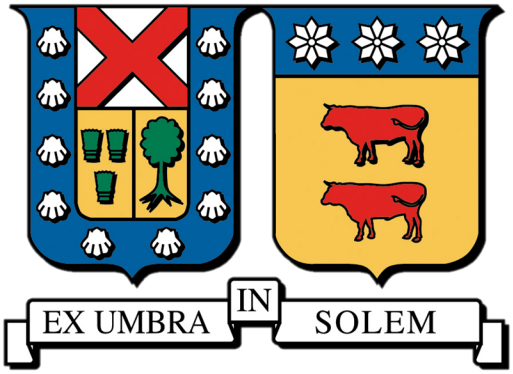
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



DISEÑO DE UN BACKEND ESCALABLE DE RECOLECIÓN y ANÁLISIS DE DATOS GEOREFERENCIADOS OBTENIDOS VÍA CROWDSOURCING

JORGE ANDRÉS COCIÑA PACHECO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

PROFESOR GUÍA : CLAUDIA ANDREA LÓPEZ MONCADA
PROFESOR CORREFERENTE : CECILIA REYES COVARRUBIAS

Diciembre 2017

Agradecimientos

Quiero agradecer profundamente a todos aquellos que me acompañaron y guiaron durante mi periodo universitario, ya que sin ellos, este trabajo no sería posible.

Glorioso Dios, se que siempre me he alejado de ti, pero no me abandonaste y me diste la fuerza para seguir hasta el final.

Amada Bethsabé, tu amor, tu apoyo y tu comprensión me mantuvieron a flote siempre, tus besos me infundieron energía. Me levantaste cuando sentía que no podía más, me diste todo para seguir adelante y estuviste en todo momento. Siempre estaré agradecido por todo lo que haces por mí y recuerda que, cuando sea grande, quiero ser como tú. Te amo hoy más que ayer, y menos que mañana.

Queridos Mamá y Papá, sé que muchas veces parezco un ingrato, pero la verdad les agradezco el apoyo y la confianza que me han otorgado. Los amo a ambos, aunque a veces no nos entendamos.

Estimados profesores, gracias por buscar que siempre me esforzara al máximo, y gracias especialmente a la profesora Claudia López por guiarme durante este largo trabajo y no eliminarme de su pizarra.

A ti querido lector, gracias por darte el tiempo de leer.

"El sabio puede cambiar de opinion, el necio nunca." -Immanuel Kant

RESUMEN EJECUTIVO

Esta memoria documenta el análisis, diseño y desarrollo del backend de un software que busca visibilizar los servicios públicos en la ciudad de Valparaíso. La solución consistió en una API RESTful, que soporta un mecanismo de crowdsourcing, que permite recopilar información de los servicios higiénicos de la ciudad y además, registrar los movimientos de los usuarios, para posteriormente poder realizar análisis de su comportamiento. Esta solución fue construida utilizando el lenguaje de programación PHP y el framework Phalcon.

El trabajo comienza con la selección de las mejores opciones de desarrollo, lo que permite tomar las decisiones sobre la arquitectura de software, lenguaje, y framework para la construcción de la solución. La toma de decisiones se centró en que la aplicación cumpliera con los requerimientos no funcionales de escalabilidad de carga, extensibilidad y reusabilidad. Posterior a esto se realizó un proceso de validación del cumplimiento de estos requerimientos.

Palabras Claves: RESTful, crowdsourcing, PHP, Arquitectura de Software.

ABSTRACT

This report documents the analysis, design, and development of a software backend that seeks to make Valparaíso's public services more visible through crowdsourcing data about them. The solution consists of a RESTful API, which supports the crowdsourcing mechanism and allows tracking user actions on the web interface.

The document reports on the selection of the development tools, which included making decisions about the software architecture, programming language, and framework. The decision-making process focused on satisfying three non-functional requirements: load scalability, extensibility, and reusability. As a result, the back-end was built using PHP and the Phalcon framework. Finally, validation of compliance with the requirements was carried out.

Keywords: RESTful, crowdsourcing, PHP, Software Architecture.

Índice de Contenidos

Índice de Tablas

Índice de Figuras

1 | Introducción

Valparaíso, conocida por sus bellos parajes, calles de casas multicolores y cerros llenos de vida, hoy se enfrenta a los desafíos de una gran ciudad que también es patrimonio de la humanidad. Es un gran centro turístico, por lo cual es fundamental darle visibilidad a servicios públicos de forma sencilla para quienes no pertenecen a la ciudad puerto, y para sus propios habitantes que muchas veces desconocen los servicios disponibles en su amada ciudad. Por ejemplo si nos encontramos en Valparaíso, y tenemos la necesidad de un baño, ¿Dónde podemos encontrar uno público y cerca? Parece información trivial, pero no siempre es fácil de encontrar. Por esto es necesario algún medio por el cual se pueda informar sobre ellos, y de esta forma contribuir a mantener limpias las calles de orines, en especial cuando hay grandes fiestas o encuentro como "La fiesta de los mil tambores".[1]

El objetivo de esta memoria es brindar un *backend* para el desarrollo de aplicaciones (web o móvil) que permitan, mediante la información reportada por los ciudadanos de Valparaíso, recolectar datos y proveer información sobre los servicios públicos básicos, principalmente baños que se pueden encontrar en la ciudad porteña. Esto significa brindar un sistema de apoyo para crear aplicaciones que permitan mediante *crowdsourcing*, levantar datos relevantes para ser utilizados de forma libre por quien la necesite, de forma que la población pueda recolectar estos datos que serían casi imposible de obtener y mantener actualizados por entes públicos como la municipalidad.

Un objetivo subyacente de esta memoria es poder contribuir, desde la informática, a la iniciativa de ciudad limpia con información dada por la comunidad. El *backend* propuesto permitirá ayudar a

[1] **Noticia de Coperativa "Vecinos de Valparaíso presentaron recurso para impedir festival":** http://www.cooperativa.cl/noticias/entretencion/carnavales-culturales/vecinos-de-valparaiso-presentaron-recurso-para-impedir-festival-de-los/2017-09-25/163739.html **Visitada:** 2017-11-10

turistas entregándoles información, que puede parecer trivial, pero que muchas veces es fundamental para satisfacer las necesidades básicas en situaciones de urgencia. Por otro lado, además permitirá a los mismos porteños contribuir conjuntamente a un mapa de los servicios públicos de Valparaíso.

Este documento cuenta con 9 capítulos (incluyendo este), donde se describe el proceso completo de documentación, análisis y desarrollo del trabajo realizado para la creación de un *backend*.

El capítulo 2 "El problema" cuenta con dos secciones. La sección 2.1 trata del problema que motiva para realizar este trabajo y la sección 2.2 de los principales problemas que se deben enfrentar para dar una solución al problema en cuestión.

El capítulo 3 "Propuesta" cuenta con dos secciones. La sección 3.1 habla de la propuesta de solución a implementar para el problema planteado y la sección 3.2 muestra el objetivo general y los objetivos específicos de este trabajo.

El capítulo 4 "Trabajo relacionado" cuenta con dos secciones. La sección 4.1 habla sobre las *APIs* y cómo éstas han contribuido al desarrollo de software y benefician a los usuarios y programadores. La sección 4.2 habla sobre las dos principales arquitecturas para la construcción de *APIs* describiendo ambas de forma general y explicando brevemente su funcionamiento.

El capítulo 5 "Metodología" cuenta con tres secciones en las que se explican las metodologías utilizadas para el desarrollo del trabajo. La sección 5.1 habla sobre la metodología utilizada para obtener los requerimientos de software, donde el proceso se describe paso a paso, dando ejemplos de cómo se realizaron éstos. La sección 5.2 describe el proceso para considerar posibles tecnologías y tomar decisiones sobre cuál utilizar dependiendo de los requerimientos de la solución. La sección 5.3 habla sobre la herramienta utilizada para medir el impacto de los componentes de la arquitectura de software en los requerimientos no funcionales.

El capítulo 6 "Resultados: Decisiones para satisfacer requerimientos no funcionales" describe en detalle a lo largo de tres secciones las decisiones tomadas para que el sistema satisfaga los

requerimientos no funcionales. La sección 6.1 muestra y explica las opciones de lenguaje a utilizar, los criterios para elegirlo y la decisión final. La sección 6.2 explica los criterios para elegir un *framework*, los posibles *frameworks* y una comparación de todos ellos mostrando sus ventajas y desventajas, finalmente se explica la decisión tomada. En la sección 6.3 se explican las decisiones tomadas para la arquitectura de software. Estas decisiones parten con el patrón de arquitectura a utilizar, donde se describe el funcionamiento de ésta y sus componentes. Luego continúa con la decisión del motor de base de datos a utilizar, considerando los tipos de motores y los más populares de cada tipo. Posteriormente se realiza una comparación entre las arquitecturas *REST* y *SOAP* y la elección para este proyecto en función de sus características. Finalmente se realiza un análisis de trazabilidad de requerimientos no funcionales contra los componentes de la arquitectura, para validar que se cumplan los requerimientos no funcionales.

El capítulo 7 “Resultados: Decisiones para satisfacer requerimientos funcionales” se describen las decisiones tomadas para que el sistema cumpla con los requerimientos funcionales a lo largo de dos secciones. La sección 7.1 explica el modelo de base de datos y cómo éste permite desarrollar las funcionalidades deseadas. La sección 7.2 muestra los micro servicios creados y explica a grandes rasgos su funcionamiento, además de mostrar a modo de ejemplo el funcionamiento de algunos métodos del sistema.

El capítulo 8 “Validación” muestra los tests y validaciones realizadas para comprobar el cumplimiento de los requerimientos a lo largo de cuatro secciones. La sección 8.1 muestra los detalles de un test de estrés para medir la escalabilidad de carga que posee el sistema. La sección 8.2 explica porqué el sistema cumple con ser reusable. La sección 8.3 explica porqué el sistema cumple con la extensibilidad. La sección 8.4 explica porqué el sistema cumple con la adaptabilidad y la interoperabilidad.

El capítulo 9 “Conclusión” detalla las conclusiones e ideas finales de todo el trabajo realizado, habla sobre los resultados del proyecto, su posible continuidad en el futuro, el logro de los objetivos y el desarrollo personal que implicó su desarrollo.

2 | El problema

2.1. Motivación

En la actualidad, no existe claridad ni visibilidad de los servicios públicos que se encuentran en la ciudad de Valparaíso. Por ejemplo, si tratamos de ingresar a la página de la municipalidad a la sección de "*Ciudad*" - "*Servicios Ciudadanos*",[2] nos encontramos con una página de error. Esta falta de información podría provocar que las inversiones tanto de privados como del Estado en proveer servicios no se aprovechen en su totalidad, causando a la ciudadanía a su vez que estos esfuerzos no generen los resultados esperados.

La información recolectada permitiría no sólo conocer los servicios disponibles en la ciudad, sino que también conocer su calidad y estado actual. Además se podría etiquetar los horarios en que se pueden utilizar e incluso tener en aviso de su valor aproximado, ya que muchas veces estos servicios son de pago, incluso en hospitales públicos.[3]

Otro punto a favor de este proyecto, es que sirve como base para formar una comunidad porteña, que de forma colaborativa y libre, entregue datos del estado de otros servicios públicos de forma *geo-referenciada*, la que es difícil de mantener actualizada por instituciones. Estos datos permitirían a las autoridades priorizar y gestionar de mejor forma los recursos destinados. Por ejemplo fácilmente podría ser expandido a reportar baches en las calles, sitios eriazos, semáforos

[2] http://www.munivalpo.cl/Valpomaps/Mapa.aspx **Visitada:** 2017-12-07

[3] Noticia de **El Mostrador**, "El Abuso de cobrar por estacionamientos y baños en edificios de uso público": http://www.elmostrador.cl/noticias/opinion/2012/05/29/el-abuso-de-cobrar-por-estacionamientos-y-banos-en-edificios-de-uso-publico/ **Visitada:** 2017-10-15

descompuestos, señaléticas rotas, etc. permitiendo a la comunidad participar activamente de la visibilidad continua de su ciudad y así sentirse parte del avance de la ciudad al ver que sus reportes permitieron generar un cambio. De esta forma el sistema partiría con el objetivo de mostrar los baños públicos de la ciudad, pero podría extenderse a ser una vía de generación de información de la situación de la ciudad. No es que con la aplicación se piense en un futuro mostrar donde están los problemas de la ciudad a quienes la utilicen, pero sí tener información útil para quien sea responsable de realizar las mejoras.

Este trabajo se centrará en servicios higiénicos públicos, pero se diseñará para dar soporte a un trabajo más extenso a futuro y abarcar diferentes problemas, más allá de los considerados actualmente. Esto se debe al posible potencial de crecimiento de la aplicación.

2.2. Problemas a solucionar

Implementar un mecanismo de *crowdsourcing* para conseguir los datos de los servicios públicos y su estado. *Crowdsourcing* es una actividad colaborativa de recolección de datos, generalmente a través de internet, para generar conocimiento colectivo [4].

Sin embargo, el mecanismo de *crowdsourcing* debe diseñarse cumplir las siguientes caracterísicas:

- **Asegurar que el sistema sea escalable**: Considerando las posibilidades de crecimiento del sistema, y dado que si se suman muchos usuarios, el sistema debe ser capaz de mantenerse estable y escalable, se debe permitir que crezca no sólo en cantidad de usuarios, sino que también en cantidad de servicios públicos y volumen de datos. La idea es que el sistema pueda ser extendido para que con el tiempo alcance una mayor utilidad.
- **Asegurar que los datos recolectados son confiables**: Esto debido a que es probable que cierta cantidad de los datos recibido por el sistema, no sean de fiar. Siempre existe la posibilidad de tener usuarios maliciosos que entreguen datos falsos o incorrectos. Por ello es necesario buscar métodos que aseguren que los datos entregados sean los correctos. Esta tarea es fundamental, ya que al depender del público, el sistema puede verse gravemente afectado si la tasa de usuarios mal intencionados es alta, contaminando la información que se quiere formar.

- **Analizar los datos de forma efectiva**: Dado que son tantas las posibles entradas de datos al sistema, es importante considerar como se va a analizar dicha data para que se convierta en información útil. Ya que por mucha ayuda que se reciba de la comunidad, si esta no es tratada de la forma correcta, probablemente no cumplirá su principal beneficio.

