapsulamiento al código y granularidad en cuanto a sus componentes.

Otra opción para abarcar la optimización de consultas o ahorrar un filtrado manual a la respuesta de la API, es incluir los filtros que sean necesarios en la misma URL a la que se está haciendo la llamada, Strapi ofrece esta posibilidad con una amplia variedad de filtros y condiciones para cada uno de los *endpoints* del sistema, además de ofrecer la posibilidad de ordenar las respuestas. Sin embargo, es una opción que se recomienda en casos sencillos y sin mucha profundidad, ya que al aumentar el nivel de profundidad de los filtros puede disminuir el rendimiento y aumentar el tiempo de respuesta de la solicitud. Strapi recomienda para estos casos construir lo mencionado anteriormente, una ruta personalizada acompañada de un controlador que se encargue de filtrar la respuesta. (Strapi Inc., 2021)

2.1.1 COMPARACIÓN DEL RENDIMIENTO DE LAS CONSULTAS

En esta parte del escrito, se compararán consultas de una API desarrollada en un framework en un lenguaje con las de otra API desarrollada en otro framework en otro lenguaje distinto, es importante destacar que se considerarán flujos claves de consultas y tráfico de datos vía HTTP de la aplicación, se compararán en cuanto a número de consultas, tamaño de la respuesta en bytes, tiempo de respuesta y tasa de errores. Es decir, si se considera un flujo como “obtener lista de talleres” u “obtener órdenes pertenecientes a un usuario”, se compararán estas métricas mencionadas anteriormente en cada API con tal de obtener una tasa de variación porcentual, ya sea de mejora o un decrecimiento en el rendimiento de la API en Ruby on Rails respecto a la API en Strapi.

2.1.2 PATRÓN N+1 Y SU OPTIMIZACIÓN EN RUBY ON RAILS

El patrón de las N+1 consultas es un problema de rendimiento en el que la aplicación realiza consultas a la base de datos en un bucle, en lugar de hacer una única consulta que devuelva toda la información de una sola vez. Cada conexión a la base de datos lleva cierto tiempo, por lo que consultar la base de datos en bucle puede ser mucho más lento que hacerlo una sola vez. Este problema suele ocurrir cuando se utiliza una herramienta de mapeo objeto-relacional (ORM) en frameworks web como Django o Ruby on Rails. (Sentry, s.f.)

Se produce cuando el código necesita cargar los registros de una asociación uno a muchos, considerando específicamente un ejemplo, un modelo *post* que tiene muchos *comments* asociados. En una situación típica, la aplicación podría tener que cargar todos los *posts* y, para cada *post*, cargar también sus *comments* asociados. Si hay N *posts*, este proceso resultaría en 1 consulta para obtener los *posts*, y luego N consultas adicionales para obtener los *comments* de cada *post*, de ahí el nombre "N+1".

Por ejemplo, si se tiene un *blog* que muestra una lista de *posts* junto con el número de *comments* para cada uno. La implementación inicial podría ser como se muestra en Ilustración 5:

```ruby
# Controlador
@posts = Post.all

# Vista
@posts.each do |post|
  puts "#{post.title}: #{post.comments.count} comments"
end
```

Ilustración 5 Fragmento de código ejemplo N+1
Fuente: Elaboración propia.

Aquí, el código realiza una consulta para obtener todos los *posts*, y luego, para cada *post*, realiza una consulta adicional para contar sus *comments*. Si hay 100 *posts*, se realizarán 101 consultas en total.

Sin embargo, como señala Sentry, este enfoque puede resultar ineficiente y lento a medida que la cantidad de datos aumenta. Idealmente, se desea minimizar el número de consultas a la base de datos. En Rails, una forma de resolver el problema de las consultas N+1 es utilizando el método *includes*, que precarga los datos asociados. Esto significa que Rails cargará todos los *posts* y sus *comments* correspondientes en solo 2 consultas, independientemente de cuántos *posts* existan, como se indica en la Ilustración 6:

```ruby
# Controlador
@posts = Post.includes(:comments)
```

Ilustración 6 Fragmento de código solución N+1.
Fuente: Elaboración propia.

2.2 ESTRATEGIAS DE REDISEÑO DE TABLAS

2.2.1 DESNORMALIZACIÓN

Las reglas de normalización no consideran el rendimiento. En algunos casos, es necesario considerar la desnormalización para mejorar el rendimiento. (“Normalización de Bases de Datos - UNAM”)

La normalización de tablas es la estrategia más recomendada a utilizar en bases de datos relacionales, su premisa es que las sentencias de SQL pueden recuperar la información uniendo las tablas. El problema es que, en algunos casos, se pueden producir problemas de rendimiento como resultado de una normalización. Por ejemplo, algunas consultas de

usuario pueden ver datos que están en una o más tablas relacionadas; el resultado es demasiadas uniones. A medida que crece el número de tablas, los costes de acceso pueden aumentar según el tamaño de las tablas, los índices disponibles, etc. Por ejemplo, si no hay índices disponibles, la unión de numerosas tablas grandes puede tardar demasiado tiempo, puede ser necesario desnormalizar las tablas. La desnormalización es la duplicación intencionada de columnas en varias tablas y esto aumenta la redundancia de datos. (IBM Corporation, 2021)

En la etapa de diseño de las tablas de una base de datos relacional, es cuando debe tomarse la decisión acerca de si se desnormalizan o no los datos. Específicamente, se necesita decidir si deben combinarse tablas o partes de tablas a las que accedan con frecuencia uniones que tienen requisitos de alto rendimiento. Para tomar esta decisión se necesita evaluar los requisitos de rendimiento, los diferentes métodos de acceder a los datos y los costes de desnormalización de los datos. Se debe tener en cuenta el coste y el resultado; ¿es la duplicación, en varias tablas, de columnas solicitadas con frecuencia menos costosa que el tiempo de llevar a cabo las uniones? (IBM Corporation, 2021)

2.2.2 TOP-DOWN VS BOTTOM-UP

El método *top-down* parte de lo general y pasa a lo específico. Básicamente, comienza con una idea general de lo que se necesita para el sistema y luego pregunta a los usuarios finales qué datos necesitan almacenar en la base de datos. El uso de este método requiere que se tenga un conocimiento detallado del sistema. En algunos casos, el diseño *top-down* puede conducir a resultados insatisfactorios porque el analista y los usuarios finales pueden pasar por alto algo que es importante y necesario para el sistema.

Este método enfatiza en un enfoque inicial basado en el conocimiento de constructos de nivel superior, como la identificación de poblaciones y colecciones de elementos y tipos de entidades, reglas de pertenencia y asociaciones entre dichas poblaciones. La adopción de un enfoque *top-down* generalmente comenzará con un conjunto de requisitos de alto nivel, como una narrativa. Estos requisitos inician un proceso de identificación de los tipos de elementos necesarios para representar datos, así como los atributos de esos elementos, que pueden convertirse en atributos en tablas. En la tradición del diseño de bases de datos *top-down*, el analista intenta inicialmente desarrollar un modelo de datos conceptual identificando objetos de datos altamente abstractos que pueden existir dentro del dominio, es decir, intenta construir una especie de ontología del dominio. Las técnicas que aplica el analista suelen incluir la realización de observaciones, entrevistas y otras estrategias de recopilación de datos. Por lo general, la inspiración para el modelo de datos también proviene de un análisis detallado de las reglas comerciales del dominio. Además,

se identifican las propiedades estructurales, como las relaciones entre los tipos de entidad y la cardinalidad de la relación. (Hsiang-Jui Kung, 2013)

Por el contrario, un enfoque *bottom-up* considera que el diseño de la base de datos procede de un análisis inicial de los elementos conceptuales de nivel inferior, como los atributos y las dependencias funcionales, y que luego se avanza hacia un modelo de datos lógico aceptable mediante agrupaciones lógicas de los atributos asociados. En otras palabras, el enfoque *bottom-up* tiende a considerar la tarea de identificación de la población como un proceso de generalización de la identidad de los objetos a partir de ejemplos de dependencias estructurales. La entrada en un enfoque *bottom-up*, por ejemplo, podría ser vistas de datos, como capturas de pantalla o informes (impresiones), o patrones de valores de atributos concurrentes identificados en grandes conjuntos de datos. Al abordar las posibles deficiencias en el diseño de un esquema relacional asociadas a diferentes niveles de forma normal, las relaciones se definen para minimizar la redundancia y la dependencia. (Hsiang-Jui Kung, 2013)

2.2.3 USO DE ÍNDICES

En los RDBMS (Sistemas de Gestión de Bases de Datos Relacionales), los índices son un objeto especial que permite al usuario recuperar rápidamente registros de la base de datos. Normalmente, un índice se implementa como una tabla de consulta que sólo tiene dos columnas: la primera columna contiene una copia de la clave primaria o candidata de una tabla; la segunda columna contiene un conjunto de punteros para mantener la dirección del bloque de disco donde se almacena ese valor clave específico. (Gravelle, 2021)

Los tipos de índices pueden clasificarse en función de sus atributos de indexación. Se dividen en dos categorías principales: índices primarios e índices secundarios.

Un índice primario es un archivo ordenado cuyos registros son de longitud fija con dos campos. El primer campo del índice replica la clave primaria del archivo de datos de forma ordenada, y el segundo campo contiene un puntero que apunta al bloque de datos en el que está disponible un registro que contiene la clave.

Los índices secundarios son índices que almacenan el valor de la clave primaria en lugar de almacenar un puntero a los datos. La ventaja es que, al acceder a los datos a través de una clave primaria, no es necesario realizar ninguna búsqueda de datos adicional, ya que todos los datos que se necesitan se pueden encontrar en las hojas de la clave primaria. (Gravelle, 2021)

El uso de índices es una estrategia para optimizar el rendimiento de las bases de datos relacionales. Un índice en una base de datos es similar a un índice en un libro: proporciona un acceso rápido a los datos sin tener que buscar en cada fila de una tabla, de la misma manera que un índice de un libro permite al lector encontrar información específica rápidamente sin tener que leer cada página.

2.3 CONSUMO DE RECURSOS

Hay dos tipos de medición del uso de recursos en la nube. Al igual que la monitorización convencional, el consumo de recursos en la nube se basa en los datos de uso actual de la CPU, memoria, red y tráfico de disco; éstos cambian dinámicamente según el tráfico, y son importantes para validar el estado del sistema con frecuencia. El otro tipo de medición es la cantidad de recursos asignados, es como un sistema de contabilidad que lleva el control de los recursos asignados. En un entorno de recursos compartidos, que es un concepto fundamental en la informática de servicios, la cantidad de recurso asignado a un usuario implica que el recurso solicitado será dominado por este usuario sin ser interrumpido por ningún otro. La asignación de recursos no mide el uso de recursos en tiempo real. En cambio, registra los recursos alquilados en un libro de contabilidad para la facturación y el cobro. Hay métricas estáticas para asignación como el número asignado de núcleos de CPU, memorias y discos. (Lee, von Laszewski, & Wang, 2014)

Tabla 1: Algunas de las métricas más utilizadas en la literatura.
Fuente: (Lee, von Laszewski, & Wang, 2014)

Resource Utilisation			
Network	The network bandwidth between two hosts	Network Capacity (NC)	byte/s
Network Utilisation	The aggregate network bandwidth between two hosts	Network Usage (NU)	byte
Memory	The total memory available on a host	Memory Capacity (MC)	byte
Memory Utilisation	The aggregate memory usage of a host	Memory Usage (MU)	byte
Processor	The processing power of a host	Processor Capacity (PC)	int/s
Processor Utilisation	The aggregate processor usage of a host	Processor Usage (PU)	int

Para efectos de monitoreo y obtención de métricas de uso y consumo de recursos se utilizará New Relic, una herramienta web que permite monitorear aplicaciones web y todas las transacciones que se reciban por parte de las API en este caso, además de los recursos que utilicen las aplicaciones.

2.4 COSTOS DE PLATAFORMAS

Existen distintos costos que abarca el desarrollo de la aplicación y el sistema en general, el primero y más importante es el de mantener el *backend* alojado en *cloud*. Son muchas las posibilidades para este apartado, en VraVa se ha ocupado Railway para el alojo de la API en el modo "Developer" con un costo mensual de $USD 10 en caso de superar ciertos límites de consumo. Como alternativa, está la plataforma Heroku, con una oferta considerablemente más amplia en cuanto a configuraciones del servicio, pero con un costo en promedio superior.

Si bien es cierto, este proyecto de memoria considera al componente IA desacoplado del *backend* que hoy por hoy se encuentra acoplado funcionando como monolito junto a la API Strapi, es importante mencionar y destacar que una vez desacoplado, actuará como artefacto mantenible, independiente y que debe estar considerado en los costos totales de VraVa. Es importante también considerar que dado el alto consumo de recursos que necesita la IA para predecir y detectar los daños, es muy posible que haya costos asociados a la frecuencia de uso de la IA. Es decir, en una u otra plataforma será más o menos conveniente dependiendo del uso y la frecuencia con la que se analizan imágenes.

2.5 TEST DRIVEN DEVELOPMENT (TDD)

El desarrollo basado en pruebas (TDD), que tiene sus raíces en la programación extrema, consiste en garantizar al equipo que el código funciona según lo esperado para un comportamiento o caso de uso. En lugar de buscar la solución óptima a la primera, el código y las pruebas se construyen de forma iterativa, caso por caso. Los equipos de desarrollo utilizan TDD como parte de muchas disciplinas de codificación para garantizar la cobertura de las pruebas, mejorar la calidad del código, sentar las bases de su canal de entrega y apoyar la entrega continua. (Acosta & Gajda, s.f.)

Esta metodología replantea la linealidad del proceso de desarrollo y lo hace ver como un ciclo iterativo que tiene como condición de término, en primer lugar, desarrollar hasta aprobar la prueba unitaria escrita, para luego de forma general, pasar la totalidad de las pruebas descritas para el conjunto de requisitos definidos para la funcionalidad que se está desarrollando (ver Ilustración 7).

El concepto básico del TDD es que todo el código de producción se escribe en respuesta a un caso de prueba. Robert C. Martin (Martin, 2005), conocido como *"Uncle Bob"*, describe las tres leyes del TDD:

1. No está permitido escribir código de producción a menos que sea para hacer pasar una prueba unitaria que falla.

2. No está permitido escribir más pruebas unitarias de las que sean suficientes para fallar; y los fallos de compilación son fallos.

3. No está permitido escribir más código de producción del que sea suficiente para superar la prueba unitaria fallida.

Tal y como menciona “Uncle Bob”, el desarrollador debe iniciar su trabajo escribiendo una prueba unitaria para la funcionalidad que busca desarrollar. Sin embargo, la regla número dos limita el alcance de esa prueba unitaria. Si el código de dicha prueba no logra compilarse correctamente, o si una aserción resulta fallida, es necesario que deje de escribir la prueba y pase a la escritura del código. Ahora bien, bajo la premisa de la regla número tres, el desarrollador solo debe escribir el código que permita la compilación exitosa de la prueba o su aprobación, excluyendo cualquier otro código adicional.

Si se realiza un análisis más detallado, se podrá entender que no se puede proceder a la escritura extensiva de un código sin realizar alguna compilación y ejecución. De hecho, esta es la meta principal. No importa si se está escribiendo pruebas, desarrollando el código o realizando una refactorización, la norma es mantener el sistema en funcionamiento en todo momento. El intervalo entre la ejecución de las pruebas es bastante breve, contado en segundos o minutos. Un periodo de 10 minutos, por ejemplo, se considera excesivamente largo. (Martin, 2005)

Dejando de lado un poco la formalidad con la que lo expresa Robert Martin (Uncle Bob), llega la simplificación con la que lo plantea Javier Saldana (Saldana, s.f.), llevando estas tres reglas a solo dos y más concisas:

1. Escriba sólo lo suficiente de una prueba unitaria para que falle.

2. Escriba solo el código de producción suficiente para que la prueba unitaria que falla pase.

Lo propuesto por Saldana hace más fácil el entendimiento y el constante recordatorio necesario a la hora de desarrollar, se debe mantener en este ciclo constantemente para desarrollar código de calidad.

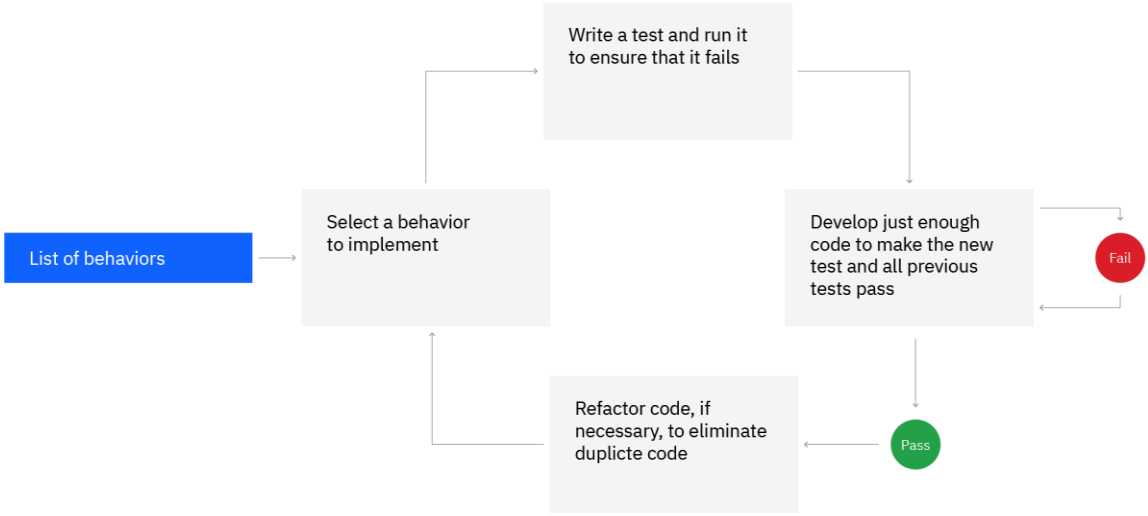


Ilustración 7 Ciclo TDD.
Fuente: (Acosta & Gajda, s.f.) – IBM Garage Methodology – Test Driven Development

2.6 ARQUITECTURA DE LA API

2.6.1 API

Una API o interfaz de programación de aplicaciones es un conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de las aplicaciones. Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero. Las API otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones; y ofrecen oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales). (Redhat, 2023)

A veces, las API se consideran como contratos, con documentación que representa un acuerdo entre las partes: si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte.

En resumen, las API le permiten habilitar el acceso a sus recursos y, al mismo tiempo, mantener la seguridad y el control. El desarrollador decide cómo habilita el acceso y a quiénes se lo otorga. La seguridad de las API depende de su buena gestión, lo cual incluye el uso de una puerta de enlace de API. (Redhat, 2023)

Se puede pensar en una API Web como una puerta de enlace entre los clientes y los recursos de la Web. Los clientes son usuarios que desean acceder a información desde la Web. El cliente puede ser una persona o un sistema de software que utiliza la API. Por ejemplo, los desarrolladores pueden escribir programas que accedan a los datos del tiempo desde un sistema de clima. También se puede acceder a los mismos datos desde el navegador cuando se visita directamente el sitio web de clima. (Amazon Web Services, s.f.)

Por otro lado, los recursos son la información que diferentes aplicaciones proporcionan a sus clientes. Los recursos pueden ser imágenes, videos, texto, números o cualquier tipo de datos. La máquina encargada de entregar el recurso al cliente también recibe el nombre de servidor. Las organizaciones utilizan las API para compartir recursos y proporcionar servicios web, a la vez que mantienen la seguridad, el control y la autenticación. Además, las API las ayudan a determinar qué clientes obtienen acceso a recursos internos específicos. (Amazon Web Services, s.f.)

2.6.2 API REST

La transferencia de estado representacional (REST) es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API. En un principio, REST se creó como una guía para administrar la comunicación en una red compleja como Internet. Es posible utilizar una arquitectura basada en REST para admitir comunicaciones confiables y de alto rendimiento a escala. Se puede implementar y modificar fácilmente, lo que brinda visibilidad y portabilidad entre plataformas a cualquier sistema de API.

Los desarrolladores de API pueden diseñar API por medio de varias arquitecturas diferentes. Las API que siguen el estilo arquitectónico de REST se llaman API REST. Los servicios web que implementan una arquitectura de REST son llamados servicios web RESTful. El término API RESTful suele referirse a las API Web RESTful. Sin embargo, los términos API REST y API RESTful se pueden utilizar de forma intercambiable. (Amazon Web Services, s.f.)

Las API RESTful incluyen los siguientes beneficios:

- Escalabilidad: Los sistemas que implementan API REST pueden escalar de forma eficiente porque REST optimiza las interacciones entre el cliente y el servidor. La tecnología sin estado (*stateless*) elimina la carga del servidor porque este no debe retener la información de solicitudes pasadas del cliente, los clientes pueden solicitar recursos en cualquier orden y todas las solicitudes son sin estado (*stateless*) o están aisladas del resto. El almacenamiento en caché bien administrado elimina de forma parcial o total algunas interacciones entre el cliente y el servidor. Todas

estas características admiten la escalabilidad, sin provocar cuellos de botella en la comunicación que reduzcan el rendimiento. (Amazon Web Services, s.f.)

- Flexibilidad: Los servicios web RESTful admiten una separación total entre el cliente y el servidor. Simplifican y desacoplan varios componentes del servidor, de manera que cada parte pueda evolucionar independientemente. Los cambios de la plataforma o la tecnología en la aplicación del servidor no afectan la aplicación del cliente. La capacidad de ordenar en capas las funciones de la aplicación aumenta la flexibilidad aún más. Por ejemplo, los desarrolladores pueden efectuar cambios en la capa de la base de datos sin tener que volver a escribir la lógica de la aplicación. (Amazon Web Services, s.f.)

- Independencia: Las API REST son independientes de la tecnología que se utiliza. Se puede escribir aplicaciones del lado del cliente y del servidor en diversos lenguajes de programación, sin afectar el diseño de la API. También se puede cambiar la tecnología subyacente en cualquiera de los lados sin que se vea afectada la comunicación. (Amazon Web Services, s.f.)

¿Cómo funcionan las API RESTful? La función básica de una API RESTful es la misma que navegar por Internet. Cuando requiere un recurso, el cliente se pone en contacto con el servidor mediante la API. Los desarrolladores de API explican cómo el cliente debe utilizar la API REST en la documentación de la API de la aplicación del servidor. A continuación, se indican los pasos generales para cualquier llamada a la API REST:

1. El cliente envía una solicitud al servidor. El cliente sigue la documentación de la API para dar formato a la solicitud de una manera que el servidor comprenda.

2. El servidor autentica al cliente y confirma que éste tiene el derecho de hacer dicha solicitud.

3. El servidor recibe la solicitud y la procesa internamente.

4. Luego, devuelve una respuesta al cliente. Esta respuesta contiene información que dice al cliente si la solicitud se procesó de manera correcta. La respuesta también incluye cualquier información que el cliente haya solicitado.

¿Qué contiene la solicitud del cliente de la API RESTful? Las API RESTful requieren que las solicitudes contengan los siguientes componentes principales:

- Identificador único de recursos: El servidor identifica cada recurso con identificadores únicos de recursos. En los servicios REST, el servidor por lo general identifica los recursos mediante el uso de un localizador uniforme de recursos

(URL). El URL especifica la ruta hacia el recurso. Un URL es similar a la dirección de un sitio web que se ingresa al navegador para visitar cualquier página web. El URL también se denomina punto de conexión de la solicitud y especifica con claridad al servidor qué requiere el cliente. (Amazon Web Services, s.f.)

- Método: Los desarrolladores a menudo implementan API RESTful mediante el uso del protocolo de transferencia de hipertexto (HTTP). Un método de HTTP informa al servidor lo que debe hacer con el recurso. A continuación, se indican cuatro métodos de HTTP comunes:
    - GET: Los clientes utilizan GET para acceder a los recursos que están ubicados en el URL especificado en el servidor. Pueden almacenar en caché las solicitudes GET y enviar parámetros en la solicitud de la API RESTful para indicar al servidor que filtre los datos antes de enviarlos.
    - POST: Los clientes usan POST para enviar datos al servidor. Incluyen la representación de los datos con la solicitud. Enviar la misma solicitud POST varias veces produce el efecto secundario de crear el mismo recurso varias veces.
    - PUT: Los clientes utilizan PUT para actualizar los recursos existentes en el servidor. A diferencia de POST, el envío de la misma solicitud PUT varias veces en un servicio web RESTful da el mismo resultado.
    - DELETE: Los clientes utilizan la solicitud DELETE para eliminar el recurso. Una solicitud DELETE puede cambiar el estado del servidor. Sin embargo, si el usuario no cuenta con la autenticación adecuada, la solicitud fallará. (Amazon Web Services, s.f.)
- Encabezados de HTTP: Los encabezados de solicitudes son los metadatos que se intercambian entre el cliente y el servidor (Amazon Web Services, s.f.). Por ejemplo, el encabezado de la solicitud indica el formato de la solicitud y la respuesta, proporciona información sobre el estado de la solicitud, etc.
- Datos: Las solicitudes de la API REST pueden incluir datos para que los métodos POST, PUT y otros métodos HTTP funcionen de manera correcta. (Amazon Web Services, s.f.)
- Parámetros: Las solicitudes de la API RESTful pueden incluir parámetros que brindan al servidor más detalles sobre lo que se debe hacer. A continuación, se indican algunos tipos de parámetros diferentes:

- Los parámetros de ruta especifican los detalles del URL.
- Los parámetros de consulta solicitan más información acerca del recurso.
- Los parámetros de *cookie* autentican a los clientes con rapidez. (Amazon Web Services, s.f.)

