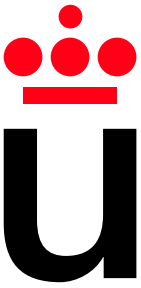


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso 2023-2024

Trabajo Fin de Grado

API - WARCRAFT RUMBLE

Autor: Santiago Ramos Gómez
Tutor: Nicolás Rodríguez Uribe

Agradecimientos

Muchas gracias a mi familia por el apoyo e insistencia constante. Espero que estéis satisfechos de que haya finalizado este proyecto de una vez. Habéis sido extremadamente pesados con el tema, comprensible.

También quiero dar las gracias a Nicolás Rodríguez Uribe por darme la oportunidad de presentar este proyecto y a Raúl Cabido Valladolid por su paciencia con los anteriores (y descartados) TFGs.

Por último quiero agradecer a Adrián Sanz Hervás y a Raúl de la Fuente Carrasco por proporcionarme la mejor experiencia universitaria que he tenido, habéis convertido asignaturas complejas en verdaderos paseos.

Resumen

Este proyecto está dedicado a proporcionar información acerca de las estadísticas de personajes de un videojuego (Warcraft Rumble), la aplicación cuenta con una base de datos en la que se almacena la información.

El servicio es público, por lo que personas de todo el mundo pueden utilizarlo, de tal manera que pueden procesar o estudiar los datos para desarrollar nuevos proyectos a partir de estos.

Palabras clave:

- Python
- API
- Framework
- FastAPI
- Base de datos no relacional
- MongoDB
- MongoDB Sharding
- BSON
- JSON
- HTTP Request
- HTTP Response
- Docker
- Docker Compose

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de códigos

1 Introducción

Este primer capítulo está dividido en 3 secciones. En la Sección 1.1, se da una perspectiva sobre Warcraft para entender de donde proviene este franquicia. En la Sección 1.2, se explica la motivación detrás del proyecto. Por último, en la Sección 1.3, se expone la estructura que mantiene el documento.

1.1. Contexto y alcance

Warcraft es una franquicia de videojuegos desarrollada por Blizzard Entertaiment. Esta hace su primera aparición de manera pública en 1994 con el videojuego Warcraft: Orcs & Humans, famoso RTS (Real Time Strategy) que impondría presencia en el mercado. Un año más tarde aparecería su secuela, Warcraft II: Tides of Darkess que posteriormente obtendría una expansión. En 2002 llegaría un título que es considerado por muchos el mejor juego RTS de la historia, Warcraft III: Reign of Chaos, juego que en 2003 obtendría, al igual que la segunda entrega, una expansión.

Finalmente, en 2004 aparecería posiblemente el videojuego más relevante de su historia y del género, World of Warcraft, el mayor MMO (multijugador masivo en linea) jamás conocido. Este, a diferencia de sus antecesores, obtuvo más de nueve expansiones, y a día de hoy se conserva como una de los títulos más jugados del mercado en línea.

Debido a la potente relevancia de la saga, surgieron otros juegos que abarcan este universo, como puede ser Hearthstone (2014), un juego de cartas con un sistema de combate por turnos y Warcraft Rumble (2023), un juego de móvil que

entra dentro de la categoría "Tower Defense" y estrategia, en el cual se deben gestionar tropas y recursos para destruir la base enemiga.

Es justamente este último título sobre el que se ha este proyecto, una API con la que se busca proporcionar información pública sobre el juego en formato JSON a cualquier persona que lo necesite.

1.2. Motivación

Cuando se trata de videojuegos competitivos en línea existen una gran cantidad de datos que pueden ser procesados para obtener estadísticas, sobre las cuales el propio juego se puede ver transformado. Un ejemplo podría ser un entorno donde varios jugadores realizan carreras con un coche, si uno de estos coches tiene estadísticas fuertemente superiores a otros y, en la mayoría de casos siempre queda primero, se puede utilizar una API como fuente de consultas y determinar una nueva manera de balancear el juego.

A nivel interno en la empresa puede existir una API privada para realizar estas consultas. En ciertas ocasiones existen API públicas donde datos concretos son proporcionados a todos los usuarios a partir de una plataforma.

Esto permite que se puedan desarrollar webs de terceros que se dedican a estudiar toda la información para poder presentarla de manera sencilla a otros jugadores.

Una API no se encuentra únicamente en el mundo de los videojuegos, está presente para obtener información del día a día como puede ser la temperatura en una localización, la hora, valores de la bolsa o para saber el Top 50 canciones más reproducidas de una plataforma de música.

En el caso de este proyecto se realizó en base a un videojuego debido a que la empresa creadora **©2024 Blizzard Entertainment, Inc.** no contaba con un servicio API para este juego en concreto.

A pesar de que existan API que no estén relacionadas con el mundo web, como podría ser Win32, a lo largo del presente se documento el termino API se referirá a las de tipo web.

1.3. Estructura del documento

Tras la introducción se presentan los objetivos, Sección donde se trata la descripción del problema junto con metodología utilizada y es concluido con un conjunto de explicaciones sobre las tecnologías elegidas y el razonamiento detrás

de estas decisiones.

La Sección 3.1 cuenta con un ambiente más técnico y se expone cada tecnología de manera aplicada, aquí se sigue un orden parcialmente cronológico respecto a su uso durante la creación del proyecto.

Para finalizar la Sección 4. muestra los resultados obtenidos y la Sección 5.2, las conclusiones y trabajos futuros.

En la región final del documento se presenta la bibliografía con todas las referencias, y el apéndice, donde se exhiben Códigos y Figuras de ciertas configuraciones.

2

Objetivos

Se busca proporcionar un recurso a la comunidad de jugadores de Warcraft Rumble. Para esto se debe de tener en cuenta que existen muchos datos que son privados e individuales y solo la entidad desarrolladora cuenta con los datos, por lo que la información que se entregue debe ser siempre de caracter público.

El acceso a la API oficial de este juego, proporcionado por **©2024 Blizzard Entertainment, Inc.** es inexistente a día de hoy, de tal manera que se tiene como meta brindar un producto similar que pueda cumplir una cantidad de requisitos suficientes para que otros desarrolladores se puedan beneficiar de la información aportada, y en consecuencia trabajar con esta.

A lo largo de este segundo capítulo se muestran aspectos como la descripción del problema en la Sección 2.1, la metodología empleada en la Sección 2.2 y finalmente, en la Sección 2.3 se muestran los diferentes conceptos y tecnologías estudiadas para llegar a una solución.

2.1. Descripción del problema

Debido a la falta de una API pública, el trabajo de los desarrolladores se complica. Recopilar datos de manera manual ralentiza mucho el proceso de creación de una aplicación, como puede ser un base de datos con interfaz gráfica detallada o AddOns para plataformas de Streaming.

El objetivo es entregar los datos de la manera más rápida, liviana y sencilla

posible, todo esto mientras se mantiene un fácil acceso y concurrencia, se proporciona una documentación apropiada y se garantiza que el servicio bajo disponible permanentemente y ser escalable.

2.2. Metodología empleada

Al ser un proyecto individual, se optó por seguir una metodología en cascada en vez de otras ágiles como Scrum o Kanban.

El enfoque de este modelo buscar ir progresando entre fases siempre que la anterior fase sea funcional. Y posteriormente comprobar que todo en conjunto funciona correctamente.

Este modelo se divide en fases y siguiendo las definiciones que se explican en un articulo de Forbes Advisor [1]:

- Requisitos: Se definen las ideas principales que el proyecto necesita.
- Diseño: Una vez que entiendes lo que el proyecto necesita, el siguiente paso es pensar en formas de crear soluciones que cumplan con esos requisitos, la idea detrás de esto es proponer diferentes ideas y más tarde ir filtrando cual es más eficiente según las necesidades. El ejemplo que proponen en Forbes Advisor, siguiendo lo que dijo el Dr. Chris Mattmann es: "Si necesitamos procesar un millón de peticiones de usuarios al día, lo más seguro es que implementar esta idea con únicamente un servidor no es suficiente, ya que este no aguantaría, por lo que posiblemente nuestro diseño indique que debemos tener varios servidores, ya que en el caso de que el primero deje de funcionar, se tienen nuevas opciones para poder cumplir la meta"
- Implementación: En esta fase se deben seleccionar el diseño que se crea más apropiado y buscar la tecnología para poder implementarlo. Aquí es totalmente factible investigar si este diseño verdaderamente satisface los requisitos.
- Verificación: Durante esta fase se toma la implementación que se ha creado en la fase anterior y se pone a prueba para ver si ya no solo teóricamente sino prácticamente cumple los requisitos. En el caso de que no se cumplan, se debe dar un paso hacia atrás e investigar cuál podría ser el problema.
- Mantenimiento: En la última fase se deben pensar maneras para mantener el proyecto a largo plazo, ya sean parches con corrección de errores, mejoras hardware o proponer un nuevo software.

Siguiendo estos pasos, se alcanzó el flujo de trabajo mostrado en la Figura 2.1.

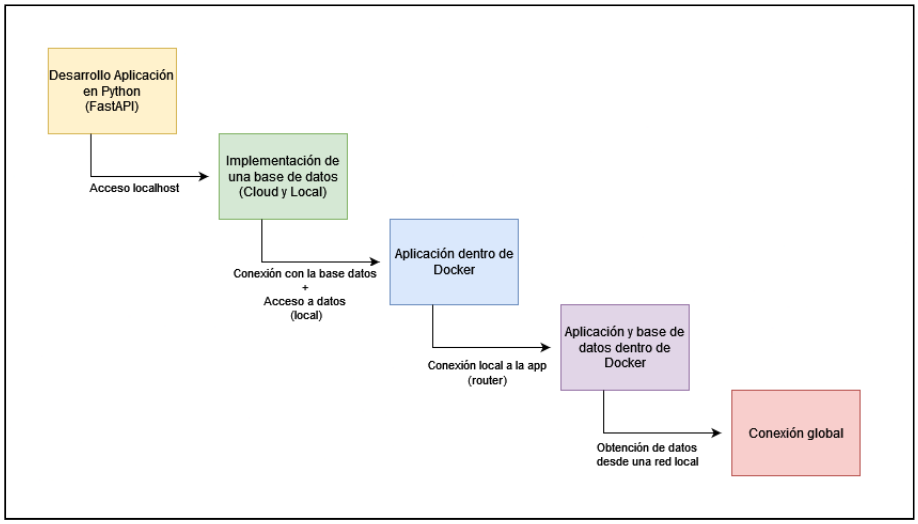


Figura 2.1: Metodología: Flujo de trabajo

2.3. Tecnologías y estudio de alternativas

En esta Sección se explican algunos conceptos sobre las API y la estructura del proyecto, y posteriormente las diferentes tecnologías utilizadas junto con comparativas de posibles herramientas.

2.3.1. API

Las siglas API provienen de Application Programming Interface, lo que en español se traduciría como Interfaz de Programación de Aplicaciones. Una definición sencilla podría ser: Conjunto de protocolos que permiten la comunicación entre dos o más entidades para compartir información. Este concepto se aprecia muy bien en la Figura 2.2.

Las APIs son herramientas que permiten la comunicación entre aplicaciones, aun cuando utilizan diferentes lenguajes. En ellas, los recursos representan datos disponibles, los endpoints son puntos de acceso para interactuar con esos datos, y las rutas son las direcciones para llegar a esos puntos.

Recursos

Son los elementos que representan datos o acciones que pueden ser manipulados a través de una API.

El acrónimo CRUD representa las acciones más comunes en una base de datos o aplicación web. Un desglose de estas sería:

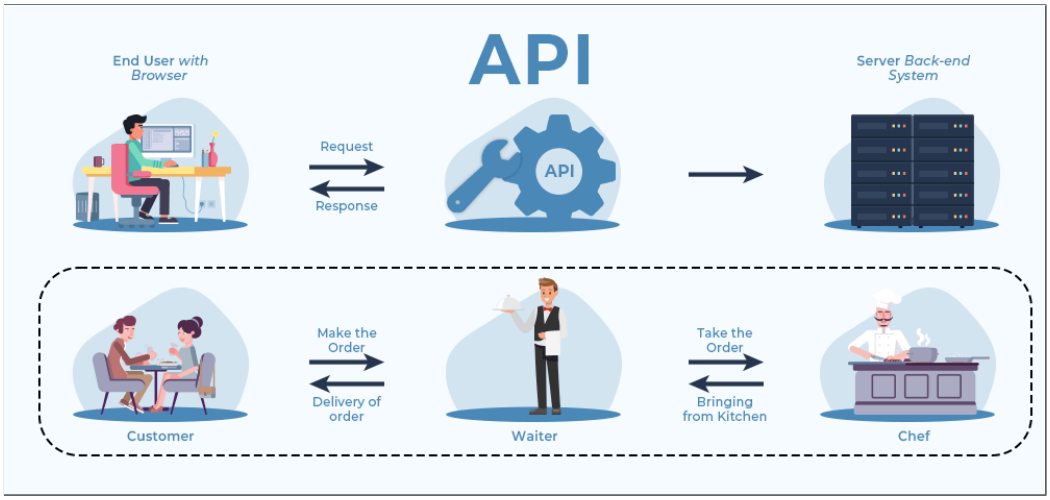


Figura 2.2: Visualización de qué es una API.
Fuente: **GeeksForGeeks**[2]

- Create: Utilizado para crear entidades (datos).
- Read: Se efectúa sobre los datos para obtener información de estos.
- Update: Se utiliza cuando se busca actualizar valores de entidades.
- Delete: Destinada a borrar entidades.