Esta memoria busca diseñar y construir un *backend* que cumpla con las características anteriormente señaladas, con el fin de darle solución al menos en parte al problema.

3 | Propuesta

3.1. Propuesta de solución

La solución propuesta es crear una *API*[4], que permita manipular la información parcial o totalmente independiente de la capa externa de presentación. La idea es que el sistema sea extensible y escalable para así alimentar varias formas de comunicación, ya sea web y/o móvil (Android e IOS), de forma que sea flexible para que fácilmente se adapte a nuevos requerimientos y abierta al público tanto en uso como en código, para así alimentarse de distintas fuentes y formas. También se busca tener la posibilidad de extender el sistema, sin afectar los métodos de conexión con éste.

La idea que el sistema sea una API permite que la parte técnica funcional esté cubierta completamente para el momento en que se quiera implementar una interfaz. De esta forma los expertos en usabilidad y experiencia de usuario puedan trabajar libremente y sin preocupaciones, con la misión de crear el "cascarón" ya que el cerebro será independiente. Por otro lado, esta forma de desarrollo permite centralizar los datos de las aplicaciones o interfaces, ya que una página web y una aplicación móvil pueden compartir el mismo núcleo, evitando tener que hacer implementaciones para comunicar sistemas separados.

Si bien la idea de una API tiene muchas ventajas, también hay que tener cuidado con la escalabilidad del sistema, pues el hecho de que centralice todo en un mismo núcleo puede causar una sobrecarga. Por ello la importancia que desde el inicio se tenga en consideración la posible expansión y crecimiento del mismo.

[4]"*Application Programing Interface´´* o "*Interfaz de Programación de Aplicaciones´´*, o sea un núcleo de procesamiento centralizado que se puede integrar con alguna interfaz.

En el fondo, el núcleo es transparente, lo que permite una abstracción del modelo de datos y del motor de base de datos que utiliza el sistema, ya que las capas externas trabajarán simplemente haciendo solicitudes al núcleo. Por lo mismo, no tienen que preocuparse de la conexión con la base de datos, simplificando el desarrollo futuro y encapsulando el negocio de la aplicación en un solo lugar.

3.2. Objetivos

- **Objetivo General:** Diseño, construcción y evaluación de un mecanismo de almacenamiento de datos de crowdsourcing que permita la escalabilidad de carga, de extensibilidad y de geografía, y que además sea reutilizable.
- **Objetivos Específicos:**
  - Comparar decisiones de diseño que permitan la construcción de un sistema escalable.
  - Implementación de la arquitectura propuesta para el backend.
  - Realizar pruebas para medir capacidad de carga en un ambiente real.

4 | Trabajo relacionado

En este capítulo se habla de la historia de las *APIs* y cómo éstas permitieron la creación de aplicaciones que hoy son de uso masivo. Por otro lado se muestran las dos principales arquitecturas para la creación de *APIs* web con sus principales características, y una breve descripción de su funcionamiento.

4.1. APIs: Información centralizada y sistemas distribuidos

Las APIs surgieron como una forma de centralizar la información de sistemas de software que antes se manejaban por separado. Esto ha traído consigo también, el hecho de una mejora a la hora de refactorizar código [9]. Las APIs han revolucionado en la facilidad de uso y desarrollo por parte de los programadores, pues permite el re-uso y la externalización del núcleo de procesamiento de ciertas cosas. Hoy, gracias a las APIs como las desarrolladas por Google, han permitido a cientos de desarrolladores crear aplicaciones de manera más rápida. Un gran ejemplo es la aplicación **Waze**.[5] Esta aplicación surgió como una iniciativa independiente del israelí Uri Levine. En un principio eran los mismos usuarios quienes creaban los mapas, lo que significaba una enorme dificultad para extender la aplicación a nuevos lugares. Aún así el 2013 el uso de la aplicación se disparó a un nivel insostenible para la pequeña empresa creadora, alrededor de 400 mil usuarios nuevos al día, es ahí cuando Google compró Waze por 1.300 millones de dólares. Google gracias a su API de mapas GoogleMaps permitió que la aplicación continuara su crecimiento alcanzando su uso a nivel mundial [20]. Es gracias a la API GoogleMaps que tanto Waze como otros softwares pueden funcionar a gran escala rápidamente.

[5] Waze es una aplicación móvil que funciona como una red social entre conductores, con el fin principal de evitar atascos y encontrar rutas alternativas.

Si vamos más a fondo en la API de GoogleMaps, se podría decir que ella ha permitido centralizar información de direcciones, y a través de un único sistema, prestar servicios a miles de personas y aplicaciones. Por ejemplo, muchos optan por utilizarla para mostrar la dirección de su empresa, haciendo que sus clientes las puedan encontrar con mayor facilidad. Por otra parte, la API GoogleMaps crece con cada persona que marca su negocio en ella, pues significa recibir nueva información.

En el fondo las APIs en general permiten centralizar información referente a ciertos temas. Pero si tomamos diferentes APIs para crear una aplicación, en el fondo, se tiene un sistema distribuido. Por ejemplo, GoogleMaps provee información de lugares, ubicaciones y mapas, la de Facebook información de personas y Global Weather de clima. Ahora si quisiera crear una aplicación que maneje la información de personas, ubicaciones y clima, se pueden utilizar los tres servicios web recién mencionados, teniendo un sistema distribuido, donde cada parte trabaja con información de forma centralizada de algún tema en particular [19]. De hecho en la actualidad, muchas aplicaciones web que requieren autenticación de usuarios, utilizan los servicios de Facebook y Google para manejar la sesión, e incluso, no registran información del usuario propiamente tal.

4.2. Arquitecturas para la creación de APIs web

4.2.1. RESTful

Es una arquitectura para la construcción de servicios web que trabaja sobre la arquitectura de *HTTP* y sus métodos. Se basa en la creación de servicios con base en *nouns*, con los que se puede interactuar utilizando los métodos *HTTP*: *POST*, *PUT*, *PUSH*, *GET*, *DELETE*, etc. Cada servicio actúa como *CRUD*, permitiendo realizar acciones sobre dicho *noun*. Por ejemplo, a través del servicio de “autos”, realizando una llamada *GET* podría obtener un auto en particular y realizando una llamada *POST* con los parámetros correctos, crearía un nuevo auto.

Esta arquitectura permite crear APIs orientadas a micro servicios, donde cada servicio puede

actuar de forma casi independiente, agrupándose sólo gracias a una pequeña capa que funciona como un "orquestador" ligero [17].

El principal problema con esta arquitectura, para traer gran cantidad de información que está relacionada entre sí, ya que implica realizar múltiples llamadas a servicios basados en sustantivos [12].

4.2.2. SOAP

Es una arquitectura compleja que permite la comunicación en sistemas distribuidos. Esta arquitectura se encuentra altamente estandarizada e incluye fuertes elementos de seguridad e integridad de la información [3]. *SOAP* puede trabajar sobre *HTTP* y utiliza mensajes *XML* codificados. Esta arquitectura sigue la lógica de llamadas a procedimientos remotos, donde un cliente llama funciones a un servidor, que según ciertas características podría o no responder [12]. Esto quiere decir que la lógica de creación de método en esta arquitectura es por función y no por sustantivo, como lo es en el caso de *RESTful*.

Esta arquitectura permite tener funciones que traigan toda la información para "pintar una vista".[6] Esto tiene la ventaja de que permite diseñar métodos que en una llamada traigan toda la información necesaria para llenar una vista. La principal desventaja es que la alta seguridad, la compleja estructura y la información manejada en *XML*, hacen que el traspaso de datos sea más lento que con otras opciones, incluyendo RESTful [12].

[6]Pintar vista: Llenar un vista de una página web con la información requerida. Esta información suele venir de la base de datos.

5 | Metodología

5.1. Definición iterativa de requerimientos

Lo primero para construir cualquier sistema, es definir los requerimientos de éste, para así saber qué debe hacer el sistema. Para definir los requerimientos de este proyecto, se realizó un análisis iterativo siguiendo los siguientes pasos[2]:

- **Extracción**: En esta etapa se buscó extraer las necesidades que debe cubrir el sistema. Para esto generalmente se realiza una investigación con el (o de el) *cliente del software*[7], ya que es para éste que se está tratando de resolver el problema. En este caso se utilizó la información disponible de la población y los servicios de Valparaíso entregada por el municipio a través de su página web[8]. De esta forma se empezaron a desentrañar los requerimientos del sistema, por ejemplo dada la gran población de Valparaíso, de alrededor de unas 295.000 personas (población estimada para la ciudad de Valparaíso según datos recopilados por el Instituto Nacional de Estadísticas durante el CENSO del año 2012 [7]), es necesario que el sistema soporte una gran cantidad de usuarios.

- **Análisis**: En esta etapa se realizó un análisis de los requerimientos encontrados en el paso de Extracción, para así poder descubrir los posibles problemas de éstos. Para ello se realizó una pequeña "maqueta" en papel de lo que se quiere construir, sólo para tener una ayuda referencial de lo que se deseaba desarrollar. De esta forma se comenzó a esbozar las alternativas de solución y las principales opciones a desarrollar.

[7] **Cliente de Software:** Es quien va a utilizar o se va a ver beneficiado por el software. No necesariamente quien paga por el mismo. Puede ser una persona una empresa, un grupo masivo de personas o incluso otro software.

[8] https://www.municipalidaddevalparaiso.cl

- **Especificación**: En esta etapa se tomó el trabajo de las dos anteriores, y se comenzó a especificar los requerimientos de manera formal, por ejemplo con una aproximación al modelo de dominio (ver figura 5.1) y un diseño preliminar de la arquitectura (ver figura 5.2) de la solución. Esta etapa se realizó de forma transversal, pues en cada paso se hicieron bosquejos de la solución planteada, se puede decir que en ésta, se pasó de bosquejos en papel a algo digital.

- **Validación**: En esta etapa se verificó que los requerimientos obtenidos en los pasos previos estuvieran completos y que conformaran una visión comprensible de lo que debe hacer el sistema y bajo qué condiciones debe hacerlo. Se revisaron uno a uno los requerimientos tanto funcionales como no funcionales para examinar su coherencia. Se revisó que cada requerimiento no se contradiga con otro o consigo mismo, y que además, no sea ambiguo (tiene sólo una interpretación).

Los pasos se repitieron cada vez que aparecieron nuevos requerimientos, ya que incluso iniciado el proceso de desarrollo surgieron nuevos requerimientos. Esto pasa porque muchas veces es difícil ver todas las funcionalidades requeridas en el primer análisis. Sin embargo, hay que considerar que a medida que avanzó el desarrollo, la aparición de nuevos requerimientos disminuyó progresivamente, como era esperable.

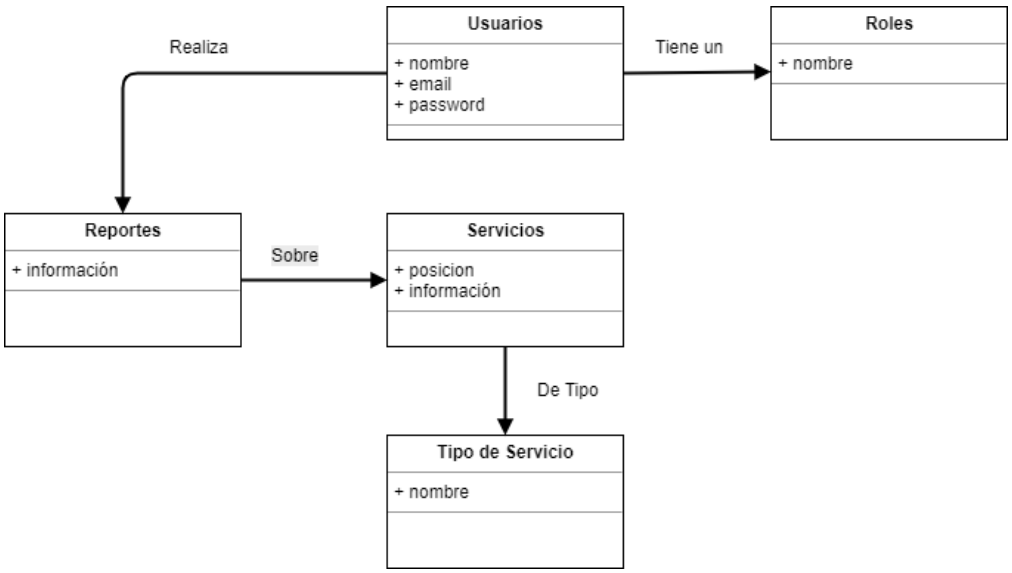


Figura 5.1: Primera aproximación al modelo de dominio.
Fuente: Elaboración Propia.

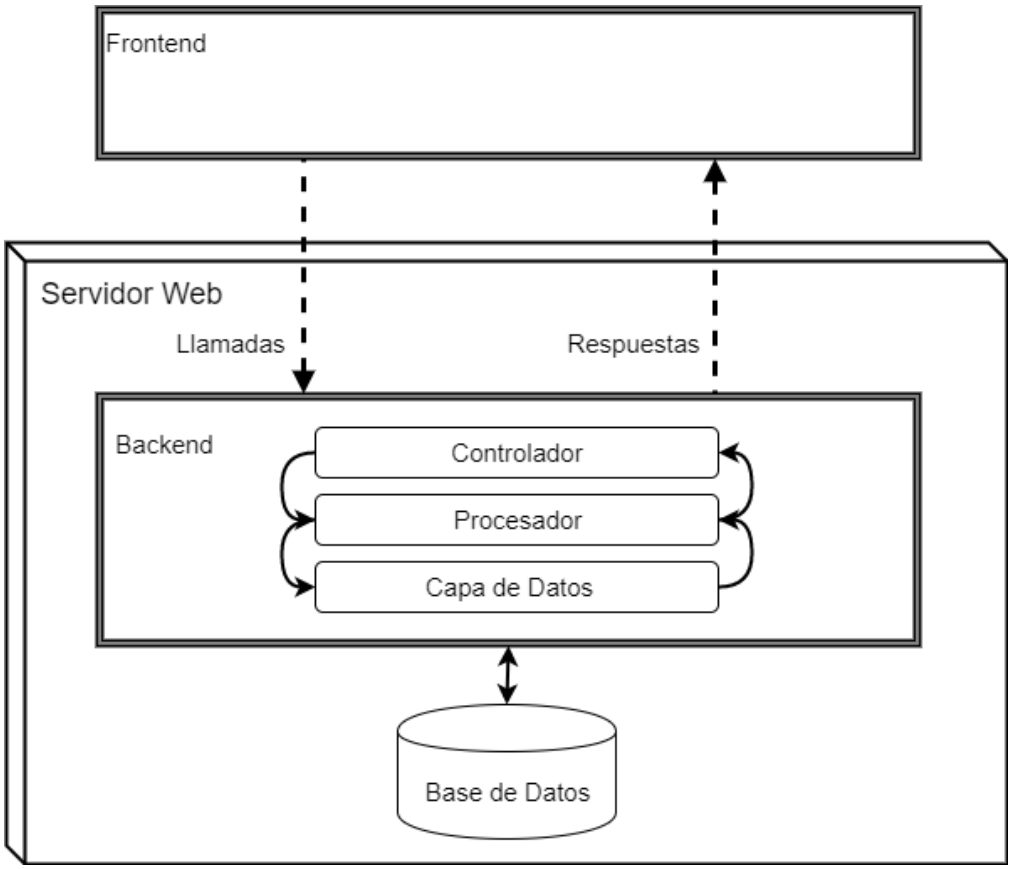


Figura 5.2: Primera aproximación a la arquitectura de software.
Fuente: Elaboración Propia.