¿Qué contiene la respuesta del servidor de la API RESTful? Los principios de REST requieren que la respuesta del servidor contenga los siguientes componentes principales:

- Línea de estado: La línea de estado contiene un código de estado de tres dígitos que comunica si la solicitud se procesó de manera correcta o dio error. Por ejemplo, los códigos 2XX indican el procesamiento correcto, pero los códigos 4XX y 5XX indican errores. Los códigos 3XX indican la redirección de URL. A continuación, se enumeran algunos códigos de estado comunes:
  - 200: respuesta genérica de procesamiento correcto
  - 201: respuesta de procesamiento correcto del método POST
  - 400: respuesta incorrecta que el servidor no puede procesar
  - 404: recurso no encontrado
- Cuerpo del mensaje: El cuerpo de la respuesta contiene la representación del recurso. El servidor selecciona un formato de representación adecuado en función de lo que contienen los encabezados de la solicitud. Los clientes pueden solicitar información en los formatos XML o JSON, lo que define cómo se escriben los datos en texto sin formato. (Amazon Web Services, s.f.)
- Encabezados: La respuesta también contiene encabezados o metadatos acerca de la respuesta. Estos brindan más contexto sobre la respuesta e incluyen información como el servidor, la codificación, la fecha y el tipo de contenido. (Amazon Web Services, s.f.)

2.7 RUBY, RAILS Y ACTIVERECORD

2.7.1 RUBY, EL LENGUAJE DE PROGRAMACIÓN

Tal y como se describe en la página oficial del propio Ruby, es un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla. (Ruby, s.f.)

Su creador, Yukihiro “Matz” Matsumoto, mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada y Lisp) para formar un nuevo lenguaje que incorporara tanto la programación funcional como la imperativa. A menudo ha manifestado que está “tratando de hacer que Ruby sea natural, no simple”, de una forma que se asemeje a la vida real. (Ruby, s.f.)

Continuando sobre esto, agrega: “Ruby es simple en apariencia, pero complejo por dentro, como el cuerpo humano.” (Matsumoto, 2000)

Inicialmente, Matz buscó en otros lenguajes para encontrar la sintaxis ideal. Recordando su búsqueda, dijo, “quería un lenguaje que fuera más poderoso que Perl, y más orientado a objetos que Python”. En Ruby, todo es un objeto. Se le puede asignar propiedades y acciones a toda información y código. La programación orientada a objetos llama a las propiedades variables de instancia y las acciones son conocidas como métodos.

Ruby es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer. Se puede agregar funcionalidad a partes ya existentes. Ruby intenta no restringir al desarrollador. (Ruby, s.f.)

Los bloques de Ruby son también vistos como una fuente de gran flexibilidad. El desarrollador puede anexar una cláusula a cualquier método, describiendo cómo debe actuar. La cláusula es llamada bloque y se ha convertido en una de las más famosas funcionalidades para los recién llegados a Ruby que vienen de otros lenguajes imperativos como PHP o Visual Basic. (Ruby, s.f.)

2.7.2 RUBY ON RAILS, EL FRAMEWORK

Ruby on Rails (o "Rails") es un marco de desarrollo de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby. Es una de las bibliotecas Ruby más populares y una de las principales razones por las que los desarrolladores deciden aprender Ruby.

Las aplicaciones web modernas pueden ser muy complejas y tener muchas capas. Rails facilita el desarrollo web, proporcionando una estructura preconstruida para el desarrollo y todo lo necesario para construir una aplicación web. Un framework precisamente facilita y simplifica la creación de éstas. Lo hace proporcionando estructuras predeterminadas para el código, cualquier base de datos que se utilice y las páginas web que servirá la aplicación. (Miller, 15)

Se puede pensar en un framework casi como en los Legos. Con un framework, se obtienen "Legos" preconstruidos de código que se pueden mezclar, combinar y modificar para

construir una aplicación web personalizada, lo que significa que no se debe crear todo desde cero.

Ruby on Rails utiliza el patrón arquitectónico Modelo-Vista-Controlador (MVC) utilizado por muchos otros frameworks web - uno de los patrones más conocidos en el mundo del desarrollo. El patrón MVC separa el código de una aplicación web en tres partes interconectadas:

- El Modelo, que contiene la estructura de datos de la aplicación. La capa Modelo representa el modelo de dominio (como Cuenta, Producto, Persona, Publicación, etc.) y encapsula la lógica empresarial específica de la aplicación.

- La capa Vista se compone de "plantillas" que son responsables de proporcionar representaciones adecuadas de los recursos de su aplicación.

- El Controlador, que conecta los datos con la Vista y contiene la lógica de negocio de la aplicación. La capa del controlador es responsable de manejar las solicitudes HTTP entrantes y proporcionar una respuesta adecuada. Por lo general, esto significa devolver HTML, pero los controladores de Rails también pueden generar XML, JSON, PDF, vistas específicas para dispositivos móviles y más. Los controladores cargan y manipulan modelos y representan plantillas de vista para generar la respuesta HTTP adecuada.

Este patrón hace que Rails sea muy flexible y útil para todo tipo de aplicaciones web. (Miller, 15)

Rails se ha utilizado para muchos tipos de aplicaciones web como herramienta para construir aplicaciones web completas que abarquen tanto el *front-end* como el *back-end*. El modelo y el controlador se considerarían la parte de *back-end* de la aplicación mientras que la vista se encarga del *front-end*, generando la página web real que el usuario ve en el navegador, junto con JavaScript y HTML incluidos.

También, se puede utilizar Rails para crear APIs que devuelvan JSON para su uso por otras aplicaciones, tal y como se propone en el presente trabajo, una API RESTful que provea JSON para el consumo de una aplicación externa. Para este caso, resulta simple diferenciar un proyecto de tipo aplicación completa de un proyecto de tipo API, puesto que para la inicialización basta con agregar un flag `--api` al comando de creación y con ello se generará solamente lo necesario para crear una API y no una aplicación con vista y plantillas que se mostrarán al usuario. Por eso, en lugar de utilizar Rails para generar archivos de vista que se comunican con el servidor a través de formularios y enlaces, se

genera un esquema de API que responde con JSON. Así que cuando se sigue los principios para construir una estructura MVC, Rails solo se encarga de los modelos y controladores.

2.7.3 ACTIVE RECORD, EL ORM

Active Record es la M en MVC - el modelo - que es la capa del sistema responsable de representar los datos del negocio y su lógica. Active Record facilita la creación y el uso de objetos de negocios cuyos datos requieren almacenamiento persistente en una base de datos. Es una implementación del patrón Active Record, que en sí mismo es una descripción de un Sistema de Mapeo Relacional de Objetos (ORM).

El mapeo relacional de objetos, comúnmente conocido por su abreviatura ORM, es una técnica que conecta los objetos enriquecidos de una aplicación con las tablas de un sistema de gestión de bases de datos relacionales (RDBMS). Mediante ORM, las propiedades y relaciones de los objetos de una aplicación pueden almacenarse y recuperarse fácilmente de una base de datos sin necesidad de escribir sentencias SQL directamente y con menos código general de acceso a la base de datos. (Rails, s.f.)

Active Record proporciona varios mecanismos, siendo los más importantes la capacidad de:

- Representar modelos y sus datos.
- Representar asociaciones entre estos modelos.
- Representar jerarquías de herencia a través de modelos relacionados.
- Validar los modelos antes de que persistan en la base de datos.
- Realizar operaciones de base de datos orientadas a objetos.

2.7.4 STI: SINGLE TABLE INHERITANCE

Definida por Martin Fowler como “representación de una jerarquía hereditaria de clases como una única tabla que tiene columnas para todos los campos de las distintas clases.” (Fowler, 2002).

En esta representación los modelos de herencia de tabla única (STI) se definen como clases separadas que heredan de una clase base, pero no están asociadas a tablas separadas, sino que comparten una tabla de base de datos. La tabla contiene una columna de tipo que define a qué subclase pertenece un objeto.

Una buena indicación de que la STI es el enfoque correcto es cuando las diferentes subclases tienen los mismos campos o columnas, pero diferentes métodos. Un indicador de

esto es que se espera que todas las columnas de la base de datos sean utilizadas por cada subclase, puesto que, de lo contrario habrá muchas columnas nulas en la base de datos.

Otra buena indicación de que STI es el método correcto es si se espera realizar consultas a través de todas las clases. Por ejemplo, si se espera encontrar aquel registro que tenga mayor cantidad de algún campo o columna entre todas las subclases, STI permite utilizar una sola consulta, mientras que en enfoques como MTI (Multiple Table Inheritance) requerirá una manipulación en memoria.

3 CAPÍTULO III: PROPUESTA DE SOLUCIÓN

3.1 MODELO DE DOMINIO

Para explicar de dónde nace la solución propuesta y el dominio de ésta, es necesario entender el contexto en el que se origina y cómo se desarrolla la situación misma de los clientes cuando recurren a un taller. Se presenta en la Ilustración 8, el modelo de dominio, que explica la interacción de las entidades y actores que se ven involucrados en este proceso.

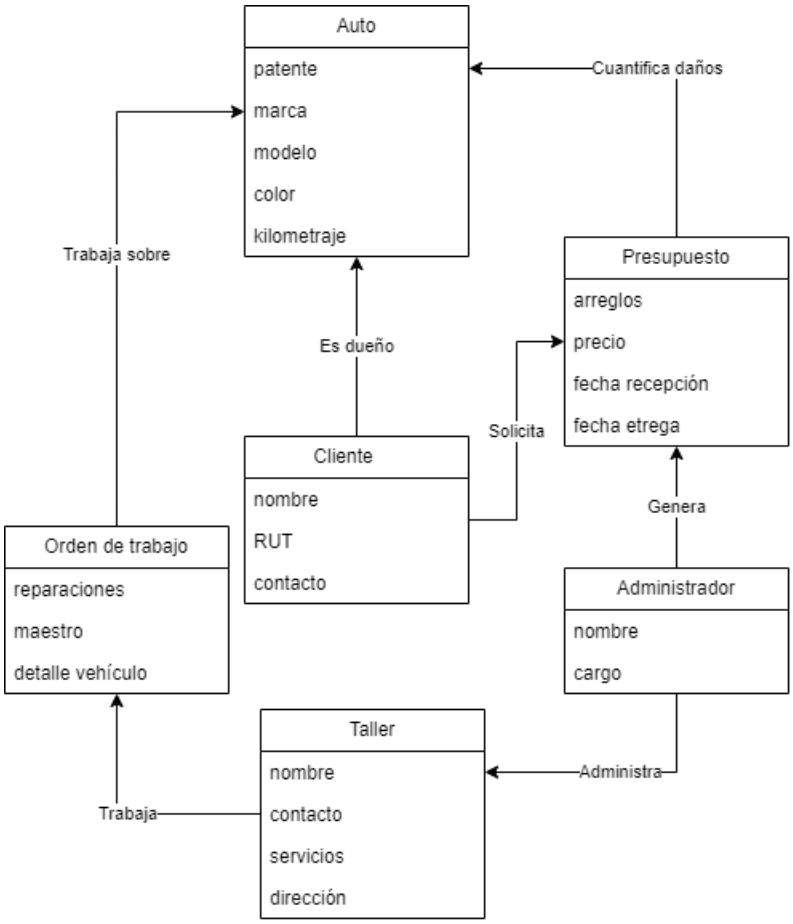


Ilustración 8 Modelo de dominio
Fuente: Elaboración propia.

Como se puede apreciar en el diagrama, actores como Cliente y Administrador (o jefe de taller), interactúan entre sí mediante el Presupuesto, entidad que se materializa a través de una propuesta escrita por parte del Administrador en representación del Taller hacia el Cliente, quien es dueño del Auto al que se le presupuestan los arreglos de los daños. Una

vez que este Presupuesto es bien visto y aceptado por el cliente, se llega al mutuo acuerdo de realizar el trabajo de arreglo, por lo que el Auto ingresa al Taller mediante la formalización de la Orden de trabajo, un documento interno del Taller utilizado para organizar, documentar y generar trazabilidad de los trabajos realizados.

Siendo esta la interacción entre las entidades y actores, para aclarar más aún la secuencia de las interacciones ver el Diagrama de Secuencia entregado en Ilustración 9.

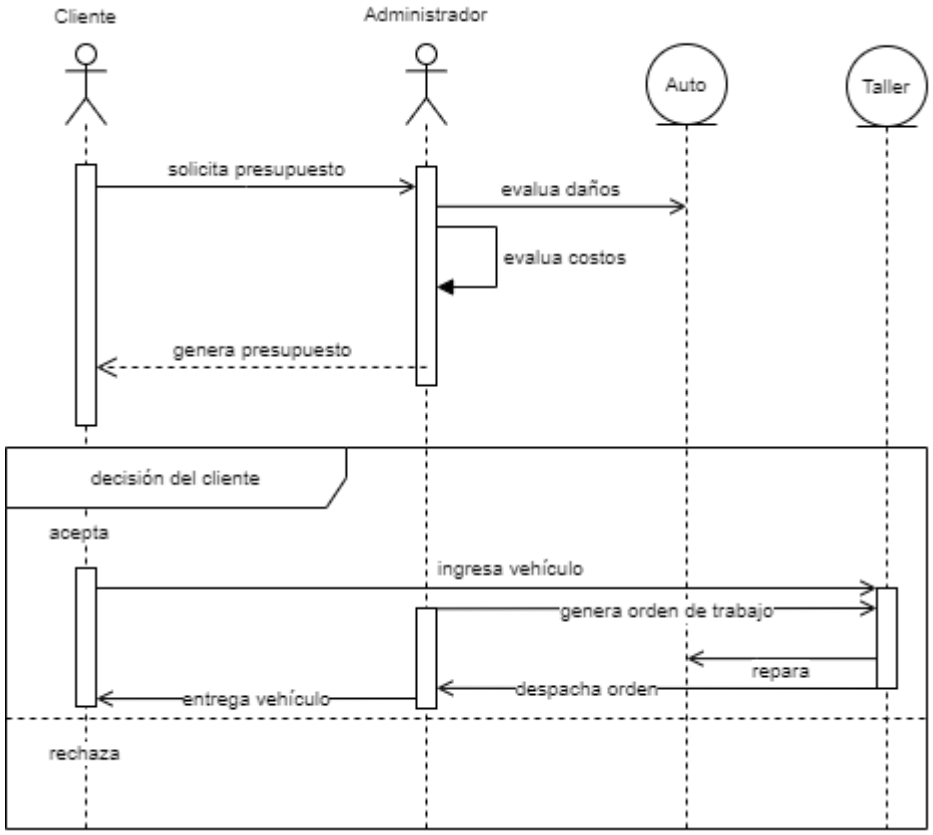


Ilustración 9 Diagrama de secuencia, cliente-administrador
Fuente: Elaboración propia

Tal como se puede ver en Ilustración 9, la interacción inicia con la solicitud de un presupuesto por parte del cliente hacia el taller, la cual es recibida y respondida por el administrador o jefe en representación del taller, esta respuesta consta de un detalle de daños y reparaciones con su costo asociado. A partir de esta respuesta en forma de documento detallado, el cliente puede aceptarla o rechazarla, en caso de aceptar, el cliente entrega el vehículo al taller para que comiencen las reparaciones y el administrador genera la orden de trabajo, un documento que nace a partir de lo detallado en el presupuesto inicial, pero más concreto y con las asignaciones internas para mantener la gestión del taller. Una vez reparado el vehículo y llegada la fecha de entrega, se despacha y entrega el

vehículo al cliente, quien en ese momento cancela el monto total o restante (en caso de existir algún pago previo) y retira su vehículo.

3.2 CLASES

Dado el dominio descrito en el punto anterior, se propone la distribución de clases para el ORM de Ruby on Rails graficada en la Ilustración 10.

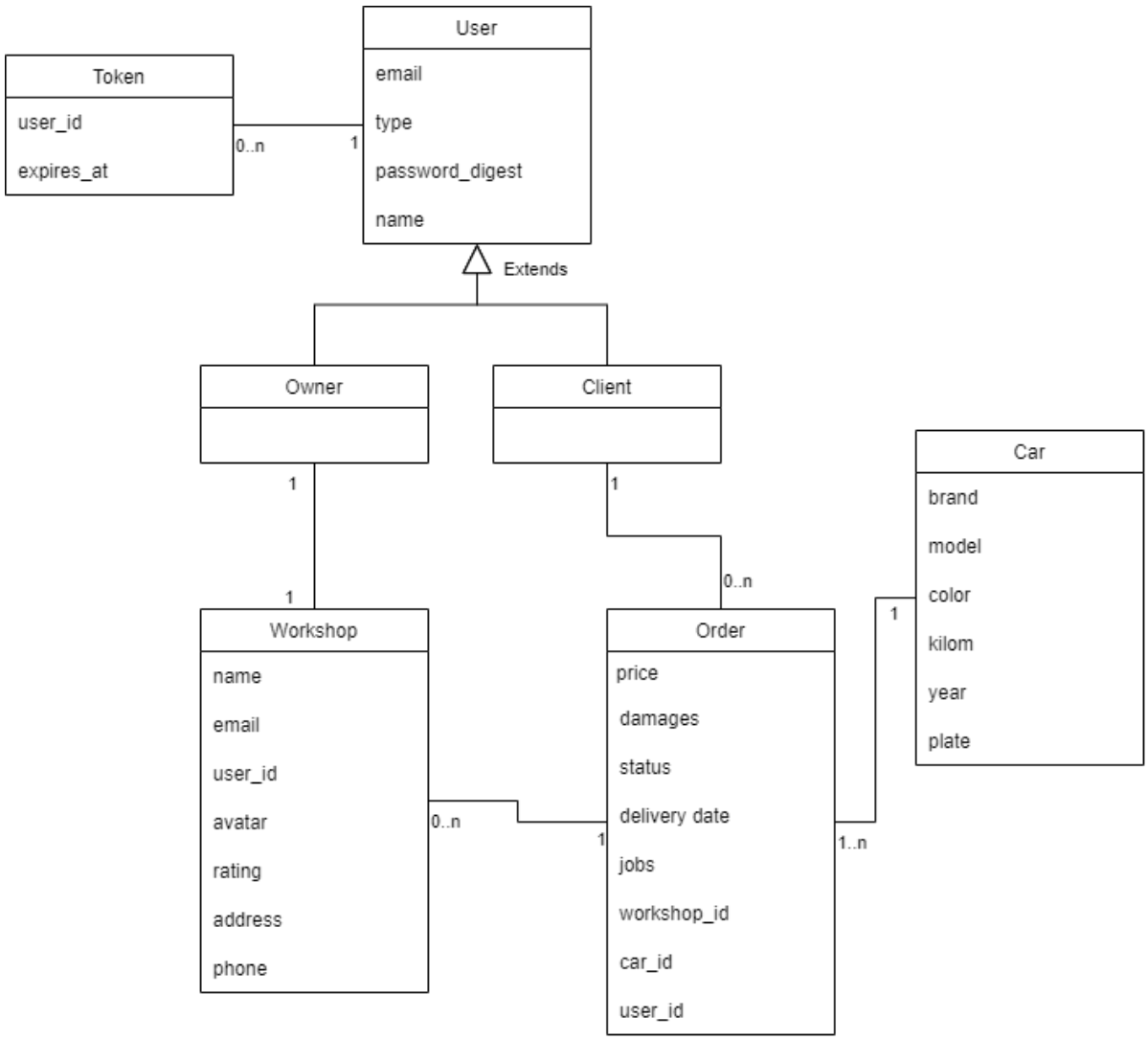


Ilustración 10 Diagrama de clases
Fuente: Elaboración propia

Como se puede apreciar, las entidades de las que se habló en el punto anterior fueron llevadas a Rails de la siguiente manera:

Se establece un User que hereda dos clases, Owner y Client que pertenecen a Administrador y Cliente respectivamente. Esto se propone de esa manera debido a que, en

términos de API y datos, tanto el Cliente como el Administrador son representados de la misma manera y con los mismos atributos, pero comportamientos y accesos distintos, por ejemplo, Owner es quien tiene la relación uno a uno con Workshop, que representa la entidad Taller, mientras que Client es quien tiene la relación uno a muchos con Order, que representa a la Orden de trabajo y el Presupuesto en la misma entidad. Lo anterior se debe a que la única diferencia teórica entre presupuesto y orden de trabajo es el estado del documento, el presupuesto es un documento sobre un vehículo que no ha ingresado al taller, por ende, aún no recibe la reparación, mientras que la orden se diferencia únicamente porque tiene fecha de entrega y trabajos en curso, lo que indica que el vehículo en cuestión, en este caso representado por Car, ya está en el taller recibiendo la reparación. En términos de API, se tratará Order con los siguientes estados: *requested* para las solicitudes de presupuesto, *in progress* para aquellos presupuestos aceptados e ingresados (órdenes de trabajo como tal), *rejected* para presupuestos rechazados por el cliente y *delivered* para órdenes de trabajo que ya fueron entregadas (o registro histórico).

Adicionalmente y debido a la estrategia de autenticación utilizada por la API, *Bearer Authentication*, se genera una entidad Token con la finalidad de que en cada inicio de sesión se genere un token de autenticación para que así, la aplicación que consuma la API pueda consultar de manera segura por cada usuario. Esto se implementa así ya que es la estrategia actual con la que se trabaja desde el lado de la aplicación móvil y es una de las más seguras para autenticar APIs.

3.3 BASE DE DATOS

Una vez implementadas las clases, en Ilustración 11 se puede ver cómo Rails y el ORM las convierten en tablas de una base de datos PostgreSQL.

Muy similar a como era el diagrama de clases, se define la tabla Users que implementa la estrategia Single Table Inheritance (STI), por lo que el atributo type es el que determina si el User es un usuario de tipo Client o de tipo Owner, para diferenciar al cliente del administrador o jefe del taller respectivamente. A esta tabla se encuentra relacionada la tabla Tokens, la cual es la encargada de almacenar los *tokens* temporales de accesos para los usuarios.

Luego se encuentra la tabla Workshops que tiene relación con Users de tal manera de que cada Workshop tiene relacionado un User de tipo Owner, que representa la relación “Administrador o dueño administra un Taller”. Además, Workshops tienen relación de cero a muchos con Orders, ya que un taller puede gestionar cero o muchas órdenes o presupuestos.