En una API se sigue esta filosofía, y las acciones que existen para tratar con los datos son las siguientes:

- POST: Se utiliza cuando se quiere añadir un dato.
- GET: Utilizado para obtener un dato.
- PUT: Dedicado a reemplazar un dato por completo.
- DELETE: Se usa cuando se quiere borrar un dato.
- PATCH: A diferencia de UPDATE, este se utiliza cuando no se quiere reemplazar un dato al completo, sino de manera parcial, como podría ser modificar uno de sus atributos.

MongoDB proporciona métodos para imitar acciones CRUD en el **manual oficial.**[3]

Routes

Dirección donde se encuentra un recurso. Esta puede contener métodos de una clase o pistas sobre las acciones que se quieren realizar respecto a un dato.

En color naranja se muestran posibles rutas:

- https://warcraftrumblejson.com/units/faction/alliance
- https://warcraftrumblejson.com/units/talents/1

Endpoints

Los endpoints son las posibles acciones que se pueden realizar sobre una ruta. No todas las rutas tienen métodos GET, POST, DELETE...Es una decisión del desarrollador.

Se podrían resumir como la suma de acción y ruta:

- GET https://warcraftrumblejson.com/units/faction/alliance
- DELETE https://warcraftrumblejson.com/units/1

2.3.2. Arquitectura

El diseño arquitectónico de una API Web puede tener diferentes opciones, algunas de ellas son:

- REST (Representational State Transfer): Arquitectura sencilla que cuenta con acciones/métodos basados en CRUD, como son GET, POST, DELETE, etc. Los recursos son identificados por URL y es fácilmente escalable.
- GraphQL: Similar a REST, la principal diferencia reside en la exposición de endpoints, mientras que REST suele contar con una amplia cantidad, GraphQL reduce las opciones (dependiendo del proyecto podría ser un único endpoint) y los clientes son los encargados de solicitar exactamente la información deseada.
- SOAP(Simple Object Access Protocol): Se trata de un protocolo algo más antiguo que los anteriores. La comunicación está basada en XML. Su uso se ha visto reducido a largo de los años por resultar menos competitiva que las anteriores opciones.

Como elección final se optó por utilizar REST, opción con menos sobrecarga y formato más limpio que SOAP. Otro punto importante es que al trabajar con un formato de respuestas en JSON facilitaba el flujo de comunicación entre las tecnologías.

Para más información sobre las diferencias entre REST y SOAP se recomienda el artículo propuesto por Amazon AWS [4].

2.3.3. Frameworks

Al traducir esta palabra al español se obtiene "marco de trabajo", este define cómo se deben seguir ciertas reglas.

Existen distintos tipos de frameworks, y dentro de estos cada uno puedo estar especializado más que otro en un campo, no son todos iguales. En lo que respecta a desarrollo de aplicaciones web, los dos más comunes son los Full-Stack y los Microframeworks.

Tipos de frameworks

- Full-stack framework: utilizado para realizar grandes proyectos, por lo general son una navaja suiza, incluyen todo lo necesario para no tener que depender tanto de módulos externos, la contraparte de esto es lo pesados que son y la poca flexibilidad a la hora de intentar trabajar de una manera distinta.
- Microframework: A diferencia de los primeros, está es su contraparte. Son perfectos para realizar proyectos a menor escala. Incluyen algunas funcionalidades esenciales que permiten construir una buena base de manera cómoda. Estos se suelen combinar con módulos externos (librerías) para añadir funcionalidades.

Opciones de frameworks

A pesar de comenzar el proyecto con Flask, se decidió utilizar FastAPI, ya que cuenta con asincronía de base y es capaz de manejar una gran cantidad de solicitudes por segundos (escenario más común en una API).

También cuenta con aspectos de seguridad como HTTP Basics, OAuth2 (JWT tokens) y claves API, parámetros de consultas o Cookies.

Adicionalmente, cuenta con otras funcionalidades como el tipado estático y la validación de datos gracias a Pydantic, aportando legibilidad, seguridad y mantenibilidad.

Django	Flask	FastAPI
Framework FullStack	Microframework	Microframework
Aplicaciones web complejas	Aplicaciones web sencillas	Especialmente diseñado para API rápidas
Nivel de abstracción alto, proporciona muchas funcionalidades integradas	Nivel de abstracción bajo, ofrece flexibilidad y control total	Nivel de abstracción medio, equilibra velocidad y funcionalidad
Basado en WSGI (se puede configurar con ASGI)	Basado en WSGI (se puede configurar con ASGI)	Basado en ASGI
Gran comunidad y amplia documentación disponible	Comunidad activa pero más pequeña en comparación con Django	Comunidad en crecimiento, con documentación clara y ejemplos

Tabla 2.1: Tabla comparativa entre Django, Flask y FastAPI

Por último, dispone de documentación autogenerada, lo que facilita la comprensión y permite que tanto usuarios como desarrolladores puedan realizar consultas utilizando una interfaz sencilla y colorida, por lo que se puede evitar el constante uso de herramientas como CURL.

FastAPI cuenta con una excelente guía donde se muestran los primeros paso en su **página oficial**[5].

2.3.4. JSON

Este formato es parecido a un diccionario en el mundo físico el cual contiene palabras y su definición. Aquí la estructura es similar, sin embargo, una palabra puede tener como definición otro diccionario mas reducido si fuera necesario. Los diccionarios se construyen a partir de llaves "{ }", estas se utilizan para delimitar cuando empiezan y terminan. Los tipos de datos (definiciones) más relevantes son strings, integers, float, boolean, array y Objects, al combinarlos se puede obtener algo como lo mostrado en el Código 2.1.

Código 2.1: Ejemplo JSON

```
"1": {
    "id": 1,
    "cost": 6,
    "name": "Abomination",
    "faction": "Undead",
    "type": "Troop",
    "description": "This Tanky mass of flesh and steel will
       Hook ranged enemies, drawing them into his Cleave attack
       .",
    "traits": [
        "Tank",
        "Hook",
        "Melee",
        "AoE"
    ],
    "stats": {
        "Area Damage": 170,
        "Health": 3400,
        "DPS": 68,
        "Attack Speed": 2.5,
        "Speed": "Slow"
    }
}
```

2.3.5. BSON

Los BSON (Binary JSON), traduciendo y atendiendo a la **información que proporciona MongoDB de manera oficial**[6] son muy parecidos a los JSON (JavaScript Object Notation), siendo los primeros ampliamente populares para tratar con datos a través de la web. A pesar de esto, JSON tiene ciertas desventajas en las bases de datos:

- JSON acepta un número muy limitado de tipos de datos, siendo estos los más simples. Por ejemplo no da soporte a datos como fechas o binario.
- Los objetos y propiedades en JSON no tienen un tamaño fijo, haciéndolos lentos de recorrer.

Para solventar este problema nacieron los BSON, una representación binaria de los datos en JSON, optimizados para ser más rápidos, ocupar menos espacio, ser más eficiente y permitir nuevos tipos de datos que antes no se podían debido a las limitaciones de JSON.

2.3.6. Docker

Se trata de una herramienta desarrollada por **Docker, Inc**, la cual permite encapsular aplicaciones junto con todas sus dependencias en un entorno portátil llamado contenedor. Esto permite poder ejecutar la aplicación en cualquier otro entorno sin tener que preocuparse por posibles errores de dependencias, versiones, compatibilidad...

La función de los contenedores Docker es generar un entorno donde únicamente se ejecute una aplicación con todas sus dependencias. Por ejemplo, este proyecto esta formado por una aplicación desarrollada en Python y una base de datos MongoDB, la cual está particionada y tiene varios nodos. La aplicación en si está en su contenedor, si este falla el problema se puede ubicar de manera sencilla, ya que está aislado. La base de datos al estar en otro contenedor se puede modificar, detenerla o ponerla en marcha sin tener que alterar la aplicación. Es importante quedarse con el concepto de aislamiento y que cada contenedor está dedicado a una única tarea.

Existen ocasiones donde es necesario ejecutar varios contenedores a la vez, ya que uno utiliza una funcionalidad de otro (aplicación realiza consultas a base de datos). Para esto se utiliza Docker Compose, utilidad para realizar despliegues coordinados. Docker Compose hace uso de un archivo de configuración YAML, donde se pueden establecer parámetros de arranque para cada contenedor.

A pesar de que desde fuera se parezcan a las máquinas virtuales, internamente son diferentes. Esto se puede comprobar en la Tabla 2.2.

Docker	Máquinas Virtuales
Más ligeros y rápidos de iniciar	Más pesados y lentos de iniciar
Comparten kernel del sistema host	Utilizan su propio sistema operativo
Recursos del sistema compartidos entre los contenedores (kernel del sistema host (kernel del SO, CPU, RAM, almacenamiento)	Recursos dedicados para cada VM (capa de virtualización sobre el hardware físico)
Escalabilidad sencilla	Escalabilidad pesada y costosa
Recursos bajo demanda	Cantidad fija y establecidas en la configuración de la imagen de la máquina virtual

Tabla 2.2: Tabla comparativa entre Docker y Máquinas Virtuales

2.3.7. Bases de datos

Antes de decidir una base de datos es importante saber con qué tipo de datos se va a trabajar. Dependiendo de la estructura y variabilidad de estos, puede que sea mejor un tipo de base de datos que otro.

Existen varias maneras de gestionar los datos, pero las más populares son SQL (Structured Query Languge) y NoSQL (Not only SQL). Algunas diferencias son expresadas en la Tabla 2.3.

Bases de datos relaciones vs no relacionales

SQL	NoSQL
Datos estructurados	Datos no estructurados o semi-estructurados
Los datos se almacenan en filas y columnas	Los datos se almacenan en documentos o colecciones
Esquema fijo, definido antes de tiempo	Esquema dinámico, flexible
Escalabilidad vertical (mejorar CPU, RAM, HDD)	Escalabilidad horizontal (distribuir carga trabajo y datos, múltiples servidores)
Suele ser más costoso debido a las mejoras hardware y licencias	Hardware de menor costo y se añaden y eliminan nodos según se necesite
Menor tolerancia a fallos, ya que uno puede afectar a toda la base de datos	Mayor tolerancia a fallos debido a que los datos están distribuidos y replicados entre múltiples nodos
Dedicadas a consultas complejas con grandes conjuntos de datos	Dedicadas a consultas simples
MySQL, PostgreSQL, Oracle	MongoDB, Cassandra, Redis

Tabla 2.3: Tabla comparativa entre SQL y NoSQL

Se decidió utilizar una base de datos NoSQL debido a que no todos los elementos contenían la misma cantidad de atributos.

Otro problema surge al realizar consultas como obtener todos los datos de un único personaje, en caso de estar trabajando con una base de datos no basada en documentos y SQL, las consultas podrían llegar a ser complejas de programar.

Dadas estas circunstancias, se optó por almacenar la información con la estructura de un JSON, y es aquí donde destacan bases de datos NoSQL como

MongoDB.

Base de datos NoSQL

Existen diferentes ejemplos de bases de datos NoSQL mostradas en la última fila de la Tabla 2.3. Entre las opciones propuesta se decidió utilizar MongoDB, sin embargo, algunos aspectos relevantes de las otras bases de datos NoSQL aparecen mostrados en la Tabla 2.4.

Cassandra	Redis
Estructura de tabla	Estructura Clave-Valor en memoria
Consultas restrictivas frente a datos complejos o variables	No permite consultas complejas, se debe de realizar la interSección de varias búsquedas
Dedicada a aplicaciones distribuidas	Dedicada a caches o sistemas RT
Transaccion a nivel de fila (ACID)	Transacciones individuales

Tabla 2.4: Tabla comparativa de Cassandra y Redis frente a MongoDB

Entre las ventajas que aporta MongoDB se encuentran la facilidad de sus métodos de inserción y consultas, así como la comodidad al trabajar con Python y JSON.

MongoDB ofrece funcionalidades como ReplicaSets o Sharding que resultan muy interesantes cuando se busca tener un servicio de manera distribuida:

- Los ReplicaSets permiten tener diferentes nodos de una base de datos en distintos puntos del mundo, dando la opción a tener los datos de manera distribuida. Esto provoca que en caso de tener un fallo en uno de los nodos y no poder estar activo, hay otros que sí que están disponibles para atender las peticiones.
- El "Sharding" por otra parte busca fragmentar la información de las bases de datos, de tal manera que a un nodo le corresponde una cantidad limitada de datos. Es decir, si a una base de datos se le incluyen como datos nombres de diferentes personas, al aplicar esta fragmentación un nodo estará dedicado a buscar datos de la A-O, mientras que el otro nodo se encargará de los valores P-Z. Esto provoca que las búsquedas sean más rápidas y se reduce la carga en los servidores.

3

Desarrollo del proyecto

En este capítulo se verá más en detalle cada tecnología y los pasos que se siguieron para realizar el proyecto.

En primer lugar, en la Sección 3.1 se mostrará la arquitectura final del proyecto en distintas Figuras.

Posteriormente se mostrarán los aspectos más relevantes utilizados en Python para poder desarrollar una API que cumpla con los objetivos propuesto, siendo sencilla, rápida y de facil acceso en la Sección 3.2.

La siguiente Sección 3.3, está dedicada a MongoDB, donde se muestran diferentes opciones de creación de una base de datos, así como breves aspectos de configuración.

Por último, en la Sección 3.4 se profundiza en el apartado de Docker. Este cuenta con mayor contenido y se adentra en áreas de configuración antes y después de tener el conjunto de contenedores objetivo. Esta Sección se complementa con la anterior, MongoDB, ya que se aplican las técnicas de Sharding y ReplicaSet para garantizar disponibilidad y escalabilidad.