5.2. Selección de tecnologías

Lo primero que se consideró es el tipo de proyecto a desarrollar. En este caso, al ser el *backend* de una aplicación web, se consideraron los principales lenguajes para programación web, con los que se armó una lista. El siguiente paso fue determinar las características deseadas para el perfil de desarrollo. Para reunir el perfil deseado se consideraron los requerimientos no funcionales. Por ejemplo algunos de los requerimientos no funcionales son que el software debe ser extensible y eficiente. Para cumplir con esos requerimientos el lenguaje utilizado debe ser popular (que tenga una comunidad de apoyo y que sea bastante utilizado) y que además sea veloz en ejecución y eficiente en el uso de memoria. De esa forma se generó una lista con los requerimientos deseados para el lenguaje de programación a utilizar.

Con la lista de lenguajes y puntos de requerimientos se realizó una comparación de las ventajas y

desventajas de cada lenguaje. Para este análisis se consideraron los lenguajes de programación *Javascript*, *PHP*, *Python* y *Ruby on Rails*, que son los 4 lenguajes para desarrollo web más demandados.[8]

El análisis se enfocó en utilizar las ventajas de cada lenguaje, para suplir los requerimientos, tratando de que los contras fueran mínimos. Por ejemplo se analizó la velocidad de ejecución y uso de los recursos de los distintos lenguajes a través de "*benchmark*". La popularidad a través de la cantidad de preguntas sobre dicho lenguaje en **stackoverflow**[9] o la cantidad de proyectos en **github**[10]. La dificultad y curva de aprendizaje en manuales o libros guía. Todo esto permitió contar con criterios para realizar un análisis sistemáticos en base a datos, y así no dejarse llevar por los intereses personales o subjetividades.

5.3. Diseño de arquitectura

Para el diseño de la arquitectura de software, se construyó una matriz de trazabilidad (ver tabla 5.1) de las características de la arquitectura respecto a los requerimientos no funcionales. Esta es una forma de ver cómo los cambios de componentes pueden afectar el desempeño del sistema con una perspectiva holística.

Analizando la tabla 5.1 se puede ver cómo los distintos componentes afectan a los requerimientos no funcionales. Por cada componente de la arquitectura, el análisis permite documentar y explicitar cómo éste afecta el requerimiento no funcional. Por ejemplo, el hecho de utilizar una arquitectura del tipo "RESTFUL" permite aumentar la escalabilidad de la aplicación de forma relativamente sencilla, ya que al estar compuesta de micro servicios, éstos se pueden parametrizar a diferentes máquinas para balancear la carga de llamadas. Por otro lado, al no tener estados, no necesita guardar información de sesión, por lo que es más eficiente en el uso de recursos. Es más, debido a que cada micro servicio puede ser utilizado de forma independiente la aplicación se torna sumamente extensible.

En la matriz 5.1, cada ticket (✓) significa que el componente (fila) afecta positivamente en el

[9] https://stackoverflow.com/
[10] https://github.com/

requerimiento (columna). A su vez los espacios en blanco significa que el componente no interviene con ese requerimiento. Por esto, para que la matriz tenga sentido, todas las columnas y filas deben tener al menos un ticket, o puede darse que existan componentes innecesarios o requerimientos sin satisfacer.

	Requerimientos				
Componenetes	Escalabilidad	Extensibilidad	Mantenibilidad	Eficiencia	Flexibilidad
API RESTFUL	✓	✓		✓	
Arquitectura por Capas			✓		✓
Base de Datos Relacional		✓		✓	

Tabla 5.1: Matriz de trazabilidad (Características de arquitectura vs Requerimientos)

6 | Resultados: Decisiones para satisfacer requerimientos no funcionales

6.1. Lenguajes de programación

Dado que éste es un proyecto que involucra desarrollo web, los lenguajes a considerar deben poseer esa característica, además de las siguientes:

- **Popularidad:** Esto debido a que uno de los objetivos del proyecto es que éste se extienda, el sistema debe ser desarrollado en un lenguaje que tenga una gran prevalencia en los desarrollos actuales.
- **Facilidad de desarrollo:** Se espera que la curva de aprendizaje sea pequeña y que la cantidad de líneas de código y configuración sea mínima para realizar una tarea, con el fin de que el lenguaje permita un desarrollo rápido.
- **Multi-plataforma:** Es fundamental que el lenguaje sea indiferente al sistema operativo que se desee usar en el servidor, para que pueda ser portado según las necesidades de crecimiento. Además debe permitir que los desarrolladores puedan trabajar desde cualquier sistema operativo, indiferente del ambiente de producción.
- **Ligero y veloz:** Se espera que el lenguaje sea veloz y ligero en el servidor, ya que el objetivo de este proyecto es el desarrollo de un *backend* (núcleo) de una aplicación. La velocidad es fundamental, pues si el núcleo es lento, no importa que tan bueno sea el *frontend*, el sistema será lento.

Para desarrollo web, los lenguajes que mejor cumplen con las características señaladas son:

- **Javascript:** Es un lenguaje fácil de utilizar y difícil de comprender inicialmente, la curva de aprendizaje es empinada. Aprender a utilizarlo en un desarrollo *full-stack* (frontend y backend) toma bastante tiempo, especialmente la configuración inicial [5]. Sin embargo, es sumamente popular porque permite desarrollos rápidos una vez dominado el lenguaje y la construcción de funcionalidades sumamente complejas, gracias a su flexibilidad, pues puede ser utilizado principalmente en tres paradigmas: imperativo, orientado a objetos y orientado a eventos [11].

- **PHP:** Es un lenguaje con una ligera curva de aprendizaje, permite desarrollos ágiles en poco tiempo sin necesidad de mucha configuración inicial [14]. Además, es un lenguaje multi-plataforma. Si bien éste se mantuvo como el rey del desarrollo web por varios años, hoy en día su popularidad ha ido en descenso. Pero, se estima que poco más del 80 % de los sitios web lo utilizan como lenguaje *server side* [16].

- **Python:** Es el lenguaje con la curva de aprendizaje más ligera entre los aquí señalados. Permite desarrollos ágiles, debido a que tiene un nivel muy alto de abstracción y en pocas líneas se pueden realizar varias tareas. Por lo mismo, es ideal para cursos de introducción a la programación [6]. La gran velocidad en desarrollo que posee se ve mermada en ejecución, pues al ser un lenguaje pensado para *scripting* es el más lento de la lista.

- **Ruby on Rails:** Su curva de aprendizaje es complicada en un principio, pero después de alcanzar cierto punto se torna ligera. Posee aceleradores de desarrollo que permiten generar mantenedores de forma automática. En cuanto a ejecución, si bien es levemente más rápido que *python*, sigue siendo lento en comparación a los otros.

Dadas las ventajas y desventajas de cada uno, el lenguaje que mejor satisface las necesidades de este proyecto es *PHP*. Esto debido a que es un lenguaje que lleva años funcionando sin quedar obsoleto, pues siguen saliendo actualizaciones (la última versión estable es la 7.1.4 liberada el 13 de abril del 2017)[11]. Además es sumamente escalable, y se puede ver en dos grandes ejemplos, *Facebook*[12] y *Wikipedia*.[13] Ambas plataformas son desarrolladas utilizando este lenguaje como principal. Otro gran beneficio de este lenguaje es que debido a sus años de historia posee una gran cantidad

[11] http://php.net/downloads.php
[12] https://www.facebook.com
[13] https://es.wikipedia.org

de proyectos, tiene una gran comunidad, manuales, códigos para reutilizar, además de bibliotecas que faciliten trabajos e integraciones. Por otra parte, posee una gran cantidad de frameworks. Esto permite elegir el que mejor se acomode a las necesidades de desarrollo. Adicionalmente su núcleo está desarrollado en *C* y es bastante veloz, a pesar de ser lenguaje interpretado.

6.2. Frameworks

Las opciones de frameworks para crear APIs es muy variada. Sin embargo, no todas las opciones son iguales. Cada una de ellas fue pensada tratando de satisfacer distintas necesidades, algunas se basan en rapidez, otras en bajo consumo de recursos. Por lo tanto, es necesario hacer un análisis de cuál es la mejor opción para el caso actual según los objetivos del sistema, los que se pueden ver reflejados en las siguientes características de la solución.

- **Desempeño:** Es importante que el sistema soporte una gran cantidad de peticiones por minuto, con el fin de que permita recibir ráfagas de un gran número de peticiones en caso de que el uso del sistema crezca, o se extienda a nuevas ciudades e incluso se le agreguen nuevas funcionalidades. Además, como el sistema funciona como una API, es fundamental que sea veloz para reducir el tiempo de espera de los usuarios producido por la interacción de diferentes capas.

- **Documentación:** Si bien en primera instancia el sistema no considera operaciones complejas, al ser un sistema que puede seguir creciendo, necesita que quién desee extenderlo, posea toda la documentación para extenderlo tanto como desee. Si bien este punto no es fundamental, es importante para que la idea de extender el sistema sea atractiva.

- **Extensibilidad:** Es fundamental que el sistema sea escalable, principalmente de forma modular e independiente, para que no sea necesario modificar las funcionalidades existentes para crear nuevas. Esto asegurara la integridad del desarrollo, trabajando de forma incremental, en la medida que sean necesarias nuevas funcionalidades.

- **Integración con sistemas de bases de datos:** Es importante que el framework posea un "ORM" ("Object-Relational Mapping" o Mapeo de objeto relacional) que permita una independencia del sistema de base de datos. Esto con el fin de que el acceso a los datos sea de

forma transparente para quien esté desarrollando, de tal forma que si es necesario se realice un cambio, no implique cambios en el núcleo del sistema.

6.2.1. Opciones de framework

- **BulletPHP:** Es un micro-framework, que tiene como único propósito realizar enrutamiento de URL's.[14]
- **Lemonade:** Es un micro-framework pensado en el desarrollo y prototipado ágil, está inspirado en Ruby on Rails.[15]
- **Fat-Free:** Liviano y modular, pero potente, diseñado para la construcción de sistemas REST.[16]
- **Lumen:** Micro-framework para aplicaciones REST dentro de Laravel.[17]
- **Phalcon:** Framework PHP, su núcleo está desarrollado en *C* y enfocado para maximizar la velocidad.[18]
- **Slim:** Micro-framework ligero y de levantamiento rápido, pensado para hacer aplicaciones REST.[19]

En la tabla 6.1, se detallan los pros y contras de cada *framework*. En la tabla 6.2 se puede ver una comparación de las características de cada "***framework***", donde cada ✓significa que éste posee esa característica, "✗" que no la posee y "-" que no es observada ni reportada. La última columna es una ponderación, donde se suman los "✓" y se restan los "✗", entregando un número que mientras mayor sea, mejor podemos considerar al *framework*.

De las diferentes opciones de frameworks para *PHP* disponibles, **Phalcon** es el más rápido, su núcleo desarrollado en *C* le permite ser el más veloz en comparación a los otros [18]. Por otro lado, al ser modular y completamente libre en cuanto a la organización de las carpetas y complementos,

[14] http://bulletphp.com/
[15] https://limonade-php.github.io/
[16] https://fatfreeframework.com/3.6/home
[17] https://lumen.laravel.com/
[18] https://phalconphp.com/es/
[19] https://www.slimframework.com/

	A favor	En contra
BulletPHP	▪ Previene la duplicación de código. ▪ Las URL's se manejan por segmentos. ▪ Rutas sin restricciones (permitiendo enrutamiento inverso)	▪ Poca documentación. ▪ Posee un nuevo modelo de enrutamiento, que si bien es sumamente potente, es difícil de aprender. ▪ No es tan flexible.
Lemonade	▪ Poca codificación para realizar más tareas.	▪ Mala documentación. ▪ Baja modularidad.
Fat-Free	▪ Liviano y multifuncional. ▪ Todo es modular. ▪ Desarrollo rápido. Es ideal para proyectos grandes.	▪ Mucha repetición de código. ▪ Demasiadas funcionalidades como para ser utilizado en proyectos básicos.
Lumen	▪ Excelente documentación y comunidad activa. ▪ Al ser el hermano pequeño de Laravel, puede ser expendido fácilmente a un proyecto más grande. ▪ Fácil sintaxis.	▪ En cuanto a velocidad de ejecución, está muy por debajo de los frameworks más veloces. ▪ Demasiadas funcionalidades como para ser utilizado en proyectos básicos.
Phalcon	▪ El más rápido de los frameworks PHP. ▪ Optimizado para ser ligero y modular. ▪ ORM y núcleo desarrollados en *C*.	▪ Relativamente nuevo, por lo que la documentación es limitada a la oficial.
Slim	▪ Uno de los frameworks más rápidos. ▪ Bien documentado.	▪ Si bien es un micro-framework, se puede considerar como demasiado pequeño aún para esa categoría, esto provoca que sea difícil de escalar a proyectos más grandes.