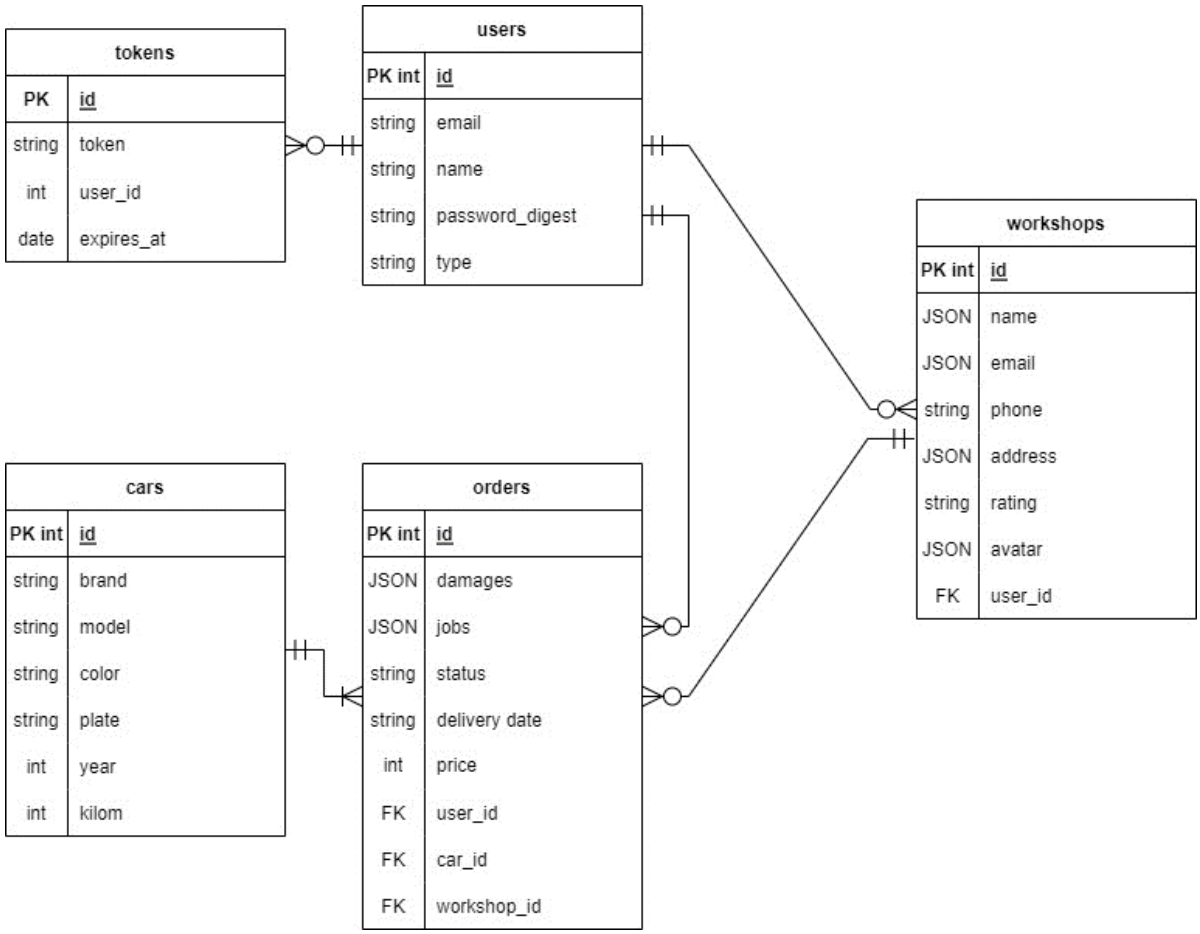


Ilustración 11 Tablas en base de datos
Fuente: Elaboración propia

Por otro lado, la tabla Orders también tiene relación con la tabla Cars, ya que cada orden tiene exactamente un auto o vehículo, pero a su vez cada auto puede tener muchas órdenes o presupuestos, ya que el mismo vehículo puede ser reparado múltiples veces y además bajo el dominio de la aplicación VraVa, un vehículo puede ser presupuestado a varios talleres a la vez, por lo que se generarían varias Orders con estado “requested” del mismo vehículo con distintos talleres.

Es importante destacar la dualidad de la entidad Order, ya que bajo la premisa de que un presupuesto y una orden de trabajo se diferencian únicamente por su estado, en ese sentido se agrupa en la misma tabla y entidad ambos documentos, que son filtrados y diferenciados mediante el atributo “status”.

3.4 OPTIMIZACIÓN DE FLUJOS

Existen diversos flujos de transferencia de datos entre la aplicación móvil y el *backend* actual de VraVa (desarrollado en Strapi), en los que dependiendo de la funcionalidad se ejecuta una serie de consultas HTTP en base a las acciones que haga el usuario, por ejemplo, si el usuario visualiza la lista de talleres y luego visualiza el perfil individual de un taller, la aplicación móvil hace dos consultas de tipo GET, en primer lugar solicita la lista de todos los talleres y luego, cuando el usuario ingresa al perfil de un taller, realiza la consulta GET del taller en específico. Lo anterior es un ejemplo particular que no forma parte de las historias de usuario declaradas como importantes en el proceso de desarrollo de VraVa, sin embargo, es un claro ejemplo de los muchos procesos que están poco optimizados y que tienen oportunidad de mejora. En este ejemplo particular, al ser un listado de datos que no están en cambio constante, ya que reciben pocas modificaciones y es de una extensión acotada, el flujo podría ser reducido a una sola consulta a la API que obtenga todos los datos necesarios.

3.4.1 CREACION DE ORDER + IMAGES

El primer flujo para abordar, y quizás el más importante, es el asociado a la creación de una Orden de trabajo. En el *backend* actual independiente del actor, ya sea cliente creando una solicitud o dueño/jefe de taller ingresando una orden de trabajo a su taller, se necesita en primer lugar subir las fotografías tomadas al automóvil, una por una, a través de una solicitud HTTP de tipo POST al *endpoint "/upload"* de Strapi, la cual almacena el registro de la imagen en base de datos y lo respalda en un bucket S3 de AWS. Es decir, si hay "n" fotografías son "n" las solicitudes POST para almacenarlas.

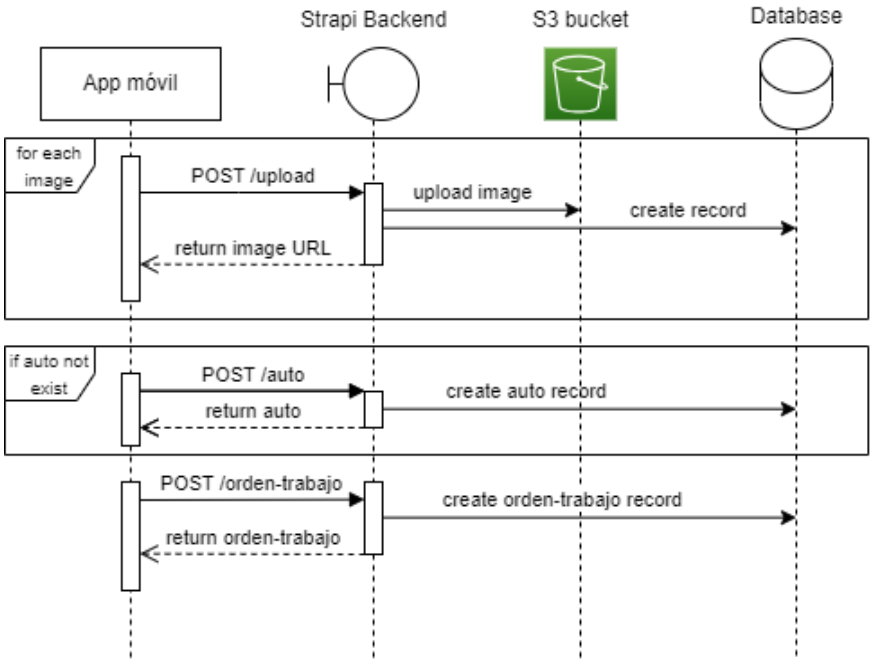


Ilustración 12 Secuencia de creación de Orden en backend Strapi.
Fuente: Elaboración propia

Siguiendo el flujo y como indica la Ilustración 12, luego de subir cada imagen se procede a crear la entidad Auto si es que no existe, para posteriormente generar la Orden de trabajo con el auto asociado y las URL de las imágenes respaldadas en S3 mediante el atributo.

Como oportunidades de mejora se propone, en primer lugar, utilizar Active Storage que es una herramienta que “facilita la subida de archivos a almacenamientos en la nube como AWS, Google Cloud Storage o Microsoft Azure Storage y adjunta esos archivos convirtiéndolos en objetos de Active Record (ORM)” (Ruby on Rails). Esta mejora cambiaría el flujo en cuanto a número de consultas necesarias, ya que como se ve en Ilustración 13 se pasaría de “n” consultas dedicadas exclusivamente para almacenar las imágenes, a solamente una consulta HTTP a la API independiente de la cantidad de imágenes de la Orden de trabajo.

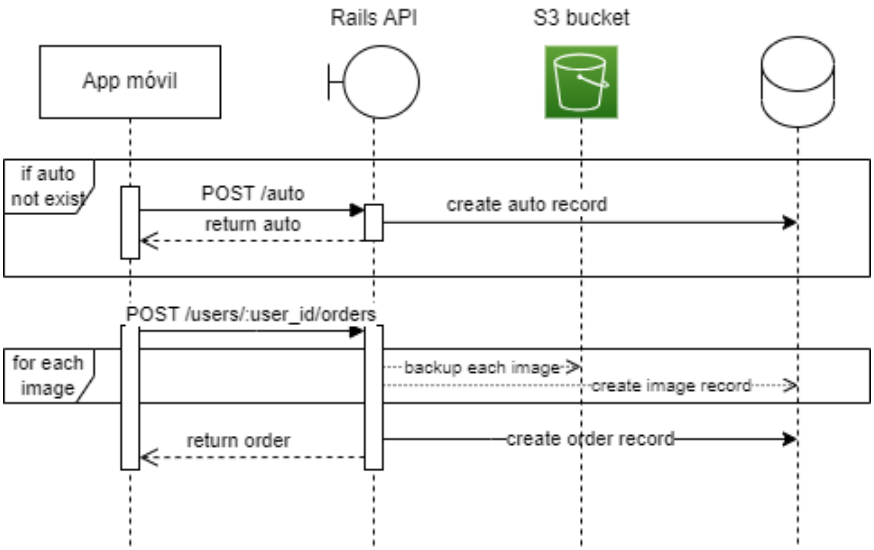


Ilustración 13 Secuencia de creación de Orden en Rails API.
Fuente: Elaboración propia

Se mantiene la conexión con S3 gracias a Active Storage, que a su vez almacena las imágenes en base de datos para su persistencia con el ORM. La condicional acerca de la existencia del auto se mantiene, ya que es una entidad por sí misma y debe persistir para que pueda ser relacionada a una Order.

Otra mejora que se obtiene al usar Active Storage es que la vinculación entre Orden – Imagen no es un atributo JSON como lo era en Strapi, ahora es una relación de tipo "`has_many_attached`" entre entidades manejada por el ORM Active Record.

3.4.2 CREACIÓN DE ORDER DESDE SOLICITUD

Las Solicitudes son las entidades en el *backend* Strapi que representan a las solicitudes de presupuesto, las cuales son construidas de forma unitaria por cada presupuesto – vehículo, es decir, independiente de a cuántos talleres solicite presupuesto el cliente, es una sola entidad que maneja estas peticiones a los talleres a través de un atributo llamado "estado" (Ilustración 14) que contiene la data sobre el estado actual de la solicitud en cada taller, su presupuesto en arreglos y monto de la reparación.

```
{
  ...
  "estado": [
    {
      "estado": "pendiente",
      "tallerId": 15,
      "comentario": "",
      "presupuesto": []
    },
    {
      "estado": "pendiente",
      "tallerId": 96,
      "comentario": "",
      "presupuesto": []
    },
    {
      "estado": "ingresado",
      "tallerId": 7,
      "comentario": "Procedimiento difícil",
      "presupuesto": [ ... ]
    }
  ],
  ...
}
```

Ilustración 14 Extracto ejemplo del estado de una solicitud.
Fuente: Elaboración propia

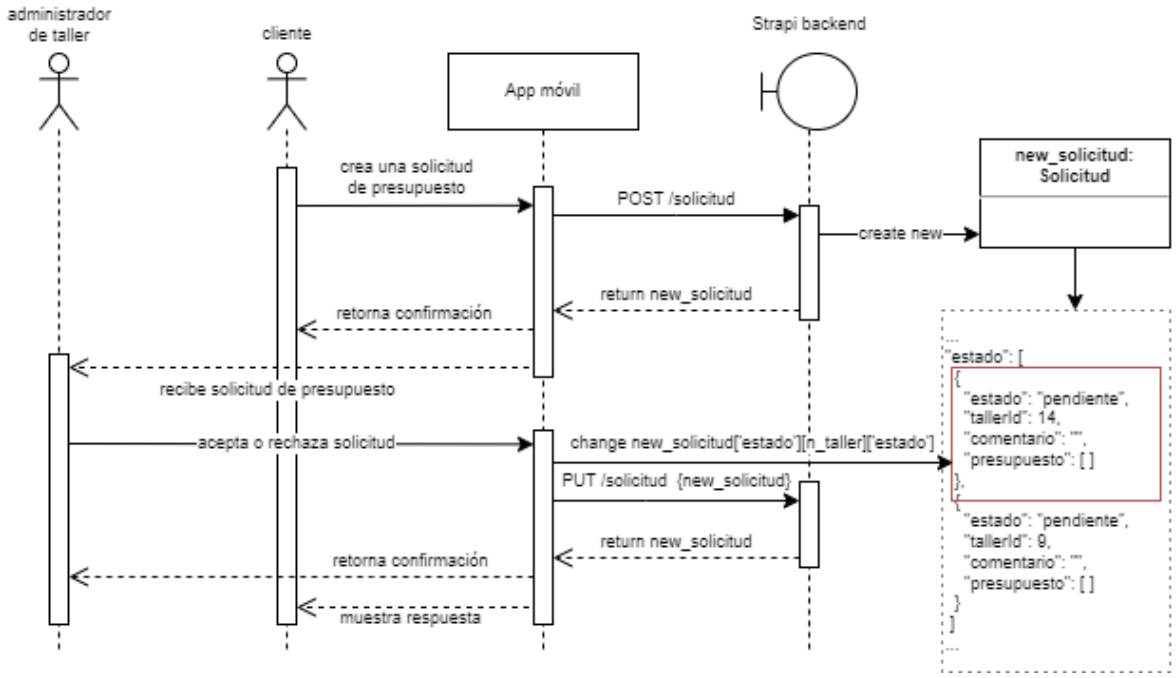


Ilustración 15 Flujo para editar estado de una Solicitud en Strapi.
Fuente: Elaboración propia

La Ilustración 15 representa al flujo necesario para editar y actualizar una entidad Solicitud, una vez creado el registro, independientemente de la respuesta de los talleres es labor del *front* (aplicación móvil) modificar los datos de "estado" para cada una de las respuestas de los distintos talleres y hacer una petición PUT para registrar el cambio en el sistema. Es decir, una vez el taller registra una respuesta a la solicitud, es la aplicación móvil quien modifica el atributo estado para enviar al *backend* la entidad ya actualizada. Lo anterior es poco mantenible y es considerado una mala práctica en el esquema MVC, ya que esta labor corresponde a la lógica del controlador y no a la vista.

Además de lo anterior, en el momento en que una Solicitud de presupuesto hacia un taller es aceptada por este último pasa a ser una Orden de trabajo, para Strapi, una entidad totalmente nueva y distinta a la de Solicitud. Por lo que es necesario, en primer lugar, modificar el atributo "estado" de la Solicitud y, posteriormente, crear una Orden de trabajo completamente nueva y con los mismos datos que ya almacena la Solicitud, lo cual, si bien implica pocas consultas HTTP a Strapi, es tremendamente ineficiente tanto por el manejo de esta data a través del atributo "estado" como por el hecho de crear una entidad completamente nueva para almacenar los mismos datos.

Para la API Rails, se propone un cambio estructural respecto a la modelación y diseño de la entidad Solicitud/Orden de trabajo que consiste en fusionarlas en una sola entidad llamada Orden que contenga toda la información, que es en esencia la misma para ambas, y manejar esta diferencia conceptual a través de un atributo de tipo status que le proporcione la identidad contextual a la orden, pasando a tener cinco posibles estados:

1. "requested": Representa a la solicitud como tal (el homólogo de la entidad Solicitud existente en Strapi), es un manifiesto de que se solicitó un presupuesto del arreglo de un vehículo a un taller.

2. "rejected": Representa a una solicitud de presupuesto rechazada, es decir, el usuario solicitó el presupuesto del arreglo del vehículo, pero el taller lo rechazó.

3. "accepted": Representa a una solicitud de presupuesto aceptada, es decir, el usuario solicitó el presupuesto del arreglo del auto y el taller aceptó recibir el vehículo entregando la información (precio, detalles y comentarios del arreglo).

4. "in progress": Representa a la orden de trabajo como tal, es la entidad digital del documento que detalla las reparaciones del vehículo, montos asociados, fechas de recepción y entrega, etc.

5. "delivered": Representa al histórico de una orden de trabajo, ya fue entregado el vehículo por ende el trabajo está finalizado.

Otro cambio es el tipo de relación con la entidad Taller, pasa de ser una solicitud a “n” talleres a convertirse en una orden por cada taller al que se solicita el presupuesto, esto tiene dos ventajas principales: en primer lugar, si una solicitud (Order con status “requested”) es rechazada por el taller, tan solo basta actualizar ese campo de la entidad en particular quedando aislada del resto de entidades que van hacia los otros talleres. En Ilustración 16, se entrega un diagrama que describe esta situación.

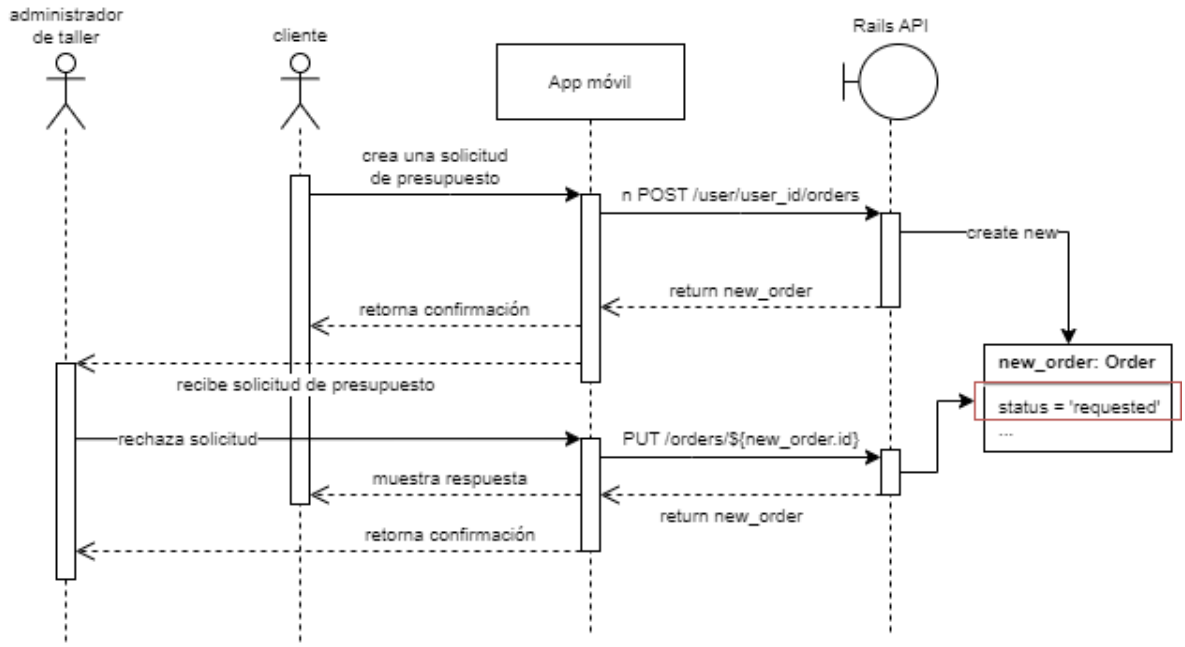


Ilustración 16 Flujo de solicitud rechazada por taller, API Rails.
Fuente: Elaboración propia

Cabe destacar que además de las diferencias ya mencionadas, como se puede ver en la primer llamada saliente desde la App Móvil hacia la Rails API, se generan “n” órdenes con estado “requested”, siendo “n” el número de talleres a los que el cliente solicita el presupuesto, esto es debido al desacople que se hizo entre taller y solicitud, ahora es una entidad solicitud por cada taller, o lo que es igual, por cada solicitud de presupuesto, se generan tantas órdenes en estado “requested” como talleres haya seleccionado el usuario cliente.

Sumado a esto, otra ventaja es que en caso de que una solicitud sea aceptada, ingresada y luego entregada posterior al trabajo realizado por el taller, quedaría toda la información y el avance del proceso registrado y unificado en una sola entidad independiente y con datos exclusivamente del proceso propio, lo cual es una gran diferencia, ya que en el modelo de solicitud de Strapi existía este atributo “estado” que mezclaba información de varios talleres, lo cual podría ser considerado como mala práctica debido a que si un taller

rechaza, modificaría el mismo campo de la entidad que otro taller podría modificar al aceptar la solicitud de presupuesto.

En cuanto a la cantidad de consultas HTTP, cabe destacar que con esta mejora aumentan, ya que el aceptar o rechazar implicaría una consulta de tipo PUT a la entidad en particular, es decir, cualquier cambio de estado implicaría esta actualización del atributo “status” mediante la consulta, sin embargo, al estar actualizando un campo de tipo string y de tamaño reducido, no debería implicar un consumo excesivo o un deterioro en el rendimiento.

3.5 DESPLIEGUE

Para desplegar la API Rails y hacerla disponible en la nube para el consumo de la aplicación móvil y poder comparar con el *backend* actual Strapi se propone desplegarla en Railway, la misma plataforma en la que se encuentra Strapi, para comparar ambas en igualdad de condiciones. Además, se propone desplegarla en 2 plataformas adicionales a Railway para comparar la API Rails en tres plataformas como servicio (PaaS) y poder concluir acerca de qué plataforma es la más idónea para una API como la desarrollada, comparando planes y costos, tiempos de respuesta y facilidad de uso.

Estas tres plataformas son Railway, Heroku y Fly.io.

3.6 MÉTRICAS

Se propone tomar las siguientes métricas asociadas al rendimiento de la aplicación, tanto del *backend* Strapi como el de la API Rails luego de ejecutar las pruebas de rendimiento correspondientes:

1. Media tiempo de respuesta (en milisegundos): Se considerará un tiempo de respuesta promedio de la aplicación para todo tipo de consulta proveniente de la prueba de rendimiento. Esto con el fin de tener una mirada global respecto a un número en concreto de cuánto toma a la aplicación responder al promedio de las consultas.

2. Mediana tiempo de respuesta (en milisegundos): Si bien es cierto, la media indica el rendimiento típico de la aplicación bajo condiciones generales, se puede ver distorsionado por aquellas peticiones que duren más que el resto o alguna petición *outlier* y hacer que parezca que el rendimiento general es peor de lo que realmente es. Por esto se considerará también la mediana, esta es el "valor medio" de los tiempos de respuesta, es decir, el valor que divide a los tiempos de respuesta en

dos mitades iguales, la mitad superior e inferior. La mediana es menos sensible a los *outliers* y puede ofrecer una mejor idea del rendimiento "típico".

3. Peticiones por minuto (RPM): Esta métrica mide cuántas solicitudes puede manejar la API en un minuto. Es una manera de medir el rendimiento bajo carga.

4. Uso de CPU (uso porcentual): Indicador de cuánto porcentaje de la unidad de procesamiento consume la API en promedio.

5. Uso de RAM (en MB): Indicador de cuánta memoria RAM utiliza la API en promedio.

6. Apdex Score: Representa una solución simplificada del Acuerdo de Nivel de Servicio (SLA) que permite evaluar la satisfacción de los usuarios con una aplicación mediante métricas como la puntuación Apdex, en lugar de métricas tradicionales que pueden ser fácilmente sesgadas, como el tiempo de respuesta promedio. Establecer umbrales adecuados para los niveles de Apdex permite visualizar de manera inmediata la satisfacción general de los usuarios finales con la aplicación y obtener la cantidad adecuada de rastreos para solucionar y mejorar el rendimiento de la aplicación.

3.7 PRUEBAS DE RENDIMIENTO

Para someter el *backend* y la API Rails a cargas y hacer pruebas de estrés se propone utilizar Locust, una herramienta de pruebas de carga de código abierto con la cual se puede describir el comportamiento de un usuario a través de código Python y simular la actividad de usuarios consumiendo la aplicación en simultáneo. Dado que se someterán ambas aplicaciones, tanto el *backend* Strapi como la API Rails, se diseñan dos *scripts* de pruebas Locust.

Estos *scripts* simulan la navegación e interacción del usuario con la aplicación, si bien las rutas y estructuras de datos varían (como se ha mostrado en los puntos anteriores) ambos *scripts* de prueba siguen los mismos flujos y patrones de consultas.

```python
@task(25)
def getAllOrders(self):
    response = self.client.get("/orden-trabajos")
    if(response.status_code != 200):
        response.failure("fetching all orders failed")

@task(25)
def getOrders(self):
    response = self.client.get("/orden-trabajos?estado=in_progress")
    if(response.status_code != 200):
        response.failure("fetching all orders failed")

@task(25)
def getOrder(self):
    response = self.client.get(f"/orden-trabajos/{order_id!r}")
    if(response.status_code != 200):
        response.failure("fetching orders failed")

@task(25)
def getRequests(self):
    response = self.client.get("/solicitudes")
    if(response.status_code != 200):
        response.failure("fetching requests failed")

```

Ilustración 17 Pruebas Locust para Strapi.
Fuente: Elaboración propia

```python
@task(25)
def getAllOrders(self):
    response = self.client.get(f"/v1/users/{self.user_id!r}/orders?status=all")
    if(response.status_code != 200):
        response.failure("fetching all orders failed")

@task(25)
def getOrders(self):
    response = self.client.get(f"/v1/users/{self.user_id!r}/orders")
    if(response.status_code != 200):
        response.failure("fetching orders failed")

@task(25)
def getOrder(self):
    response = self.client.get(f"/v1/orders/{order_id!r}")
    if(response.status_code != 200):
        response.failure("fetching orders failed")

@task(25)
def getRequests(self):
    response = self.client.get(f"/v1/users/{self.user_id!r}/orders?status=in_progress")
    if(response.status_code != 200):
        response.failure("fetching requests failed")
```

Ilustración 18 Pruebas Locust para API Rails.
Fuente: Elaboración propia.

