**CESUR**

Proyecto Intermodular de Fin de Grado

Grado Superior

Desarrollo de Aplicaciones Multplataforma



**RESTful Leaderboard Crawler**

*Andy López Rey*

*Curso 2023 - 2025*

Índice

[Resumen 3](#_Toc193729525)

[Palabras clave 4](#_Toc193729526)

[Introducción 5](#_Toc193729527)

[Contexto 5](#_Toc193729528)

[Motivación 5](#_Toc193729529)

[Objetivo general 5](#_Toc193729530)

[Estrategia de Desarrollo 6](#_Toc193729531)

[Conclusión 6](#_Toc193729532)

[Objetivos 8](#_Toc193729533)

[Objetivo general 8](#_Toc193729534)

[Objetivos específicos 8](#_Toc193729535)

[Técnicos y funcionales 8](#_Toc193729536)

[Enfoque en la UI y UX 8](#_Toc193729537)

[Estrategia comercial y propuesta de valor 8](#_Toc193729538)

[Contextualización 10](#_Toc193729539)

[Análisis de Requisitos 11](#_Toc193729540)

[Requisitos funcionales 11](#_Toc193729541)

[Requisitos no funcionales 12](#_Toc193729542)

[Diseño 13](#_Toc193729543)

[Lenguajes y Herramientas 13](#_Toc193729544)

[Frontend 13](#_Toc193729545)

[Backend 16](#_Toc193729546)

[Base de datos 19](#_Toc193729547)

[Despliegue 23](#_Toc193729548)

[Decisión final 23](#_Toc193729549)

[Descripción de la arquitectura 24](#_Toc193729550)

[Diagramas y Modelos 26](#_Toc193729551)

[Planificación 27](#_Toc193729552)

[Desarrollo 28](#_Toc193729553)

[Metodología de trabajo 28](#_Toc193729554)

[Prototipo de la solución 28](#_Toc193729555)

[Conclusiones 29](#_Toc193729556)

[Bibliografía y fuentes 30](#_Toc193729557)

Resumen

aa

Palabras clave

* BDO
* MMORPG
* Guild
* API
* API REST
* MVC
* Single page application (SPA). Es una aplicación que carga una única página HTML y actualiza dinámicamente su contenido sin necesidad de recargar toda la página. Esto tiene como consecuencia un menor consumo de ancho de banda, experiencia fluida y JavaScript sólo en el lado del cliente, no servidor.  
  Esto quiere decir sin embargo que dependen enormemente en JavaScript, por lo motores de búsqueda para indexar (SEO) lo tendrán más difícil. Además, ha de cargarse toda la aplicación por adelantado, lo cual podría suponer un mayor tiempo de carga inicial.
* Progressive web application (PWA). Es una aplicación web optimizada para móviles que puede funcionar sin conexión. Para ello, almacena recursos en caché para trabajar offline.
* Librería.
* Framework.

Introducción

Los videojuegos en línea han evolucionado significativamente en los últimos años, generando comunidades activas y sistemas de competencia altamente organizados. Black Desert Online (BDO), un MMORPG de gran alcance global, es un claro ejemplo de esta tendencia. Sin embargo, a pesar de su popularidad, no existen herramientas oficiales que permitan a los jugadores realizar un seguimiento preciso de su progreso dentro del juego. En particular, la ausencia de sistemas eficientes de clasificación impide la comparación de rendimiento entre jugadores y guilds, lo que limita la competitividad y la interacción dentro de la comunidad.

Contexto

Actualmente, las herramientas disponibles para el seguimiento de rankings en BDO son ineficientes y están desactualizadas. Los jugadores que las utilizan lo hacen por falta de alternativas, enfrentándose a problemas de rendimiento y funcionalidad limitada. Esta situación resalta la necesidad de desarrollar una solución más robusta y eficiente que satisfaga las expectativas de la comunidad.​

Motivación

La motivación principal de este proyecto radica en la demanda real de una gran parte de la comunidad de BDO por una herramienta que les permita:​

* Visualizar rankings actualizados de jugadores y guilds.​
* Comparar niveles y estadísticas de manera eficiente.
* Análisis comparativo entre jugadores y guilds para adoptar estrategias de optimización y mejora.
* Acceder a datos históricos para identificar tendencias y evaluar el progreso individual y colectivo.

Esta necesidad se ve acentuada por la gran cantidad de datos y el volumen de jugadores que participan en BDO, especialmente en regiones como Europa (EU), Norteamérica (NA) y Corea del Sur (KR). Por ejemplo, según datos de Steam Charts, BDO ha alcanzado picos de hasta 59,917 jugadores concurrentes en la plataforma de Steam. Además, se estima que la base de jugadores diaria supera los 230,000 a nivel global *(sólo en Steam; el juego tiene su propio cliente que es bastante más antiguo y usado)*.

Objetivo general

Desarrollar una página web dinámica que consuma una API REST externa para mostrar clasificaciones de jugadores y guilds en BDO. Los datos obtenidos se almacenarán en una base de datos para:​

* Incrementar la eficiencia de las consultas.​
* Mantener un histórico de datos.
* ​Presentar un top 30 de jugadores y guilds.​
* Comparaciones entre jugadores y guilds.