Tabla 6.1: Comparativa de pros y contras de los *frameworks* para *PHP*

	Características						
Framework	Documentación	Modularidad	Flexibilidad	Eficiencia (memoria)	Eficiencia (ejecución)	Eficiencia (desarrollo)	**Ponderación**
BuletPHP	×	-	×	-	-	✓	-1
Lemonade	×	×	✓	-	-	✓	0
Fat-Free	-	✓	✓	✓	-	✓	+4
Lumen	✓	✓	-	-	×	✓	+2
Phalcon	✓	✓	✓	✓	✓	-	+5
Slim	✓	✓	-	✓	✓	-	+4

Tabla 6.2: Comparación de frameworks PHP para API Rest

permite flexibilidad a la hora de elegir arquitectura y capas a utilizar. Por lo mismo, es sumamente escalable. Además, su ORM permite una independencia del motor de base de datos, pues permite trabajar directamente con los modelos sin utilizar *SQL* directamente, realizando filtros de sanitización para impedir ataques comunes de inyección de SQL u otros asociados a la alteración de la base de datos [15].

En la figura 6.1, se observa una comparación en la velocidad de ejecución de los métodos de un *CRUD* en distintos frameworks. El primer elemento en el eje inferior del gráfico muestra los tiempos de ejecución en *PHP* plano (sin ningún framework), luego hacia la derecha se muestran los diferentes tiempos obtenidos con cada frameworks. En el gráfico se observa como **phalcon**, es el framework más veloz, casi tan rápido como utilizar *PHP* plano (lo más rápido que se puede utilizar *PHP*).

6.3. Arquitectura de software

La idea es que este proyecto funcione como un backend que posteriormente puede ser re-usado por un entorno gráfico. El entorno gráfico puede ser una página web, una aplicación móvil o incluso de escritorio, permitiendo centrar gran parte del desarrollo en un solo *backend*. Por ello, es fundamental que la arquitectura permita la conexión de múltiples sistemas.

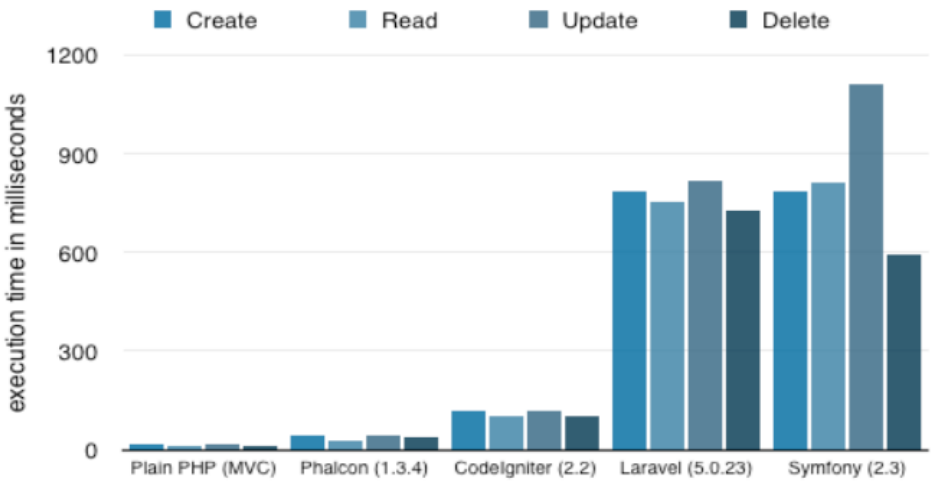


Figura 6.1: Benchmarck de algunos Frameworks populares de PHP. (Tiempo de ejecución de operaciones de un *CRUD*) [18]

Para que la construcción del *backend*, cumpla con lo expuesto en los objetivos (sección 3.2), que el sistema sea escalable y reutilizable, se debió realizar un análisis del patrón de arquitectura a seguir, y cómo según ese patrón, se diseñaría el sistema. La elección de patrón y diseño fue un paso fundamental, porque afectan de forma crítica el comportamiento de la solución, siendo factores determinantes en el cumplimiento de los requerimientos no funcionales. En la siguiente sección se explican los patrones que se siguieron para el diseño, y el diseño mismo de la aplicación.

6.3.1. Patrón de Arquitectura

La idea de que sea un modelo por capas es importante, pues permite una independencia de la base de datos, debido a que no importa cómo sea su estructura o su método de acceso, lo que importa es que venga la data necesaria. A su vez, permite mejor mantenibilidad del sistema, pues la aislación de las capas permite testearlas de forma independiente para encontrar más fácilmente problemas y o *bugs* que afecten el código.

Por otro lado, el modelo orientado a micro-servicios aporta a la escalabilidad y reuso de código, sin mencionar además su funcionamiento más veloz. Otra consideración importante es el hecho de que la arquitectura de micro-servicios es ideal para el desarrollo de *APIs*, ya que permite cons-

truirla de forma modular en la medida que se van necesitando nuevos recursos o funcionalidades. Además permite tener módulos totalmente independientes entre sí, donde cada módulo se puede hacer cargo de cierta funcionalidad, permitiendo granularizar o aglutinar las operaciones tanto como sea necesario, lo cual significa una gran flexibilidad en desarrollo.

En la figura 6.2 se puede observar cómo es el diseño de la arquitectura de software, donde el primer contenedor es un servidor web para *http*, encargado de levantar el servicio, por ejemplo con *Apache*, *Tomcat*, *Lighttpd*, *Caddy*, etc. Dado que *http* (Hypertext Transfer Protocol) es un protocolo estandarizado que permite la comunicación de sistema de diferentes características. Es una excelente opción para la comunicación del sistema, ya que hoy en día son cada vez más lo aparatos que se comunican a través de este protocolo.

La aplicación debe estar construida de tal forma que sea escalable, independiente del motor de base de datos, para esto la mejor opción es utilizar un híbrido entre un modelo por capas y uno orientado a micro-servicios [17]. En este caso se consideró *Apache*, por ser el servidor *http* más popular [13]. Dentro del servidor web se encuentra alojada la **API**, que a su vez es un contenedor de un micro-servicios. Cada micro-servicio encapsula el funcionamiento completo de una pequeña parte del proyecto. Por ejemplo, el micro-servicio de usuarios funciona como *CRUD*,[20] y algunas otras pequeñas funcionalidades de usuarios, tal como validarse como usuario (*Login*). Esto quiere decir que cada micro-servicio es independiente, permitiendo un desarrollo incremental y generando una gran adaptabilidad a los cambios de requerimientos. Además, los sistemas basados en micro servicios aumentan la escalabilidad del sistema, pues permiten incluso migrar parte de la aplicación a un servidor externo, sin afectar el funcionamiento del resto [17].

A su vez cada micro-servicio está compuesto de 3 capas, que permiten aislar el comportamiento de la aplicación y de esa forma facilitar la lectura de código, y detectar de forma sencilla errores o bugs, ya que quedan encapsulados en una capa en específico [17]. Las capas de cada micro-servicio son:

- **Capa de Acceso (*Controller*):** Esta capa se encarga de recibir la petición *http* y los parámetros

[20]CRUD: Controlador que agrupa las funciones relacionadas a crear (Create), leer (Read), actualizar (Update) y eliminar (Delete) de cierta clase de objetos.

necesarios para cada acción, validando que se esté entregando la información requerida para la ejecución de la petición realizada y haciendo control del acceso y sesión. Posteriormente, se comunica con la capa de negocio, analiza la respuesta verificando que esté correcta y la envía a quien realizó la petición inicial.

- **Capa de Negocio (*Business*):** Esta capa se encarga de realizar todos los cálculos y transformaciones para pasar la información de la capa de acceso a la capa de datos o viceversa. Se encarga del negocio mismo de la aplicación, realizando las validaciones y transformaciones más profundas a los datos para generar información. Es la capa intermedia entre la capa de acceso y la capa de datos.

- **Capa de Datos (*Model Manager*):** Esta capa se encarga de la conexión directa con el motor de base de datos, manejando errores, y validando la información de los registros de la base de datos a los objetos de datos y realizando todas las consultas necesarias. Permite una abstracción completa al motor de base de datos, haciendo que para la capa de negocios sea transparente el motor de base de datos, y trabaje solo con objetos. De esta forma aisla y controla todos los errores al nivel de la base de datos, siendo la única que tiene acceso a la base de datos y genera el puente entre ésta y la aplicación, siendo su otra contraparte la capa de negocios.

6.3.2. Motor de Base de Datos

Elegir el tipo de base de datos a utilizar, es fundamental, ya que existen bases de datos relacionales y no relacionales, cada una con sus ventajas y desventajas. Para este proyecto se decide utilizar un motor de base de datos relacional en contra de un motor no relacional (como **MongoDB**[21] o **Cassandra**[22]) debido a que los motores de bases de datos no relacionales se comportan mejor ante la búsqueda de gran cantidad de datos (mayor crecimiento horizontal). Sin embargo tienen problemas en consultas complejas que permiten por ejemplo, entrelazar y estudiar el comportamiento de los usuarios (crecimiento vertical), problemas que las bases de datos relacionales, a través de su experiencia y ventaja en el tiempo, tienen resuelta mediante el lenguaje *SQL* [10].

[21] https://www.mongodb.com/es
[22] http://cassandra.apache.org/

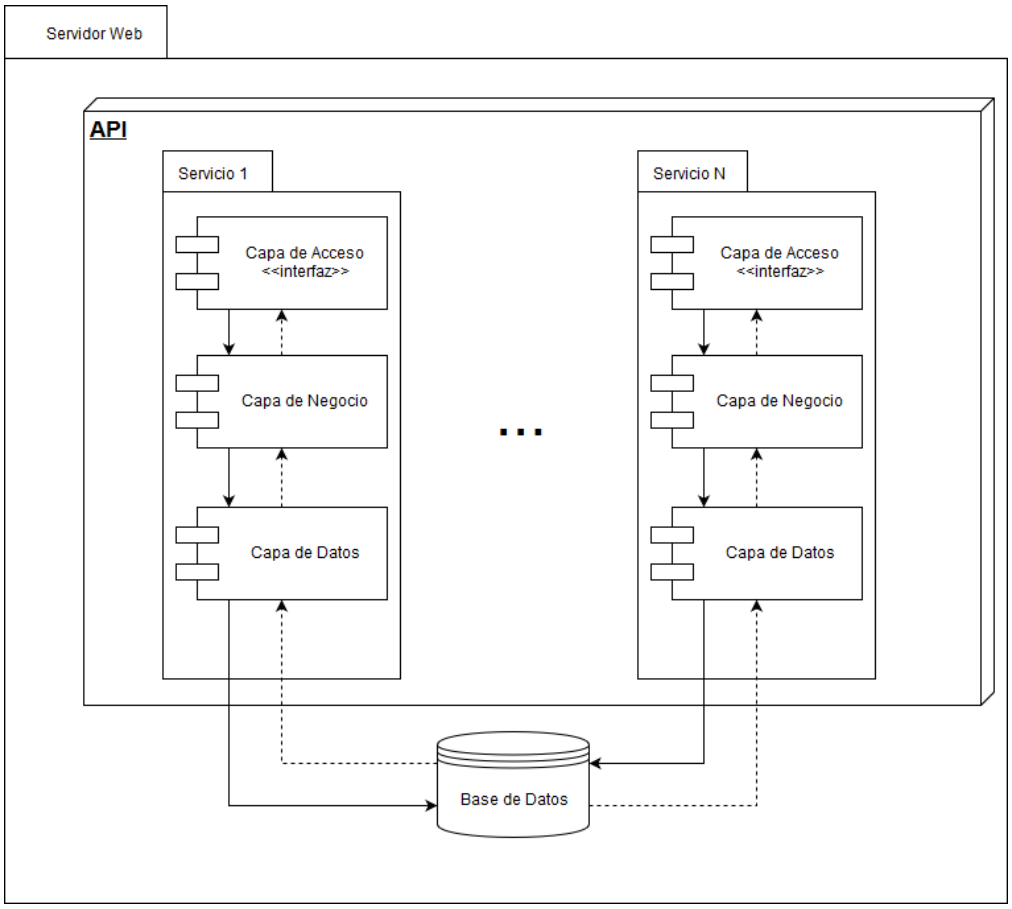


Figura 6.2: Diagrama de Arquitectura de la API.
Fuente: Elaboración Propia.

Ya seleccionado el tipo de base de datos relacional, se consideraron los motores **MySql**,[23] **PostgreSQL**[24] y **Microsoft SQL Server**.[25] Finalmente para el desarrollo del sistema se ha considerado el motor de base de datos **MySql**, perteneciente a la empresa **ORACLE**,[26] y que es de licencia gratuita para proyectos de software open source.[27] Esto debido a que según **DB-Engines Ranking**,[28] desde octubre del 2016 a octubre del 2017, se ha mantenido como el segundo motor de base de datos más popular (por debajo del motor de base de datos **Oracle**), y el primero de uso libre para proyectos licencia de código abierto. Esta popularidad permite garantizar un mayor soporte, una mejor comunidad de usuarios dispuestos a responder dudas y puede ser un indicio de su calidad.

[23] https://www.mysql.com/
[24] https://www.postgresql.org/
[25] www.microsoft.com/\discretionary{-}{}{}en-us/\discretionary{-}{}{}sql-server
[26] https://www.oracle.com/
[27] Software Open Source: Es software de código abierto, es decir que quien tiene los derechos de autor permite que cualquiera tome el código y lo modifique como desee.
[28] https://db-engines.com/en/ranking

6.3.3. SOAP vs REST

SOAP (Simple Object Access Protocol) y *REST* (Representational State Transfer) son protocolos para servicios web, cada uno con sus respectivas características que le otorgan ciertos beneficios y desventajas. Ambos protocolos son viables, pero hay que escoger el que mejor se acomode al proyecto.

- **SOAP:** Nació como protocolo para darle acceso a internet a tecnologías antiguas. La comunicación funciona a base de *XML* y si bien está pensado para trabajar sobre *HTTP*, permite también la comunicación mediante otros protocolos como por ejemplo *SMTP*. En general, tiene los siguientes aspectos a favor:
  - Altamente estandarizado.
  - Independiente del protocolo de transporte.
  - Ideal para sistemas distribuidos.
- **REST:** Nació como una arquitectura ligera para mejorar la escalabilidad en sistemas distribuidos, pero hoy hace referencia también a un protocolo de conexión sobre *HTTP*. Sus aspectos a favor son:
  - Eficiente en el traspaso de información. Dado que permite utilizar formatos de mensajes más ligeros que *XML*.
  - Rápido ya que no requiere procesamiento extra.
  - Flexible para las nuevas tecnologías web.