Como se ve en las ilustraciones 17 y 18, son muy similares las pruebas para cada *endpoint* homólogo y en cada una de ellas se verifica que el código de la respuesta sea exitoso, en caso contrario arroja el error correspondiente. Cabe destacar que estas son sólo algunas de las peticiones que se incluyeron en el test completo que simula el recorrido del usuario, éste se puede ver en el Anexo 3 y Anexo 4.

Una vez escrito el conjunto de pruebas y solicitudes a simular, es necesario definir un plan de pruebas para poder comparar el rendimiento de ambas aplicaciones. La idea detrás de este plan es poder simular situaciones reales de carga o estrés en la aplicación VraVa, lo que se traduce, debido al flujo de información, en peticiones a las APIs.

Se propone simular tres situaciones distintas: una situación de carga alta en un período de tiempo acotado, una situación de carga media por un tiempo moderado y una situación de carga baja por un período de tiempo más prolongado.

Tabla 2 Plan de ejecución de pruebas.
Fuente: Elaboración propia

Situación	Máximo usuarios concurrentes	Tiempo [s]	Usuarios cada segundo
Masiva	80	40	4
Media	40	80	1
Baja	20	90	0.4

Estas pruebas se proponen con el objetivo de simular puntos de sobrecarga a la API donde se vea sometida a múltiples consultas recurrentes. Las cantidades de máximo de usuarios y tiempo de ejecución de la prueba fueron determinados en base a la experiencia previa con la aplicación *backend* de Strapi y con Railway en base a lo vivido y probado en el proceso de despliegue del proyecto para FESW. Se estima que el *backend* comienza a colapsar y mostrar alzas considerables en los tiempos de respuestas sobre los 50 a 60 usuarios simultáneos.

Finalmente, cabe destacar que estas mediciones se harán en cada ambiente desplegado, es decir, se someterán a estas pruebas los siguientes artefactos:

- Strapi *backend* en Railway
- Rails API en Railway
- Rails API en Heroku
- Rails API en Fly

Esto con el objetivo de comparar, por un lado, Strapi vs Rails API en Railway y, por otro lado, comparar Rails API en cada una de las PaaS.

4 CAPÍTULO IV: VALIDACIÓN DE LA SOLUCIÓN

4.1 CONSTRUCCIÓN

4.1.1 RUBY ON RAILS API

Se crea una API en el framework Ruby on Rails en su versión 6.1.7, a través del comando nativo de Rails mostrado en Ilustración 19.

```
1  rails new vrava-api -d posgresql --api
```

Ilustración 19 Comando inicialización Rails.
Fuente: Elaboración propia

Tal y como sería una app clásica de Rails, se crea una API especificando su nombre como *vrava-api*, se establece PostgreSQL en su versión 14.6 como motor de base de datos y se le indica que es una aplicación de tipo API-only con el parámetro --api.

Según lo especificado en la misma instrucción Rails new --help, esta opción --api preconfigura un stack más pequeño para aplicaciones de API-only e instala las dependencias “mínimas”, excluyendo las vistas.

Una vez inicializada la API, es necesario *dockerizar* el proyecto de tal manera de generalizar su instalación y que no dependa de la arquitectura del dispositivo o máquina en la que se instale y ejecute el servidor, esto ayudará a que en pasos futuros como el despliegue sea mucho más fácil y simple el uso de distintas plataformas, además de todas las ventajas que conlleva el uso de *containers*. Para esto, es que se implementa el Dockerfile que se muestra en Anexo 1 y el archivo docker-compose que se muestra en Anexo 2. Con esto es que se logra encapsular la API por completo y hacerla independiente de la arquitectura que se esté utilizando para su ejecución, estando en condiciones de comenzar a desarrollar y generar los recursos necesarios para la construcción de la API.

4.1.2 RSPEC Y TDD

Una vez que la API ya se encuentra inicializada, dockerizada y lista para comenzar el desarrollo, es necesario la incorporación de una gema que permita al desarrollador aplicar TDD, de tal manera de que, al implementar modelos, controladores y rutas, sean los test

quienes dirijan el desarrollo. RSpec es la gema elegida en este caso debido a su amplia documentación y lo fácil que es la lectura de los test, uno de los objetivos además de dirigir el desarrollo a través de las pruebas es que estas mismas pruebas generan documentación viva de los componentes de la aplicación.

Como plan de desarrollo se plantea seguir una estructura basada en pruebas unitarias por componentes: modelos, controladores y solicitudes (requests), es decir, se sigue el ciclo dictado por TDD partiendo por escribir las pruebas unitarias de cada funcionalidad como, por ejemplo, el modelo de cierta entidad y sus restricciones, luego ejecutar las pruebas y detectar los fallos, aplicar los cambios necesarios para pasar los test y volver a ejecutar.

4.1.2.1 TDD: MODELOS

A modo de ejemplificar y demostrar la estrategia, se utilizará la entidad Order como muestra, pero es importante notar que para el resto se implementó de forma análoga.

En primer lugar, dadas las reglas del negocio y la definición como tal de la entidad, se definen las validaciones por las que debe pasar la definición del modelo, como se detalla en Ilustración 20.

```ruby
RSpec.describe Order, type: :model do
  describe "Order model validation" do
    subject { build(:order) }
    it "validates if damages exists" do
      should validate_presence_of(:damages)
    end
    it "validates if jobs exists" do
      should validate_presence_of(:jobs)
    end
    it "validates if status exists" do
      should validate_presence_of(:status)
    end
    it "validates if delivery_date exists" do
      should validate_presence_of(:delivery_date)
    end
    it "validates if price exists" do
      should validate_presence_of(:price)
    end
    it "validates if car relationship exists" do
      should belong_to(:car)
    end
    it "validates if client relationship exists" do
      should belong_to(:client)
    end
    it "validates if client relationship exists" do
      should belong_to(:workshop)
    end
  end
end
```

Ilustración 20 Archivo spec de modelo Order.
Fuente: Elaboración propia

Como se puede notar en el código correspondiente al spec del modelo de Order, se valida la presencia de los atributos definidos como obligatorios y se valida que Order tenga relaciones de tipo *belongs to* con Car, Client y Workshop.

Es claro que al momento de ser generado el modelo por consola a través del comando *"rails generate model"* no contiene ninguna declaración, por lo que el test falla al momento de validar las cláusulas especificadas. Por ende, el siguiente paso es escribir el código necesario para aprobar las pruebas (ver Ilustración 21).

```ruby
class Order < ApplicationRecord
  belongs_to :car
  belongs_to :workshop
  belongs_to :client, foreign_key: 'user_id'
  validates :damages, :jobs, :status, :delivery_date, :price, presence: true
  has_many_attached :images

  attribute :damages, :jsonb
  attribute :jobs, :jsonb

  accepts_nested_attributes_for :car, :workshop, :client

end
```

Ilustración 21 Modelo de Order y sus validaciones.
Fuente: Elaboración propia.

Considerando el código destacado, se tienen aquellas validaciones necesarias para aprobar los test declarados en el archivo spec y cumplir con lo que se determina en el modelo de dominio de la Orden, se declaran las relaciones correspondientes y se valida que existan los atributos considerados como obligatorios. Tal y como se espera en el ciclo TDD se procede a ejecutar nuevamente las pruebas y estas ahora las pasa de manera exitosa (ver Ilustración 22).

```
$ rspec spec/models/order_spec.rb

Randomized with seed 1280

Order
  Order model validation
    validates if client relationship exists
    validates if status exists
    validates if price exists
    validates if delivery_date exists
    validates if jobs exists
    validates if car relationship exists
    validates if client relationship exists
    validates if damages exists

Finished in 1.63 seconds (files took 2.39 seconds to load)
8 examples, 0 failures
```

Ilustración 22 Ejecución de pruebas modelo Order.
Fuente: Elaboración propia

Finalmente se considera completo el modelo de la entidad, en este caso Order, por lo que al menos por ahora que no necesita modificaciones se procede con otro modelo. En caso de que el modelo de la entidad necesite ser modificado, ya sea por algún desarrollo de otra funcionalidad, como por ejemplo el controlador, se procede a ejecutar nuevamente las pruebas y verificar que éstas sigan siendo aprobadas exitosamente.

Se procede con el resto de las entidades de manera análoga.

4.1.2.2 TDD: CONTROLADOR

Para el caso de los controladores se necesita, como en cualquier API, métodos de tipo CRUD que permitan manejar los recursos, entregarlos y almacenarlos según corresponda la petición. Bajo el esquema de Rails se suele trabajar con cuatro métodos base que gestionan el recurso: *index* para listar todos los recursos, *show* para mostrar un recurso en específico, *create* para generar un nuevo recurso, *update* para actualizarlo y *destroy* para eliminarlo. Para cada uno de estos se trabajará con un archivo spec por separado, a modo de ordenar y mantener las pruebas aisladas y legibles.

Continuando con Order como ejemplo, en Ilustración 23 e Ilustración 24 se muestra el proceso para el método *create* ya que es de las funciones más importantes y que requiere

más lógica, en cualquier caso, el resto de los métodos sigue de manera análoga y su desarrollo no es muy alejado.

En primer lugar, se diferencian contextos dependiendo del caso que se está probando en cada test y en base a ello se tiene un contexto exitoso, uno con fallo asociado a una consulta mal estructurada y otro fallo asociado a la falta de autenticación.

```ruby
RSpec.describe V1::OrdersController, type: :controller do
  describe "orders#create method" do
    let!(:client) { create(:client) }
    let!(:bearer) { create(:token, user: client) }
    let!(:headers) { { Authorization: "Bearer #{bearer.token}" } }
    let!(:car) { create(:car) }
    let!(:workshop) { create(:workshop) }

    context "success" do
      before do
        @order_params = attributes_for(:order)
        request.headers.merge! headers
        post :create, format: :json, params: {
          order: @order_params,
          car_id: car.id,
          client_id: client.id,
          workshop_id: workshop.id
        }
      end

      it "returns a successful status" do
        expect(response).to have_http_status(:created)
      end

      it "returns the order successfully" do
        expect(payload_test).to include(
          :id, :damages, :jobs, :status, :delivery_date, :price,:created_at,
          :car, :client, :workshop)
        expect(payload_test[:client]).to include(:id, :email, :name)
        expect(payload_test[:car]).to include(:id, :brand, :model, :year, :plate, :color, :kilom)
        expect(payload_test[:workshop]).to include(:id, :name, :email, :phone, :address, :rating)
      end
    end

    # ... #
```

Ilustración 23 Archivo spec de método create de controlador Order.
Fuente: Elaboración propia

Para el caso exitoso (ver Ilustración 23), se procede con un bloque predecesor que se encarga de generar el JSON con la estructura y los datos correspondientes a la nueva Order a crear, agrega los *headers* correspondientes a la autenticación y ejecuta la acción apuntando al método *create* del controlador. Una vez ejecutado, se verifica que la respuesta contenga un código de estado creado (201) y, además, que el cuerpo de la respuesta contenga la Order creada con todos sus atributos declarados en la vista jbuilder (Anexo 5 Archivo jbuilder de Order) y las entidades relacionadas anidadas.

```ruby
# ... #

context "fails with invalid param" do
  before do
    @order_params = {damages: "{}", jobs: "{}"}
    request.headers.merge! headers
    post :create, format: :json, params: {
      order: @order_params,
      car_id: car.id,
      client_id: client.id,
      workshop_id: workshop.id
    }
  end

  it "returns bad request" do
    expect(response).to have_http_status(:bad_request)
  end
end

context "fails with no authorization token" do
  before do
    @order_params = attributes_for(:order)
    post :create, format: :json, params: {
      order: @order_params,
      car_id: car.id,
      client_id: client.id,
      workshop_id: workshop.id
    }
  end

  it "returns unauthorized" do
    expect(response).to have_http_status(:unauthorized)
  end
end

# ... #
```

Ilustración 24 Archivo spec del método create de Order.
Fuente: Elaboración propia

Respecto a los casos de fallo (ver Ilustración 24), se simula una petición con su cuerpo incorrecto, en este caso con atributos obligatorios vacíos, a la cual se espera una respuesta con código de petición incorrecta (400). Además, se incluye un caso donde se genera una petición correcta, pero sin autenticación por lo que se espera que el controlador responda con código no autorizado (401).

Siguiendo con la estructura de TDD, en Ilustración 25 se muestra el código necesario en el método create para aprobar las pruebas descritas recientemente.

```ruby
class V1::OrdersController < ApplicationController
  before_action :authenticate_user!

  # ... #
  def create
    @order = Order.new(order_params)
    @order.client = @client
    @order.car = @car
    @order.workshop = @workshop

    if @order.save
      if params[:order][:images].present?
        @order.images.attach(params[:order][:images])
      end
      render :show, status: :created
    else
      render json: @order.errors, status: :bad_request
    end
  end
  # ... #
end
```

Ilustración 25 Método create Order.
Fuente: Elaboración propia

Cabe destacar que, en primer lugar, antes de cualquier acción o método del controlador se ejecuta la autenticación del usuario. Para el método *create* además se crea una nueva entidad Order, se le asigna el cliente, auto y taller y también se le adjuntan las imágenes, si es que la petición las incluye. Una vez estructurada de esta manera la Order, se intenta persistir en la tabla Orders, en un caso exitoso se retorna una respuesta con estado creado y se renderiza la vista parcial de mostrar la Order, en caso contrario se renderiza un JSON con los errores encontrados en la estructura y un código de petición malformada.

Volviendo a ejecutar las pruebas escritas, se ve que ahora sí son pasadas con éxito en cada uno de sus casos (ver Ilustración 26).

```
$ rspec spec/controllers/orders/create_spec.rb

Randomized with seed 17275

V1::OrdersController
  orders#create method
    fails with no authorization token
      returns unauthorized
    success
      returns the order successfully
      returns a successful status
    fails with invalid param
      returns bad request

Finished in 0.9476 seconds (files took 1.55 seconds to load)
4 examples, 0 failures
```

Ilustración 26 Ejecución spec de create Order.
Fuente: Elaboración propia

4.1.2.3 TDD: CONSULTAS

Una vez que se tienen los controladores y modelos de todos los recursos de la API, es necesario generar las rutas de los *endpoints* que consumirá la aplicación cliente, entendiendo a cliente como la aplicación móvil o web que mostrará al usuario final los datos y todo el flujo a través de las propias vistas.

Estos *endpoints* serán quienes recibirán las peticiones (HTTP *requests*) e interpretarán lo solicitado por el cliente, a lo que responderán con el código correspondiente y el JSON de respuesta solicitado. Si anteriormente se testearon los modelos y controladores, el testeo de consultas y rutas encapsula todo lo anterior, desde que llega la consulta hasta que se despacha la respuesta final, pasando por el controlador, el modelo y las vistas (estructuras JSON) definidas, por lo que se comprende como un test unificado de cada proceso.

En Ilustración 27, como se ha hecho para los dos puntos anteriores, se propone el desarrollo y pruebas de una petición en particular, ya que el resto de las peticiones para cada entidad es análogo. En este caso se muestra la petición POST para la ruta *"/v1/workshops/:workshop_id/orders"* la cual estaría simulando la creación de una orden desde un usuario Owner, para añadir un vehículo a su taller. Esta ruta tiene su homónima que realiza el mismo procedimiento, pero desde el lado de un usuario cliente. *"/v1/users/:user_id/orders"*

```ruby
describe "POST /v1/workshops/:workshop_id/orders" do
  it "creates a new order for the specified workshop" do
    order_params = {
      order: {
        damages: { "dent": [1, 5], "scratch": [2], "breakage": [7, 6, 8] },
        jobs: { "dent_remove": [1, 5], "paint": [2], "replace": [7, 6, 8] },
        status: "in_progress",
        delivery_date: "20/02/2023",
        price: 150000,
      },
      workshop_id: workshop.id, client_id: client.id, car_id: car.id
    }
    post "/v1/workshops/#{workshop.id}/orders", params: order_params, headers: headers

    expect(response).to have_http_status(201)
    expect(payload_test[:delivery_date]).to eq(order_params[:order][:delivery_date])
    expect(payload_test[:workshop][:id]).to eq(workshop.id)
    expect(payload_test[:client][:id]).to eq(client.id)
    expect(payload_test[:car][:id]).to eq(car.id)
  end
end
```

Ilustración 27 Prueba para petición POST nueva orden para taller.
Fuente: Elaboración propia

En primer lugar, se genera la estructura JSON a enviar como parámetros del cuerpo de la petición para crear una nueva orden de trabajo. Posteriormente, se valida que la respuesta tenga un código de estado creado (201) y que la data que retorna sea equivalente a la que fue enviada, tanto para el recurso Order, como para los recursos anidados de Workshop, Client y Car. Este chequeo asegura que la orden se le asigne y pertenezca a las entidades correctas mediante la igualación del identificador.

Se agregan también los test negativos para esta ruta, en los que se intenta acceder sin autenticación y con un formato de parámetros incorrectos, para los cuales se espera recibir un código de estado 401 y 400 respectivamente. No se muestran ya que son del mismo tipo que los mostrados anteriormente en el controlador (Ilustración 24 Archivo spec del método create de Order.).

```ruby
Rails.application.routes.draw do
  namespace :v1, defaults: { format: 'json' } do
    resources :users, only: %i[create update] do
      post 'login', on: :collection
      delete 'logout', on: :collection
      put 'update', on: :collection
      resources :orders, shallow: true
    end
    resources :workshops do
      resources :orders, shallow: true
    end
    resources :cars do
      resources :orders, shallow: true
    end
    resources :orders
  end
end
```

Ilustración 28 Archivo routes.rb
Fuente: Elaboración propia

Ya teniendo los métodos de los controladores fue necesario implementar las rutas de la API para que el cliente (aplicación móvil) pueda acceder, crear y modificar los recursos, esto se hace a través del archivo de rutas, donde se declara el esquema y los métodos que se ofrecen para las consultas.

Como se puede apreciar en la Ilustración 28, primero se nombra el espacio de rutas a través del "v1", esto con el objetivo de aplicar versionado a las rutas y controladores de los recursos, lo cual es una buena práctica para las API RESTful, ya que mantiene el código y la estructura de la API versionada.

En primer lugar, se define el recurso de Users que detalla algunas operaciones como login, logout y update, esto porque son métodos propios del controlador de User que no corresponden con los CRUD clásicos como *create* o *index* y contienen la lógica necesaria para la autenticación, proceso que es testeado en cada prueba unitaria de cada controlador y cada una de las solicitudes, ya que es requisito para ambos la autorización del usuario.

Luego se procede a definir los recursos disponibles, a través de la declaración "`resources :<nombre del recurso>`" se declaran las rutas RESTful de cada entidad, es decir, las acciones CRUD de cada uno de ellos. Para el caso de Order, al pertenecer a cada uno de los recursos está declarada dentro de cada uno de forma anidada con el parámetro "`shallow: true`", lo que permite que para la creación, actualización y eliminación de un recurso Order en particular sea necesario acceder por la ruta base del recurso padre, por ejemplo

"/v1/workshops/:workshop_id/orders/:order_id" para obtener o modificar una orden en específico de cierto taller. Además de esto, se declara a Orders como un recurso independiente al final para que la acción GET de una Order en específico quede desacoplada de cualquier entidad, así se puede consultar tanto desde el cliente como desde el taller.

```
$ rspec spec/requests/v1/orders_spec.rb

Randomized with seed 11214

V1::Orders
  GET /v1/orders/:id
    returns the specified order
  GET /v1/users/:user_id/orders
    returns 400 to owner user
  GET /v1/workshops/:workshop_id/orders
    returns a list of orders for the specified workshop
  GET /v1/cars/:car_id/orders
    returns a list of orders for the specified car
  GET /v1/users/:user_id/orders
    returns a list of orders for the specified user (client)
  POST /v1/workshops/:workshop_id/orders
    creates a new order for the specified workshop
  PUT /v1/orders/:id
    updates the specified order
  POST /v1/cars/:car_id/orders
    creates a new order for the specified car
  DELETE /v1/orders/:id
    deletes the specified order
  POST /v1/users/:user_id/orders
    creates a new order for the specified user (client)

Finished in 1.39 seconds (files took 1.52 seconds to load)
10 examples, 0 failures

```

Ilustración 29 Resultado pruebas para solicitudes para Order.
Fuente: Elaboración propia

Nuevamente, al ejecutar las pruebas se ve como son pasadas con éxito todas las solicitudes (ver Ilustración 29), incluida la prueba para la solicitud POST para crear un nuevo registro de Order descrita anteriormente.

A modo de resumen de esta sección, se sigue el ciclo TDD para cada uno de los componentes asociados a los recursos definidos, tanto para modelos, controladores y solicitudes, estas últimas actuando como un conjunto entre modelo, controlador y ruta ya que son las rutas las encargadas de ser este punto de acceso visible a la API hacia el cliente o aplicación que la consuma.

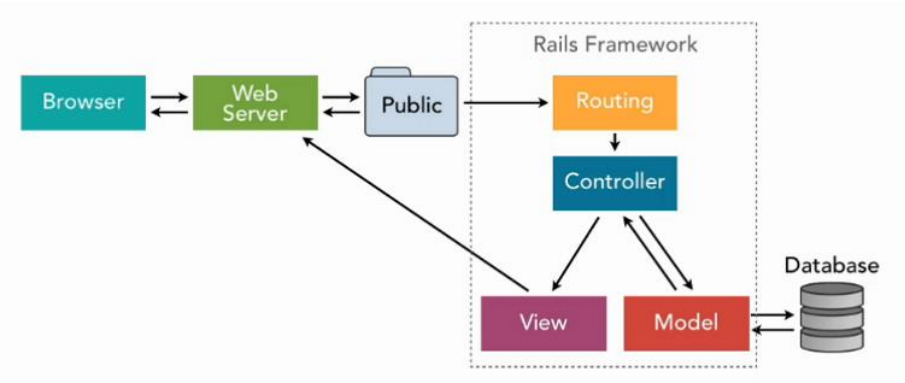


Ilustración 30 Esquema de funcionamiento de aplicaciones Rails
Fuente: (Abdelhamid, 2021)

Tal y como se muestra en la Ilustración 30, son las rutas la primera capa que recibe la consulta y la dirige hacia el método controlador correspondiente, según lo declarado en el archivo de rutas. Ahí es cuando el controlador en conjunto del modelo elabora la vista (en este caso JSON de respuesta) y es respondida al cliente o web server como indica la figura.

4.1.3 DESPLIEGE EN PaaS

Como se ha propuesto en la sección 3.5 se desplegará la API Rails en 3 plataformas distintas, a fin de poder obtener una comparativa de estas y poder determinar una mejor opción para el alojamiento de la API.

Railway

Como primera opción se tiene Railway, una plataforma que se basa en el pago por uso de recursos y ofrece un rápida y sencilla puesta en marcha para las aplicaciones independiente del lenguaje o *framework* que utilicen. Es la aplicación que se utilizó para desplegar lo que fue el *backend* Strapi para su funcionamiento en etapas de desarrollo y producción para la FESW, por lo que será el principal punto de comparación entre el *backend* Strapi y la API Rails.

Siguiendo las instrucciones y guías de despliegue resulta fácil y rápido tener la aplicación disponible en un dominio en la nube, en este caso, luego de hacer *deploy,* el proyecto ya está disponible y mediante la consola de Railway es posible monitorear y realizar distintas acciones como hacer un redespliegue, levantar servicios adicionales, instalar *plugin*, establecer variables de ambiente, ejecutar comandos, etcétera.

Una vez desplegada en el dominio https://api.vrava-prod.up.railway.app/ se puede chequear mediante la consola el estado del servicio (Ilustración 31).

```
$ railway status
  Project: vrava-api
  Environment: production
  Plugins:
    postgresql
  Services:
    vrava_api
```

Ilustración 31 Consola de Railway verificando estado Rails API.
Fuente: Elaboración propia.

En esta plataforma se tiene una visibilidad media – baja respecto a las métricas de consumo, al tener un cobro basado en los recursos utilizados en el tiempo, Railway ofrece tres gráficos como métricas de la aplicación: el uso porcentual de las vCPU, memoria RAM consumida (bytes) y el tráfico de la red HTTP (bytes).