3.1. Arquitectura del proyecto

En la Figura 3.1 se muestra a gran escala la estructura final del proyecto. Todo está incluido en un Docker Compose y los usuarios pueden realizar peticiones a

la API a traves de **warcraftrumblejson.com**.

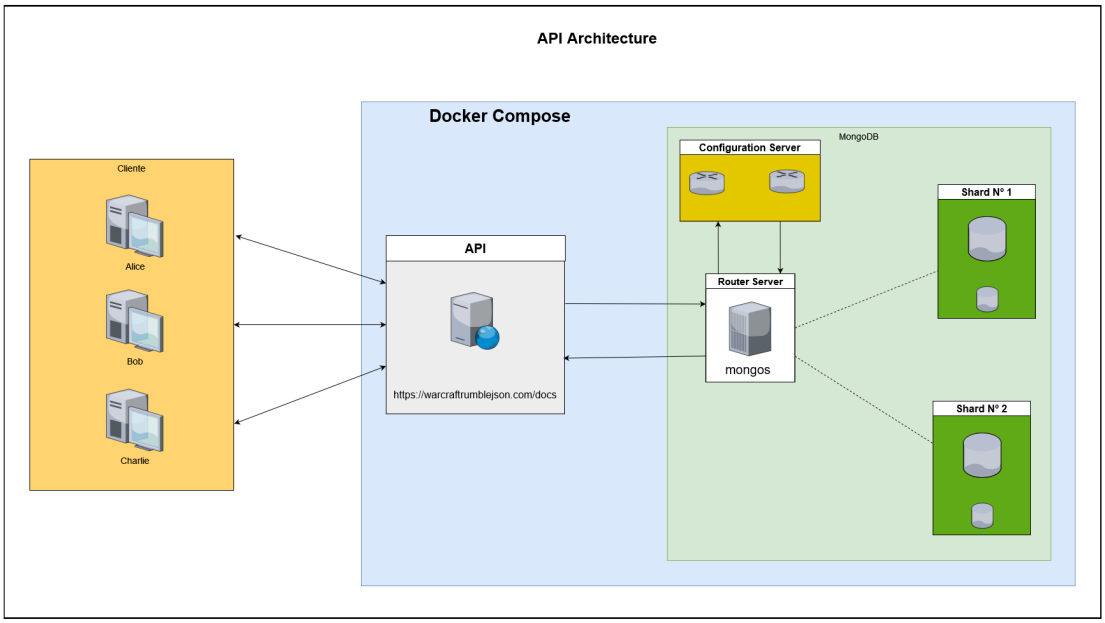


Figura 3.1: Arquitectura del proyecto

Una mejor representación arquitectónica de la Base de Datos se encuentra en la Figura 3.2. Donde se muestra la relación entre los servidores de configuración, el router server y los shards, así como la importancia de los replica sets.

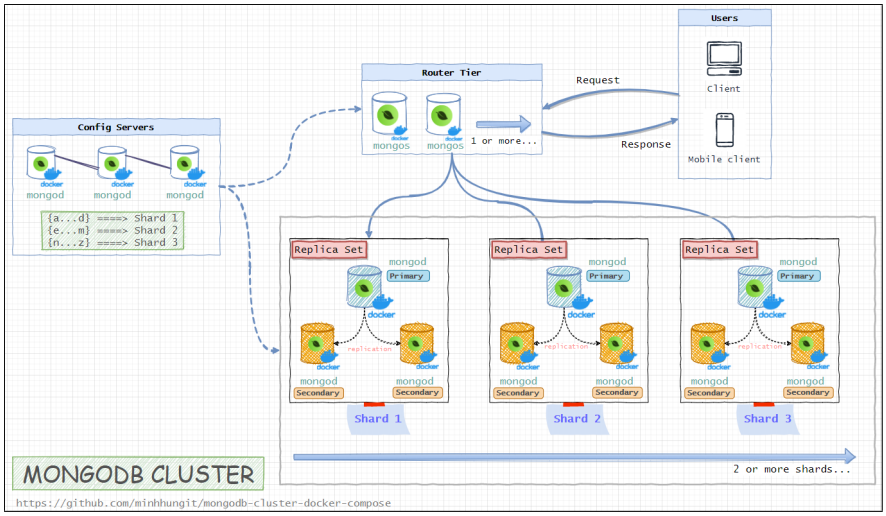


Figura 3.2: Arquitecura MongoDB. Sharding y Replica Sets.
Fuente: **Github minhhungit**[7]

3.2. Python

Lo primero que se debe realizar es instalar FastAPI, framework que permite el desarrollo de la API; y Uvicorn, servidor web asíncrono que gestiona las peticiones.

Se deben de instalar las dependencias e importar los módulos:

```
pip install fastapi uvicorn[standard]
```

El servidor se ejecutará en localhost para realizar pruebas.

Si se desea ponerle otro nombre que no sea localhost, se debe acceder al siguiente archivo:

`Windows: C:\Windows\System32\drivers\etc\hosts`

`Linux: /etc/hosts`

Se aconseja utilizar un puerto en el rango 49152–65535, estos son puertos privados no registrados por la IANA (Internet Assigned Numbers Authority), aunque se pueden utilizar otros como 1234, 8080 o 4444 sin problema, pero es recomendable comprobar que ese puerto no está en uso. Ya que se trata de un servicio web, el uso del puerto 80 es la opción por defecto y recomendada.

Existen mas parámetros que se pueden añadir al Uvicorn, como autoreload, ruta a archivos de claves SSL o un limitador de peticiones, se pueden consultar en la **página oficial de Uvicorn**[8].

3.2.1. Redirección a la documentación

Se puede redireccionar "/docs" a "/",de tal manera que la página principal siempre serán los /docs.

Código 3.1: Python: Redirección a los docs

```
1
2 from fastapi.responses import RedirectResponse
3
4 @app.get("/", include_in_schema=False)
5 def redirect():
6     return RedirectResponse(url="/docs")
```

Para más detalles sobre la configuración de rutas se recomienda consultar la **documentación oficial de FastAPI**[9].

3.2.2. Documentación auto-generada

En "/docs" aparecen todas las rutas que se han declarado, aunque se puede configurar cuales mostrar y cuales no.

Si se experimenta con la página se llega a:

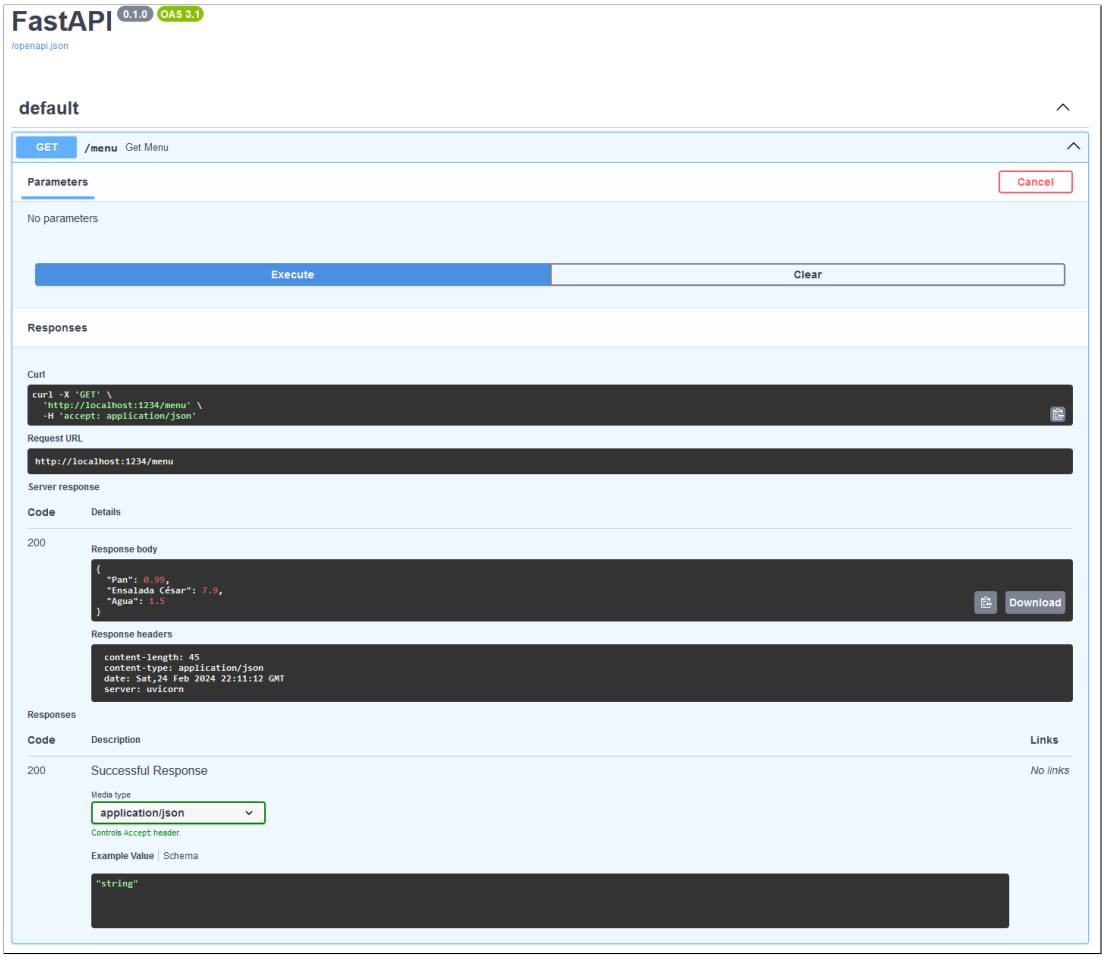


Figura 3.3: FastAPI: Documentación auto-generada

Se puede apreciar que han aparecido nuevos bloques en pantalla como:

- **Curl**
- Códigos de respuesta HTTP. Para obtener más información sobre cada uno se puede acceder a la **documentación oficial de Mozilla**[10].
- Body response: respuesta del servidor e información adicional como el tamaño, tipo de contenido, fecha y servidor.

3.2.3. BaseModel

BaseModel es un módulo de **pydantic**[11] que se puede añadir a una clase.

Entre las diferentes funcionalidades que trae se incluye la validación, restricción y modelación de datos.

Útil al realizar métodos POST en los que se busca mantener una estructura concreta, como por ejemplo asegurarse que un dato es siempre de tipo integer en vez de string. Estas comparaciones se realizan de manera automatica con BaseModel.

3.3. MongoDB

Existen varias maneras de crear una base de datos con MongoDB, por ejemplo, **MongoDB Atlas**, un servicio Cloud que ofrece MongoDB. De manera local se puede realizar con **MongoDB Community Server**.

Cabe destacar que todas las consultas desde Python a MongoDB se realizan utilizando un Cursor [12].

3.3.1. MongoDB Atlas

El servicio Cloud de MongoDB Atlas es excelente para tener los datos de manera distribuida, sin ocupar espacio en un ordenador local y auto-escalados. Proporciona un servicio gratuito, el cual está limitado a 512MB pero suele ser espacio suficiente para realizar pruebas y probar el entorno. La configuración es muy similar a la versión de MongoDB en local.

3.3.2. MongoDB Community Server

La opción de Community Server permite acceder a los datos de manera local sin tener que depender de una conexión a internet. Se puede obtener desde la **página oficial de MongoDB.**

Al terminar el proceso de instalación se abrirá MongoDB Compass con una configuración predeterminada:

La caja de color blanco contiene el Título de URI (Identificador de recursos uniformes), esto se puede pensar como la ruta de acceso. Al hacer click en Advanced Options, dentro de la Sección de General aparecen dos opciones:

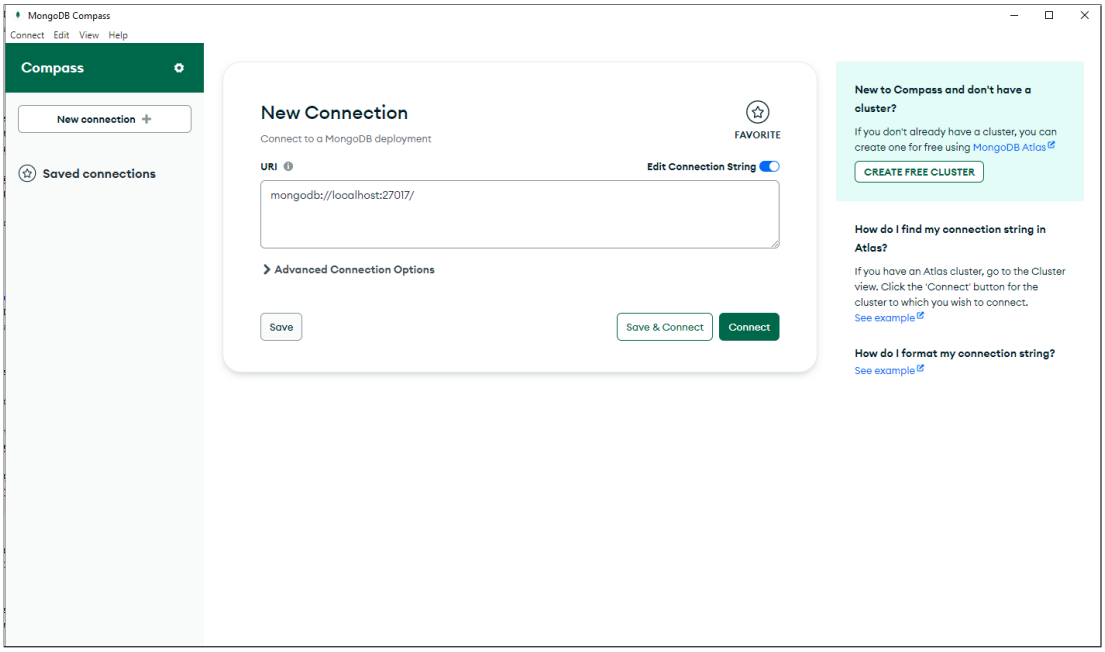


Figura 3.4: MongoDB Compass: Pantalla de inicio.