Dado que *REST* es una forma de comunicación más moderna, que en el fondo nació para buscar una alternativa ligera para *SOAP*, se considera que es la mejor opción. Además, es de construcción simple, permite arquitecturas más flexibles y mejor integración con *frameworks web* modernos, como *nodejs* o similares que trabajan con *javascript*. Esto debido principalmente a que *REST* puede trabajar con *JSON* (Javascript Object Notation) y que no se necesitan bibliotecas adicionales para trabajar ni procesos para convertir la respuesta en objetos, porque los textos en formato *JSON* se interpretan de forma natural como objetos.

6.3.4. Análisis de trazabilidad de requerimientos de calidad

Es importante identificar los requerimientos no funcionales del sistema, para poder medir de alguna forma la calidad del mismo, puesto que si bien cumpliendo los requisitos funcionales, el sistema "*hace lo que tiene que hacer*". Es necesario también asegurar un nivel de calidad, puesto que no sirve que el sitema realice las tareas si le toma el doble del tiempo esperado, o que a la hora de querer mejorarlo, el código sea inteligible.

	Requerimientos				
Componentes de Software / Arquitectura	**Eficiencia**	**Interoperabilidad**	**Escalabilidad**	**Extensibilidad**	**Adaptabilidad**
Phalcon PHP	✓				✓
RESTful		✓	✓		
Arquitectura Mixta (Microservicios y Capas)			✓	✓	
Base de Datos Relacional	✓			✓	

Tabla 6.3: Matríz de trazabilidad (Requerimientos no funcionales v.s Componentes)

La tabla 6.3 es la versión final de la matríz de trazabilidad. En ésta se agrega el *framework* utilizado y se cambian los requerimientos no funcionales de **mantenibilidad** y **flexibilidad** por **interoperabilidad** y **adaptabilidad**. El primer cambio se debe a que un software extensible debe ser mantenible, por lo que ese descriptor mantenibilidad no entrega nada nuevo, reemplazándose con interoperabilidad, que resultó ser más importante, puesto que, el software debía funcionar en múltiples plataformas. Además se reemplazó flexibilidad con adaptabilidad, ya que el software debe tener la posibilidad de adaptarse a nuevos usos, como uno de sus requerimientos principales. Estos cambios no se consideraron desde el principio, ya que en la iteración de requerimientos inicial no fueron visibles, apareciendo más adelante.

7 | Resultados: Decisiones para satisfacer requerimientos funcionales

7.1. Modelo de datos

Considerando los requerimientos funcionales, el modelo de datos quedó finalmente tal como se observa en la figura 7.2. Una de las diferencias más destacables entre esta versión y la inicial (ver figura 7.1), es que en este modelo de datos se incluyen las tablas "*movements*" y "*user_tracks*", que tienen como función registrar cada movimiento de los usuarios para poder recrear y analizar los movimientos que realice y las rutas a las que ingrese. Para esto la tabla "*movements*" registra todas las llamadas **HTTP** recibidas por el sistema, mientras que a su vez, la tabla "*user_tracks*" registra los movimientos del mouse, "*clicks*" y deslizamientos en el mapa de un usuario. Con ambas tablas se puede analizar por completo la traza de un usuario y se podrían obtener datos de cuántos "*clicks*" o tiempo toma a un usuario realizar cierta acción. Esto combinado con un análisis del "*frontend*" podría permitir identificar las posiciones idóneas para ciertos botones, por ejemplo.

Por otra parte la tabla "*service_types*" permite la flexibilidad de tener diferentes tipos de servicios, como por ejemplo basureros, puntos de reciclaje, etc. Esto da flexibilidad al sistema, para que pueda utilizarse con propósitos más generales. A su vez la tabla "*price_ranges*" permite guardar los rangos de precios posibles para cada tipo de servicio. Además, a través de sus relaciones con las tablas "*reports*" y "*services*", permite saber el rango de precio actual para un servicio o la votación de precio entregada en un reporte. Estas dos últimas tablas son las principales del sistema, pues almacenan los servicios reportados por los usuarios y los reportes (o valoraciones) realizadas por los

mismos. En la tabla "*reports*", se guarda el registro de cada vez que un usuario reporta un nuevo servicio o realiza una valoración de un servicio. El registro en la tabla "*services*" se crea junto con el reporte de nuevo servicio.

La tabla "*users*" tiene el registro de los usuarios en el sistema y la tabla "*roles*" tiene los tipos de roles de usuario. En este caso son dos: "Admin" y "Normal". Además está la tabla "*service_types*", que denota los tipos de reportes, esto quiere decir, que permite diferenciar cuándo se reporta un nuevo servicio de cuándo se califica un servicio.

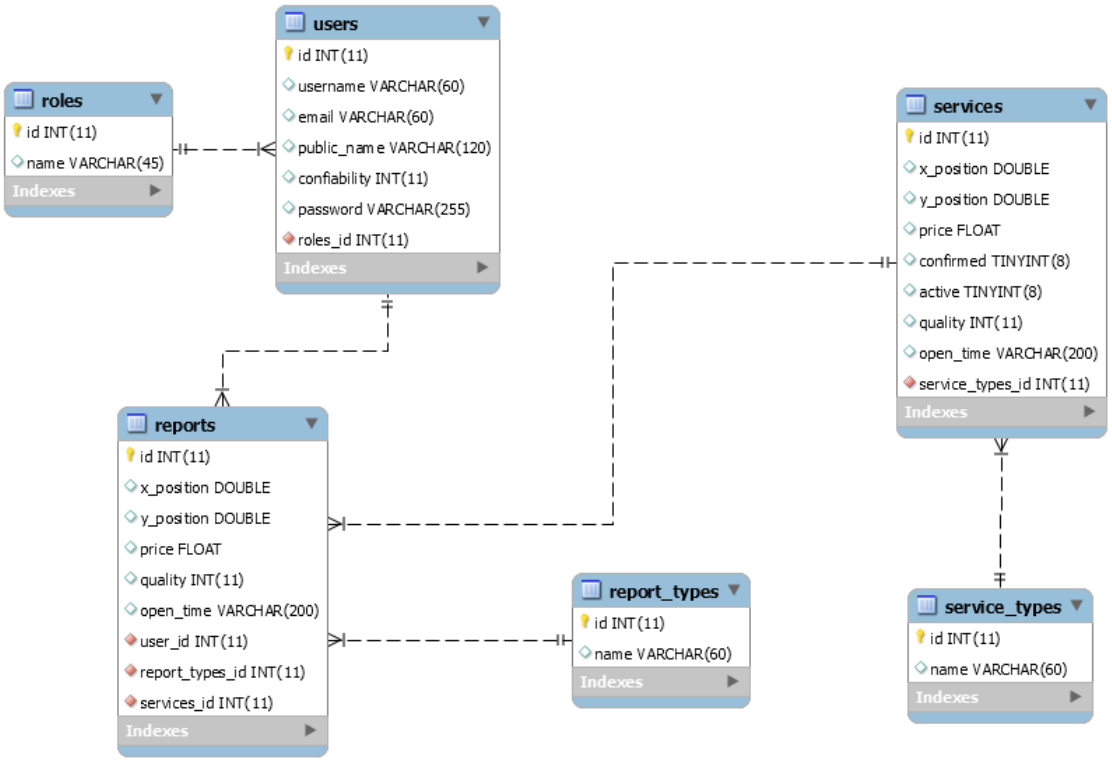


Figura 7.1: Modelo de Datos Inicial.
Fuente: Elaboración Propia.

7.2. Funcionalidades del sistema

En esta sección se explican las principales funcionalidades de la API, para mayor detalle de ellas o para ver todas las funcionalidades incluidas en el sistema, revisar el anexo A "Manual de configuración y uso API". La aplicación cuenta con seis micro servicios y un total de 34 métodos

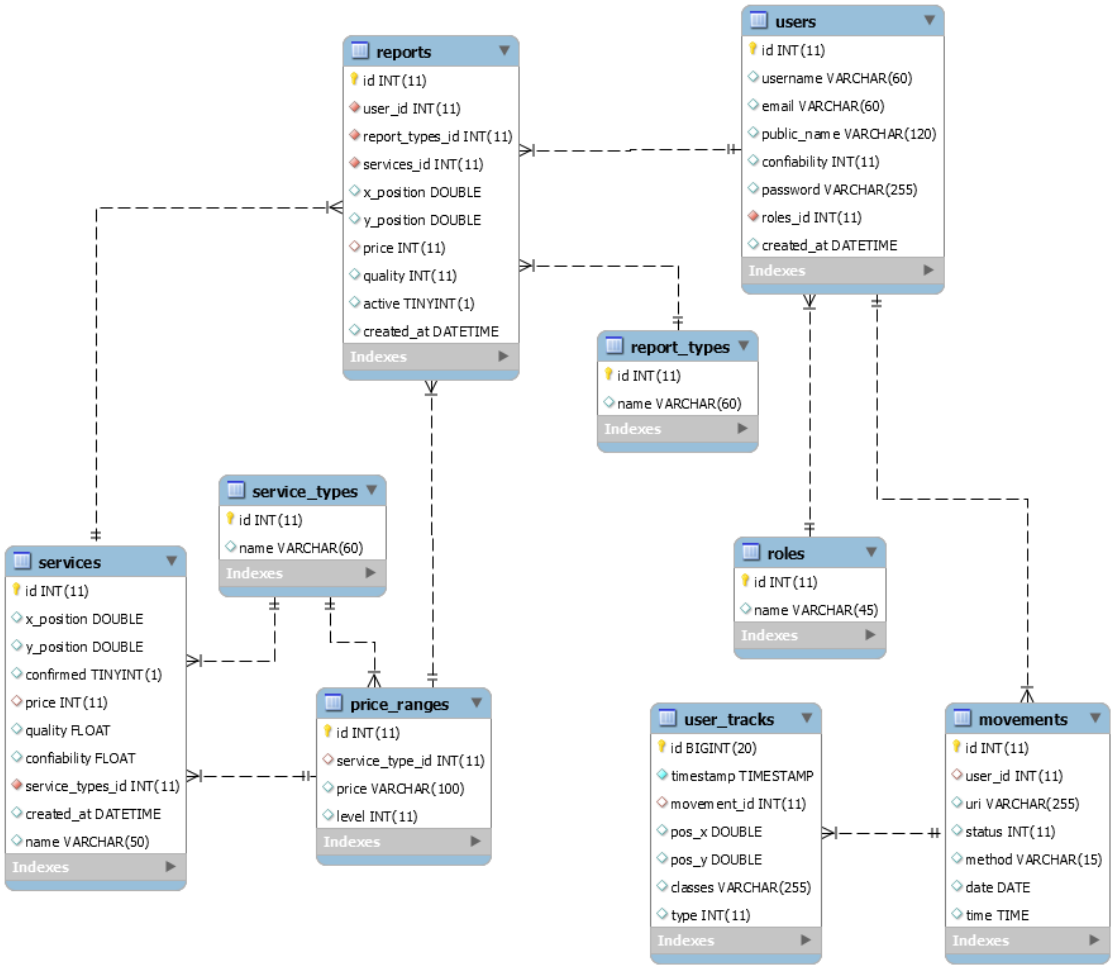


Figura 7.2: Modelo de Datos Final.
Fuente: Elaboración Propia.

que permiten interactuar con el sistema.

Los seis micro servicios son:

- **Usuarios (“*users*”):** Permite crear y traer la información de usuarios y además obtener un token de acceso para el sistema.
- **Reportes (“*reports*”):** Permite crear reportes de nuevos servicios o de valoración de los mismos.
- **Servicios (“*services*”):** Permite buscar servicios por ubicación y además obtener la información detallada de un servicio en específico.

- **Tipos de servicios (“*service_types*”):** Permite obtener, crear y eliminar tipos de servicios.
- **Rangos de precio (“*price_ranges*”):** Permite obtener la lista de rango de precios para los distontos tipos de servicios.
- **Analíticas (“*analytics*”):** Permite obtener información para hacer análisis del sitema, por ejemplo, cantidad de visitas o cantidad de movimientos para realizar cierta tarea.

7.2.1. Funcionalidades básicas del sistema

En esta sección se señalan y se da una pequeña descripción de algunas de las funcionalidades del sistema:

- **POST login**: Permite validarse como usuario y obtener un *token* de acceso que permite realizar otras llamadas que requieren de un usuario validado.
- **GET services find**: Permite localizar servicios disponibles en un determinado radio a partir de un punto en el mapa. Esta búsqueda puede filtrar resultados también por tipo de servicio y/o por nombre.
- **POST reports add new**: Crea un nuevo servicio en la posición indicada y del tipo de servicio señalado. Es la función que permite a los usuarios generar información de nuevos servicios.
- **GET service_type**: Trae la lista de tipos de servicios disponibles.

7.2.2. Funcionalidades de análisis dentro del sistema

En esta sección se muestran algunas funcionalidades que permiten realizar un análisis del sistema, todas las funciones requieren que los usuarios estén validados con rol de administrador. Sin embargo, esto es modificable desde el archivo de configuración A.1. Entre las funcionalidades disponibles están:

- **GET visits (day|month|year)**: Entrega las visitas realizadas al sistema ya sea agrupadas por día, mes o año. También se pueden sub-agrupar por usuario y por ruta http.

- **GET reports quantity (day|month|year)**: Entrega la cantidad de reportes realizados en el sistema ya sea agrupadas por día, mes o año, como también sub-agrupadas por usuario.
- **GET reports movements (day|month|year)**: Entrega la cantidad de movimientos promedio requeridos por un usuario para realizar un reporte, incluyendo (*clicks*, movimientos en el mapa y *hover* sobre elementos). Además incluye el tiempo promedio en segundos que toma realizar la acción. El promedio puede ser diario, mensual o anual y además permite agrupar los resultados por usuario.

8 | Validación

Este capítulo muestra las pruebas y validaciones realizadas para verificar el cumplimiento de los requerimeintos no funcionales señalados en la sección de objetivos (ver sección 3.2). Los requerimientos ahí presentados son: escalabilidad y reusabilidad.

8.1. Escalabilidad de carga: test de estrés

Para los test se utilizó una máquina con las siguientes características:

- Sistema Operativo: CentOS Linux 7.1.1503
- Procesador: Intel 1 núcleo de 2.5 [GHz]
- Memoria RAM: 2 [GB]
- Disco duro: SSD 20 [GB]
- Latencia de conexión: 27 [ms] promedio

Para medir escalabilidad de carga se utilizó un "*test de estrés*" que consiste en realizar múltiples llamadas al sistema en un lapso corto de tiempo para ver si es capaz de soportar la carga o si colapsa en algún punto.