Heroku

Para el caso de Heroku es muy similar, ya que al igual que Railway puede ser configurado con despliegues automáticos asociados a una rama en específico del repositorio GitHub. Heroku utiliza los denominados *dynos* para gestionar las aplicaciones web, en este caso es el encargado de ejecutar el servidor Rails y disponibilizar la API. Luego de la configuración inicial la API queda alojada en la dirección vrava-api.herokuapp.com y se valida que el estado de la aplicación sea correcto y sin errores (ver Ilustración 32).

```
$ heroku status
  Apps: No known issues at this time.
  Data: No known issues at this time.
  Tools: No known issues at this time.
```

Ilustración 32 Consola de Heroku verificando estado.
Fuente: Elaboración propia.

En cuanto a la visibilidad de métricas y recursos, Heroku ofrece un poco más de opciones asociadas a los recursos que está consumiendo la aplicación, por ejemplo, un gráfico que muestra los eventos importantes de la aplicación (logs críticos, advertencias, informativos y

actividad en general), un gráfico acerca del consumo de memoria RAM, uno con los tiempos de respuesta de la aplicación, uno con el porcentaje de carga del *dyno* y otro donde visibiliza el *throughput*.

Fly

Nuevamente el caso de Fly no es muy distinto al de las otras dos plataformas, la única diferencia es que aquí el despliegue inicial y creación de la app es enteramente mediante la consola que genera un *release* en base al directorio local, a diferencia de Railway y Heroku, que dan la opción de desplegar mediante GitHub. En ese sentido, la misma consola da la posibilidad de generar la base de datos de forma automática, lo que reduce la complejidad de configuración dejando todo conectado desde el inicio. La aplicación queda disponible en la dirección vrava-api.fly.dev y es posible verificar el estado a través de la consola (ver Ilustración 33).

```
$ flyctl apps list
NAME          OWNER         STATUS         PLATFORM
vrava-api     personal      deployed       machines
vrava-api-db  personal      deployed       machines
```

Ilustración 33 Comando para visualizar estado de aplicaciones en Fly.
Fuente: Elaboración propia.

Para el caso de Fly, se cuenta con un extenso reporte de métricas de monitoreo de la aplicación, ya que cuenta con el soporte de Grafana, una herramienta dedicada para aquello. Es posible diseñar *dashboards* con la información necesaria con todas las opciones que ofrecen las otras dos plataformas y muchas más.

4.2 RESULTADOS

4.2.1 TDD

Una vez desarrollada la API en su totalidad cumpliendo todos los requisitos funcionales, siendo capaz de sustituir al *backend* Strapi y más importante aún, desarrollada con la metodología TDD, es cuando se pueden obtener métricas y resultados cualitativos del código respecto a su calidad, entre ellos la cobertura de los tests, que indica cuánto código está cubierto por las pruebas escritas. Para este caso, RSpec permite identificar la cobertura por cada componente probado y por la totalidad de la aplicación.

```
$ rspec

Randomized with seed 30157
.....................................................................
.....................................................................

Finished in 4.64 seconds (files took 1.88 seconds to load)
147 examples, 0 failures

```

Ilustración 34 Resultado test integración.
Fuente: Elaboración propia.

Se puede ver en la Ilustración 34 que al ejecutar los 147 specs escritos para probar unitariamente los componentes de la API se obtiene un resultado exitoso en cada uno de ellos, lo cual es resultado de ir desarrollando las funcionalidades guiado por los tests. Esto da pie a dos aspectos importantes respecto al código de la API, en primer lugar, es un código que realiza de manera correcta las funciones descritas en los test, cumple con los requisitos planteados en ellos, además de esto, los tests generan una documentación viva de las mismas funcionalidades y aportan a la hora de la mantenibilidad.

Ejemplificando lo anterior, si en un futuro es necesario un cambio en una funcionalidad o bien agregarle un criterio adicional, para el desarrollador le será de suma ayuda contar con las pruebas unitarias, ya que con ellas puede saber exactamente qué hace cierta función o componente sin necesidad de leer el código, con lo que se da pie a que comience a desarrollar lo nuevo inmediatamente iniciando por agregar los casos de prueba necesarios para cumplir esta funcionalidad que se quisiera implementar.

La Ilustración 35 muestra un poco más de detalle acerca de la cobertura que alcanzan las pruebas RSpec implementadas, llegando a ser un total de 99.44% de cobertura total en el código fuente de la API Rails pasando por un promedio de 6.45 veces cada línea.

Para el caso específico de los controladores se muestra que llega solo a un 97.47%, esto es debido a que las líneas de código del controlador de Orders que se encargan del almacenamiento de imágenes para las órdenes de trabajo y solicitudes (Active Storage) no son cubiertas por las pruebas, ya que requiere adjuntar a la estructura JSON de la Order los archivos de las imágenes como tal, lo cual no es posible implementar en RSpec. Sin embargo, esta parte del código fue testeada de forma tradicional con consultas de prueba con Postman.

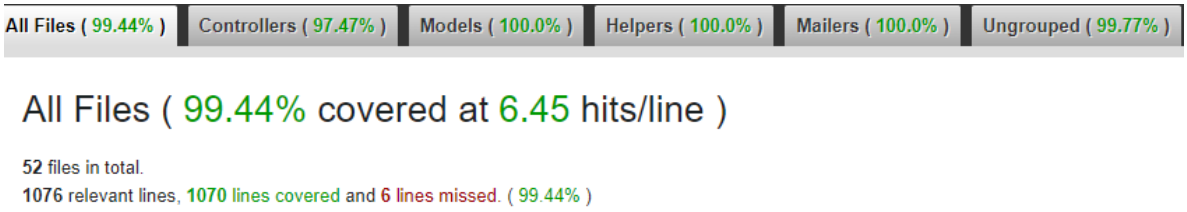


Ilustración 35 Resultados test de cobertura.
Fuente: RSpec coverage report.

4.2.2 DOCUMENTACIÓN

En cuanto a la documentación generada, se obtiene las mismas pruebas RSpec como un gran complemento a la documentación técnica de cada controlador, modelo y solicitud, que bajo la idea de TDD el propósito de estas pruebas es mantenerlas actualizadas y que sean las encargadas de guiar el desarrollo.

Como principal medio de documentación se propuso Postman, un completa herramienta de testing y depuración de sistemas que, entre muchas otras opciones, permite documentar APIs mediante el almacenamiento de las peticiones HTTP o *endpoints* disponibles en la API, se presenta aquí la Documentación Postman.

4.2.3 PERFORMANCE TESTING

Se procede con la ejecución de los escenarios descritos en Tabla 2 Plan de ejecución de pruebas., para cada uno de los artefactos desplegados. Se ejecutan los escenarios bajo, medio y masivo de manera consecutiva, para luego proceder a obtener data asociada al rendimiento de cada uno de los artefactos a través de la herramienta New Relic e información variada que entrega cada PaaS sobre las métricas de los despliegues. Finalmente, se genera una tabla que resume las métricas obtenidas más importantes que permitan comparar todos los artefactos desplegados.

Cabe destacar que en esta sección se presentarán imágenes de gráficos con el objetivo de comparar y obtener datos importantes, estas imágenes se mostrarán en un tamaño reducido a modo de no saturar esta sección, pero se encontrarán como anexo al final del documento, para visualizarlas con mayor definición.

4.2.3.1 COMPARACIÓN: STRAPI BACKEND Y RAILS API EN RAILWAY

Se procede con la comparación del backend desarrollado con Strapi y la nueva Rails API, ambos artefactos desplegados en la plataforma Railway. El objetivo principal de este apartado es obtener una mirada comparativa de las métricas de Strapi versus la API Rails bajo el mismo ambiente de despliegue Railway.

La primera comparación nace a partir de los gráficos de tiempo consumido por las cinco transacciones HTTP con mayor *throughput* (ver Ilustración 36).

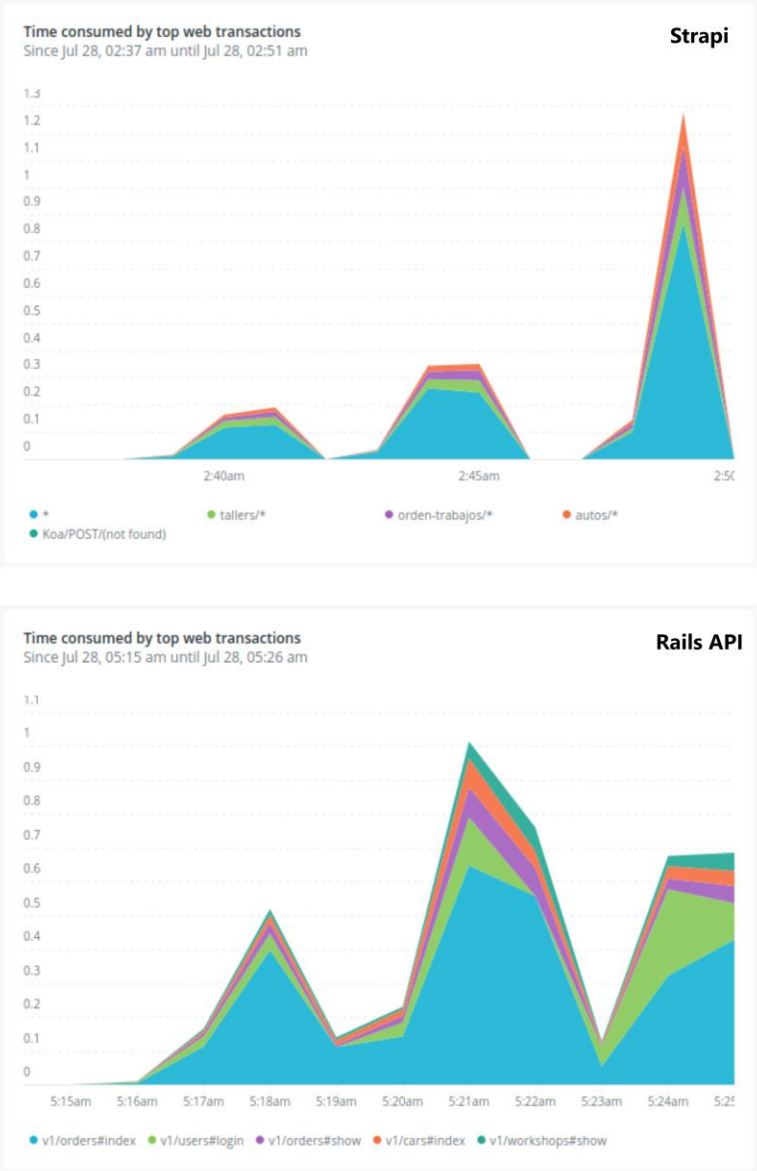


Ilustración 36 Comparativa gráficos de tiempo consumido por las principales transacciones HTTP. Fuente: New Relic, transacciones

Según New Relic, este gráfico muestra la distribución del tiempo del reloj de pared consumido por las principales transacciones con mayor *throughput*. Según explican "El tiempo de reloj de pared es la cantidad de tiempo registrada por el reloj. New Relic utiliza el tiempo de reloj de pared para todas las transacciones y luego suma ese valor en todas las transacciones." y en adición a esto, se menciona que es posible obtener valores porcentuales superiores a 100 ya que el *host* puede ejecutar solicitudes en paralelo, es

decir, 100% indicaría que el tiempo de ejecución a través de todas las transacciones seleccionadas es igual al tiempo empleado al registrar el tiempo del reloj de pared.

Se puede notar que para el caso de Strapi se tiene una transacción etiquetada como "*" que abarca el mayor porcentaje en todos los escenarios de pruebas, alcanzando cerca del 90% en el caso de prueba masivo y según se puede monitorear con New Relic, corresponde en gran parte a procesos de *middleware* relacionados con el enrutador de los *endpoints*. Esto es también lo que explica el por qué no aparece en el listado las peticiones asociadas al *endpoint* "/solicitudes", seguramente por algún tema de configuración realizado al momento de desarrollar el *backend* el *endpoint* quedó encapsulado en la ruta base "*" y New Relic lo interpreta como tal. Tampoco se descarta un posible *bug* relacionado a la versión de Strapi, que con el pasar del tiempo sin mantención se quedó en una versión desactualizada.

Por el lado de la API Rails, destaca el gran porcentaje que acumula el método "/orders#index" llegando a un poco más del 60% del tiempo consumido en el escenario medio, siendo superior a los escenarios bajo y masivo. En segundo lugar, y cercano al resto de las otras peticiones se encuentra el método de *login*.

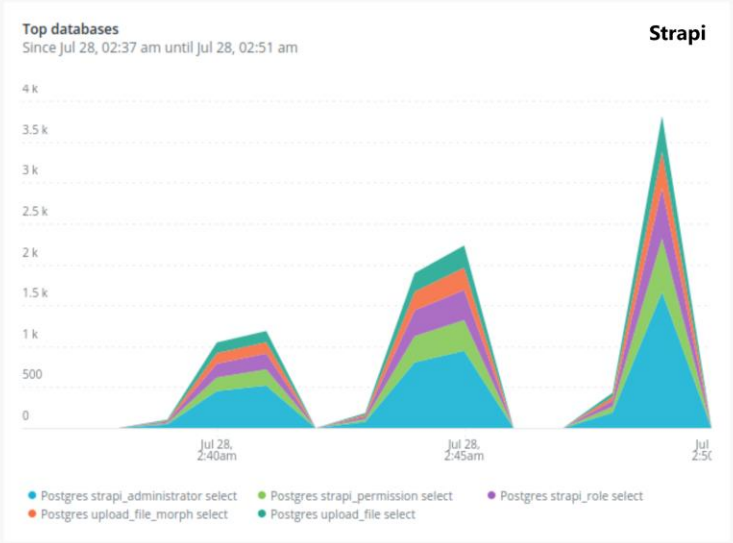
Ilustración 37 Comparativa de gráficos de tiempos de respuesta por percentiles. Fuente: New Relic APM, transacciones

En el gráfico de la Ilustración 37 es posible apreciar cómo para ambos artefactos, se marcan perfectamente en el tiempo los *peaks* causados por los escenarios bajo, medio y masivo.

Para el caso de Strapi, se tiene un promedio que oscila entre los 50 y 60 milisegundos para los escenarios bajo y medio, mientras que para el escenario masivo el promedio sube hasta los 125 milisegundos. Casos similares para el percentil 95 y percentil 99 que se mantienen estables para los casos bajo y medio, variando entre los 100 y 150 milisegundos para el

percentil 95 y entre los 190 y 250 milisegundos para el percentil 99. Mientras que para el caso masivo se disparan hacia los 350 y 430 milisegundos respectivamente.

Considerando a la API Rails se obtienen tiempos de respuesta, a la vista, superiores a los de Strapi. El promedio oscila sorprendentemente entre los 150 a 350 milisegundos en el escenario bajo y procede a mantenerse más estable para el caso medio y masivo, variando entre los 150 y 250 milisegundos. Como es de esperar los percentiles se vieron afectados de la misma manera que el promedio, alcanzando valores máximos de entre 500 y 600 milisegundos para el percentil 95 y entre 600 y 700 milisegundos para el percentil 99 en los escenarios medio y masivo.



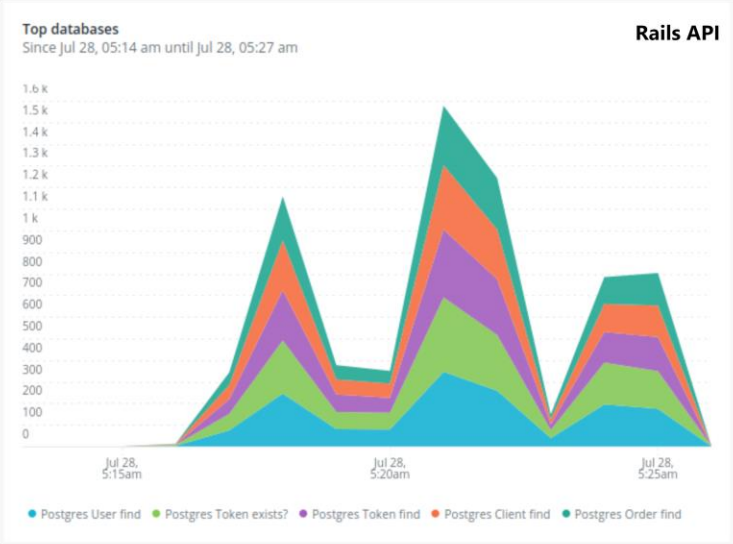


Ilustración 38 Comparativa gráficos de consultas más frecuentes a base de datos. Fuente: New Relic APM, base de datos

En la Ilustración 38 se tienen los gráficos respectivos para cada artefacto de las cinco operaciones a base de datos más frecuentes (las con mayor *throughput*) mostrando el acumulado de las totales en cada escenario de pruebas.

Es posible notar que para Strapi se tiene nuevamente muy marcado el patrón de los *peaks* provocados por los escenarios de prueba, alcanzando las 2.200 operaciones en el caso bajo y llegando hasta las 3.800 operaciones en el caso masivo. En este caso se puede apreciar que la llamada "strapi_administrator select" es la con mayor *throughput* (*calls per minute*) alcanzando un promedio de 335 llamadas por minuto. Cabe mencionar que de las cinco operaciones a base de datos con mayor *throughput* ninguna es asociada directamente a un recurso como órdenes de trabajo o talleres, las cinco están relacionadas con chequeos de permisos y roles o archivos.

Para el caso de Rails API, se ve un comportamiento considerablemente parejo en los tres escenarios de prueba, incluso obteniendo el valor más bajo en el escenario masivo de prueba, donde solamente se alcanza cerca de las 800 consultas, por debajo del *peak* más alto alcanzado en el escenario medio que fue de alrededor de las 1.550 consultas.

A lo anterior se le suma el hecho de que las consultas con mayor *throughput* (*calls per minute*) corresponden a las de "*posgres token exists*" y *"posgres user find"* que ambas alcanzan un *throughput* de 124 y son gatilladas por la acción de autenticación para cada petición HTTP.

La comparativa de gráficos de la Ilustración 39, muestra el consumo porcentual de CPU por cada aplicación. Se puede distinguir claramente que para el caso de Strapi, el consumo se ve afectado de forma proporcional a la carga que se ve sometida la aplicación, ya que es notoria la subida conforme a los casos de prueba que van aumentando uno tras otro la carga simultánea.

Ilustración 39 Comparativa gráficos de uso porcentual de CPU. Fuente: New Relic APM

Nuevamente es visible cómo a pesar de que la carga aumenta a medida que avanza el tiempo y van ejecutándose los escenarios de prueba, el uso del recurso CPU no se ve reflejado un comportamiento al alza, sino más bien un comportamiento parejo y constante. Incluso es llamativo el hecho de que el máximo de uso se logra mientras se ejecuta el escenario de pruebas bajo.

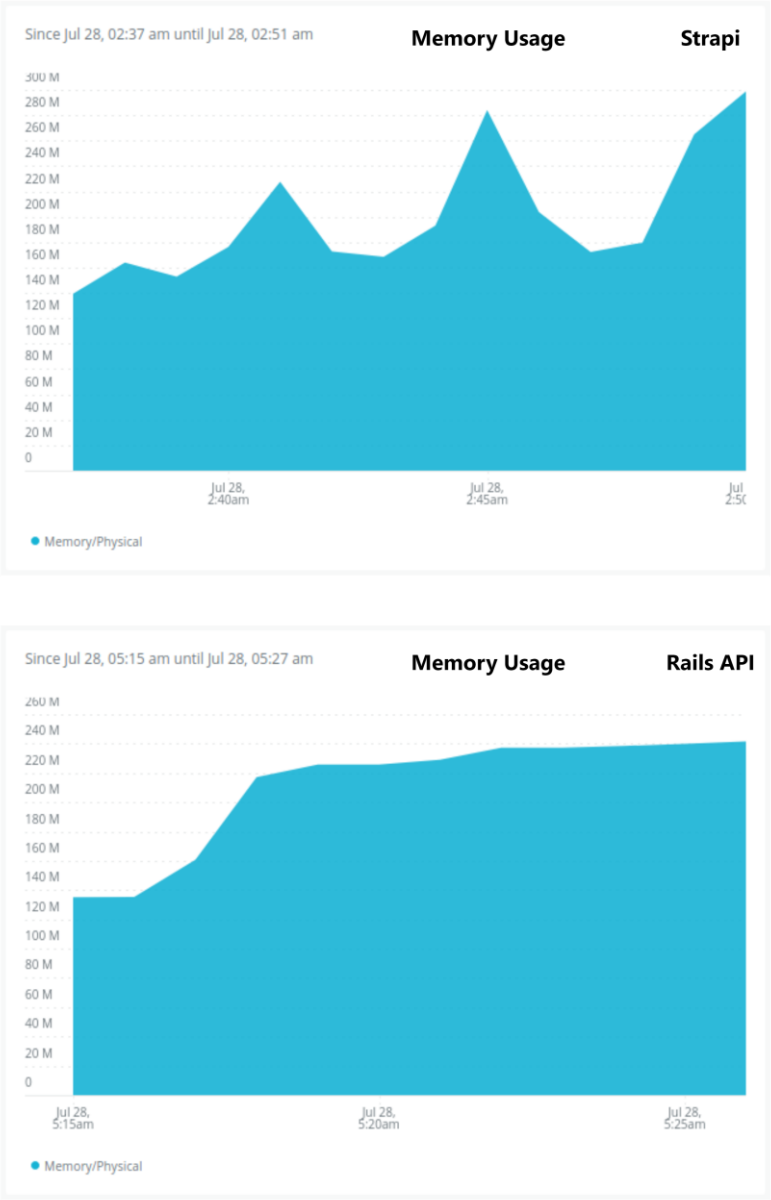
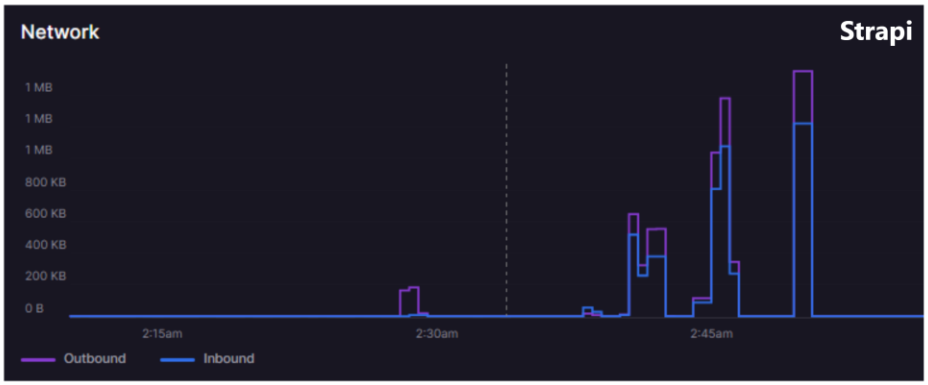


Ilustración 40 Comparativa gráficos de uso de memoria.
Fuente: New Relic APM, *metrics.*

En cuanto a Ilustración 40 se puede ver cómo el uso de la memoria en el caso de Strapi tiene *peaks* que se corresponden con los escenarios de carga, en cada uno se ven alzas que llegan hasta los 300 MB en el caso del *peak* asociado al escenario masivo.

En el caso de Rails es distinto, se ve cómo al inicio hay un alza considerable desde los 120 MB hasta los 220 MB aproximadamente, para luego quedar estabilizada en torno a los 240 MB a lo largo de toda la ejecución de los escenarios. A diferencia de Strapi, no se genera

esta liberación de memoria luego de cada *peak*, sin embargo, no sufre alzas bruscas de consumo.



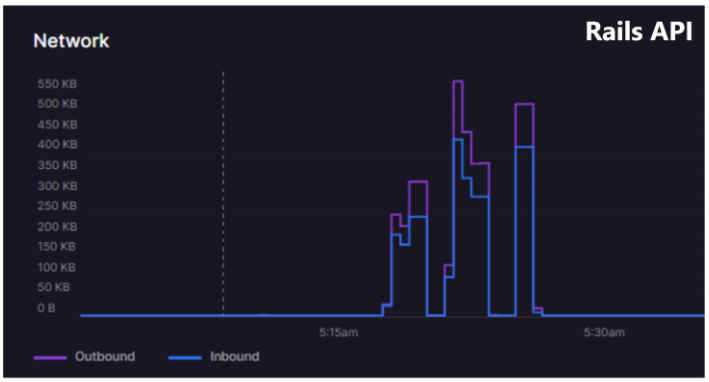


Ilustración 41 Comparativa gráficos de flujo de red. Fuente: Railway App Metrics.