- mongodb: Se utiliza al realizar una conexión a partir de la dirección del servidor y puerto.
- mongodb+srv: Se utiliza al realizar una conexión a una base de datos MongoDB compartida, un cluster. Esta es la configuración que se utiliza en MongoDB Atlas. Es útil para no tener que especificar IP fijas y se sustituye por la dirección del cluster y el nombre de la base de datos dentro de este. Esta ruta es proporcionada por el servicio en cuestión (p.e MongoDB Atlas).

Por último, existe la opción de Direct Conection. Se utiliza cuando el servidor está particionado y existe un nodo intermedio que actúa como router, que guía y redirecciona la petición a un Shard concreto. Si se activa, las conexiones se realizan directamente sobre el nodo sin tener que pasar por el intermediario, esto es útil cuando se busca velocidad y robustez al realizar pruebas específicas sobre un nodo.

Existen más opciones para proporcionar una capa de seguridad, sin embargo, no se explicarán en este documento.

Si el servicio mongod no se ha ejecutado previamente, la base de datos no estará activa, apareciendo el error mostrado en la Figura 3.5.

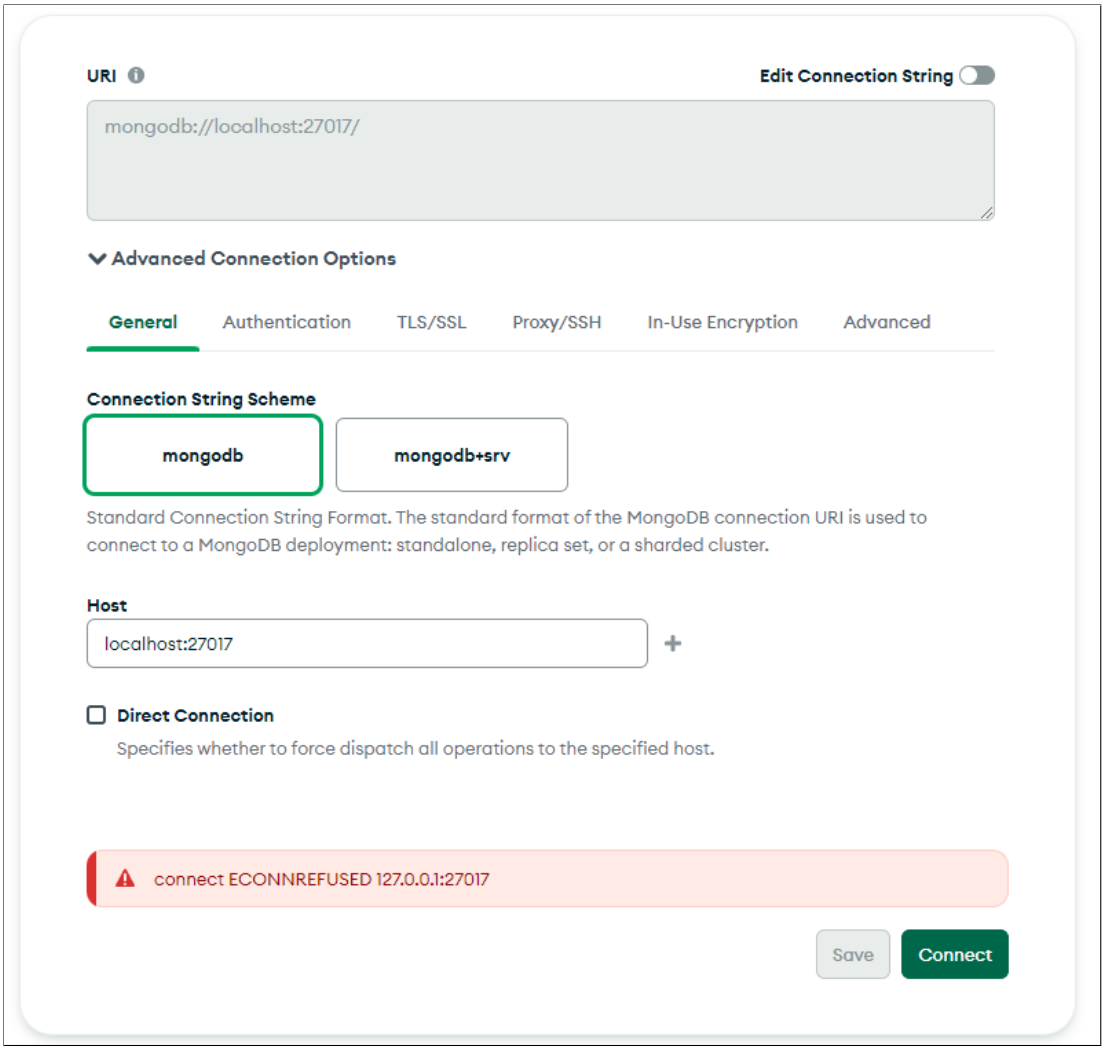


Figura 3.5: MongoDB Compass: Error de conexión.

Para iniciar la base de datos con mongod.exe, se puede realizar desde el intérprete de comandos escribiendo:

```
mongod --port 27017 --dbpath ../data
```

3.4. Docker

En esta última Sección se explica como generar contenedores Docker para la aplicación de Python y la base de datos MongoDB. Esta última utiliza una nueva imagen en Docker, por lo que habrá que introducir de nuevo los datos. Además se utilizarán las técnicas de Sharding y creación de ReplicaSets para poder disponer de un servicio distribuido y balanceado (en cuanto a datos se refiere), imitando

lo mostrado en la Figura 3.2.

3.4.1. Docker Desktop

Este proyecto fue diseñado y desarrollado en un entorno Windows, utilizando **Docker Desktop**, una aplicación de escritorio que aporta una GUI. A lo largo del documento se mostrarán los siguientes pasos haciendo uso de esta herramienta.

3.4.2. Dockerfile y cambios en la aplicación

Para poder realizar peticiones a la aplicación dentro de Docker, se cambia la configuración con localhost a 0.0.0.0, permitiendo que cualquier interfaz de red (de momento de manera local) se pueda conectar al servicio.

A continuación se necesita un fichero llamado Dockerfile, el cual será utilizado para crear una imagen de nuestra aplicación en un contenedor Docker, es decir, una copia de esta aplicación. Para ello se debe de tener primero un archivo con todas las dependencias que utiliza la aplicación (uvicorn, fastapi y muchas más que están por debajo o que se instalan como dependencias de otras). Se puede crear este archivo utilizando pip, gestor de paquetes de Python.

```
pip freeze > requirements.txt
```

La configuración de un archivo Dockerfile puede variar dependiendo de la configuración del proyecto de cada persona (si se está utilizando un entorno virtual o no, versión de Python, etc).

Toda la documentación sobre Dockerfile se puede encontrar **su página oficial**[13]. Aquí se pueden encontrar nuevas maneras de configurar el Dockerfile así como las definiciones de cada cosa.

El código 3.2 contiene la configuración utilizada en este proyecto.

3.4.3. Creación de una imagen

Para generar la imagen:

```
docker build -t <nombre de la imagen> <ruta de creacion>
```

Se pueden ver todas las imágenes creadas hasta el momento escribiendo:

```
docker images
```

Se pueden encontrar más parámetros y maneras de realizar un build en la **documentación oficial de Docker**[14].

Código 3.2: Docker: Dockerfile

```
FROM alpine:latest

RUN apk update && \
    apk add --no-cache python3 py3-pip bash

WORKDIR /app

COPY . /app

RUN python3 -m venv venv && \
    source venv/bin/activate && \
    pip install --no-cache-dir -r requirements.txt

CMD ["/bin/sh", "-c", "source venv/bin/activate && python app.
   py"]
```

3.4.4. Redes en Docker

Al crear un contenedor con una imagen también se está generando un entorno de redes dentro de este, además, dentro del contenedor Docker se gestionan las reglas del firewall de manera ajena al servidor en el que se almacenan.

Dentro de un contenedor existen 65535 posibles puertos que se pueden abrir. Al crear un contenedor es necesario dirigir un puerto del servidor a uno del contenedor. De tal manera que se obtiene la estructura:

```
<puerto servidor> : <puerto contenedor>
```

Las conexiones desde el exterior se deben de realizar al puerto del servidor.

3.4.5. Contenedor Docker a partir de una imagen

Para crear un contenedor a partir de una imagen:

```
docker create --name <nombre contendor> --publish
5000:1234 <nombre imagen>
```

Se pueden encontrar más parámetros y maneras de realizar un create en la **documentación oficial de Docker**[15].

En Docker Desktop, en el apartado Containers debería aparecer uno nuevo.

Al ejecutar este nuevo servicio, el acceso estará disponible para cualquier persona dentro de la misma red local.

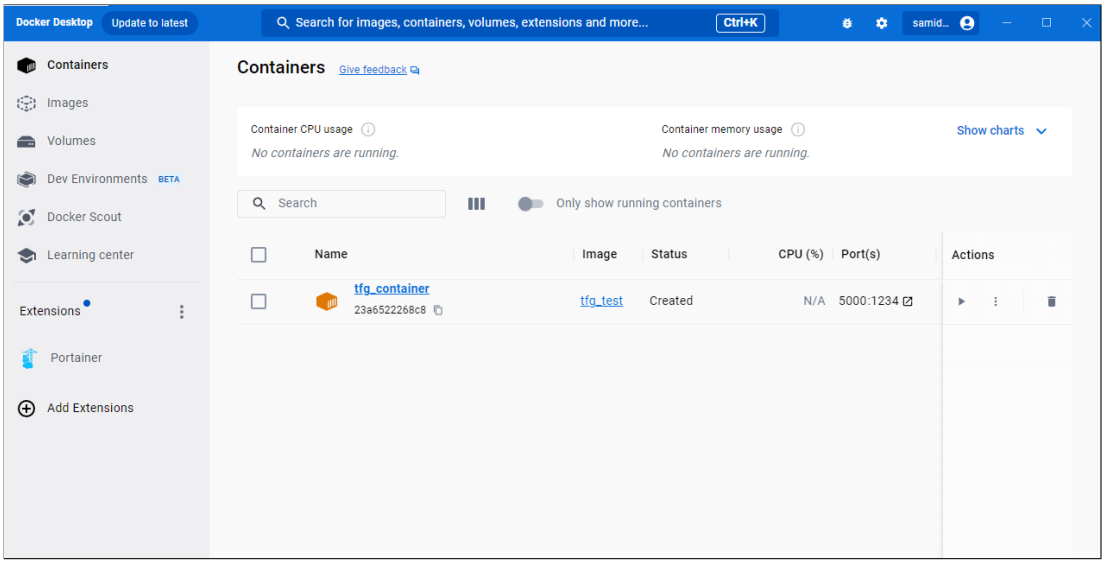


Figura 3.6: Docker: Contenedor Docker

3.4.6. Arquitectura de las imágenes mongo

Se debe realizar Docker Compose junto con imágenes mongo y la aplicación si no se desea utilizar MongoDB Atlas, el servicio Cloud.

De esta manera se busca obtener la arquitectura mostrada en la Figura 3.7:

Se pueden diferenciar varios componentes en este esquema:

- Client: en el contexto del proyecto sería la API realizando una petición a la base datos, a pesar de que es un usuario el que hizo en primer lugar una petición a la API.
- Routers: Nodos encargados de gestionar las peticiones, solicitando datos y enviándolos.
- Config Servers: Se encargan de indexar las peticiones y de esta manera localizar los datos pedidos por los nodos Router.
- ReplicaSets: Conjunto de nodos que existen para generar disponibilidad, son divididos en nodos primarios y secundarios, el primario se encarga de realizar las operaciones de escritura y los secundarios irán copiando los datos de manera asíncrona. En caso de que el nodo primario deje de funcionar, se realizará un proceso en el cual un nodo secundario tomará el papel del primario. De esta manera a pesar de que un nodo deje de funcionar, los datos se pueden seguir leyendo debido a que los secundarios los han copiado (no siempre se asegura que los datos se hayan copiado al completo en los secundarios).