Para realizar el test de estrés se utilizó la herramienta JMETER de Apache, que simuló el siguiente escenario: 100 usuarios al mismo tiempo, creando un servicio nuevo cada 5 segundos, 10 veces. Esto quiere decir crear 1.000 servicios nuevos en un lapso de 50 segundos. También se dejó en 5 segundos como tiempo máximo de espera. Esto quiere decir que cualquier petición que tomara

más de 5 segundos se consideraría como error. Se decidió utilizar la llamada para reportar un nuevo servicio (ver sección A.2.3.1), debido a que es una de las más complejas en el sistema y que requiere almacenar más información en la base de datos.

Como resultado, los 1.000 servicios fueron creados satisfactoriamente, el menor tiempo de respuesta fue de 131[ms], el mayor 4393[ms] y el promedio 2.924[ms], tal como se muestra en el gráfico 8.1. Esto muestra que si bien el servidor utilizado para las pruebas no posee muchos recursos, logró responder a todas las peticiones realizadas en un tiempo inferior a los 5 segundos. Aún así el tiempo promedio de respuesta fue de casi 3 segundos, tiempo que se puede considerar crítico para la tarea realizada.

Finalmente el resultado entregado por la prueba, es que el rendimiento promedio del sistema es de 1.694 operaciones por minuto, lo que significa que si un usuario realiza alrededor de 1 operación cada 10 segundos, entonces el sistema soportaría aproximadamente 284 usuarios concurrentes, lo que parece una gran cifra considerando las características del servidor utilizado.

En el gráfico mostrado en 8.1 aparecen puntos de 4 colores distintos, los azules representan el tiempo promedio que toman las operaciones, el color rojo representa la desviación estándar, el color verde el rendimiento de la aplicación y el color negro el tiempo de respuesta por llamada.

8.2. Reusabilidad

Que el sistema sea reusable, quiere decir que éste pueda ser reutilizado para otra aplicación con las mismas características, realizando la menor cantidad de cambios en el menor tiempo. O sea, utilizando el mínimo esfuerzo posible. Para ello la API es ideal, ya que por definición los servicios RESTful son reutilizables, puesto que permiten la integración con diferentes interfaces o cascarones, o la reutilización sólo de algunos micro servicios [1]. Esto se debe a que la API recibe y entrega información en formato estándar, permitiendo que actúe de forma independiente a la aplicación externa, incluso permitiendo trabajar con más de una a la vez. Por ejemplo, puede al mismo tiempo alimentar una página web y una aplicación móvil. Además considerando que cada micro servicio es

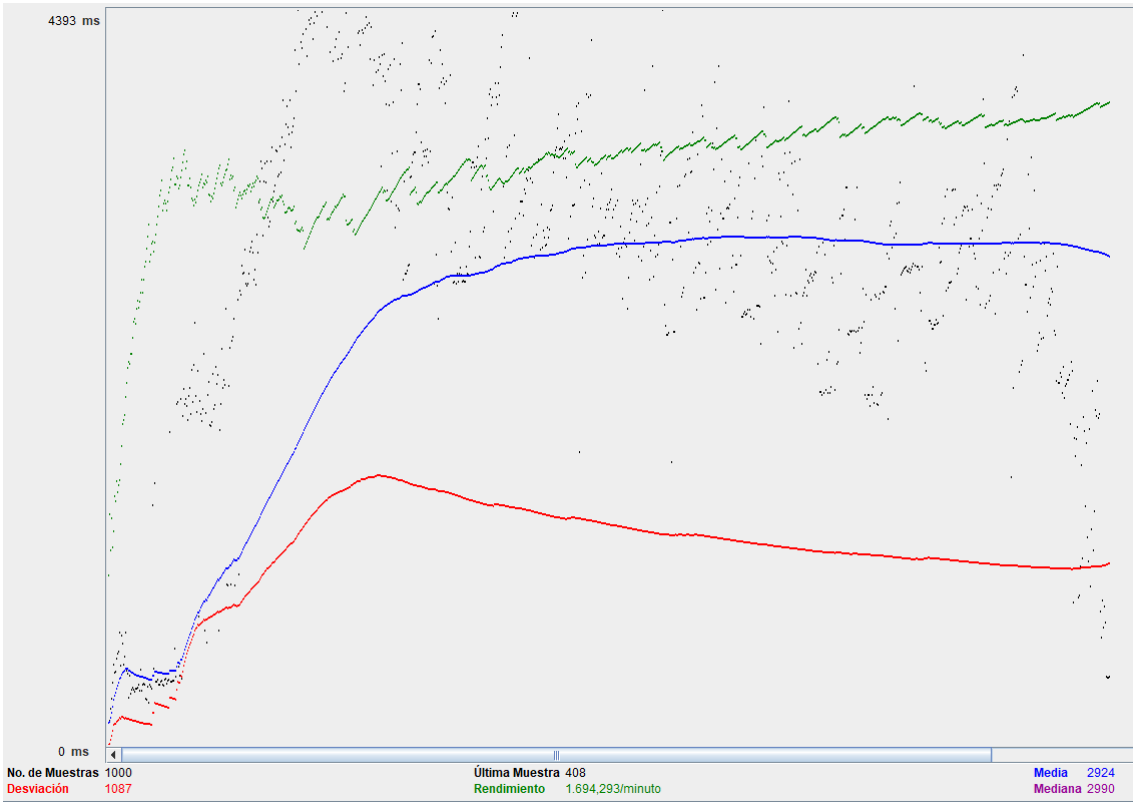


Figura 8.1: Gráfico prueba de estrés.
Elaborado con *Apache JMETER*.

independiente del otro, éstos se pueden tomar de forma individual para integrarlos a otros sistemas, tal como si fueran componentes.

8.3. Extensibilidad

La extensibilidad quiere decir que el sistema pueda ser extendido en funcionalidad de forma sencilla y sin afectar las funcionalidades ya existentes. Esto es posible gracias a la arquitectura de micro servicios, pues permite que sea sencillo agregar funcionalidades sin tener que realizar cambios en los micro servicios ya existentes, puesto que se pueden agregar nuevos servicios sin tocar una línea de código de los otros micro servicios [1]. Esto quiere decir que gracias a la arquitectura utilizada el sistema posee intrínsecamente la escalabilidad de extensibilidad horizontal.

Dado que el sistema posee como sub-arquitectura una estructura por capas, se puede decir que el sistema posee extensibilidad vertical. Esta arquitectura permite agregar capas superiores para

extender fácilmente la funcionalidad de forma vertical, para trabajar sin la necesidad de alterar las capas inferiores y de forma mínima las superiores [17].

Por lo anterior y según lo señalado por la literatura, se puede asegurar que dada la arquitectura utilizada en el proyecto, el sistema es extensible, tanto vertical como horizontalmente.

8.4. Adaptabilidad e interoperabilidad

La adaptabilidad quiere decir que el sistema es capaz de adaptarse a cambios en el entorno, con la menor necesidad de cambios y/o esfuerzo. Por otro lado, la interoperabilidad es la capacidad de interactuar con diferentes sistemas sin requerir ningún cambio. Ambos requerimientos no funcionales están cubiertos por la arquitectura de software, tal como se puede ver en la tabla (o matriz) de trazabilidad (ver tabla 6.3). Ahí se puede ver como la interoperabilidad está cubierta por ser una aplicación *RESTful*, ya que sin la necesidad de ningún cambio puede entablar comunicación con cualquier otro software que utilice el protocolo *HTTP*. A su vez, la adaptabilidad está cubierta por el uso del *framework*, *phalcon*. Esto debido a que su facilidad de uso y simplicidad, lo hacen sumamente adaptable a cambios.

9 | Conclusión

9.1. Resultados del proyecto

El resultado del proyecto fue una *API RESTFul* completa con cada etapa del desarrollo justificada y documentada a lo largo de este trabajo. Después del análisis de posibles lenguajes se decidió usar *PHP*, esto porque facilita la posibilidad de que en un futuro otros decidan tomar el proyecto para extenderlo, agregarle nuevas funcionalidades o simplemente crear un *frontend* que se comunique con éste. Esto se debe a que *PHP*, tal como se mostró en el proyecto, es un lenguaje sumamente popular, de fácil aprendizaje y de rápida ejecución.

También se creó una arquitectura que permite la escalabilidad y la extensibilidad del proyecto, esto posibilita que el proyecto pueda ser retomado en el futuro, ya sea para agregarle funcionalidades nuevas como para hacerlo crecer en capacidad de usuarios soportados. Esto gracias a su arquitectura de micro-servicios que permite modularizar completamente la aplicación, y que a su vez, permite agregar nuevas funcionalidades a través de nuevos micro-servicios. El modelo de base de datos da además la posibilidad de agregar nuevos tipos de servicios.

Si bien el proyecto se centró en la creación de un *backend* para trabajar con *crowdsourcing*, también incluye funcionalidades de análisis de datos, puesto que registra por completo los movimientos de los usuarios y los vincula a las llamadas realizadas en el sistema. Esto permite analizar el comportamiento de los usuarios, ya que registra información de la cantidad de movimientos realizados en promedio para cierta acción y la duración promedio de esos movimientos. De esta forma, se puede medir más objetivamente los cambios en la capa *frontend* para cuantificar por ejemplo, la

usabilidad dependiendo de si un botón está en cierta posición o en otra, si se cambia su color o si definitivamente se elimina.

En el trabajo se presentó una matriz de trazabilidad como herramienta para para validar que los componentes de arquitectura utilizados apoyaran y permitieran cumplir con los requerimientos no funcionales. Esta herramienta resultó sumamente valiosa, pues permite ver cómo cada componente afecta distintos puntos críticos a considerar dentro del proyecto. Esto posibilita una mejor toma de decisiones y una mayor consciencia a la hora de realizar cambios.

En el trabajo también se explica a grandes rasgos el modelo de datos. En este caso el modelo de datos surge (tal como sucede muchas veces) como una extensión del modelo de dominio, que resulta en base, principalmente a una refinación, pues implicó tomar algunas decisiones de diseño, como por ejemplo, tener una tabla de roles de usuario en vez de que el rol sólo sea un campo. Estas decisiones de diseño se tomaron considerando la funcionalidad del sistema, pues había que asegurarse de que el modelo de datos permitiera registrar y obtener todos los datos requeridos.

9.2. Continuidad del proyecto

Una de las principales características del proyecto es que es extensible, adaptable y reusable. Esto se debe a que su arquitectura de micro servicios permite que la aplicación se pueda extender de forma sencilla agregando nuevos micro servicios con nuevas funcionalidades. Por otro lado es adaptable y reusable, pues es totalmente independiente de la capa externa, por lo que funciona para cualquier tipo de interfaz, ya sea una aplicación de escritorio, una aplicación móvil o una página web, permitiendo incluso tener diferentes aplicaciones concentradas trabajando con un mismo núcleo. Esto permite que la *API* pueda ser retomada en el futuro, ya sea para extenderla como también para crearle una determinada capa externa.

Ya que la *API* contiene un servicio de análisis, permite que futuros estudiantes y profesionales puedan trabajar con ella, probando por ejemplo, la usabilidad del sistema en función de los cambios de su capa externa, o diseñando un cascarón que permita la interactividad con el sistema.

Otra característica sumamente importante, es que la arquitectura interna de la aplicación es por capas, esto permite que cualquiera que quiera agregar mejoras dentro de la aplicación, lo pueda hacer con mayor facilidad. Esto se debe a que una arquitectura por capas es más fácil de leer, pues cada capa tiene sus propias funcionalidades permitiendo ir directo al punto deseado, además es más fácil de *debuguear*, pues los errores quedan encapsulados en su capa, permitiendo identificarlos de forma más eficiente. Esto quiere decir que la aplicación es de fácil mantención y que además permite leer el código con mayor facilidad. Además se incluye el anexo A con la explicación de configuración del sistema, para así facilitar la integración y modificación del proyecto.

Si bien el proyecto se concibió inicialmente con servicios higiénicos en el sector de Valparaíso y Viña del Mar, dada su estructura fácilmente se puede trabajar con otro tipo de elementos dentro de la ciudad, para por ejemplo generar un informe general sobre el estado de las calles de una ciudad, las deficiencias en el alumbrado público, semáforos dañados, árboles públicos que requieren podarse o señaléticas faltantes. Todo a base de la información entregada por los usuarios, esto podría permitir a un municipio o a quien sea el encargado, ahorrar dinero público en fiscalizadores, inspectores u otro encargado de documentar este tipo de fallas. Estos son sólo algunos de los ejemplos de la utilidad de este sistema base, que si bien por si solo no es para que un usuario final lo utilice, es una base sólida para continuar trabajando.

9.3. Objetivos del proyecto

El objetivo principal del proyecto (diseñar, construir y evaluar un software que fuera escalable y reutilizable), se cumplió en su totalidad. Se construyó una *API RESTful*, que permite almacenar y consultar datos obtenidos vía *crowdsourcing*. Gracias a las comprobaciones realizadas se pudo verificar el cumplimiento de los requerimientos no funcionales descritos en este objetivo. Estas validaciones (test de estrés, matriz de trazabilidad, etc), resultaron fundamentales para verificar el cumplimiento, debido a que muchas veces, los requerimientos no funcionales son difíciles de medir. Por ejemplo, si un requerimiento no funcional es la rapidez de respuesta, entonces basta con definir un punto de aceptación (tiempo máximo de demora en responder) para verificar que se cumpla.

Sin embargo, si el requerimiento es que el sistema sea extensible y adaptable, la verificación es un poco más compleja. En este caso se utilizó una matriz de trazabilidad de la arquitectura del sistema, pera ver si ésta brinda las características requeridas para el cumplimientos de los requerimientos no funcionales.

Por otra parte el test de estrés permitió medir de forma objetiva el rendimiento del software, bajo ciertas condiciones de hardware, lo que permitirá que en el futuro, quien lo desee pueda realizar un benchmark para compararlo con otras posibles soluciones.