Claramente los escenarios de prueba generan un impacto en la entrada y salida de data a través de la red (ver Ilustración 41), así como entran datos en forma de JSON, salen aún más en cada respuesta de las aplicaciones. En ambos artefactos se ve el mismo patrón, sin embargo, en Rails API se tienen números menores en lo que respecta a tamaño de las entradas y salidas. Si bien el gráfico que proporciona Railway no es exacto, da una clara idea de que los tamaños de los datos que entran y salen son menores para Rails, el máximo de entrada es cercano a los 450 KB y el máximo de salida es cercano a los 600 KB, mientras que para Strapi se alcanzan máximos de entrada cercanos a 1,1MB y máximos de salida cercanos a 1,3 MB.

Para finalizar la comparativa, resulta muy útil ver el comportamiento del puntaje Apdex a través de los escenarios de prueba en cada artefacto y ver el nivel de satisfacción con el que responden cada una de las aplicaciones.

Ilustración 42 Comparativa gráficos de Apdex Score. Fuente: New Relic APM, metrics.

Dada la Ilustración 42 es posible ver un comportamiento decreciente para el puntaje Apdex en el *backend* Strapi que decae de manera importante sobre todo en lo que corresponde al escenario masivo de pruebas, obteniendo un valor promedio de 0,87. Por otro lado, Rails obtiene un Apdex mucho más estable que no decae más allá del valor 0,93 en ningún momento, manteniéndose entre 1 y 0,93 durante toda la ejecución de las pruebas y obteniendo un valor promedio de 0,97.

4.2.3.2 COMPARACIÓN: RAILS API EN RAILWAY, HEROKU Y FLY

Se procede con la comparación del rendimiento y las métricas obtenidas por la API Rails en los distintos ambientes o plataformas seleccionadas: Railway, Heroku y Fly. El objetivo principal de este apartado es obtener una mirada comparativa de las métricas que obtiene la API Rails desarrollada en cada uno de estos ambientes de despliegue y con ello determinar cuál es el mejor para el caso. Más allá de analizar los gráficos métrica por métrica, como se hizo en el punto anterior, resulta más interesante resaltar algunas de las pocas diferencias de comportamiento que se encontraron tras comparar las métricas y revisarlas a través de tablas numéricas.

Se detectaron las siguientes diferencias significativas:

1. Diferencia considerable en tiempos de login: Se pudo detectar una clara diferencia en la distribución del porcentaje de tiempo consumido por cada *endpoint*, específicamente en el caso de Railway se observa que la petición hacia "orders#index" abarca durante los tres escenarios la mayoría del tiempo ocupando significativamente gran parte del área del gráfico con un 57,5%, luego por detrás aparece "users#login" con solo el 14%. Por el lado de Heroku se observa un ascenso hasta del 23,25% del tiempo total que fue consumido por el método "users#login", mientras que "orders#index" alcanza el 50,7%.

Tabla 3 Comparativa de transacciones listado de órdenes y login por cada plataforma.
Fuente: Elaboración propia

Transaction	Total time	Avg	Min	Max	Median	95th %	99th %	Apdex	Error rate	Throughp	Total count
					RAILWAY						
v1/orders#index	57.49%	317 ms	156 ms	895 ms	260 ms	590 ms	723 ms	0.93	0	44 rpm	525
v1/users#login	14.20%	294 ms	231 ms	672 ms	252 ms	502 ms	553 ms	0.97	0	12 rpm	140
					HEROKU						
v1/orders#index	50.67%	104 ms	53.1 ms	423 ms	77.6 ms	231 ms	295 ms	1	0	57 rpm	735
v1/users#login	23.25%	252 ms	230 ms	380 ms	239 ms	306 ms	328 ms	1	0	11 rpm	140
					FLY						
v1/orders#index	40.93%	155 ms	59.2 ms	884 ms	107 ms	400 ms	574 ms	0.99	0	57 rpm	685
v1/users#login	40.12%	743 ms	235 ms	1.33 s	840 ms	1.25 s	1.29 s	0.71	0	12 rpm	140

Es posible notar que es aún mayor esta diferencia para el caso de Fly, llegando a utilizar el mismo porcentaje de tiempo que "orders#index" con un *throughput* casi cinco veces menor y un promedio de más de 700 milisegundos.

Resulta muy alarmante que, para la misma aplicación, con el mismo plan de pruebas y en distintas plataformas de despliegue se produzcan diferencias tan significativas para procesos repetitivos para los usuarios como el login.

2. Diferencias en tiempos de respuesta (Ilustración 43): Se detectan diferencias sustanciales en las métricas asociadas a tiempos de respuesta, los comportamientos de alzas y *peaks* son muy similares, pero hay una gran diferencia en los valores alcanzados por cada plataforma.

Ilustración 43 Comparativa gráficos de tiempos de respuesta por percentiles para cada plataforma. Fuente: New Relic APM, transacciones.

En cuanto a los promedios se puede notar que el valor máximo se alcanza para el caso de Railway con un valor aproximado de 350 milisegundos, que curiosamente ocurre en el peak asociado al escenario de pruebas bajo, posterior a alcanzar dicho valor tiende a estabilizarse alrededor de los 150 y 250 milisegundos. Aparentemente el caso más favorable terminó siendo el de Heroku donde el promedio se mantiene estable en valores cercanos a los 60 milisegundos con picos de hasta los 180 milisegundos. El caso de Fly es similar al de Heroku, sin embargo, se obtiene un *peak* mayor en el escenario masivo, donde alcanza casi los 300 milisegundos.

Para el caso de la mediana, su comportamiento no es muy distinto al del promedio, estando incluso por debajo de este último la mayoría del tiempo para cada plataforma.

Para los percentiles 95 y 99 el comportamiento es claro, a mayor el *throughput* es mayor el tiempo de respuesta y aparecen peticiones *outliers* con duraciones que se escapan de promedio y la mediana. El caso más exagerado ocurre en Fly donde estos percentiles alcanzan valores por sobre los 1000 milisegundos (ver Ilustración 44), un resultado asociado a peticiones de login, donde cuatro de ellas alcanzaron un tiempo de respuesta superior a 950 milisegundos.

Transaction	↓ Duration
v1/users#login Controller/v1/users/login	1161 ms
v1/users#login Controller/v1/users/login	1117 ms
v1/users#login Controller/v1/users/login	966 ms
v1/users#login Controller/v1/users/login	960 ms
v1/orders#index Controller/v1/orders/index	434 ms

Ilustración 44 Lista de cinco peticiones con mayor tiempo para Fly. Fuente: New Relic APM, transacciones.

En el caso de Heroku se tiene una situacion particular donde las 21 primeras peticiones de las que tienen mayor tiempo de respuesta corresponden a login, superando los 230 milisegundos y alcanzando un máximo de 307 milisegundos, seguidas de la número 22 correspondiente a una petición “orders#index” de tan solo 149 milisegundos. Lo anterior explica de cierta forma que la dispersión de los percentiles 95 y 99 respecto a la mediana y el promedio, se deben a motivos particulares del login como funcionalidad y, en cierto punto, relacionados con la plataforma de despliegue, ya que ocurre en Heroku y en Fly (con la evidente y ya mencionada diferencia de valores alcanzados, siendo Heroku mucho más estable) mientras que en Railway no.

3. Apdex Score: Dada la importancia de esta métrica resulta interesante ver la comparativa de las plataformas respecto a la misma (ver Ilustración 45).

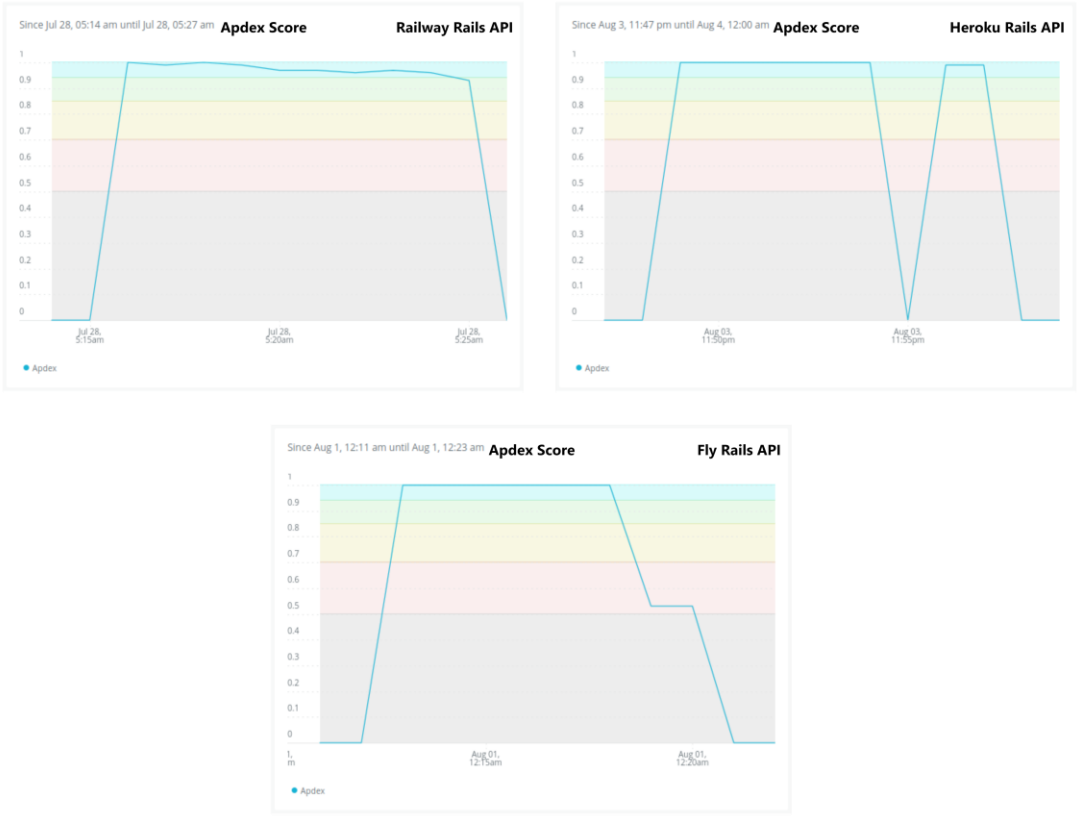


Ilustración 45 Comparativa gráficos de Apdex score para cada plataforma. Fuente: New Relic APM, métricas.

Es clara la diferencia presente en la comparativa, la aplicación desplegada en Heroku presenta un constante puntaje Apdex que alcanza su mínimo de 0,99 para el caso masivo de pruebas, siendo superior en todo aspecto a Railway que se mantiene estable durante toda la ejecución, pero alcanzando mínimos de 0,93. Por otro lado, Fly es la que tiene mayor problema respecto al Apdex Score alcanzando mínimos graves de 0,53 en el escenario de pruebas masivo, donde se condice con las anteriores métricas que reflejaban los *outliers* existentes en este escenario.

4. Histograma de distribución de tiempos: Como se puede apreciar en la Ilustración 46, comparativa de histogramas de la frecuencia de tiempos de respuestas (en el eje abscisas el tiempo en milisegundos, en el eje ordenado la frecuencia), es notable cómo la plataforma que mejor se comporta en este aspecto es Heroku, concentrando su mayor densidad por debajo de los 60 milisegundos. En los casos de Railway y Fly se ven los *outliers* reflejados en aquellas barras cercanas a los 900 milisegundos (0,9 en el gráfico) siendo Fly el caso donde se supera incluso el umbral de los 1000 milisegundos.

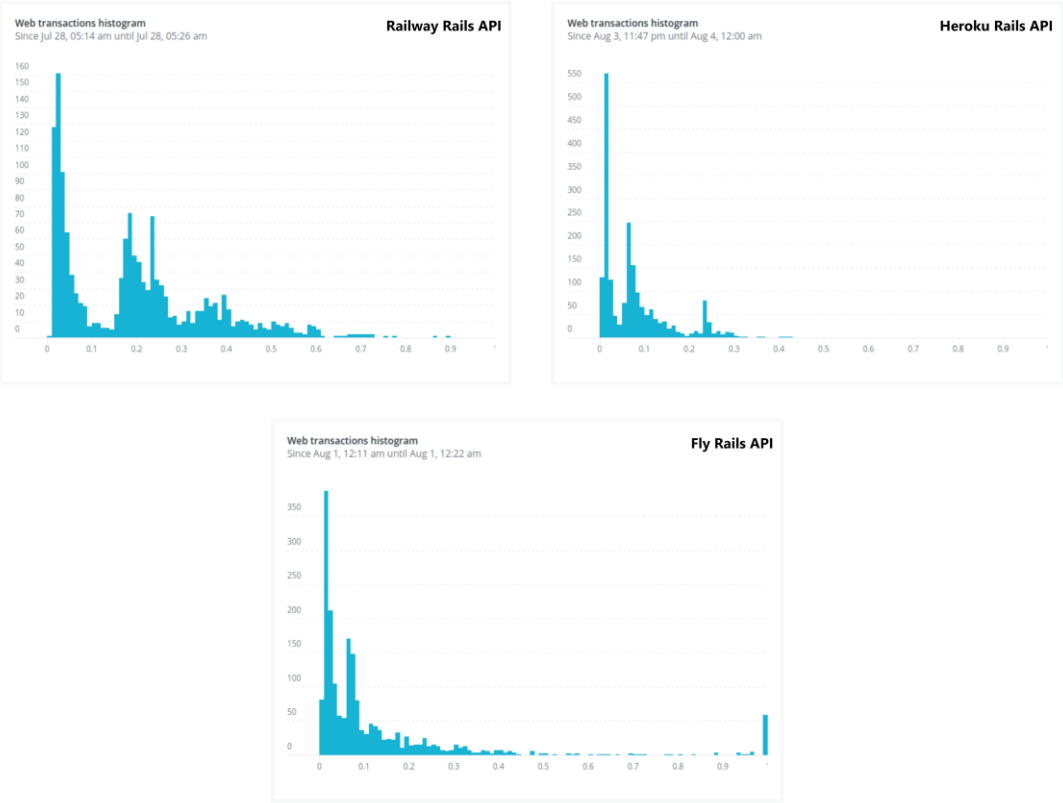


Ilustración 46 Comparativa de histogramas por tiempo de respuesta por cada plataforma. Fuente: New Relic APM, métricas.

4.3 RESUMEN COMPARATIVO

A continuación, se presentan tablas resumen de las transacciones recibidas por cada artefacto (*backend* Strapi en Railway, Rails API en Railway, Rails API en Heroku y Rails API en Fly) con los promedios por cada métrica y su desviación estándar.

Tabla 4 Tabla resumen de transacciones Strapi en Railway.
Fuente: Elaboración propia / New Relic APM

Transaction	Total time	Avg [ms]	Min [ms]	Max [ms]	Median [ms]	95th % [ms]	99th % [ms]	Apdex	Error rate	Throughpu	Total count
*	69.76%	85.50	16.7	495	69.3	239	400	0.86	0.0081	88 rpm	1.23k
tallers/*	11.06%	92.40	43.4	416	72.2	303	424	0.87	0	13 rpm	181
orden-trabajos/*	10.79%	79.20	25.2	380	44.8	303	354	0.89	0	15 rpm	206
autos/*	8.27%	59.20	19.2	272	36.8	194	268	0.93	0	15 rpm	211
Koa/POST/(not found	0.05%	71.20	71.2	71.2	71.3	71.3	71.3	1	0	0.07 rpm	1.00
Koa/GET/(not found)	0.05%	70.00	70	70	70.2	70.2	70.2	1	0	0.07 rpm	1.00
GET/(not found)	0.02%	34.60	34.6	34.6	35	35	35	1	0	0.07 rpm	1.00
Promedio		**70.30**	**40.04**	**248.40**	**57.09**	**173.64**	**231.79**	**0.94**			**7**
Desv. Estandar		**19.14**	**22.76**	**189.61**	**17.33**	**114.42**	**169.35**	**0.06**			

Tabla 5 Tabla resumen de transacciones Rails API en Railway.
Fuente: Elaboración propia / New Relic APM

Transaction	Total time	Avg [ms]	Min [ms]	Max [ms]	Median [ms]	95th % [ms]	99th % [ms]	Apdex	Error rate	Throughput	Total count
v1/orders#index	57.49%	317	156	895	260	590	723	0.93	0	44 rpm	525
v1/users#login	14.20%	294	231	672	252	502	553	0.97	0	12 rpm	140
v1/orders#show	6.77%	115	16	515	45.1	395	488	1	0	14 rpm	171
v1/cars#index	5.95%	96.4	27.7	393	49.4	334	392	1	0	15 rpm	179
v1/workshops#show	5.15%	91	18.6	686	40.5	348	473	1	0	14 rpm	164
v1/cars#show	4.46%	70.6	9.66	427	22.6	262	387	1	0	15 rpm	183
v1/orders#update	2.22%	174	19.5	439	175	438	439	1	0	3.08 rpm	37
v1/orders#create	1.89%	131	19.6	507	60.7	365	507	0.99	0	3.50 rpm	42
v1/cars#create	1.50%	104	14.4	389	38.5	342	389	1	0	3.50 rpm	42
v1/workshops#index	0.37%	153	37.6	342	61.2	342	342	1	0	0.6 rpm	7
Promedio		**154.6**	**55.006**	**526.5**	**100.5**	**391.8**	**469.3**	**0.989**			**1490**
Desv. Estandar		**85.2178**	**75.4899**	**173.603**	**92.053161**	**95.077512**	**110.28957**	**0.023**			**1490**

Tabla 6 Tabla resumen de transacciones Strapi en Heroku.
Fuente: Elaboración propia / New Relic APM

Transaction	Total time	Avg [ms]	Min [ms]	Max [ms]	Median [ms]	95th % [ms]	99th %	Apdex	Error rate	Throughput	Total count
v1/orders#index	50.67%	104	53.1	423	77.6	231	295	1	0	57 rpm	735
v1/users#login	23.25%	252	230	380	239	306	328	1	0	11 rpm	140
v1/orders#show	6.65%	40	8.79	245	14.7	138	212	1	0	19 rpm	252
v1/cars#index	5.76%	33.7	12.8	404	17.9	101	229	1	0	20 rpm	259
v1/workshops#show	5.25%	33.3	9.24	213	15.1	110	155	1	0	18 rpm	239
v1/cars#show	4.61%	28.2	5.92	159	10.9	99.1	146	1	0	19 rpm	248
v1/orders#update	1.30%	42	10.2	216	15.9	112	161	1	0	3.62 rpm	47
v1/orders#create	1.17%	34.8	11.6	138	15.9	119	138	1	0	3.92 rpm	51
v1/cars#create	1.06%	31.5	8.78	147	12.5	121	147	1	0	3.92 rpm	51
v1/workshops#index	0.27%	45.2	21.4	112	27.3	112	112	1	0	0.7 rpm	9
Promedio		**64.47**	**37.183**	**243.7**	**44.68**	**144.91**	**192.3**	**1**			
Desv. Estandar		**69.4473**	**69.1448**	**116.988**	**71.0791546**	**68.364813**	**72**	**0**			

Tabla 7 Tabla resumen de transacciones Rails API en Fly.
Fuente: Elaboración propia / New Relic APM

Transaction	Total time	Avg [ms]	Min [ms]	Max [ms]	Median [ms]	95th % [ms]	99th % [ms]	Apdex	Error rate	Throughput	Total count
v1/orders#index	40.93%	155	59.2	884	107	400	574	0.99	0	57 rpm	685
v1/users#login	40.12%	743	235	1.33	840	1.25	1.29	0.71	0	12 rpm	140
v1/cars#index	4.68%	49.5	17.8	272	28.2	159	228	1	0	20 rpm	245
v1/workshops#show	4.54%	45	12.4	352	22	163	220	1	0	22 rpm	261
v1/orders#show	4.31%	48.1	12.2	256	24.7	171	231	1	0	19 rpm	232
v1/cars#show	2.66%	27.3	6.94	304	11.8	85	209	1	0	21 rpm	252
v1/orders#create	0.91%	52.3	13.5	225	20.5	151	225	1	0	3.75 rpm	45
v1/orders#update	0.83%	49.1	14.7	272	19.5	173	272	1	0	3.67 rpm	44
v1/cars#create	0.74%	42.7	10.7	162	21.1	142	162	1	0	3.75 rpm	45
v1/workshops#index	0.30%	128	25.8	338	37	338	338	1	0	0.5 rpm	6
Promedio		**134**	**40.824**	**306.633**	**113.18**	**178.325**	**246.029**	**0.97**			
Desv. Estandar		**217.92**	**69.8488**	**226.607**	**256.817089**	**114.24025**	**144.02854**	**0.091**			

En verde se resaltan las métricas que obtienen los mejores promedios y desviaciones estándar (los valores más bajos). Entendiendo que el promedio es solo una métrica que representa el general de los casos para los tiempos de respuesta, es que se acompañan con la desviación estándar para ver qué tanto varían estos tiempos de respuesta por cada artefacto y sus respectivos *endpoints*.

Resulta evidente que es Heroku la plataforma que obtiene los mejores resultados, ya que no solo obtiene los mejores tiempos promedios en cada métrica, sino que además obtiene las menores dispersiones respecto a la media en cada métrica (desviación estándar). Solo en el caso de la media, los mínimos y la mediana es que Strapi obtiene menores desviaciones, lo cual se puede explicar con la agrupación que se vio en el *endpoint* "/*" por parte de New Relic, ya que representa un gran porcentaje del total de las transacciones.

Finalmente, se presenta una tabla comparativa de los costos asociados a cada plataforma para comparar sus planes y cuánto cuesta mantener tanto el *backend* Strapi como la API Rails. Se muestran los planes de cada plataforma que cumplan con las mismas o similares especificaciones técnicas, debido a que las plataformas ofrecen sus planes y posibilidades de despliegue de manera particular, resulta difícil compararlas en exactamente las mismas condiciones, pero resulta muy útil tener estos parámetros y precios para tener una idea de cuál sería la opción más conveniente.

En el caso de Railway, se ofrece un plan basado en el uso que se le dé a la plataforma (*Usage-based subscription*). Esto es se genera una factura según la cantidad de recursos utilizados durante el mes, en Tabla 8 se muestran los costos asociados al plan "Hobby Plan" que ofrece hasta 8 GB de RAM, 8 vCPU y 100 GB de disco compartido. Railway cobra por los tres ítems mencionados una tarifa por minuto de utilización, la cual, para efectos prácticos de este análisis, se convierte a un costo por hora mediante una multiplicación simple del costo por minuto resultando los valores por hora de la Tabla 8.

Tabla 8 Precios por ítem Railway.
Fuente: Railway Pricing

Railway	Costo unitario [USD]
RAM [GB/h]	0.01386
CPU [vCPU/h]	0.02778
Disk [per GB]	0.1

Tal como indica la tabla, se cobra un cierto monto por cada hora de utilización de 1 GB de RAM y por cada núcleo virtual que consuma el contenedor como CPU, además de otro monto fijo por cada GB de almacenamiento en disco. De lo visto en las pruebas realizadas (Ilustración 39), tanto Strapi como la API Rails tienen máximos no superiores al 70% de utilización en Railway por lo que en un caso pesimista donde consumen el 100% de CPU de manera permanente, junto con un aproximado de 500 MB de memoria utilizada, se obtiene un aproximado de un monto en torno a los 25 USD mensuales. Nuevamente hay que destacar que esto sería en un caso pesimista donde a cada hora se consumen los recursos mencionados anteriormente, llevando los valores por hora a valores mensuales en un ejercicio de extrapolación para obtener un costo estimado.

En lo que respecta a Heroku se obtiene que su manera de cobrar es un tanto distinta a la de Railway, ya que en esta ocasión se cuantifican los *dynos* y sus tipos, Heroku ejecuta la app en contenedores Linux ligeros y aislados llamados "*dynos*". La plataforma ofrece diferentes tipos de *dyno* dependiendo del tipo de app y el precio por *dyno* al mes indicado para cada tipo es el máximo que se cobra si se ejecuta el *dyno* 24 horas al día, 7 días a la semana.