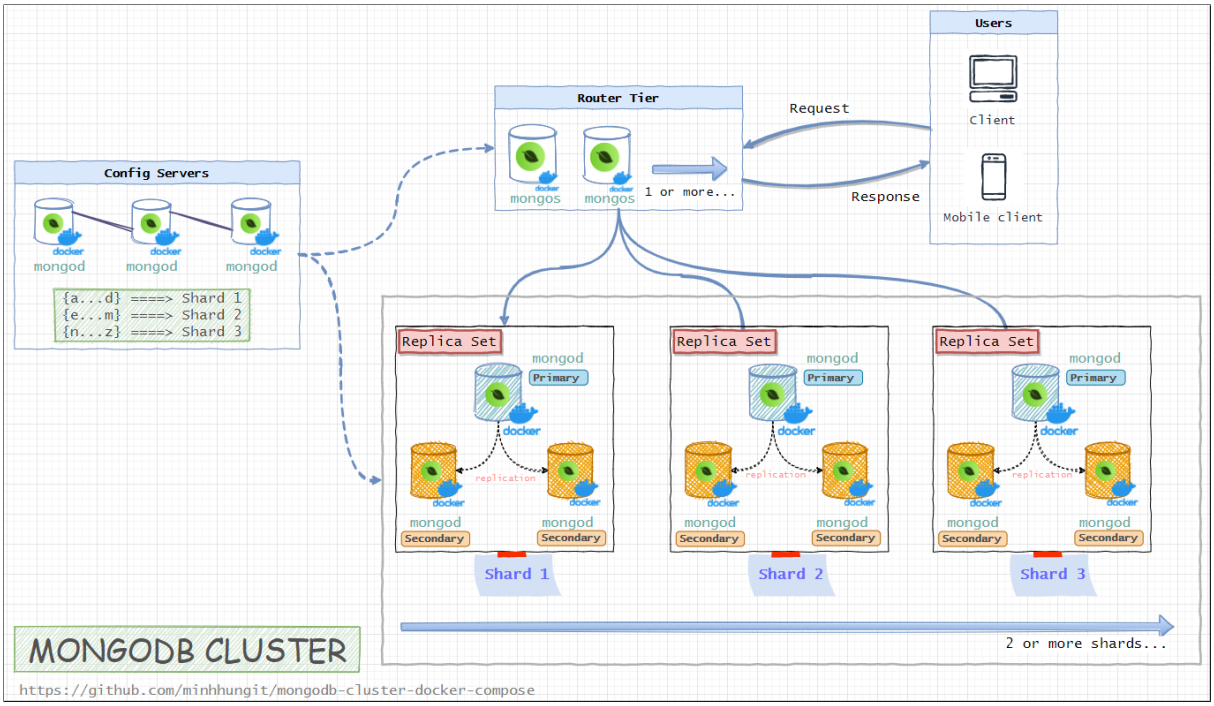


Figura 3.7: Arquitecura MongoDB: Config Server, Router y Shards.
Fuente: **Github minhhungit**[7]

- Shards: Este concepto o técnica se refiera a la distribución de datos en conjuntos de nodos concretos, que por lo general suelen ser ReplicaSets. De esta manera un ReplicaSet puede contener datos de la A-M y el segundo ReplicaSet de N-Z.

3.4.7. Arranque de Docker Compose

Como se ha mencionado anteriormente, se utilizará el archivo de configuración proporcionado por **minhhungit en su GitHub**[7]. Se han realizado pequeñas modificaciones. Consultar Código A.1.

Para obtener información sobre cada uno de los parámetros se puede acceder a **la documentación oficial de Docker**[13].

Para iniciar la creación del Docker Compose, se debe escribir en el intérprete de comandos :

```
docker-compose up -d
```

En la aplicación de Docker Desktop aparecerá el conjunto de contenedores bajo un mismo nodo. Se puede apagar, reiniciar y ejecutar cada uno de manera

independiente.

Ahora se debe de configurar cada tipo de nodo mongo, siendo los servidores de configuración los primeros.

En el **GitHub de minhhungit**[7] aparece una carpeta llamada scripts, con ficheros ".js", estos incluyen unos comandos que se deben de utilizar en la shell propia de mongo, llamada mongosh. Estos comandos son muy similares a los propuesto en el **manual oficial de MongoDB**[16].

Servidor de configuración

Código 3.3: Docker: Servidor de Configuración

```
rs.initiate(
    {
        _id: "rs-config-server",
        configsvr: true,
        version: 1,
        members: [
            { _id: 0, host : 'configsvr01:27017 },
            { _id: 1, host : 'configsvr02:27017' }
        ]
    }
)
```

Con rs.status() se debe comprobar que se han generado correctamente un host primario y otro secundario, como se puede apreciar en la Figura 4.13.

Se recomienda dejar entre 5 y 10 segundos antes de poner rs.status(), ya que puede estar procesando el comando anterior.

Shards

Empezando por el rs-shard-01 y posteriormente el rs-shard-02. Se establecerá que el shard-01-node-a será el primario y shard-01-node-b, shard-01-node-c los secundarios.

Al terminar, se escribe nuevamente rs.status() para comprobar que la configuración ha sido completada correctamente. Se debe realizar lo mismo para el rs-shard-02.

Código 3.4: Docker: Configuración de los Shards

```
rs.initiate(
    {
        _id: "rs-shard-01",
        version: 1,
        members: [
            { _id: 0, host : "shard01-a:27017" },
            { _id: 1, host : "shard01-b:27017" },
            { _id: 2, host : "shard01-c:27017" }
        ]
    }
)
```

Router Servers

Por último quedan los Router Server, donde se utilizará router-01 como principal. En la consola de mongosh se deben de añadir todos los Shards creados, tanto primarios como secundarios.

Código 3.5: Docker: Router Servers

```
sh.addShard("rs-shard-01/shard01-a:27017")
sh.addShard("rs-shard-01/shard01-b:27017")
sh.addShard("rs-shard-01/shard01-c:27017")
sh.addShard("rs-shard-02/shard02-a:27017")
sh.addShard("rs-shard-02/shard02-b:27017")
sh.addShard("rs-shard-02/shard02-c:27017")
```

Al utilizar sh.status(), se podrá ver la configuración de los Shards junto con los componentes de cada uno, esto se aprecia en la Figura 4.14

3.4.8. Creación base de datos MongoDB en Docker

Finalmente queda crear la base de datos junto con su colección y activar el Sharding.

```
sh.enableSharding("<nombre de la base de datos>")
```

Solo queda decidir como se quieren distribuir los datos, a partir de rangos o un atributo del dato. Es decir:

- rango: Se centran en juntar los datos que están dentro de un rango de valores similares en un Shard. Por ejemplo: KM de carreteras, fechas de

transacciones, coordenadas, sucesión de números.

- atributo: Se puede elegir un atributo de un objeto que se guarda en la base de datos para generar una función de hashing sobre este, y clasificar los datos respecto al resultado de la función.

Esto es algo que cada persona debe de valorar y no es una decisión para nada fácil. En muchos casos se puede requerir un estudio de los datos, por lo que no es trivial.

Se recomienda consultar **el manual oficial de Mongo**[17] para obtener información más específica y detallada.

```
sh.shardCollection("<base de datos>.<coleccion>",
{"<atributo>": "hashed"})
```

Se puede obtener una visión mas clara de toda la distribución con el siguiente comando:

```
db.<coleccion>.getShardDistribution()
```

Resultado disponible en la Figura 4.15.

Por último, queda tomar la nueva URI de la base de datos, para esto se ha de acceder a al contenedor del router01 y escribir mongosh en el intérprete de comandos (Exec). Se ha de copiar la ruta que aparece después de "Connecting to:" y cambiarla en la app.

Se debe de cambiar la URI actual por la nueva y sustituir el 127.0.0.1, por el nombre del servicio (archivo docker-compose.yml) que utiliza mongos. En este caso es router01.

Finalmente se deberán de rellenar nuevamente las base de datos.

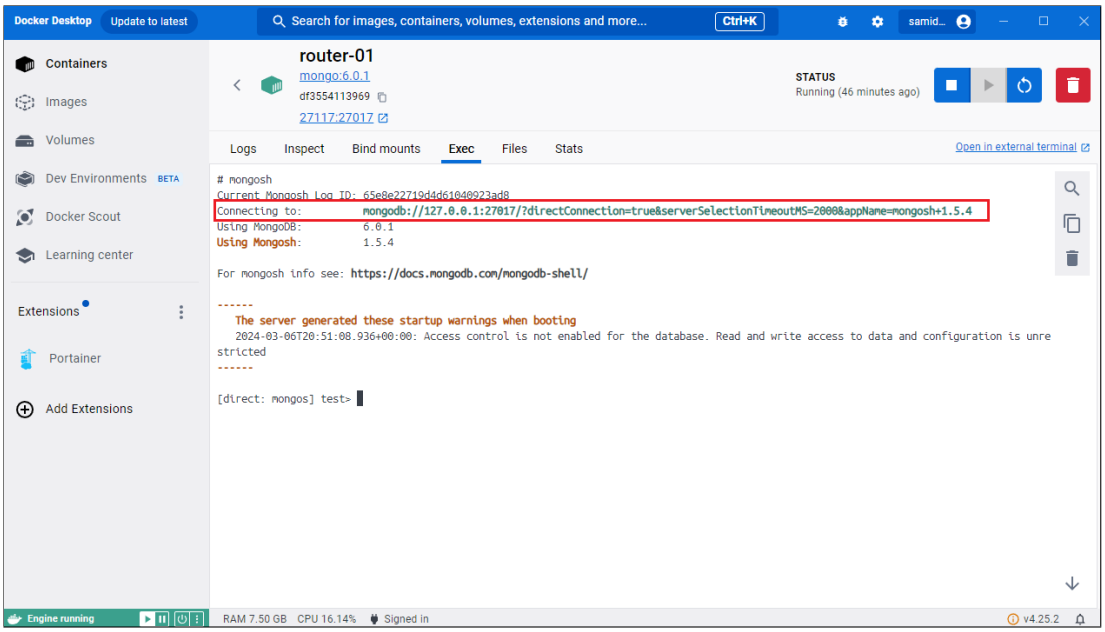


Figura 3.8: Docker: URI del servidor de configuración

4

Resultados

A lo largo de este capítulo se mostrarán ejemplos sobre los resultados obtenidos en base a los objetivos, una vez finalizado el proyecto.

4.1. API

A continuación se muestran los diferentes entornos visuales de la API. Esto incluye tanto GitHub como los docs generados por FastAPI.

4.1.1. GitHub

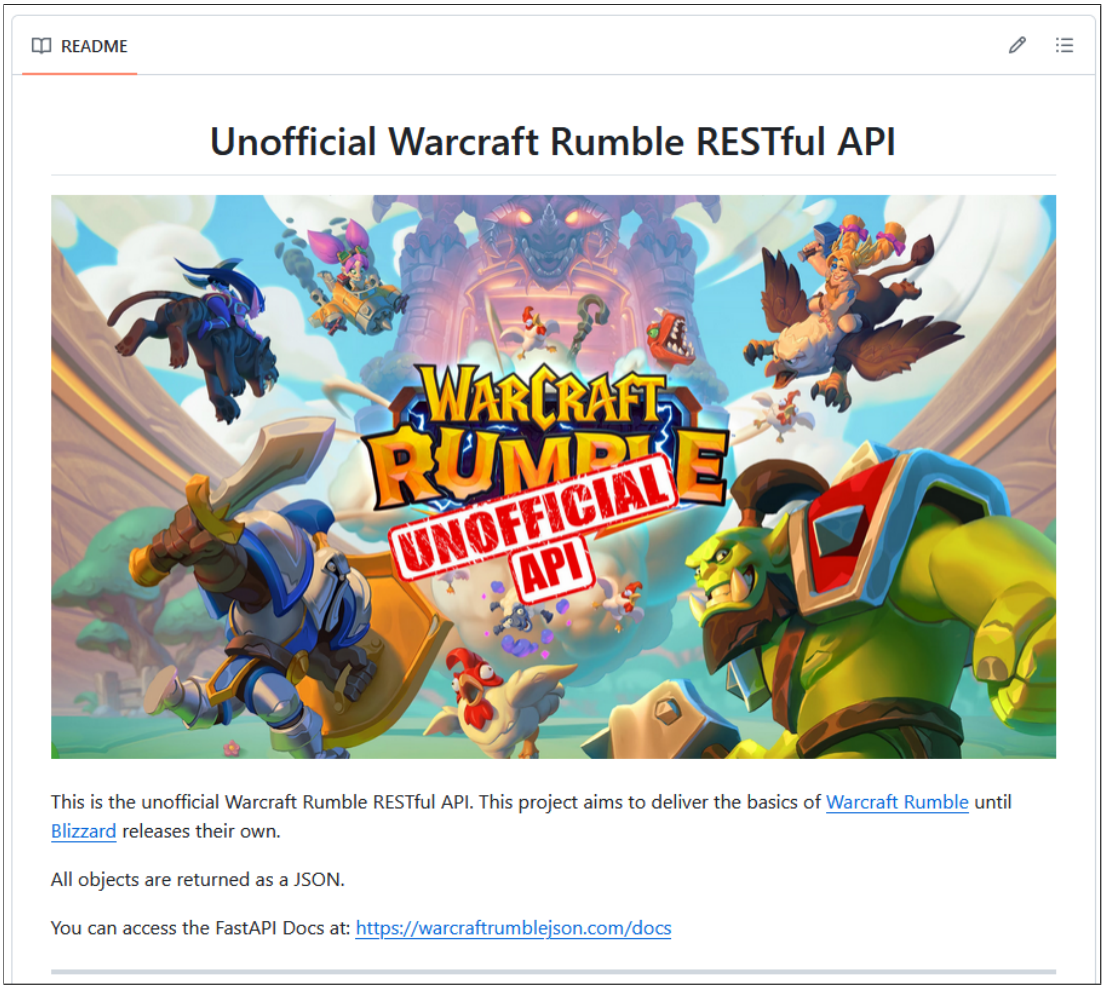
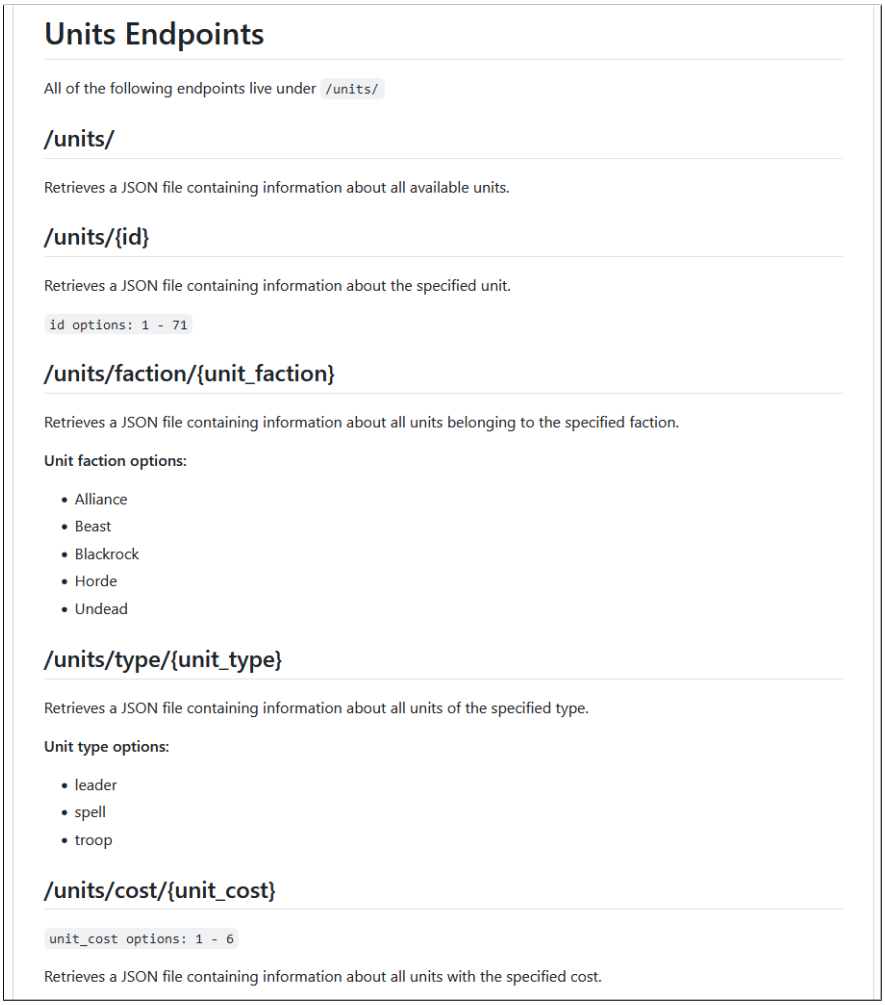