Otro punto importante son los problemas a solucionar descritos en la sección 2.2, donde se explica la importancia de superar las dificultades para que el sistema sea escalable, cosa que ya se mencionó, que el sistema asegure que los datos sean confiables y que los datos se analicen de forma correcta. Para los dos últimos puntos mencionados cabe decir que, el sistema utiliza un método de ponderación de confiabilidad de los datos en base a la cantidad de personas que la ratifiquen, y es en base a esos datos que se realizan análisis. Si bien esta medida pretende sopesar a los usuarios mal intencionados, siempre se puede burlar al sistema. Esta vulnerabilidad se puede controlar con los usuarios administradores, que pueden actuar como filtros finales para cuando la ponderación de los otros usuarios se vea sobrepasada.

9.4. Toma de requerimientos

Si bien a lo largo de este documento, el tema de los requerimientos se toca sólo en el capítulo "Metodología", mostrando los pasos utilizados para la toma de éstos, este proceso resultó ser complejo dadas las condiciones de los posibles *"clientes"* (quienes construyen una capa externa para la *API*), como de los usuarios (quienes reportan y consultan datos en la *API*). Esto considerando que el software debe brindar una forma simple y efectiva para que otros sistemas se comuniquen con él y además, brindar toda la funcionalidad requerida por los usuarios, ambos desconocidos al comienzo.

En este proceso de toma de requerimientos resultó complejo dar un límite a las iteraciones, esto se debió a que el sistema siempre puede ser mejorado, pero debe existir un límite claro de las

funcionalidades a crear. Además, para la creación de los requerimientos, se tuvo que trabajar con los objetivos del proyecto, ya que estos definieron los requerimientos no funcionales.

9.5. Desarrollo profesional

Este proyecto permitió demostrar habilidades ingenieriles, ya que contempla el proceso completo del desarrollo de un proyecto de software, pasando por las etapas de análisis, desarrollo y documentación del mismo. Esto hizo que se tomaran diferentes roles para cada paso en la confección, pasando desde ser desarrollador-programador o analista de software, a ser arquitecto de software, alternándose dependiendo de la etapa del proyecto. También significó realizar un trabajo metódico y completamente justificado. Cada decisión relacionada con el desarrollo del proyecto se analizó, se impusieron criterios bien definidos, se agruparon las mejores posibilidades y finalmente se optó por una, o la mezcla de algunas.

Bibliografía

[1] Subbu Allamaraju. *RESTful: web services: Solutions for improving scalability and simplicity*. O'Reilly Media, Inc., primera edition, 2010. 8.2, 8.3

[2] Michael Arias Chaves. La ingeniería de requerimientos y su importancia en el desarrollo de proyectos de software. *InterSedes: Revista de las Sedes Regionales*, 6(10), 2005. 5.1

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000. 4.2.2

[4] Enrique Estellés-Arolas and Fernando González-Ladrón-de Guevara. Towards an integrated crowdsourcing definition. *Journal of Information science*, 38(2):189–200, 2012. 2.2

[5] David Flanagan. *JavaScript: The definitive guide: Activate your web pages*. O'Reilly Media, Inc., sexta edition, 2011. 6.1

[6] José Carlos García Monsálvez. Python como primer lenguaje de programación textual en la enseñanza secundaria. *Teoría de la Educación. Educación y Cultura en la Sociedad de la Información*, 18(2), 2017. 6.1

[7] Instituto Nacional de Estadísticas. Actualización 2002-2012 y proyección de población 2012 - 2020: Total región de valparaíso: por comuna y sexo. http://www.inevalparaiso.cl/archivos/files/xls/Censos/2012/Poblacion%20por%20Comunas%20Valparaiso.xls, 2013. Accesado: 2017-10-28. 5.1

[8] Jeff Kaflin. The five most in-demand coding languages. https://news.netcraft.com/archives/2017/09/11/september-2017-web-server-survey.html, 2017. Accesado: 2017-11-04. 5.2

[9] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. pages 151–160, 2011. 4.1

[10] Lokesh Kumar, Shalini Rajawat, and Krati Joshi. Comparative analysis of nosql (mongodb) with mysql database. *International Journal of Modern Trends in Engineering and Research*, 2(5):120–127, 2015. 6.3.2

[11] Mozilla Inc. Mdn web docs: Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2017. Accesado: 2017-10-14. 6.1

[12] Gavin Mulligan and Denis Gračanin. A comparison of soap and rest implementations of a service based interaction independence middleware framework. pages 1423–1432, 2009. 4.2.1, 4.2.2

[13] Netcraft. September 2017 web server survey. https://www.forbes.com/sites/jeffkauflin/2017/05/12/the-five-most-in-demand-coding-languages/, 2017. Accesado: 2017-12-05. 6.3.1

[14] Robin Nixon. *Learning PHP, MySQL, JavaScript, and CSS: A step-by-step guide to creating dynamic websites*. .º'Reilly Media, Inc.", 2012. 6.1

[15] Phalcon. Un framework full-stack de php entregado como una extensión de c. https://phalconphp.com/es/, 2017. Accesado: 2017-10-15. 6.2.1

[16] Natalya Prokofyeva and Victoria Boltunova. Analysis and practical application of php frameworks in development of web information systems. *Procedia Computer Science*, 104:51–56, 2017. 6.1

[17] Marck Richards. *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*. TSbu, primera edition, 2015. 4.2.1, 6.3.1, 8.3

[18] Jone Samra. Thesis project: Comparing performance of plain php and four of its popular frameworks, 2015. Linnaeus University, Sweden. (document), 6.2.1, 6.1

[19] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice: Hypermedia and sistem architecture*. O'Reilly Media, Inc., primera edition, 2010. 4.1

[20] Estratégias y Negocios. La historia de waze contada por uri levine. http://www.estrategiaynegocios.net/lasclavesdeldia/913286-330/la-historia-de-waze-contada-por-uri-levine, 2015. Accesado: 2017-12-08. 4.1

A | Manual de configuración y uso API

A.1. Configuración

El archivo de configuración permite establecer las configuraciones básicas del sistema, como por ejemplo la conexión a la base de datos, tiempos de duración de tokens y llaves hash. En el archivo config.php (ver sección A.1.1) se pueden encontrar las siguientes configuraciones:

- **Base de datos:** De las línea 5 a la 12 está la configuración de la base de datos donde aparecen los siguientes campos:
  - **adapter**: Adaptador de conexión para los distintos motores de base de datos. Para este proyecto 'Mysql'.
  - **host**: Ip o nombre de dominio donde se encuentra el motor de base de datos. Puede ser 'localhost' en caso de que esté en la misma máquina que la aplicación.
  - **port**: Puerto de conexión con la base de datos. 'Mysql' usa por defecto el puerto 3306.
  - **username**: Nombre de usuario con el que se ingresará a la base de datos.
  - **password**: Contraseña correspondiente al 'username' para autenticarse.
  - **dbname**: Nombre de la base de datos con la que se establece la conexión.
- **Aplicación:** De las línea 14 a la 23 está la configuración básica para la aplicación, que parte por las rutas de las distintas carpetas del proyecto. De esta sección los campos interesante son:
  - **baseUri**: Directorio/ruta base donde se encentra el proyecto, puede ser una ip o un dominio.

  - **publicUrl**: Similar a 'baseUri', sin embargo es la ip pública con la que se accederá a la aplicación.
  - **cryptSalt**: Texto utilizado para generar la encriptación de contraseñas. Se recomienda que se cree de forma aleatoria, con un largo de más de sesenta caracteres y que mezcle letras mayúsculas y minúsculas, números, y signos en general.

- **Token para login**: De las líneas 25 a 29 aparece la configuración requerida para generar y validar las autenticaciones de los usuarios. Los campos editables son:
  - **hashkey**: Texto utilizado para generar la encriptación de los tokens. Se recomienda que se cree de forma aleatoria, con un largo de más de sesenta caracteres y que mezcle letras mayúsculas y minúsculas, números, y signos en general.
  - **tokenExpiration**: Tiempo en segundos que dura la validez de un token. Fuerza a que después del tiempo estimado los usuarios deban volver a autenticarse.
  - **protected**: Booleano que enciende y apaga la protección por token. Estando en 'false' no es necesario autentificarse. Se recomienda dejarlo siempre en 'true'.

- **Tiempo máximo de edición**: En la línea 31 aparece la variable 'max_edition_time' que es el tiempo en segundo máximo para permitir la edición de un elemento.

- **Tiempo para votar**: En la línea 32 aparece la variable 'votation_delay' que determina el tiempo mínimo en segundos que hay que esperar para poder volver a valorar un mismo servicio.

- **Rutas de libre acceso**: Este es un array en la línea 34 con todas las rutas a las que se podrá acceder sin la necesidad de un token válido. Para armar las rutas se utiliza como primera clave del array, luego el tipo de método de conexión, luego las rutas a de libre acceso. el carácter '*' señala que son todas las funciones.

- **Rutas para admin**: Este es un array en la línea 57 con todas las rutas que requieren de un usuario administrador. Para armar las rutas se utiliza como primera clave del array, luego el tipo de método de conexión, luego las rutas a de acceso limitado administrador. el carácter '*' señala que son todas las funciones.

A.1.1. Archivo de configuración

```php
<?php

return new \Phalcon\Config(
    [
        'database' => [
            'adapter'           => 'Mysql',
            'host'              => '',
            'port'              => ,
            'username'          => '',
            'password'          => '',
            'dbname'            => '',
        ],

        'application' => [
            'controllersDir'    => APP_DIR.'/Controllers/',
            'modelsDir'         => APP_DIR.'/models/',
            'businessDir'       => APP_DIR.'/business/',
            'helpersDir'        => APP_DIR.'/helpers/',
            'exceptionsDir'     => APP_DIR.'/Exceptions/',
            'baseUri'           => "/apimemoria/",
            'publicUrl'         => "/apimemoria/",
            'cryptSalt'         => '',
        ],

        'jwt' => [
            'hashkey'           => '',
            'tokenExpiration'   => '', //tiempo en [s]
            'protected'         => true
        ],

```

```
31          'max_edition_time' => '', //tiempo en [s]
32          'votation_delay' => '', //tiempo en [s]
33
34          'noLoginRequired' => [
35              'users' => [
36                  'post' => [
37                      '*' => true
38                  ]
39              ],
40              'services' => [
41                  'get' => [
42                      '*' => true
43                  ]
44              ],
45              'service_types' => [
46                  'get' => [
47                      '*' => true
48                  ]
49              ],
50              'price_ranges' => [
51                  'get' => [
52                      '*' => true
53                  ]
54              ]
55          ],
56
57          'adminLoginRequired' => [
58              'service_types' => [
59                  'post' => [
60                      '*' => true
61                  ],
62                  'put' => [
```

```
63                      '*' => true
64                  ],
65                  'delete' => [
66                      '*' => true
67                  ]
68              ],
69              'analitics' => [
70                  'get' => [
71                      '*' => true
72                  ]
73              ]
74          ]
75      ]
76  );
```

A.2. Funcionalidades

Para efectos de esta sección las llamadas de prueba mostradas se realizaron con la herramienta **Postman**[29]. Esta herramienta permite realizar llamadas a aplicaciones **RESTFUL** mediante el protocolo **HTTP**.

A.2.1. Usuarios

A.2.1.1. GET /users/:id:

Trae la información referente a un usuario.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - id: Número identificador del usuario. Obligatorio.
- **Respuesta:**

[29] https://www.getpostman.com/postman

- id: Número identificador del usuario.
- username: Nombre de usuario.
- email
- public_name: Nombre público visible para otros usuarios.
- confiability: Porcentaje de confiabilidad que el sistema le otorga al usuario.
- password: contraseña encriptada. Si se posee el criptSalt se puede validar si otro string es la contraseña. Virtualmente indescifrable.
- roles_id: Id del rol al que pertenece el usuario.
- created_at: Fecha y hora de creación del usuario.

A.2.1.2. POST /users

Registra un nuevo usuario en el sistema.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - username: Nombre de usuario. Obligatorio.
  - email. Obligatorio.
  - public_name: Nombre público visible para otros usuarios. Obligatorio.
  - password: Contraseña para el nuevo usuario. Obligatorio.
- **Respuesta:**
  - id: Número identificador del usuario.

A.2.1.3. PUT /users/:id:

Trae la información referente a un usuario.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**

- id: Número identificador del usuario. Obligatorio.
- username: Nombre de usuario.
- email
- public_name: Nombre público visible para otros usuarios.
- password: nueva contraseña

- **Respuesta:**
  - true

A.2.1.4. POST /users/login

Valida un usuario del sistema y entrega un "*token*" que sirve para realizar llamadas al sistema que requieren de un usuario validado.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - username: Nombre del usuario a validarse. Obligatorio.
  - password: Contraseña secreta del usuario. Obligatorio.
- **Respuesta:**
  - token: Texto alfanumérico que permite realizar llamadas que requieren un usuario validado. El token se envía como **header** con nombre *accesstoken*.
  - id: Número identificador del usuario.

A.2.2. Servicios

A.2.2.1. GET /services/find

Trae una lista de servicios en un determinado radio según los parametros de búsqueda.

- **Requisitos:** Ninguno.

- **Parámetros:**
  - position_x: Decimal de longitud de la posición central para la búsqueda. Obligatorio.
  - position_y: Decimal de latitud de la posición central para la búsqueda. Obligatorio.
  - radius: Decimal radio de búsqueda. Opcional. Por defecto es 11.12 [Kms].
  - service_type: Entero identificador del tipo de servicio. Opcional. Por defecto busca todos.
  - name: Texto nombre de servicio. Opcional. Por defecto bsuca todos.
- **Respuesta:**
  - array: Arreglo de servicios
    - id: Número identificador del servicio.
    - x_position: Decimal de longitud de la posición del servicio.
    - y_position: Decimal de latitud de la posición del servicio.
    - confirmed: Binario de largo 1. Posee valor 1 si el servicio ha sido confirmado por otra persona aparte del creador.
    - price: Rango de precio votado por la mayoría.
    - price_level: Entero nivel de precio. Permite determinar que tan económico es el servicio, mientras menor sea, más económico se considera el servicio.
    - quality: Decimal nota del servicio. Rango 1 a 5, donde 5 es excelente y 1 muy deficiente.
    - confiability: Decimal confiabilidad del servicio. Representa el porcentage de probabilidades de encontrar el servicio disponible.
    - service_types_id: Número identificador del tipo de servicio.
    - created_at: Fecha en formato "YYYY-MM-DD hh:mm:ss' de creación del servicio.
    - name: Texto nombre del servicio.