Tabla 9 Precios por tipos de Dyno.
Fuente: Heroku Pricing

Dyno Type	Price per dyno/month
Eco	US$5 for 1,000 dyno hours per month, shared across all your Eco dynos
Basic	US$7
Standard-1X	US$25
Standard-2X	US$50
Performance-M	US$250
Performance-L	US$500

Como indica la Tabla 9, un *dyno* standard-2x con 1 GB de memoria tendría un costo asociado de 25 USD, siendo el recomendado por Heroku para aquellas aplicaciones de negocio que se ejecutan en producción. Recién con el *dyno performance-M* es que se obtiene una arquitectura similar en cuanto a memoria y unidades de procesado a la de Railway, pero con un costo casi diez veces mayor, pero que según Heroku, es recomendado para aplicaciones de alto tráfico y baja latencia. (Heroku)

En cuanto a Fly, se tienen precios por cada máquina según su configuración (ver Tabla 10):

Tabla 10 Precios por máquina alojada en Fly.io.
Fuente: Fly.io Pricing

	CPU(s)	RAM	Price
shared-cpu-1x	1 shared	256MB	$0.0000008/s ($1.94/mo)
		512MB	$0.0000012/s ($3.19/mo)
		1GB	$0.0000022/s ($5.70/mo)
		2GB	$0.0000041/s ($10.70/mo)
shared-cpu-2x	2 shared	512MB	$0.0000015/s ($3.89/mo)
		1GB	$0.0000029/s ($7.64/mo)
		2GB	$0.0000049/s ($12.64/mo)
		4GB	$0.0000087/s ($22.65/mo)
shared-cpu-4x	4 shared	1GB	$0.0000030/s ($7.78/mo)
		2GB	$0.0000064/s ($16.53/mo)
		4GB	$0.0000102/s ($26.54/mo)
		8GB	$0.0000180/s ($46.55/mo)
shared-cpu-8x	8 shared	2GB	$0.0000060/s ($15.55/mo)
		4GB	$0.0000132/s ($34.31/mo)
		8GB	$0.0000210/s ($54.32/mo)
		16GB	$0.0000364/s ($94.34/mo)
performance-1x	1 performance	2GB	$0.0000120/s ($31.00/mo)
		4GB	$0.0000158/s ($41.01/mo)
		8GB	$0.0000235/s ($61.02/mo)
performance-2x	2 performance	4GB	$0.0000239/s ($62.00/mo)
		8GB	$0.0000355/s ($92.02/mo)
		16GB	$0.0000509/s ($132.04/mo)
performance-4x	4 performance	8GB	$0.0000478/s ($124.00/mo)
		16GB	$0.0000749/s ($194.04/mo)
		32GB	$0.0001057/s ($274.08/mo)
performance-8x	8 performance	16GB	$0.0000957/s ($248.00/mo)
		32GB	$0.0001536/s ($398.08/mo)
		64GB	$0.0002153/s ($558.16/mo)
performance-16x	16 performance	32GB	$0.0001914/s ($496.01/mo)
		64GB	$0.0003110/s ($806.16/mo)
		128GB	$0.0004345/s ($1126.33/mo)

Los servicios de Fly.io se facturan por organización. La facturación se basa en los recursos aprovisionados para las aplicaciones, prorrateados por el tiempo en que se aprovisionaron. Caso similar al de Heroku en la forma de facturar, sin embargo, por una máquina similar al *dyno standard-2x* en Fly se obtiene un precio de alrededor de 7,6 USD. Un elemento para destacar positivamente a favor de Fly.io es que permite alojar aplicaciones en servidores ubicados en Santiago (SCL código de servidor), lo que teóricamente permitiría reducir aún más la latencia y tiempos de respuesta. Lo anterior se contrasta con el caso de las otras dos plataformas (y la gran mayoría de las opciones del mercado) que tan solo ofrecen alojar aplicaciones en servidores estadounidenses, europeos o asiáticos.

Dado que resulta difícil comparar las tres plataformas por sus diferencias en el modo de facturación y su manera de ofrecer los planes y las distintas arquitecturas disponibles, se genera esta comparativa teórica a modo de establecer un plan equivalente para las tres plataformas, en Railway según su estimador de costos para una aplicación que utiliza dos unidades de procesado CPU dedicado y 4 GB de RAM sería un costo de $85 USD mensuales, como figura en la Ilustración 47.

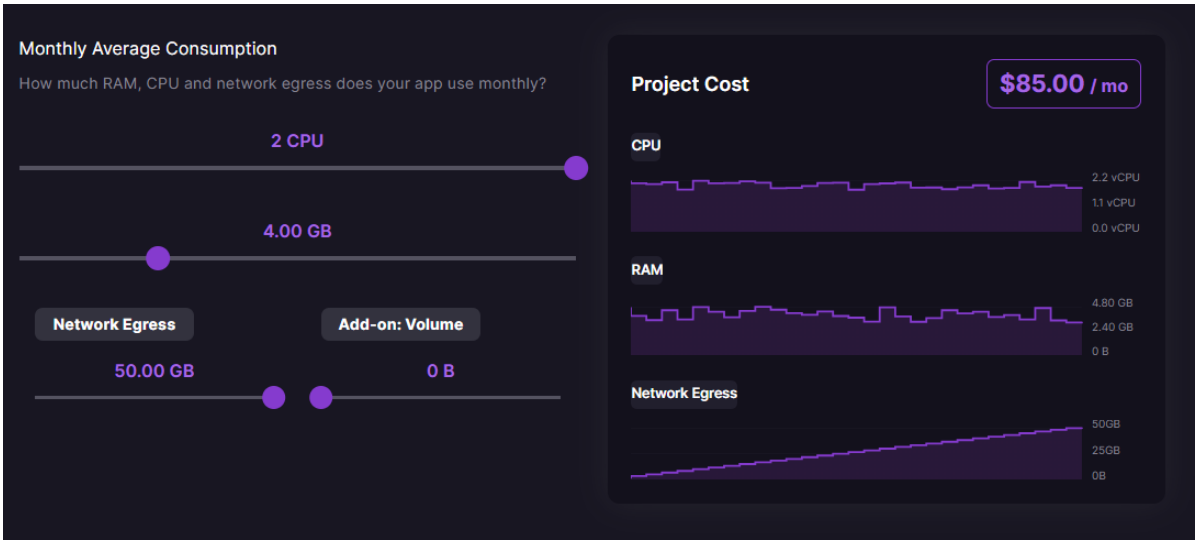


Ilustración 47 Estimación de precio para aplicación en Railway.
Fuente: Railway Pricing.

Dadas esas características correspondería seleccionar en Fly.io una máquina equivalente, la performance-2X con 4 GB de RAM que tiene un costo de $62 USD mensuales (Tabla 10), mientras que en Heroku una máquina performance-M sería el equivalente con un costo mensual máximo de $250 USD (ver Ilustración 48).

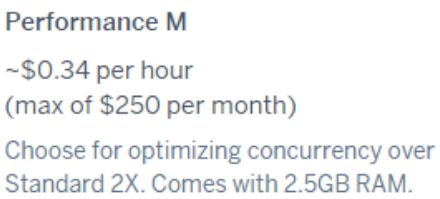


Ilustración 48 Detalles de pricing para Heroku en su plan Performance M.
Fuente: Heroku Pricing

Es evidente que en la comparativa teórica resulta mucho más económica la opción de Fly de tan solo $62 USD, sin embargo, en ningún caso se puede asegurar que la aplicación se comporte de la misma manera en las tres plataformas eligiendo planes y máquinas teóricamente similares, en primer lugar porque si bien pueden sonar como la misma

opción, no se sabe realmente las especificaciones técnicas al detalle de los contenedores que utilizan ni la arquitectura que emplean, estos son detalles técnicos que por lo general no comparten a los usuarios de forma sencilla, solo otorgan breves detalles para ofrecer una cierta variedad de opciones además de escalamiento horizontal.

Ahora bien, los resultados discutidos anteriormente ¿Con qué plan para cada plataforma se obtuvieron? Por los alcances y limitaciones de un proyecto como éste, no se contó con un presupuesto de nivel empresarial con el que podría funcionar VraVa en un futuro, si no que se optó por elegir la opción de entrada de cada plataforma con el objetivo de medir también cuál es la más idónea y conveniente para comenzar a desarrollar un proyecto como VraVa en un contexto como el de la FESW. De hecho, la plataforma elegida al momento del levantamiento y desarrollo del proyecto en FESW fue Railway con el plan básico.

A continuación, se listan los planes escogidos para desplegar las aplicaciones en cada plataforma.

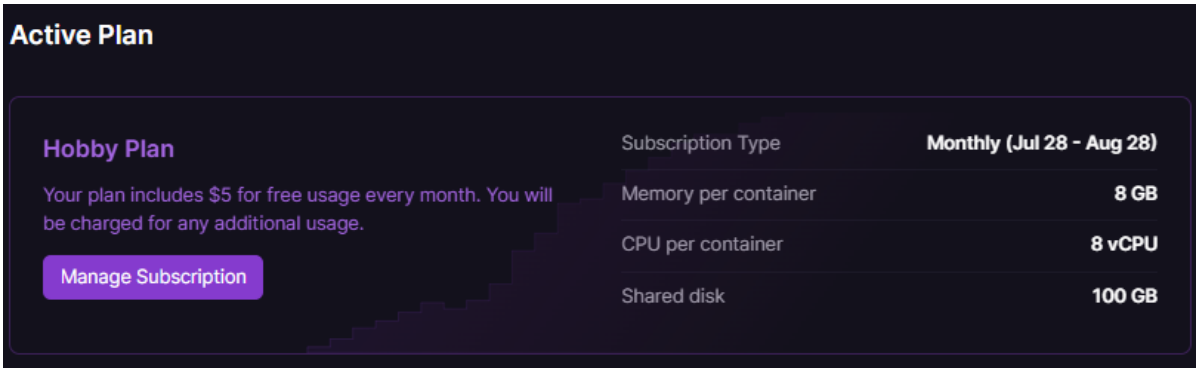


Ilustración 49 Plan utilizado en Railway. Fuente: Railway Personal Billing.

En Railway se usó el “Hobby Plan” que se describe en Ilustración 49, con 8 GB de memoria, 8 vCPU y 100 GB de espacio de disco compartido, este incluye todos los meses $5 USD de consumo gratuito y cobro a partir de ese punto con los valores mencionados en la Tabla 8.

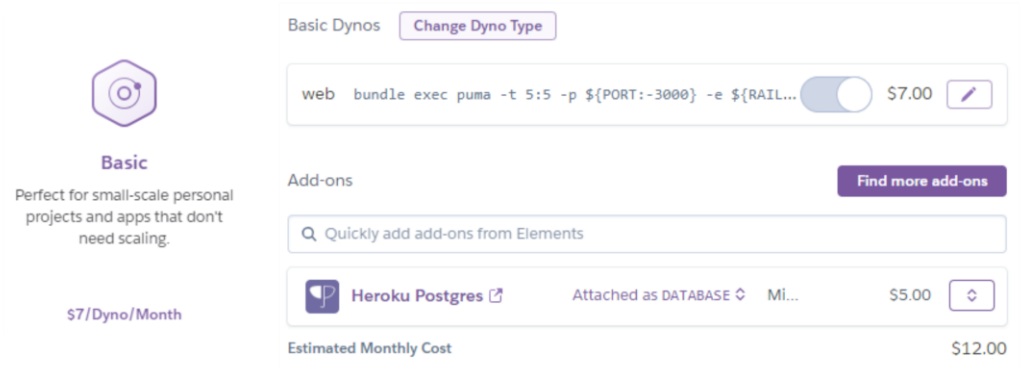


Ilustración 50 Items utilizados en Heroku. Fuente: Heroku Resources.

Con Heroku se hizo uso de un Dyno de tipo “Basic” (descrito en Ilustración 50) con valor fijo mensual de $7 USD más un *add-on* para levantar la base de datos PostgreSQL que conlleva un valor fijo mensual de $5 USD, ambos items sumando una tarifa total de $12 USD.

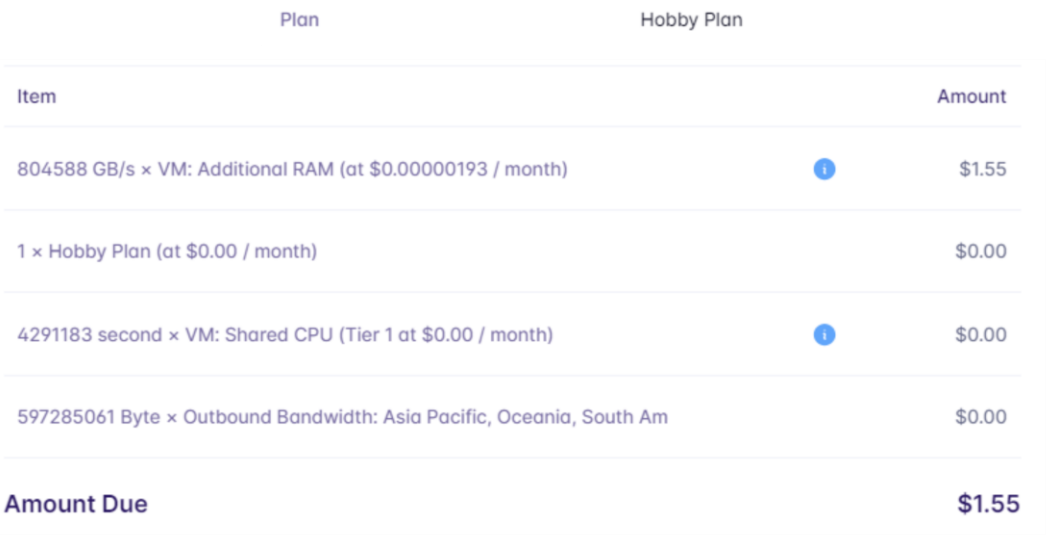


Ilustración 51 Items utilizados y facturados en Fly.io. Fuente: Fly dashboard.

Finalmente en Fly.io y similar a Railway, se hace uso del plan “Hobby Plan” (ver Ilustración 51) que cobra los montos detallados en la imagen de forma mensual dependiendo del uso, donde además ofrece $5 USD de consumo de manera gratuita mensualmente.

4.4 VALIDACIÓN

En el punto 4.2 se exponen los resultados obtenidos en las pruebas tal y como son, métricas, gráficos y comparativas de rendimiento, sin embargo, resulta necesario explicar y contextualizar algunas diferencias que explican en su gran medida lo obtenido.

Por parte del autor se considera que una API desarrollada en el framework Ruby on Rails posee una complejidad técnica y de diseño superior a una API implementada con un CMS como Strapi. Ruby on Rails, desde su concepción en 2004, ha sido diseñado como un marco de desarrollo completo y versátil. Esta versatilidad implica una gama más amplia de funcionalidades y herramientas, las cuales permiten a Rails adaptarse a una variedad más amplia de escenarios y necesidades de desarrollo, lo que, por su naturaleza, introduce una mayor complejidad técnica. Un principio fundamental de Rails es "Convención sobre Configuración" y aunque este enfoque minimiza la toma de decisiones arbitrarias en el desarrollo, también establece un conjunto de prácticas y estructuras predeterminadas que refuerzan la consistencia y la eficiencia del código. Esto, en términos técnicos, proporciona una base más robusta y estandarizada para la creación de aplicaciones.

Desde una perspectiva arquitectónica, Ruby on Rails emplea una estructura Modelo-Vista-Controlador (MVC). Esta arquitectura no sólo exige una separación clara de las responsabilidades del sistema, sino que también permite una modulación y una escalabilidad más efectivas. Además, la capacidad de Rails para ser altamente personalizable garantiza que se puedan crear soluciones más específicas y adaptadas a requisitos complejos. Strapi, al ser un CMS, tiene una orientación más hacia la gestión de contenido con configuraciones predefinidas, lo que podría limitar ciertas personalizaciones técnicas.

De forma más particular, en el caso de VraVa, existe una diferencia notable en el cómo se realizan ciertos procesos para cumplir con las funcionalidades cada una de las APIs. Respecto a la funcionalidad de crear una orden de trabajo o solicitud, hay un paso clave que corresponde a la subida y respaldo de las imágenes de la orden que en Strapi se aísla y se deja como una subida de un recurso independiente, esto es, desde la aplicación móvil se sube la imagen a Strapi y luego el URL generado se almacena en el recurso de la orden de trabajo, compartiendo la responsabilidad en el proceso y aumentando la posibilidad de fallas, mientras que en el caso de Rails al crear una nueva orden o solicitud las imágenes constituyen una entidad de ActiveRecord y son respaldadas automáticamente en el *bucket* S3. Esto por un lado reduce la cantidad de llamadas HTTP por cada proceso, pero por otro, complejiza y agranda el proceso completo de una creación de nueva orden. Este cambio sin duda es una decisión cuestionable y tiene ventajas y desventajas, el autor considera que incluso esta responsabilidad de almacenar y respaldar imágenes correspondería a un artefacto externo como lo es la inteligencia artificial que detecta los daños en las imágenes, artefacto que para efectos de este trabajo se consideró como una caja negra, sin embargo, esta solución comprendería un enfoque propio en este artefacto acerca de su diseño y arquitectura y una serie de decisiones y consideraciones que escapan del alcance y limitación del presente trabajo.

¿Por qué para Rails en Railway se obtienen peores resultados que Strapi y que Rails en Heroku? Es una interrogante que destaca porque por parte del autor no se esperaba que ocurriera algo así, sin embargo, tras una investigación en la comunidad activa de Railway y consultas al soporte, se llegó a la conclusión de que Railway está ciertamente optimizado y configurado para priorizar aplicaciones basadas en frameworks JavaScript como lo son Express o KOA (del cual se basa Strapi) ya que son los más frecuentes en la plataforma, mientras que Rails representa una minoría en cuanto a la cantidad de aplicaciones desplegadas en Railway. Esto puede explicar por qué se obtienen números desfavorables para Rails en Railway en la comparación con Strapi en Railway, pero a su vez obteniendo los mejores resultados por Rails en Heroku donde se utilizó un plan teóricamente con menos recursos y una arquitectura más básica.

Retomando el punto anterior relacionado con la funcionalidad de subida de imágenes, se propone una idea relacionada a la distribución de responsabilidades respecto a la arquitectura que se le dé al proyecto completo de VraVa.

Hoy en día y como lo fue para la FESW, VraVa funcionaba con tres artefactos ejecutándose de manera relacionada: la aplicación móvil (ReactNative), el *backend* Strapi y un servicio de IA ejecutándose en la nube cuya única funcionalidad es recibir una imagen como URL para luego analizar y detectar daños y, finalmente subir la imagen con las detecciones dibujadas y retornar su URL. Esta IA tiene formato y arquitectura de una API (FastAPI) que se ejecuta en una máquina con unidad de procesado gráfico dedicado y queda totalmente aislado su funcionamiento de cualquier componente de Strapi, comunicándose únicamente a través de métodos y llamadas HTTP. Es aquí donde el autor plantea una idea de mejora a futuro y una oportunidad de desarrollo que consta en hacer más robusta esta API-IA y asignarle la responsabilidad de la gestión de medios gráficos (todas las imágenes que se requieran para el negocio) así se simplificaría el trabajo de la API principal, ya sea Rails o Strapi, quedando estas únicamente encargadas de la lógica de negocio asociada a las órdenes de trabajo, talleres y clientes, cada entidad sin perder la relación con sus respectivas imágenes o medios gráficos, pero siendo esta nueva API-IA la encargada de gestionar estos datos (imágenes como tal, URL, relaciones con entidades). Existe un mundo de posibilidades para llevar a cabo esta idea incluso para testear si es mejor que la solución actual, el autor considera que luego de lo visto y detectado con la API Rails sumado a la experiencia adquirida en contexto laboral, es una gran opción para tener en cuenta en caso de profesionalización e impulso comercial de VraVa considerando como principal objetivo buscar reducir aún más el grado de unificación de las tareas y la arquitectura monolito que se pueda llegar a tener.

Una de las decisiones técnicas más difíciles de tomar y que más trabajo comparativo y analítico llevó fue la elección de la tecnología o biblioteca a utilizar para generar las “vistas” JSON que retornaría la API, un punto importante, ya que era crucial determinar la cantidad de información, la profundidad de las anidaciones, la facilidad de modificar y condicionar y siempre velando por el rendimiento de las respuestas. Para esto había dos principales opciones, por un lado, existe la alternativa de utilizar la serialización nativa de ActiveModel y por otro, utilizar la gema Jbuilder y su estructuración mediante parciales. Esta comparación entre JBuilder y ActiveModel::Serializers (AMS) en Rails es esencial en cualquier proyecto o aplicación que busque la mejor herramienta para estructurar y presentar datos. Ambas herramientas poseen distintas filosofías y características técnicas que influyen en su elección.

JBuilder se destaca principalmente por su naturaleza específica para Rails, permite la construcción de respuestas JSON con gran precisión otorgando un control minucioso sobre la estructura final del JSON. Además, JBuilder aprovecha eficientemente el sistema de vistas de Rails, utilizando “partials” para reutilizar fragmentos comunes de JSON, siguiendo el paradigma tradicional de Rails para el HTML. Esta integración íntima con Rails facilita su uso y adopción en proyectos basados en este marco de trabajo.

Por otro lado, AMS adopta un enfoque más orientado a objetos, ya que utiliza serializadores para determinar la representación de los objetos, brindando una estructura clara, especialmente en aplicaciones de gran envergadura. Sin embargo, AMS posee convenciones más rígidas en cuanto a la estructuración del JSON, lo que, si bien puede ser beneficioso en términos de consistencia, puede limitar su flexibilidad en comparación con JBuilder. Es importante señalar que el rendimiento de AMS ha sido objeto de debate, y en ciertos contextos, ha mostrado ser menos eficiente que JBuilder, en especial al serializar objetos de gran complejidad.

Al analizar ambas herramientas en términos de facilidad de uso, mantenimiento y rendimiento, JBuilder parece llevar la delantera sobre todo por su capacidad de adaptarse con precisión a las necesidades del desarrollador y su integración con Rails, por ende, se presenta como una herramienta más versátil. Esta versatilidad no solo beneficia a proyectos en sus etapas iniciales (como lo sería la etapa actual de VraVa), sino que asegura un mantenimiento más sencillo y adaptativo a largo plazo, por lo tanto, para proyectos basados en Rails que buscan flexibilidad, eficiencia y un balance entre personalización y rendimiento, se determinó que JBuilder sería la opción recomendada.

4.4.1 VALIDACIÓN DE OBJETIVOS

En este apartado se presentará una perspectiva de lo logrado y resultado en relación con los objetivos propuestos al inicio de este proyecto de forma directa y a modo de resumen de lo conseguido y aprendido.

4.4.1.1 OBJETIVO GENERAL

"Rediseñar el backend del software VraVa, de tal forma de optimizar el rendimiento actual y buscando disminuir los tiempos de respuesta, recursos utilizados y costos asociados al uso y mantenimiento de la aplicación."

Se da este objetivo por cumplido porque se rediseñó el *backend* de VraVa resultando en una nueva API desarrollada en Ruby on Rails con una arquitectura y diseño completamente nuevo, donde como se pudo ver en el punto 4.3 se obtienen los mejores tiempos de respuesta y en 4.2.3.2 donde se observa que se obtiene un Apdex Score superior en la API desarrollada y desplegada en Heroku. En cuanto a los recursos resulta difícil considerar una optimización clara ya que, si bien los gráficos reportan menos uso de RAM y CPU para la API nueva, las plataformas que no entregan un detallado extenso de la arquitectura y componentes utilizados, por lo que no se puede concluir explícitamente que consume menos recursos físicos. Donde sí hubo una baja tangible en uso de recursos fue en el contexto de consultas a la base de datos, siendo reducidas respecto a lo que se obtuvo con el *backend* Strapi. En cuanto a costos nuevamente es necesario más detalle, ya que por sí sola la API nueva consume menos recursos físicos y de base de datos, pero como se vió en 4.3, los costos asociados a la aplicación dependen mucho de la elección de la plataforma a desplegar. Finalmente, en cuanto al mantenimiento se considera una mejora considerable, generando pruebas para guiar el desarrollo y una documentación actualizada y mantenible.

4.4.1.2 OBJETIVOS ESPECIFICOS

"Medir la cantidad de consultas a la API REST de los flujos de información de las funcionalidades importantes para definir y probar una estrategia de optimización del flujo lógico que permita disminuir la cantidad de llamadas a la API, mejorando el uso y mantenimiento de la aplicación."

Como se pudo ver en 3.4 Optimización de Flujos, se desarrollaron nuevos flujos para la información entre la aplicación móvil, la API y el bucket S3 resultando en una optimización y disminución en la cantidad de consultas a la API que incluso se vió reflejado en una baja en la cantidad de consultas a la base de datos. Esto además genera por consecuencia una simplificación en los flujos y, por ende, se mejora el mantenimiento de la aplicación.

"Desarrollar una API en Ruby on Rails con la metodología Test Driven Development (TDD) para mejorar la calidad del código y su mantenimiento."

Por lo visto en 4.1.2 se desarrolló una API completamente desde cero y guiada íntegramente por pruebas unitarias de cada funcionalidad. Esto provoca que en un futuro para nuevas funcionalidades las pruebas sean documentación para desarrollar bajo el mismo esquema TDD, generando mantenibilidad y fácil acceso a nuevos desarrolladores para agregar o modificar funcionalidades. Nuevamente, por lo visto en 4.2.1 se alcanza un porcentaje de cobertura de código de un 99,44%, lo que da cuenta de que casi en su totalidad el código está cubierto por pruebas y cumple con lo estipulado en ellas.