Figura 4.1: GitHub: Presentación

Cuenta con una breve introducción al proyecto donde se da a entender que este no es un producto oficial y busca tomar el rol de producto temporal sustitutivo. También se indica el link para acceder a los docs de FastAPI para poder trabajar de manera mas visual con los endpoints.

La Figura 4.2 muestra el formato que sigue la documentación dentro de GitHub. Se da una breve descripción sobre la utilidad de cada endpoint y posteriormente se proporciona el rango de opciones posibles en cada caso.



Units Endpoints

All of the following endpoints live under `/units/`

/units/

Retrieves a JSON file containing information about all available units.

/units/{id}

Retrieves a JSON file containing information about the specified unit.

`id options: 1 - 71`

/units/faction/{unit_faction}

Retrieves a JSON file containing information about all units belonging to the specified faction.

Unit faction options:

- Alliance
- Beast
- Blackrock
- Horde
- Undead

/units/type/{unit_type}

Retrieves a JSON file containing information about all units of the specified type.

Unit type options:

- leader
- spell
- troop

/units/cost/{unit_cost}

`unit_cost options: 1 - 6`

Retrieves a JSON file containing information about all units with the specified cost.

Figura 4.2: GitHub: Ejemplo de la documentación

4.1.2. FastAPI

Dentro de los docs autogenerados existen diferentes endpoints. Tanto el método POST y DELETE están ocultos en la versión final.

Los endpoints están conectados a una imagen en Docker de la base de datos MongoDB, , la cual está particionada y dispone de varios servidores de tipo Replica, dificultando que la disponibilidad del servicio sea nula.

Entre las distintas peticiones que se pueden realizar, a continuación se muestran un conjunto de ellas utilizando tanto el servicio que proporciona FastAPI (gracias a OpenAPI) y una consola de comandos:

- Método GET que proporciona la toda información de una unidad específica (OpenAPI). Figura 4.4.
- Método GET que proporciona la toda información de una unidad específica (CMD). Figura 4.5.
- Método GET que proporciona la toda información de todas las unidades (OpenAPI). Figura 4.6.
- Método GET que proporciona únicamente la estadísticas de una unidad específica (OpenAPI). Figura 4.7.
- Método POST que añade una unidad con todos sus datos (OpenAPI). Figuras 4.8, 4.9, 4.10.
- Método DELETE que elimina una unidad según su ID (OpenAPI). Figuras 4.11, 4.12.

Figura 4.3: FastAPI: Métodos GET, POST y DELETE

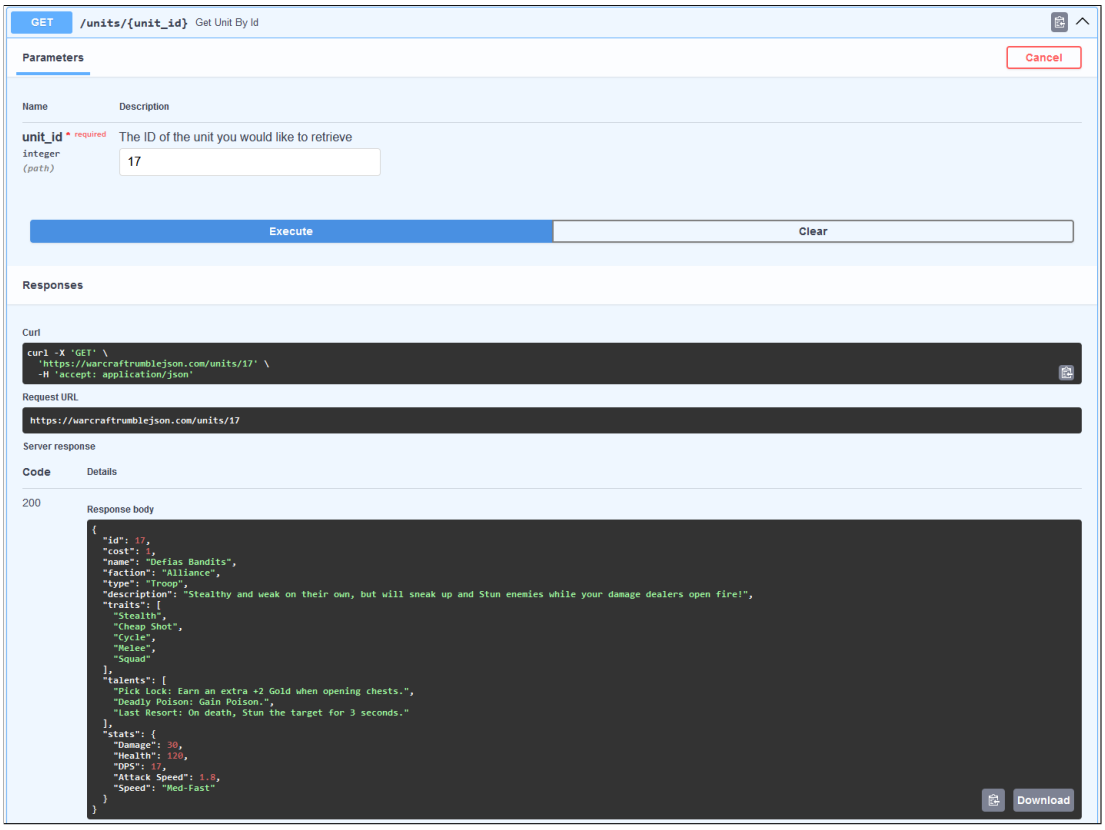


Figura 4.4: FastAPI: Demostración método GET individual.

Figura 4.5: FastAPI: Demostración método GET individual utilizando CMD

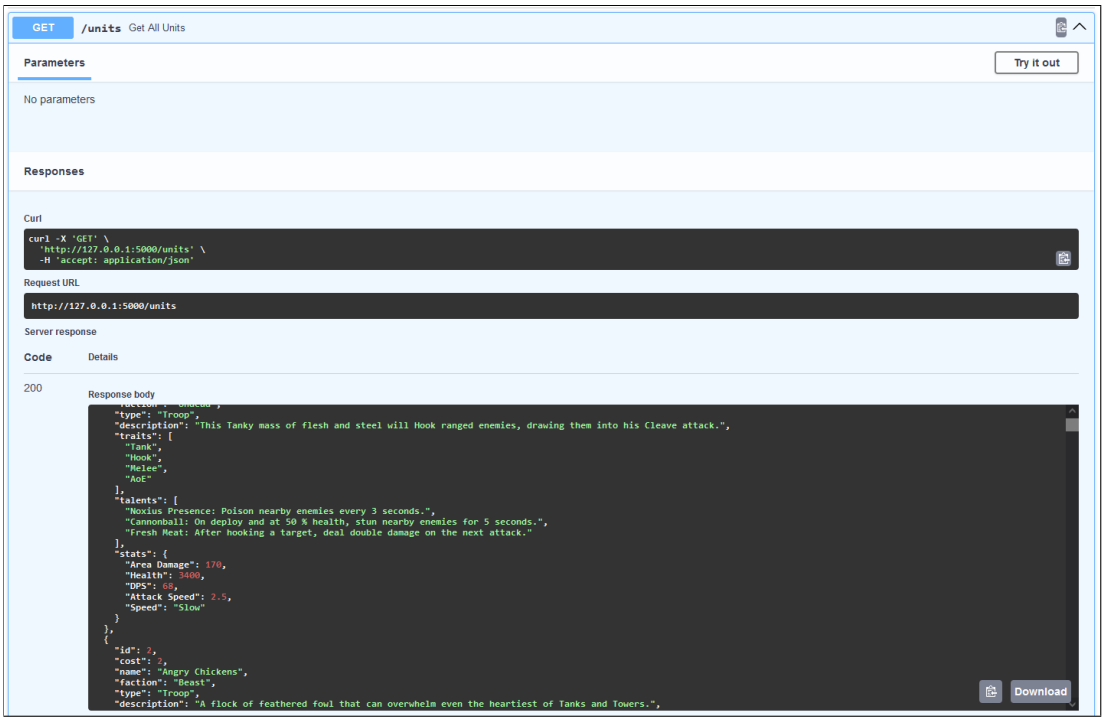


Figura 4.6: FastAPI: Demostración método GET general.

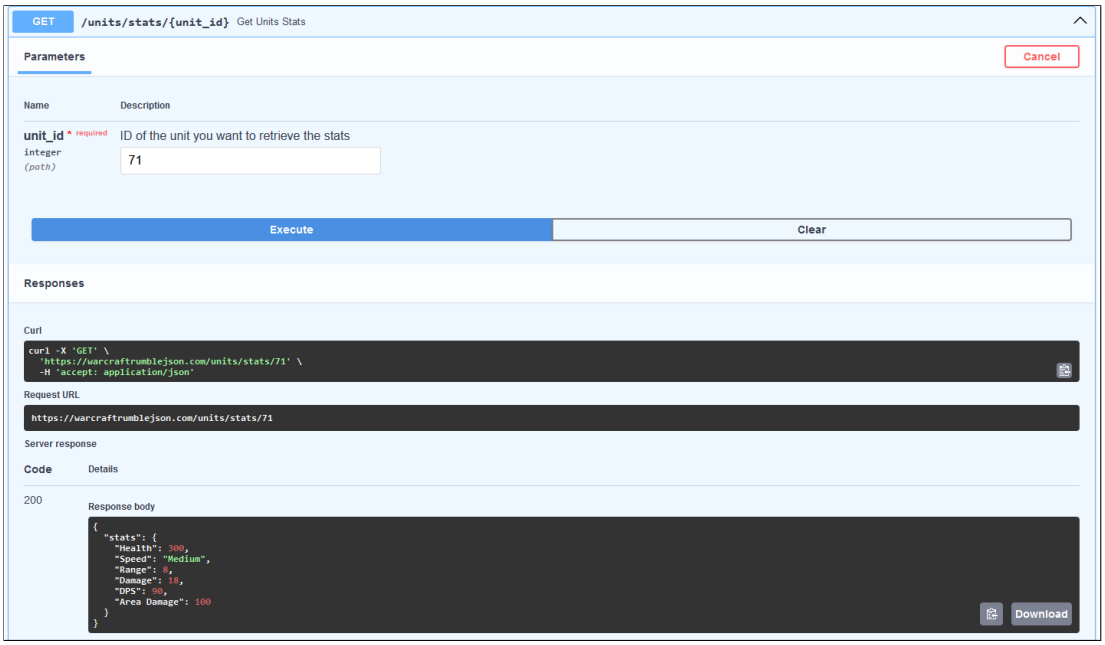


Figura 4.7: FastAPI: Demostración método GET Stats.