A.2.2.2. GET /services/:id:

Trae un servicio según su número identificador.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - id: Número identificador del servicio. Obligatorio.
- **Respuesta:**
  - id: Número identificador del servicio.
  - x_position: Decimal de longitud de la posición del servicio.
  - y_position: Decimal de latitud de la posición del servicio.
  - confirmed: Binario de largo 1. Posee valor 1 si el servicio ha sido confirmado por otra persona aparte del creador.
  - price: Rango de precio votado por la mayoría.
  - price_level: Entero nivel de precio. Permite determinar que tan económico es el servicio, mientras menor sea, más económico se considera el servicio.
  - quality: Decimal nota del servicio. Rango 1 a 5, donde 5 es excelente y 1 muy deficiente.
  - confiability: Decimal confiabilidad del servicio. Representa el porcentage de probabilidades de encontrar el servicio disponible.
  - service_types_id: Número identificador del tipo de servicio.
  - created_at: Fecha en formato "YYYY-MM-DD hh:mm:ss' de creación del servicio.
  - name: Texto nombre del servicio.

A.2.2.3. DELETE /services/:id:

Elimina un servicio según su número identificador. Existe un tiempo máximo para la eliminación por parte del creador. Sólo pueden eliminar el creador y el administrador del sistema.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - id: Número identificador del servicio. Obligatorio.

- **Respuesta:**
  - true

A.2.3. Reportes

A.2.3.1. POST /reports/addnew

Crea un nuevo servicio con la información entregada.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - service_type_id: Número identificador del tipo de servicio. Obligatorio.
  - x_position: Decimal de longitud de la posición del servicio. Obligatorio.
  - y_position: Decimal de latitud de la posición del servicio. Obligatorio.
  - name: Texto nombre del servicio. Obligatorio.
  - price: Número identificador del rango de precio. Por defecto es nulo.
  - quality: Entero de 1 a 5 con la calidad del servicio. Por defecto es nulo.
  - active: Binario de largo 1. Determina si el servicio estaba activo. Por defecto es 0.
- **Respuesta:**
  - id: Número identificador del servicio recién creado.

A.2.3.2. POST /reports/add

Crea un nuevo reporte acerca de un servicio con la información entregada.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - services_id: Número identificador del servicio. Obligatorio.
  - x_position: Decimal de longitud de la posición del servicio. Opcional.

- y_position: Decimal de latitud de la posición del servicio. Opcional.
- price: Número identificador del rango de precio. Opcional.
- quality: Entero de 1 a 5 con la calidad del servicio. Opcional.
- active: Binario de largo 1. Determina si el servicio estaba activo.Opcional.

- **Respuesta:**
  - id: Número identificador del reporte recién creado.

A.2.3.3. DELETE /reports/:id:

Elimina un reporte según su número identificador. Existe un tiempo máximo para la eliminación por parte del creador. Sólo pueden eliminar el creador y el administrador del sistema.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - id: Número identificador del reporte. Obligatorio.
- **Respuesta:**
  - true

A.2.4. Tipos de servicios

A.2.4.1. POST /service_types/

Crea un nuevo tipo de servicio.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - name: Nombre del tipo de servicio. Obligatorio.
- **Respuesta:**
  - id: Número identificador del tipo de servicio.

A.2.4.2. GET /service_types/

Trae una lista de tipos de servicio.

- **Requisitos:** Ninguno.
- **Parámetros:** Ninguno.
- **Respuesta:**
  - array: Arreglo de tipos de servicios
    - id: Número identificador del tipo de servicio.
    - name: Nombre del tipo de servicio.

A.2.4.3. GET /service_types/:id:

Trae un tipo de servicio por su número identificador.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - id: Número identificador del tipo de servicio. Obligatorio.
- **Respuesta:**
  - id: Número identificador del tipo de servicio.
  - name: Nombre del tipo de servicio.

A.2.4.4. PUT /service_types/:id:

Actualiza un tipo de servicio por su número identificador.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - id: Número identificador del tipo de servicio. Obligatorio.

- name: Nombre del tipo de servicio. Obligatorio.

- **Respuesta:**

  - true

A.2.4.5. DELETE /service_types/:id:

Elimina un tipo de servicio por su número identificador.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.

- **Parámetros:**

  - id: Número identificador del tipo de servicio. Obligatorio.

- **Respuesta:**

  - true

A.2.5. Rangos de precios

A.2.5.1. GET /price_ranges/

Trae la lista completa de rangos de precios

- **Requisitos:** Ninguno.

- **Parámetros:** Ninguno.

- **Respuesta:**

  - array: Arreglo de rangos de precios.
    - id: Número identificador del tipo de servicio.
    - service_type_id: Número identificador del tipo de servicio.
    - price: Texto de precio para mostrar.
    - level: Nivel de precio. Determina que tan caro es el precio, mientras menor es, más barato se considera el precio.

A.2.5.2. GET /price_ranges/service_tipe/:id:

Trae la lista de rangos de precios por tipo de servicio.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - id: Número identificador de tipo de servicio. Obligatorio.
- **Respuesta:**
  - array: Arreglo de rangos de precios.
    - id: Número identificador del tipo de servicio.
    - service_type_id: Número identificador del tipo de servicio.
    - price: Texto de precio para mostrar.
    - level: Nivel de precio. Determina que tan caro es el precio, mientras menor es, más barato se considera el precio.

A.2.5.3. GET /price_ranges/:id:

Trae un rango de precio por su número identificador.

- **Requisitos:** Ninguno.
- **Parámetros:**
  - id: Número identificador del rango de precio. Obligatorio.
- **Respuesta:**
  - id: Número identificador del tipo de servicio.
  - service_type_id: Número identificador del tipo de servicio.
  - price: Texto de precio para mostrar.
  - level: Nivel de precio. Determina que tan caro es el precio, mientras menor es, más barato se considera el precio.

A.2.6. Análisis del sistema

A.2.6.1. GET /analytics/visits/:date_group:

Trae la lista de visitas el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - date_group: Forma de agrupar las fechas, puede ser 'day', 'month' o 'year'. Obligatorio.
  - dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
  - status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.
  - exclude: Excluye de la búsqueda url que contengan la palabra señalada.
  - grouped: Forma en la que se agruparán los registros. Por defecto es sólo fechas. Los posibles valores son 'users' y 'routes'.
- **Respuesta (sin 'grouped'):**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
      - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
      - count: Cantidad de visitas para la fecha señalada arriba.
- **Respuesta (con 'grouped=users'):**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.

        - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
        - count: Cantidad de visitas para la fecha señalada arriba.

- **Respuesta (con 'grouped=routes'):**

    - array: Arreglo de fechas.
        - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
            - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
                - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
                - count: Cantidad de visitas para la fecha señalada arriba.

A.2.6.2. GET /analytics/reports/quantity/:date_group:

Trae la cantidad de reportes realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
    - date_group: Forma de agrupar las fechas, puede ser 'day', 'month' o 'year'. Obligatorio.
    - dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
    - status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.
    - exclude: Excluye de la búsqueda url que contengan la palabra señalada.
    - grouped: Forma en la que se agruparán los registros. Por defecto es sólo fechas. Los posibles valores son 'users' y 'routes'.
- **Respuesta (sin 'grouped'):**
    - array: Arreglo de fechas.
        - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.

        - ◇ user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
        - ◇ uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
        - ◇ count: Cantidad de reportes para la fecha señalada arriba.

- **Respuesta (con 'grouped=users'):**
    - array: Arreglo de fechas.
        - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
            - ◇ user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
                - ◇ uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
                - ◇ count: Cantidad de reportes para la fecha señalada arriba.

- **Respuesta (con 'grouped=routes'):**
    - array: Arreglo de fechas.
        - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
            - ◇ uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
                - ◇ user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
                - ◇ count: Cantidad de reportes para la fecha señalada arriba.

A.2.6.3. GET /analytics/reports/duration/:date_group:

Trae la lista de reportes realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
    - date_group: Forma de agrupar las fechas, puede ser 'day', 'month' o 'year'. Obligatorio.

- dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
- status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.
- exclude: Excluye de la búsqueda url que contengan la palabra señalada.

- **Respuesta:**

  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - user_id: Número identificador de usuario.
      - route: Ruta la que accedió para el creación del reporte.
      - status: Número de estado *HTTP* de la respuesta.
      - seconds: Cantidad de segúndos que el usuario tardo en realizar la acción.
      - clicks: Cantidad de clicks que realizó el usuario para realizar la acción.
      - hovers: Cantidad de objetos por los que el usuario pasó el cursor.
      - map_clicks: Cantidad de clicks que realizó el usuario sobre el mapa para realizar la acción.

A.2.6.4. GET /analytics/reports/movements/:date_group:

Trae los reportes promediados realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.

- **Parámetros:**

  - date_group: Forma de agrupar las fechas, puede ser 'day', 'month' o 'year'. Obligatorio.
  - dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
  - status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.

- exclude: Excluye de la búsqueda url que contengan la palabra señalada.

- **Respuesta:**

  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - ◇ route: Ruta la que accedió para el creación del reporte.
      - ◇ status: Número de estado *HTTP* de la respuesta.
      - ◇ seconds: Cantidad de segúndos promedio que los usuarios tardaron en realizar la acción.
      - ◇ clicks: Cantidad de clicks promeio que realizaron los usuarios para realizar la acción.
      - ◇ hovers: Cantidad de objetos promedio por los que los usuarios pasaron el cursor.
      - ◇ map_clicks: Cantidad de clicks promedio que realizaron los usuarios sobre el mapa para realizar la acción.

A.2.6.5. GET /analytics/reportsnew/quantity/:date_group:

Trae la cantidad de reportes servicios nuevos realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.

- **Parámetros:**

  - date_group: Forma de agrupar las fechas, puede ser 'day', 'month' o 'year'. Obligatorio.
  - dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
  - status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.
  - exclude: Excluye de la búsqueda url que contengan la palabra señalada.
  - grouped: Forma en la que se agruparán los registros. Por defecto es sólo fechas. Los posibles valores son 'users' y 'routes'.

- **Respuesta (sin 'grouped'):**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
      - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
      - count: Cantidad de reportes de servicios nuevos para la fecha señalada arriba.
- **Respuesta (con 'grouped=users'):**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
        - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
        - count: Cantidad de reportes de servicios nuevos para la fecha señalada arriba.
- **Respuesta (con 'grouped=routes'):**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - uri: Array de rutas a las que se accedieron en la fecha señalada arriba.
        - user_id: Array de números identificadores de usuarios que accedieron al sistema en la fecha señalada arriba.
        - count: Cantidad de reportes de servicios nuevos para la fecha señalada arriba.

A.2.6.6. GET /analytics/reportsnew/duration/:date_group:

Trae la lista de reportes de servicios nuevos realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.

- **Parámetros:**
  - date_group: Forma de agrupar las fechas, puede ser ‘day’, ‘month’ o ‘year’. Obligatorio.
  - dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato ‘YYYY-MM-DD’, ‘YYYY-MM-DD[,YYYY-MM-DD]’ o en el formato ‘YYYY-MM-DDtoYYYY-MM-DD’. Obligatorio.
  - status: Estado de la respuesta http entregada. Formaro ‘ddd[,ddd]’.
  - exclude: Excluye de la búsqueda url que contengan la palabra señalada.
- **Respuesta:**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato ‘YYYY[-MM[-DD]]’ según corresponda.
      - user_id: Número identificador de usuario.
      - route: Ruta la que accedió para el creación del reporte de servicio nuevo.
      - status: Número de estado *HTTP* de la respuesta.
      - seconds: Cantidad de segúndos que el usuario tardo en realizar la acción.
      - clicks: Cantidad de clicks que realizó el usuario para realizar la acción.
      - hovers: Cantidad de objetos por los que el usuario pasó el cursor.
      - map_clicks: Cantidad de clicks que realizó el usuario sobre el mapa para realizar la acción.

A.2.6.7. GET /analytics/reportsnew/movements/:date_group:

Trae los reportes de servicios nuevos promediados realizados en el sistema por día, mes o año.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - date_group: Forma de agrupar las fechas, puede ser ‘day’, ‘month’ o ‘year’. Obligatorio.

- dates: Fecha o rango de fechas para la búsqueda. Pueden ser en el formato 'YYYY-MM-DD', 'YYYY-MM-DD[,YYYY-MM-DD]' o en el formato 'YYYY-MM-DDtoYYYY-MM-DD'. Obligatorio.
- status: Estado de la respuesta http entregada. Formaro 'ddd[,ddd]'.
- exclude: Excluye de la búsqueda url que contengan la palabra señalada.

- **Respuesta:**
  - array: Arreglo de fechas.
    - fecha: Fecha en formato 'YYYY[-MM[-DD]]' según corresponda.
      - route: Ruta la que accedió para el creación del reporte de servicio nuevo.
      - status: Número de estado *HTTP* de la respuesta.
      - seconds: Cantidad de segúndos promedio que los usuarios tardaron en realizar la acción.
      - clicks: Cantidad de clicks promeio que realizaron los usuarios para realizar la acción.
      - hovers: Cantidad de objetos promedio por los que los usuarios pasaron el cursor.
      - map_clicks: Cantidad de clicks promedio que realizaron los usuarios sobre el mapa para realizar la acción.

A.2.6.8. GET /analytics/services/ranking

Trae un ranking de calidad con los mejores servicios.

- **Requisitos:** *header accesstoken* obtenido vía **POST /users/login**.
- **Parámetros:**
  - service_type_id: Tipo de servicio a considerar dentro del ranking. Opcional.
  - limit: Limita la cantidad de resultados a la indicada. Opcional.
- **Respuesta:**
  - array: Arreglo de servicios.

- id: Número identificador del servicio.
- x_position: Decimal de longitud de la posición del servicio.
- y_position: Decimal de latitud de la posición del servicio.
- price: Entero nivel de precio. Permite determinar que tan económico es el servicio, mientras menor sea, más económico se considera el servicio.
- qualities: Decimal nota del servicio. Rango 1 a 5, donde 5 es excelente y 1 muy deficiente.
- confiabilities: Decimal confiabilidad del servicio. Representa el porcentage de probabilidades de encontrar el servicio disponible.
- service_type_name: Texto nombre del tipo de servicio.
- service_name: Texto nombre del servicio.
- ponderation: Decimal que toma valores de 0 a 1 y que representa el puntaje obtenido por el servicio.