"Evaluar opciones de tecnologías para levantar la instancia del backend comparando sus costos, de tal manera de encontrar la alternativa más rentable para el desarrollador."

El tercer objetivo específico se declara cumplido dado que se presentó en 4.3 una comparativa y evaluación acerca de los servicios disponibles y utilizados en este proyecto, disponibilizando los datos necesarios para en un futuro comercial determinar qué plataforma podría ser la mejor opción para desplegar la aplicación.

5 CAPÍTULO V: CONCLUSIONES

Como resultado del presente proyecto se obtiene una aplicación de tipo API desarrollada en el *framework* Ruby on Rails que es producto de un rediseño aplicado a una aplicación previamente desarrollada para fines del proyecto VraVa perteneciente a la FESW, esta última aplicación desarrollada con el CMS Strapi, se utilizó como *backend* para gestionar la data e información necesaria y entregarla a la aplicación móvil que estaba de cara al usuario final. Esta aplicación inicial cumplía un rol de solución *backend* en el sistema completo de VraVa, donde existe la antes mencionada aplicación móvil además de un artefacto de Inteligencia Artificial de tipo red neuronal que se encarga de analizar imágenes de vehículos y detectar los daños presentes, estos tres artefactos funcionaban en conjunto para brindarle a los usuarios las funcionalidades definidas en el proceso de diseño de la aplicación.

La nueva aplicación, denominada por el autor a lo largo de este informe como Rails API, cumple con todas las funcionalidades dispuestas en la etapa de diseño que además ya cumplía el backend a rediseñar, es completamente funcional y bastarían ciertas modificaciones en los artefactos restantes para acoplar y migrar de una aplicación a otra. Además de cumplir con las mismas funcionalidades posee una considerable documentación actualizada, un set completo de pruebas unitarias por cada método de cada controlador, por cada modelo y por cada solicitud HTTP existente en la API, que son resultado de la metodología utilizada en el proceso de desarrollo, Test Driven Development. Por otro lado, la aplicación Rails API considera ciertas mejoras en aspectos lógicos y de negocio que fueron diseñadas en la búsqueda de optimizar recursos y métricas como tiempo de respuesta, número consultas a base de datos por cada petición, uso de CPU, entre otras. Lo anterior nace en la búsqueda de una solución a un dolor detectado en etapas de desarrollo de la aplicación Strapi por parte de los mismos desarrolladores que fueron parte del equipo MalayApps (equipo creador de VraVa), un dolor asociado a la difícil tarea de mantenimiento de la aplicación Strapi provocada por el desarrollo inexperto y a contrarreloj que sufrió la misma aplicación, lo cual es importante destacar ya que en ningún momento se considera por parte del autor que Strapi sea una tecnología insuficiente o carente de nivel técnico, sino que el proceso de desarrollo que dio a luz al *backend* Strapi fue a prueba y error, con el tiempo justo y sin tener en mente la escalabilidad a futuro.

Una vez VraVa finaliza su etapa como producto participante en la FESW y es galardonado como el producto más innovador de su categoría, es cuando el autor considera la oportunidad de generar un reemplazo para esta aplicación *backend* buscando optimizar lo

anteriormente mencionado y generar un artefacto más robusto, sólido y escalable en un futuro comercial, apuntando a utilizar tecnologías ampliamente reconocidas por la industria, como Ruby on Rails, y desarrollando bajo el esquema TDD, sentando las bases para un artefacto futuro que escale bien y soporte una puesta en marcha comercial.

En cuanto a las limitaciones presentes en el proyecto se tiene el hecho de que se hace necesario un proceso de migración y adaptación tanto de la aplicación móvil como el componente IA, donde se deben ver sometidos a una modificación y posible reestructuración debido a los cambios en algunas reglas de negocio generados por esta solución como lo es, por ejemplo, la transición de Orden de trabajo y Solicitud como dos entidades distintas a una sola entidad denominada Orden, cambiando consigo una serie de relaciones y lógicas que se deben ver reflejadas en la aplicación móvil y componente IA. Si bien la mayoría son invisibles al usuario final, estas modificaciones tienen impacto en los demás componentes. Otra limitación existente, aunque un poco más técnica, es el uso de ActiveStorage como motor principal para la gestión de imágenes, si bien esto no es un problema ni una limitación como tal, sería deseable a futuro implementar una solución como la propuesta en el apartado de validación, donde la responsabilidad de los recursos gráficos quede totalmente cubierta por el componente IA, quien por principio de responsabilidad única es el que debería encargarse de la gestión de imágenes ya que por definición es quien trabaja sobre las mismas. ¿Por qué no se integró esta solución a la solución propuesta en el presente trabajo? En primer lugar, se buscaba suplir al completo las funcionalidades con las que cuenta hoy en día el backend Strapi dado que el objetivo principal habla sobre un rediseño de este, el cual contempla únicamente los límites actuales del mismo backend, es por esto por lo que, en segundo lugar, el autor considera que el modificar o agregar nuevas funcionalidades a otros componentes de VraVa se escapa del alcance del proyecto.

Otra limitación que se considera importante para VraVa en sí, pero externa a lo que puede ser el *backend* o API, es la red neuronal que compone el núcleo del componente IA. Esta es una red neuronal de tipo FastCNN que fue entrenada utilizando *transfer learning* y apoyado de un banco muy limitado de imágenes dada la gran dificultad de obtención de estas y el etiquetado de los daños imagen por imagen. Si bien la estructura y arquitectura de la red neuronal es considerablemente robusta y está ajustada tras varias sesiones de pruebas, resulta muy difícil obtener resultados aceptables en términos de aciertos en la detección sin contar con un banco robusto de imágenes por cada tipo de daño categorizado.

Por último, como limitación particular de este proyecto se considera el límite presupuestario relacionado a los costos de despliegue para las pruebas realizadas, si bien

se eligieron los planes básicos para cada plataforma, estos no son iguales en términos de arquitectura llevando a una comparativa un tanto dispareja para las plataformas debida a esta diferencia técnica, por otro lado, existen plataformas más profesionales como Azure o Google Cloud Computing que quedan fuera del abanico de opciones por esta misma razón presupuestaria. Sin embargo, frente a este último punto es importante destacar que Railway, que fue la plataforma elegida para desplegar el *backend* para la FESW, no es competencia directa de estas plataformas más profesionales por sus propias limitaciones y alcances, además de la diferencia sustancial en términos de complejidad de despliegue. Según el autor, se puede decir que apuntan a mercados objetivos distintos y que son una gran alternativa para proyectos emergentes o contextos como el de la Feria de Software.

Una vez desarrollada la Rails API utilizando TDD, generando las propuestas de optimización y desplegada en las distintas plataformas propuestas, se realizó la comparación de los resultados obtenidos luego de someter a pruebas de carga los artefactos desplegados (Strapi backend en Railway y Rails API en Railway, Heroku y Fly.io). Cabe destacar que estas pruebas de carga fueron diseñadas con Locust, una herramienta que simula el comportamiento de usuarios interactuando con las aplicaciones a través de consultas HTTP para cada aplicación, Strapi backend y Rails API. Estas pruebas fueron ejecutadas con parámetros de carga como tiempo total, cantidad de usuarios y tiempo entre cada usuario estimadas a partir de la experiencia propia del autor en base a lo vivido en FESW y pruebas aisladas para determinar estas variables. Además, se propuso levantar dos comparativas distintas, por un lado, comparar Strapi backend con Rails API en la misma plataforma (Railway, la que fue elegida para el contexto FESW) y por otro, comparar el desempeño de Rails API en tres distintas plataformas para obtener noción de cuál sería la mejor opción para desplegar esta aplicación.

Respecto a la primera comparativa los resultados fueron exitosos, pero con ciertos matices, si bien Rails API disminuyó el uso de recursos utilizados, los tiempos de respuesta fueron incluso mayores para el caso de la media y los percentiles 95 y 99. Esto fue provocado principalmente por peticiones *outliers* asociadas al *endpoint* del listado completo de órdenes y al de autenticación que empeoraron estas métricas, principalmente porque el *endpoint* del listado de órdenes de la API Rails tiene una respuesta más completa en información y con una profundidad mayor en cuanto a su anidación de relaciones, lo cual es una característica totalmente configurable y sujeta a una decisión técnica asociada a cuánta información se necesita responder en ese listado, por otra parte, el *endpoint* de autenticación es el encargado de autenticar y generar un *token* de sesión para el usuario y es ahí donde se genera un tiempo adicional dada la cantidad de chequeos y verificaciones que hace el controlador, dejándola como una de las operaciones más lentas de la API. Si bien lo anterior sin duda es un punto por mejorar, es algo que no es tan alarmante debido

a que las peticiones login se hacen únicamente una vez por sesión de cada usuario y que existan en ocasiones algunos outliers respecto al tiempo de respuesta, es algo que no debería impactar de sobremanera en la experiencia de este. A pesar de lo anterior, se consideran exitosos los resultados ya que en general se obtienen mejores métricas y sobre todo considerando que el Apdex Score fue mayor y más estable durante las pruebas ejecutadas, dando cuenta de una satisfacción mayor en cuanto a los resultados de las transacciones.

Considerando la segunda comparativa propuesta, se obtuvieron resultados muy positivos en cuanto a los números obtenidos en las métricas comparadas. Se determina que la mejor opción para desplegar la Rails API es Heroku, donde se obtienen los mejores tiempos de respuesta, con menos variaciones (lo que indica que los tiempos son más estables) y con un porcentaje de satisfacción superior al del resto de opciones. Incluso, si se llevara el artefacto Rails API desplegado en Heroku a la primera comparativa, saldría como la mejor en todas las métricas consideradas para la comparación, superando a su homóloga en Railway y al Strapi backend actual de VraVa. Si bien no es la opción más económica en cuanto a costos de mantención, resulta sorprendente que incluso en condiciones de arquitectura teóricamente inferiores, la Rails API en Heroku es la que obtiene las mejores métricas. Esto se puede atribuir, según el autor, a que Heroku cuenta con una trayectoria como PaaS mucho más longeva, una comunidad más grande y un soporte más robusto, siendo estas características notorias en la experiencia de uso de estas plataformas de cara al desarrollador.

Como resultado final de este trabajo, se obtiene una API completamente funcional, robusta en términos de calidad de código, documentada y con un conjunto de pruebas unitarias que conforman la metodología TDD y que cubren casi la totalidad del código fuente. En adición a esto, se obtiene una oportunidad de despliegue con mejor desempeño, menor consumo y con mayor oportunidad de escalar en un futuro respecto a con lo que hoy en día cuenta VraVa.

Este trabajo aporta significativamente al área de la optimización de software al proponer soluciones específicas para un caso real, donde se llevó a cabo identificación de problemas y la propuesta de soluciones basadas en tecnologías y metodologías modernas, lo que puede servir como referencia para otros proyectos similares. Además, el software VraVa ofrece una solución digital para la cotización y reparación de vehículos, optimizar su arquitectura backend no solo mejorará la experiencia del usuario, sino que también fortalecerá la posición de VraVa en el mercado, beneficiando a todos los actores involucrados. Este trabajo ha proporcionado una visión detallada de los desafíos y oportunidades asociados con el software VraVa, y las conclusiones y propuestas

presentadas tienen el potencial de llevar a VraVa a un futuro comercial donde pueda entrar a competir en el mercado con una característica diferenciadora, la inteligencia artificial.

Como trabajo a futuro se propone una serie de ideas para impulsar VraVa a un producto comercial rentable y con oportunidades de surgir en el mercado, entre ellas se encuentra un remodelado completo de los artefactos involucrados en la aplicación completa, vale decir, remodelar el componente IA para adaptarlo a los esquemas de datos propuestos por Rails API y que contemple la responsabilidad completa de los recursos de imágenes y además, reformular la aplicación móvil para adaptarla a las mejoras propuestas en este trabajo. Cada uno de los artefactos desarrollados en contexto FESW tienen una gran oportunidad de mejora dadas las condiciones mencionadas anteriormente, el autor considera que llevar al siguiente nivel cada uno de estos artefactos teniendo en cuenta la escalabilidad y la optimización de recursos es el camino para convertir a VraVa en un producto competitivo en el mercado.

Como se mencionó anteriormente, la red neuronal que comprende el elemento diferenciador de VraVa en el mercado debe ser mucho más robusta y precisa en sus detecciones, para lo cual se hace necesario en un futuro potenciar el entrenamiento de la red neuronal a través de diversificar y aumentar el volumen de datos de entrenamiento.

Otro punto con potencial trabajo futuro es el de generar una vista web para la administración interna de los talleres, si bien la aplicación móvil cumple como gestora de las ordenes de trabajo pertenecientes a cada taller, sería un elemento clave el tener una vista más cómoda y completa del estado actual del taller desde un tamaño web o de escritorio, con lo cual el administrador del taller podría gestionar de manera más fácil y con una interfaz diseñada especialmente para dispositivos más grandes.

Es a raíz de lo desarrollado en el presente proyecto que el autor genera un aprendizaje considerable en relación a la importancia de los esquemas de desarrollo, donde pasa de generar una solución en un contexto universitario para un evento como FESW a proponer una solución más sólida y robusta pensada en la escalabilidad a la que se debe ver sometido VraVa para un futuro comercial, donde la evaluación, análisis y uso de tecnologías como Ruby on Rails, New Relic y Locust y prácticas como TDD generan un impacto positivo en los resultados y en el camino recorrido para llegar a ellos. Bajo la opinión del autor es muy distinto desarrollar sin un horizonte claro y sin la especificación técnica de qué se debe desarrollar a utilizar prácticas como TDD que guían el desarrollo a través de pruebas concretas, medibles y que funcionan como herramienta de validación al componente o funcionalidad desarrollada.

En conclusión, el proceso de reestructuración y mejora del núcleo técnico o *backend* de VraVa refleja la esencia de la evolución y la búsqueda de la perfección en el ámbito del software. Este proyecto evidencia que, al seleccionar las herramientas y estrategias correctas, se puede convertir un sistema con desafíos en una solución preparada para enfrentar las demandas futuras. VraVa, con su enfoque vanguardista en la detección de daños vehiculares a través de la IA, sueña como producto con marcar un antes y un después en la industria. Este esfuerzo no solo ha consolidado su estructura técnica, sino que también ha preparado el terreno para que VraVa se destaque como una referencia en su sector, subrayando que con dedicación y una visión clara, se pueden superar obstáculos y trazar nuevos caminos en el mundo digital. El autor sostiene que ideas disruptivas y soluciones integrales como VraVa son el futuro de la industria de software enfocado a atender problemáticas tanto de las empresas como de los clientes consumidores, buscando ser indispensables para su labor diaria y simplificándoles procesos que hoy en día presentan carencias y dolores a través de tecnología y sistemas que velen por la disponibilidad, escalabilidad y la experiencia del usuario.

6 REFERENCIAS

Abdelhamid, A. (30 de Julio de 2021). *Active Record (Rails Model) - Introduction.* Obtenido de DEV Community: https://dev.to/ahmedhamid13/active-record-introduction-37hb

Acosta, J., & Gajda, K. (s.f.). *Test-driven development*. Obtenido de IBM Garage Methodology: https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/

Amazon Web Services. (s.f.). *¿Qué es una API de RESTful? - Explicación de API de RESTful - AWS*. Obtenido de https://aws.amazon.com/es/what-is/restful-api/

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Obtenido de https://www.martinfowler.com/eaaCatalog/singleTableInheritance.html

Gravelle, R. (17 de Septiembre de 2021). *Overview of RDBMS Index Types*. Obtenido de Navicat Blog: https://www.navicat.com/en/company/aboutus/blog/1782-overview-of-rdbms-index-types-2

IBM Corporation. (2021). *IBM Documentation*. Obtenido de Diseño de base de datos con desnormalización: https://www.ibm.com/docs/es/db2-for-zos/12?topic=design-database-denormalization

Kung, H.-J., Kung, L., & Gardiner, A. (Febrero de 2013). Comparing Top-down with Bottom-up Approaches: Teaching Data Modeling. *Information Systems Education Journal*, págs. 15-24.

Lee, H., von Laszewski, G., & Wang, F. (2014). *Towards Understanding Cloud Usage through Resource Allocation Analysis on XSEDE.* Community Grids Lab Publications. Obtenido de http://dsc.soic.indiana.edu/publications/xsede14_submission_166.pdf

Martin, R. C. (6 de Octubre de 2005). *The Three Laws of TDD.* Obtenido de http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

Matsumoto, Y. (12 de Mayo de 2000). Re: More code browsing questions.

Miller, S. (2021 de Septiembre de 15). *What Is Ruby on Rails?* Obtenido de Codecademy Blog: https://www.codecademy.com/resources/blog/what-is-ruby-on-

rails/#:~:text=Ruby%20on%20Rails%20(or%20%E2%80%9CRails,very%20complex%20with%20many%20layers.

Rails. (s.f.). *Active Record Basics*. Obtenido de Ruby on Rails Guides: https://guides.rubyonrails.org/active_record_basics.html

Redhat. (Enero de 2023). *¿Qué es una API y cómo funciona?* Obtenido de https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

Ruby. (s.f.). *Acerca de Ruby*. Obtenido de https://www.ruby-lang.org/es/about/

Ruby. (s.f.). *Lenguaje de Programación Ruby*. Obtenido de https://www.ruby-lang.org/es/

Ruby on Rails. (s.f.). *Active Storage Overview.* Obtenido de Rails Guides: https://guides.rubyonrails.org/active_storage_overview.html

Saldana, J. (s.f.). *Refactoring the three laws of TDD.* Obtenido de On Programming Well: https://www.javiersaldana.com/articles/tech/refactoring-the-three-laws-of-tdd

Sentry. (s.f.). *N+1 Queries*. Obtenido de Sentry Documentation: https://docs.sentry.io/product/issues/issue-details/performance-issues/n-one-queries/?original_referrer=https%3A%2F%2Fwww.google.com%2F

Strapi Inc. (2021). *Strapi Documentation, APIs Reference*. Obtenido de REST API Filters: https://docs.strapi.io/developer-docs/latest/developer-resources/database-apis-reference/rest-api.html#filters

Strapi Inc. (2021). *Strapi Documentation, Backend Customization*. Obtenido de Controllers: https://docs.strapi.io/developer-docs/latest/development/backend-customization/controllers.html#implementation

7 ANEXOS

Anexo 1 Dockerfile

```
FROM ruby:3.1.3

RUN useradd developer

RUN mkdir -p /home/developer

WORKDIR /home

RUN chown -R developer:developer developer

RUN chmod 755 developer

USER developer

WORKDIR /tmp

RUN mkdir -p /tmp/cache

RUN mkdir -p /home/developer/app
WORKDIR /home/developer/app

COPY --chown=developer:developer ./ .

RUN bundle install

EXPOSE 3000

ENTRYPOINT [ "./entrypoints/docker-entrypoint.sh" ]
```

Anexo 2 Archivo docker-compose.yml

```
version: "3"
volumes:
  postgresql:
  bundle_path:
services:
  db_vrava:
    restart: on-failure:3
    image: postgres:14.6
    volumes:
      - postgresql:/var/lib/postgresql/data
    ports:
      - 5432:5432
    env_file:
      - .env
  vrava_api:
    restart: on-failure:3
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 3000:3000
    volumes:
      - ./:/home/developer/app
      - bundle_path:/usr/local/bundle
    env_file:
      - .env
```

Anexo 3 Archivo pruebas locust para Strapi completo.

```python
class ClientTasks(TaskSet):

    def on_start(self):
        self.client.headers["Authorization"] = "Bearer " + token

    @task(25)
    def getAllOrders(self):
        response = self.client.get("/orden-trabajos")
        if(response.status_code != 200):
            response.failure("fetching all orders failed")

    @task(25)
    def getOrders(self):
        response = self.client.get("/orden-trabajos?estado=in_progress")
        if(response.status_code != 200):
            response.failure("fetching all orders failed")

    @task(25)
    def getOrder(self):
        response = self.client.get(f"/orden-trabajos/{order_id!r}")
        if(response.status_code != 200):
            response.failure("fetching orders failed")

    @task(25)
    def getRequests(self):
        response = self.client.get("/solicitudes")
        if(response.status_code != 200):
            response.failure("fetching requests failed")

    @task(25)
    def getWorkshops(self):
        response = self.client.get("/tallers")
        if(response.status_code != 200):
            response.failure("fetching workshops failed")

    @task(25)
    def getWorkshop(self):
        response = self.client.get(f"/tallers/{workshop_id!r}")
        if(response.status_code != 200):
            response.failure("fetching workshops failed")

    @task(25)
    def getCars(self):
        response = self.client.get("/autos")
        if(response.status_code != 200):
            response.failure("fetching cars failed")

    @task(25)
    def getCar(self):
        response = self.client.get(f"/autos/{car_id!r}")
        if(response.status_code != 200):
            response.failure("fetching car failed")

    @task(5)
    def createCarAndOrder(self):
        car = generate_car(strapi=True)
        response = self.client.post(
            "/autos",
            json=car
        )
        if(response.status_code not in [200, 201]):
            response.failure("creating car failed")
        else:
            self.new_car_id = response.json()["id"]
            order = generate_order(
                user_id=self.user_id,
                car_id=self.new_car_id,
                workshop_id=workshop_id,
                strapi=True,
                status="in_progress")
            response = self.client.post(
                "/ordern-trabajos",
                json=order
            )
            if(response.status_code != 201):
                response.failure("creating order failed")
```

Anexo 4 Archivo pruebas locust para API Rails completo.

```python
class ClientTasks(TaskSet):
    def login(self):
        response = self.client.post(
            "/v1/users/login",
            json=client_login_payload
        )
        user_id = response.json()["id"]
        token = response.json()["token"]["token"]
        return token, user_id

    def on_start(self):
        token, self.user_id = self.login()
        self.client.headers["Content-Type"] = "application/json"
        self.client.headers["Authorization"] = "Bearer " + token

    @task(25)
    def getAllOrders(self):
        response = self.client.get(f"/v1/users/{self.user_id!r}/orders?status=all")
        if(response.status_code != 200):
            response.failure("fetching all orders failed")

    @task(25)
    def getOrders(self):
        response = self.client.get(f"/v1/users/{self.user_id!r}/orders")
        if(response.status_code != 200):
            response.failure("fetching orders failed")

    @task(25)
    def getOrder(self):
        response = self.client.get(f"/v1/orders/{order_id!r}")
        if(response.status_code != 200):
            response.failure("fetching orders failed")

    @task(25)
    def getRequests(self):
        response = self.client.get(f"/v1/users/{self.user_id!r}/orders?status=in_progress")
        if(response.status_code != 200):
            response.failure("fetching requests failed")

    @task
    def getWorkshops(self):
        response = self.client.get("/v1/workshops")
        if(response.status_code != 200):
            response.failure("fetching workshops failed")

    @task(25)
    def getWorkshop(self):
        response = self.client.get(f"/v1/workshops/{workshop_id!r}")
        if(response.status_code != 200):
            response.failure("fetching workshops failed")

    @task(25)
    def getCars(self):
        response = self.client.get("/v1/cars")
        if(response.status_code != 200):
            response.failure("fetching cars failed")

    @task(25)
    def getCar(self):
        response = self.client.get(f"/v1/cars/{car_id!r}")
        if(response.status_code != 200):
            response.failure("fetching car failed")

    @task(5)
    def createCarAndOrder(self):
        car = generate_car()
        response = self.client.post(
            "/v1/cars",
            json={"car": car}
        )
        if(response.status_code not in [200, 201]):
            response.failure("creating car failed")
        else:
            self.new_car_id = response.json()["id"]
            order = generate_order(
                user_id=self.user_id,
                car_id=self.new_car_id,
                workshop_id=workshop_id,
                status="in_progress")
            response = self.client.post(
                f"/v1/users/{self.user_id!r}/orders",
                json=order
            )
            if(response.status_code != 201):
                response.failure("creating order failed")

    @task(5)
    def changeOrderStatus(self):
        order = { "status": "delivered" }
        response = self.client.put(
            f"/v1/orders/{order_id!r}",
            json=order
        )
        if(response.status_code not in [200, 201]):
            response.failure("change order status failed")
```

Anexo 5 Archivo jbuilder de Order

```
json.extract! order, :id, :damages, :jobs, :status, :delivery_date, :price, :created_at
json.client do
  json.partial! 'v1/users/user', user: order.client
end
json.workshop do
  json.partial! 'v1/workshops/workshop', workshop: order.workshop
end
json.car do
  json.partial! 'v1/cars/car', car: order.car
end
if order.images
  json.images do
    json.array! order.images do |image|
      json.url image.url
    end
  end
end
```