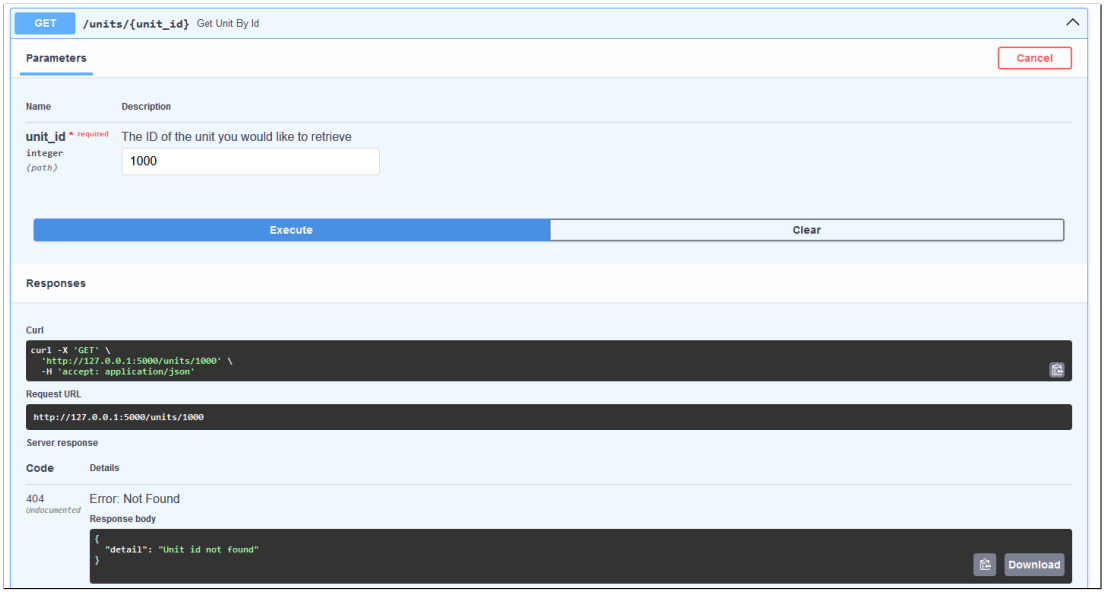


Figura 4.8: FastAPI: Demostración método POST (1/3)

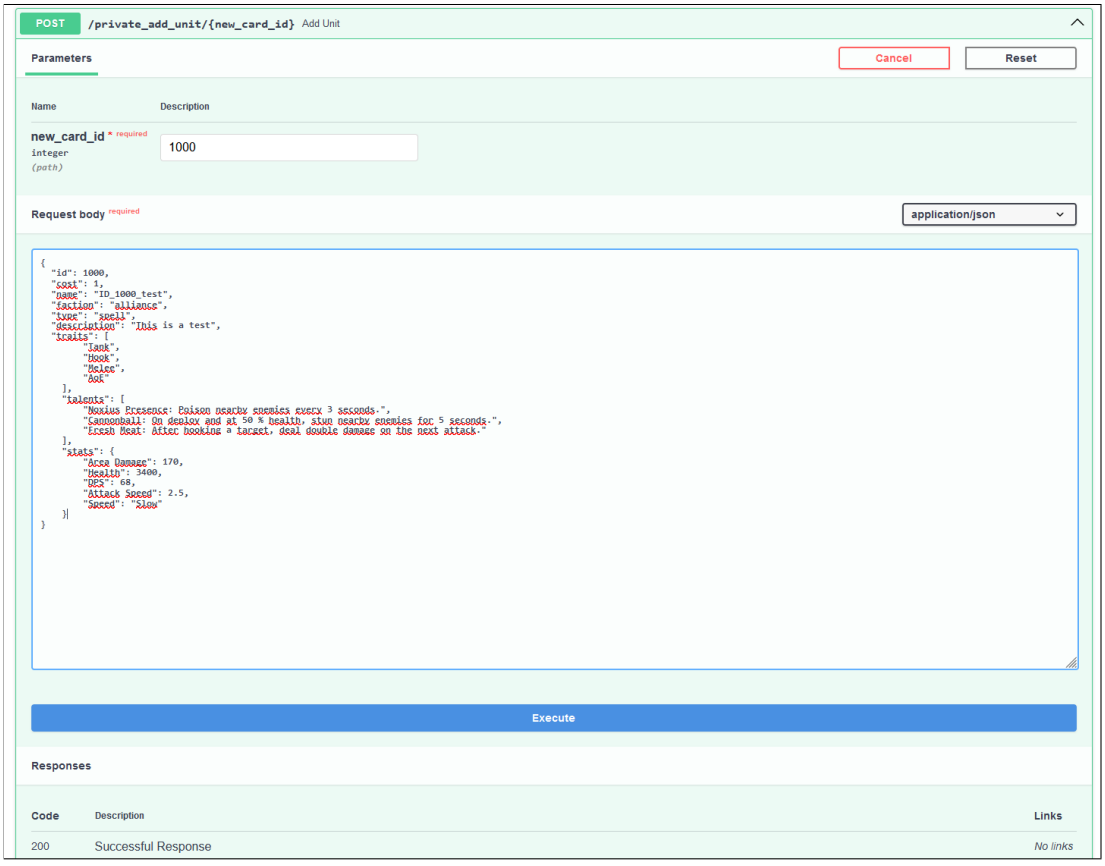


Figura 4.9: FastAPI: Demostración método POST (2/3)

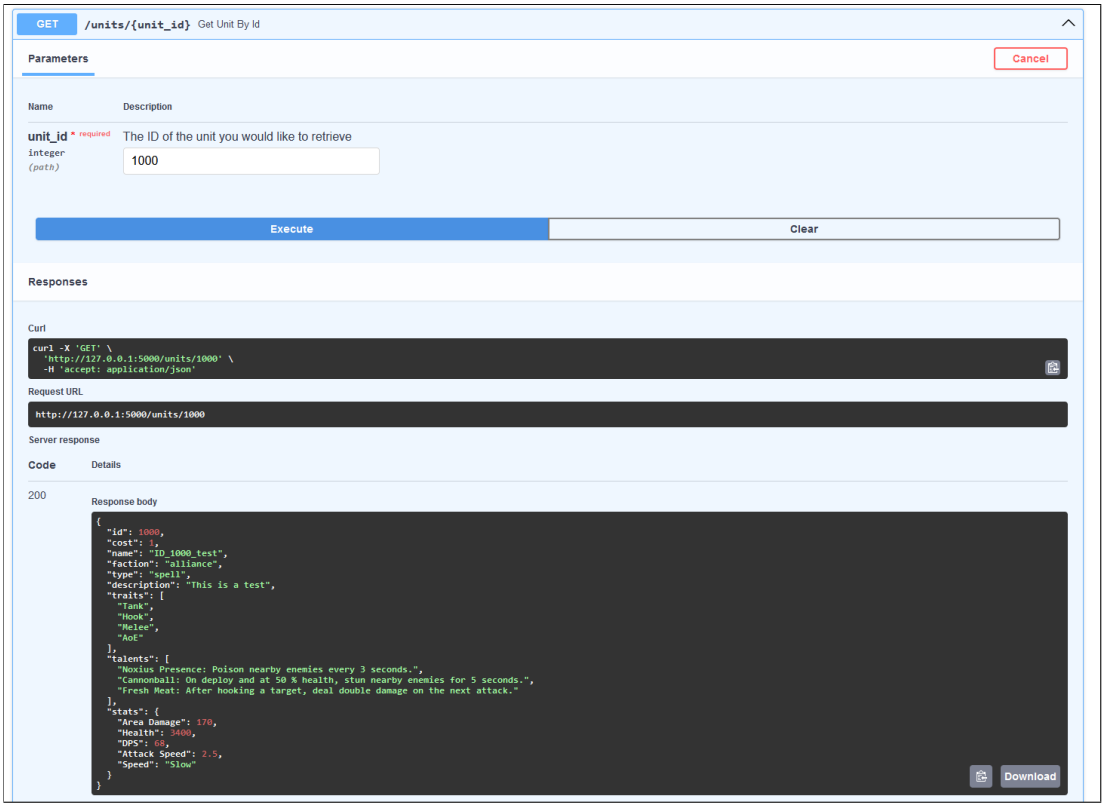


Figura 4.10: FastAPI: Demostración método POST (3/3)

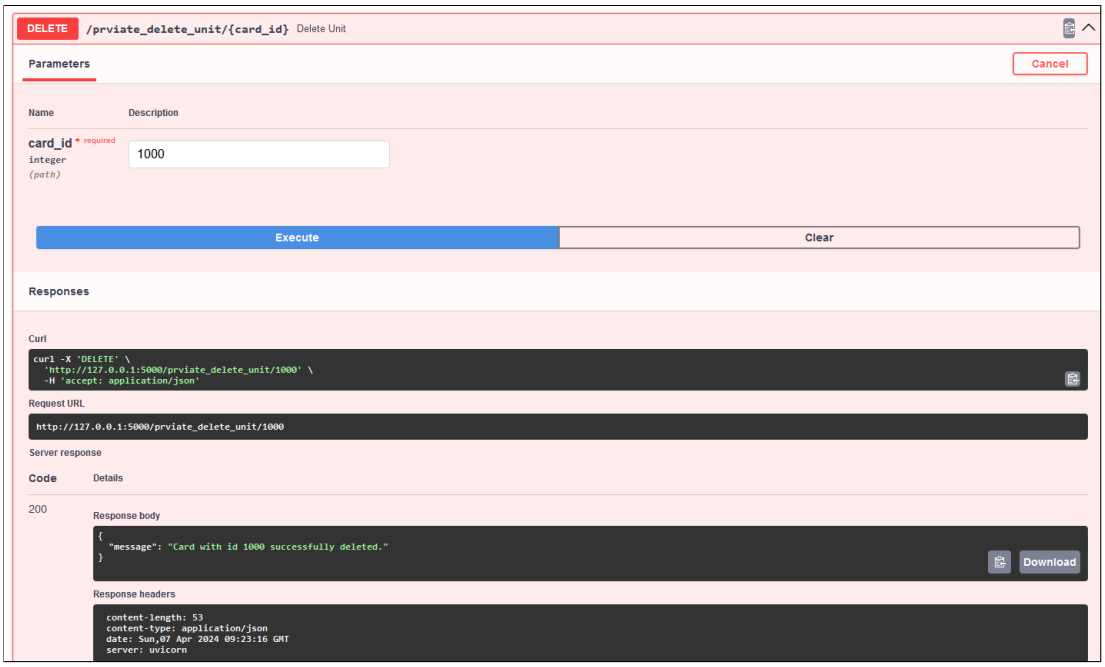


Figura 4.11: FastAPI: Demostración método DELETE (1/2)

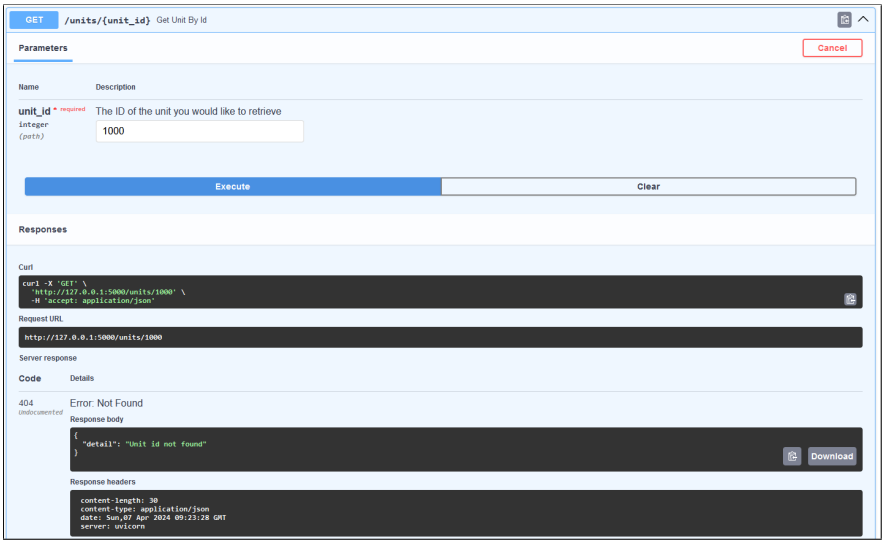


Figura 4.12: FastAPI: Demostración método DELETE (2/2)

4.2. Docker

El apartado de Docker comparte protagonismo con MongoDB. A continuación se muestran los siguientes resultado obtenidos:

- Conversión de un replicaSet en primario y otros en secundarios en los servidores de configuración. Figura 4.13
- Distribución de los replicaSets dentro de cada Shard. Figura 4.14
- Distribución de los datos en cada Shard antes de introducir datos. Figura 4.15
- Distribución de los datos en cada Shard tras introducir 70 datos. Figura 4.164.15

```
  members: [
    {
      _id: 0,
      name: 'configsvr01:27017',
      health: 1,
      state: 1,
      stateStr: 'PRIMARY',
      uptime: 1281,
      optime: { ts: Timestamp({ t: 1709744206, i: 1 }), t: Long("1") },
      optimeDate: ISODate("2024-03-06T16:56:46.000Z"),
      lastAppliedWallTime: ISODate("2024-03-06T16:56:46.271Z"),
      lastDurableWallTime: ISODate("2024-03-06T16:56:46.271Z"),
      syncSourceHost: '',
      syncSourceId: -1,
      infoMessage: '',
      electionTime: Timestamp({ t: 1709743042, i: 1 }),
      electionDate: ISODate("2024-03-06T16:37:22.000Z"),
      configVersion: 1,
      configTerm: 1,
      self: true,
      lastHeartbeatMessage: ''
    },
    {
      _id: 1,
      name: 'configsvr02:27017',
      health: 1,
      state: 2,
      stateStr: 'SECONDARY',
      uptime: 1175,
      optime: { ts: Timestamp({ t: 1709744204, i: 1 }), t: Long("1") },
      optimeDurable: { ts: Timestamp({ t: 1709744204, i: 1 }), t: Long("1") },
      optimeDate: ISODate("2024-03-06T16:56:44.000Z"),
      optimeDurableDate: ISODate("2024-03-06T16:56:44.000Z"),
      lastAppliedWallTime: ISODate("2024-03-06T16:56:46.271Z"),
      lastDurableWallTime: ISODate("2024-03-06T16:56:46.271Z"),
      lastHeartbeat: ISODate("2024-03-06T16:56:45.059Z"),
      lastHeartbeatRecv: ISODate("2024-03-06T16:56:46.145Z"),
      pingMs: Long("0"),
      lastHeartbeatMessage: '',
      syncSourceHost: 'configsvr01:27017',
      syncSourceId: 0,
      infoMessage: '',
      configVersion: 1,
      configTerm: 1
    }
  ],
  ok: 1,
  lastCommittedOpTime: Timestamp({ t: 1709744206, i: 1 }),
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1709744206, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1709744206, i: 1 })
}
rs-config-server [direct: primary] test>
```


Figura 4.13: Docker: Servidor de configuración

```
[direct: mongos] test> sh.status()
shardingVersion
{
  _id: 1,
  minCompatibleVersion: 5,
  currentVersion: 6,
  clusterId: ObjectId("65e89bc7031bf1589ba1f633")
}
---
shards
[
  {
    _id: 'rs-shard-01',
    host: 'rs-shard-01/shard01-a:27017,shard01-b:27017,shard01-c:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1709746588, i: 1 })
  },
  {
    _id: 'rs-shard-02',
    host: 'rs-shard-02/shard02-a:27017,shard02-b:27017,shard02-c:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1709746590, i: 2 })
  }
]
---
active mongoses
[ { '6.0.1': 2 } ]
---
autosplit
{ 'Currently enabled': 'yes' }
---
balancer
{
  'Currently enabled': 'yes',
  'Currently running': 'no',
  'Failed balancer rounds in last 5 attempts': 0,
  'Migration Results for the last 24 hours': 'No recent migrations'
}
---
databases
[
  {
    database: { _id: 'config', primary: 'config', partitioned: true },
    collections: {}
  }
]
[direct: mongos] test>
```

Figura 4.14: Docker: Servidor de Shard

```
[direct: mongos] menu_database> db.menu_collection.getShardDistribution()
Shard rs-shard-01 at rs-shard-01/shard01-a:27017,shard01-b:27017,shard01-c:27017
{
  data: '0B',
  docs: 0,
  chunks: 2,
  'estimated data per chunk': '0B',
  'estimated docs per chunk': 0
}
---
Shard rs-shard-02 at rs-shard-02/shard02-a:27017,shard02-b:27017,shard02-c:27017
{
  data: '0B',
  docs: 0,
  chunks: 2,
  'estimated data per chunk': '0B',
  'estimated docs per chunk': 0
}
---
Totals
{
  data: '0B',
  docs: 0,
  chunks: 4,
  'Shard rs-shard-01': [ '0 % data', '0 % docs in cluster', '0B avg obj size on shard' ],
  'Shard rs-shard-02': [ '0 % data', '0 % docs in cluster', '0B avg obj size on shard' ]
}
[direct: mongos] menu_database>
```

Figura 4.15: Docker: Distribución de los Shards

```
[direct: mongos] wcrumble_database> db.wcrumble_collection.getShardDistribution()
Shard rs-shard-02 at rs-shard-02/shard02-a:27017,shard02-b:27017,shard02-c:27017
{
  data: '19KiB',
  docs: 34,
  chunks: 2,
  'estimated data per chunk': '9KiB',
  'estimated docs per chunk': 17
}
---
Shard rs-shard-01 at rs-shard-01/shard01-a:27017,shard01-b:27017,shard01-c:27017
{
  data: '20KiB',
  docs: 36,
  chunks: 2,
  'estimated data per chunk': '10KiB',
  'estimated docs per chunk': 18
}
---
Totals
{
  data: '39KiB',
  docs: 70,
  chunks: 4,
  'Shard rs-shard-02': [
    '48.09 % data',
    '48.57 % docs in cluster',
    '575B avg obj size on shard'
  ],
  'Shard rs-shard-01': [
    '51.9 % data',
    '51.42 % docs in cluster',
    '586B avg obj size on shard'
  ]
}
```

Figura 4.16: Docker: Distribución de los datos (34-36) tras aplicar Sharding.

5

Conclusiones y trabajos futuros

La Sección 5.1 está dedicada a la valoración de los objetivos y sub-objetivos cumplidos, donde se refrescan las tecnologías utilizadas y el aporte que proporciona cada una al conjunto global del proyecto.

En segundo lugar se expresan posibles mejoras o adiciones para generar un producto más robusto y completo en la Sección 5.2.

5.1. Conclusiones

Este proyecto surgió con el ánimo de aportar algo a la comunidad de Warcraft. De todas las opciones posibles, tanto juegos como proyectos, se decidió entregar un servicio aun no disponible, una API.

A pesar de que este servicio tiene un colectivo más reducido (desarrolladores), es importante remarcar que gracias a este tipo de proyectos e iniciativas se pueden crear páginas webs y otros servicios que pueden aprovechar los jugadores para obtener información de manera maquetada y sintetizada.

A continuación se indexarán los objetivos que se han cumplido al finalizar el proyecto:

- La información se transmite de manera rápida, ligera y sencilla gracias al formato JSON y la simbiosis que realiza con MongoDB.

- La API cuenta con un nombre sencillo y directo, haciendo que motores de búsqueda como Google o DuckDuckGo la posicionen al principio.
- Cuenta con una documentación detallada en GitHub para facilitar la comprensión de las consultas.
- Gracias a la unión de FastAPI y Uvicorn, se dispone de una concurrencia sólida.
- El servicio está construido bajo una arquitectura que aporta escalabilidad, disponibilidad y flexibilidad gracias a las técnicas de Sharding y Replica-Sets, así como la aplicación de Docker Compose sobre los diferentes microservicios.

5.2. Trabajos futuros

En esta última Sección se manifiestan un conjunto de mejoras que se podrían aplicar a lo largo del tiempo para mejorar la calidad del servicio.

- Aplicar diferentes capas de seguridad a la API. Entre las diferentes opciones se propone el uso de un token para poder realizar llamadas, limitar el número de llamadas por minuto o certificados SSL.
- Introducir una nueva versión de la API donde se conceda información de otros aspectos del juego como pueden ser los mapas y sus subniveles o las recompensas al subir de nivel tanto en el entorno PVE (Player Versus Enviroment) como PVP (Player Versus Player).
- Mejorar la UX con las herramientas que proporciona FastAPI. Ahora la API cuenta con diferentes endpoints y se asume que el usuario ha leído la documentación expuesta en GitHub. La idea es generar desplegables y proporcionar más descripciones para que no sea necesario consultar la información más básica en el GitHub, como puede ser el número total de unidades existentes o el tipo de facciones.

Bibliografía

[1] R. W. Leeron Hoory, Cassie Bottorff, "What is waterfall methodology?" *Forbes Advisor*, 2022. [Online]. Available: https://www.forbes.com/advisor/business/what-is-waterfall-methodology/

[2] AkshitaKumawat, "What is an api," *Geeks For Geeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/what-is-an-api/

[3] MongoDB, *MongoDB Manual*. [Online]. Available: https://www.mongodb.com/docs/manual/crud/#std-label-crud

[4] A. AWS, "¿cuál es la diferencia entre soap y rest?" *Amazon AWS*. [Online]. Available: https://aws.amazon.com/es/compare/the-difference-between-soap-rest/

[5] F. Tiangolo, *Tiangolo FastAPI Documentation*. [Online]. Available: https://fastapi.tiangolo.com/#requirements

[6] MongoDB, *MongoDB JSON and BSON*. [Online]. Available: https://www.mongodb.com/json-and-bson

[7] minhhungit, *MongoDB (6.0.1) Sharded Cluster with Docker Compose*. [Online]. Available: https://github.com/minhhungit/mongodb-cluster-docker-compose

[8] U. Encode, *Uvicorn Settings*. [Online]. Available: https://www.uvicorn.org/settings/

[9] F. Tiangolo, *FastAPI: Custom Response - HTML, Stream, File, others*. [Online]. Available: https://fastapi.tiangolo.com/es/advanced/custom-response/#redirectresponse

[10] M. W. Docs, *Mozilla: HTTP response status codes*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[11] Pydantic, *Pydantic Documentation*. [Online]. Available: https://docs.pydantic.dev/latest/

[12] techni621, "What is a pymongo cursor?" *Geeks For Geeks*, 2022. [Online]. Available: https://www.geeksforgeeks.org/what-is-a-pymongo-cursor/

[13] Docker, *Docker Compose File*. [Online]. Available: https://docs.docker.com/compose/compose-file/

[14] ——, *Docker Image Build*. [Online]. Available: https://docs.docker.com/reference/cli/docker/image/build/

[15] ——, *Docker Container Create*. [Online]. Available: https://docs.docker.com/reference/cli/docker/container/create/

[16] MongoDB, *MongoDB Manual*. [Online]. Available: https://www.mongodb.com/docs/manual/tutorial/deploy-replica-set/

[17] ——, *MongoDB Manual*. [Online]. Available: https://www.mongodb.com/docs/manual/reference/method/sh.shardCollection/

Apéndices

A

Modificaciones Código .yml

Código A.1: docker-compose.yml (1/5)

```
1
2 version: '3'
3 services:
4
5 ## App
6   WcrAPI:
7     image: wcrumble-image
8     container_name: wcrumble-container
9     ports:
10      - 7000:1234
11
12 ## Router
13   router01:
14     image: mongo:6.0.1
15     container_name: router-01
16     command: mongos --port 27017 --configdb rs-config-server/
          configsvr01:27017,configsvr02:27017 --bind_ip_all
17     ports:
18       - 27117:27017
19     restart: always
20     volumes:
21       - mongodb_cluster_router01_db:/data/db
22       - mongodb_cluster_router01_config:/data/configdb
23   router02:
24     image: mongo:6.0.1
25     container_name: router-02
26     command: mongos --port 27017 --configdb rs-config-server/
          configsvr01:27017,configsvr02:27017 --bind_ip_all
27     volumes:
28       - mongodb_cluster_router02_db:/data/db
29       - mongodb_cluster_router02_config:/data/configdb
30     ports:
31       - 27118:27017
32     restart: always
```

Código A.2: docker-compose.yml (2/5)

```
## Config Servers
  configsvr01:
    image: mongo:6.0.1
    container_name: mongo-config-01
    command: mongod --port 27017 --configsvr --replSet rs-
       config-server
    volumes:
      - mongodb_cluster_configsvr01_db:/data/db
      - mongodb_cluster_configsvr01_config:/data/configdb
    ports:
      - 27119:27017
    restart: always

  configsvr02:
    image: mongo:6.0.1
    container_name: mongo-config-02
    command: mongod --port 27017 --configsvr --replSet rs-
       config-server
    volumes:
      - mongodb_cluster_configsvr02_db:/data/db
      - mongodb_cluster_configsvr02_config:/data/configdb
    ports:
      - 27120:27017
    restart: always
```

Código A.3: docker-compose.yml (3/5)

```
## Shards
  ## Shards 01

  shard01-a:
    image: mongo:6.0.1
    container_name: shard-01-node-a
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -01
    volumes:
      - mongodb_cluster_shard01_a_db:/data/db
      - mongodb_cluster_shard01_a_config:/data/configdb
    ports:
      - 27122:27017
    restart: always

  shard01-b:
    image: mongo:6.0.1
    container_name: shard-01-node-b
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -01
    volumes:
      - mongodb_cluster_shard01_b_db:/data/db
      - mongodb_cluster_shard01_b_config:/data/configdb
    ports:
      - 27123:27017
    restart: always
  shard01-c:
    image: mongo:6.0.1
    container_name: shard-01-node-c
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -01
    volumes:
      - mongodb_cluster_shard01_c_db:/data/db
      - mongodb_cluster_shard01_c_config:/data/configdb
    ports:
      - 27124:27017
    restart: always
```

Código A.4: docker-compose.yml (4/5)

```
    ## Shards 02
  shard02-a:
    image: mongo:6.0.1
    container_name: shard-02-node-a
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -02
    volumes:
      - mongodb_cluster_shard02_a_db:/data/db
      - mongodb_cluster_shard02_a_config:/data/configdb
    ports:
      - 27125:27017
    restart: always

  shard02-b:
    image: mongo:6.0.1
    container_name: shard-02-node-b
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -02
    volumes:
      - mongodb_cluster_shard02_b_db:/data/db
      - mongodb_cluster_shard02_b_config:/data/configdb
    ports:
      - 27126:27017
    restart: always
  shard02-c:
    image: mongo:6.0.1
    container_name: shard-02-node-c
    command: mongod --port 27017 --shardsvr --replSet rs-shard
       -02
    volumes:
      - mongodb_cluster_shard02_c_db:/data/db
      - mongodb_cluster_shard02_c_config:/data/configdb
    ports:
      - 27127:27017
    restart: always
```

Código A.5: docker-compose.yml (5/5)

```
volumes:
  mongodb_cluster_router01_db:
  mongodb_cluster_router01_config:

  mongodb_cluster_router02_db:
  mongodb_cluster_router02_config:

  mongodb_cluster_configsvr01_db:
  mongodb_cluster_configsvr01_config:

  mongodb_cluster_configsvr02_db:
  mongodb_cluster_configsvr02_config:

  mongodb_cluster_shard01_a_db:
  mongodb_cluster_shard01_a_config:

  mongodb_cluster_shard01_b_db:
  mongodb_cluster_shard01_b_config:

  mongodb_cluster_shard01_c_db:
  mongodb_cluster_shard01_c_config:

  mongodb_cluster_shard02_a_db:
  mongodb_cluster_shard02_a_config:

  mongodb_cluster_shard02_b_db:
  mongodb_cluster_shard02_b_config:

  mongodb_cluster_shard02_c_db:
  mongodb_cluster_shard02_c_config:
```