La visualización de estos datos se realizará de forma intuitiva y atractiva para mejorar la experiencia del usuario.​

Estrategia de desarrollo

El sistema se desarrollará utilizando una arquitectura basada en:​

* Frontend: React.js.
* Backend: Spring Boot.​​
* Base de Datos: MongoDB.
* Deploy: *Aún por determinar.*

Se implementarán estrategias de caché, actualización selectiva y procesamiento por lotes para optimizar el rendimiento y reducir la carga en la API externa. Además, se priorizará la optimización y el rendimiento del sistema, garantizando una interfaz de usuario intuitiva y una gestión eficiente de la base de datos para minimizar las llamadas a la API REST.​

Se optará por un patrón de arquitectura de software llamado **Modelo – Vista – Controlador** (MVC). Con este patrón se separarán claramente las responsabilidades del software por capas muy bien definidas y aisladas. El objetivo principal de esta decisión es facilitar la modularidad, mantenibilidad y escalabilidad de la aplicación. Brevemente, cada capa:

* Vista. Encargada de la interfaz de usuario (UI) y la interacción. Tiene como responsabilidad principal mostrar datos y recoger eventos de entrada por parte del usuario. React.js cumplirá esta función.
* Modelo. Representa la capa de datos y lógica de negocio de la aplicación. Por tanto, manejará: (1) acceso a la base de datos, (2) validación y manipulación de datos y (3) reglas de negocio (por ejemplo el cálculo del ranking).
* Controlador. Es el intermediario entre la vista y el modelo. Se encargará de recibir solicitudes de la vista (es decir, peticiones del usuario), consultar la base de datos y procesar la respuesta de la petición y devolver una respuesta y exponer la API REST para tratar datos.

*Es posible que algunas de estas herramientas cambien a futuro. Aunque se han analizado otras opciones para cada capa del producto (Angular, Vue, Python, Hibernate, MySQL, PostgreSQL…), nuevos requerimientos a la hora de profundizar en el proyecto pueden hacer que se opte por un cambio en la arquitectura.*

Conclusión

Este proyecto busca satisfacer una necesidad palpable dentro de la comunidad de BDO, ofreciendo una herramienta eficiente y actualizada que mejore la competitividad y la interacción entre los jugadores. Al centrarse en la optimización del rendimiento y en una experiencia de usuario de calidad, se espera que esta solución se convierta en un recurso valioso para todos los entusiastas de Black Desert Online.

Dado que la compañía detrás de BDO no cuenta actualmente con una herramienta oficial que ofrezca este tipo de funcionalidades, y considerando la demanda por parte de la comunidad de jugadores, el desarrollo de esta plataforma representa una oportunidad potencialmente rentable. Si el software se diseña con altos estándares de calidad, optimización y usabilidad, no solo podría consolidarse como una solución indispensable para los jugadores, sino que también abriría la posibilidad de una eventual adquisición o colaboración con la empresa desarrolladora del juego.

Objetivos

Objetivo general

Desarrollar una plataforma web dinámica e integrada que permita la gestión y visualización eficiente de rankings de jugadores y guilds en BDO.

Esta herramienta se basará en el consumo de una API REST externa y en la optimización del almacenamiento y procesamiento de datos, a fin de mejorar la experiencia de usuario, proporcionar análisis detallados y establecer una solución escalable y comercialmente viable.

Objetivos específicos

Técnicos y funcionales

* Implementación de una API REST. Se desarrollará un backend usando Spring Boot para procesar y almacenar los datos obtenidos a través de la API REST a consumir *(BDO REST API)*. Enfoque en la optimización, mejorando tiempos de espera y evitando consultas innecesarias / pesadas que puedan disrumpir el flujo del programa.
* Base de datos. Se utilizará MongoDB para almacenar los rankings históricos, clasificación actual, jugadores, guilds, entre otros datos. Se hará uso de técnicas como indexación para asegurar consultar rápidas y escalabilidad al manejar grandes volúmenes de datos.
* Optimizar el consumo de datos a través de estrategias como: *crowdsourced data influx*, actualización selectiva, procesamiento por lotes *(batching)* y uso de caché *(¿Redis?)*.
* Garantizar la modularidad, legibilidad y mantenibilidad del código.

Enfoque en la UI y UX

* Diseñar un sistema accesible y de alto rendimiento, reduciendo los tiempos de carga y una experiencia fluida en cualquier dispositivo.
* Soporte multilingüe para que la plataforma sea accesible a distintas regiones.

Estrategia comercial y propuesta de valor

El producto también se desarrollará en mente para poder venderlo a la empresa desarrolladora de BDO. Para ello:

1. Posicionar el producto como una solución atractiva. Ideal para la comunidad del juego, que no tiene herramientas similares.
2. Diferenciarse de herramientas existentes. Actualmente sólo existe una plataforma similar. Enfatizando mejoras en la eficiencia, diseño y funcionalidades extras, se pretende hacer que el producto supere lo existente.
3. Propuestas de monetización. Venta de la licencia del software y posibilidad de implementar funcionalidades premium.
4. Evaluar la posibilidad de integrar la herramienta en el propio juego. El juego ya posee diversas herramientas integradas en él que consultan webs eternas.

Contextualización

Descripción de la empresa

Organigrama

Descripción del departamento

Descripción de los perfiles profesionales

Riesgos laborales

Análisis de Requisitos

La fase de análisis de requisitos constituye una etapa fundamental en el proceso de desarrollo de software, cuyo objetivo principal es identificar, analizar, especificar y validar todas las necesidades que el sistema debe satisfacer para ser considerado exitoso.

Esta fase es esencial porque proporciona claridad acerca de lo que el cliente o usuario final espera del sistema, asegurando que tanto los desarrolladores como los interesados compartan una visión común sobre el producto a construir.

Al concluir esta etapa, se espera obtener un conjunto claro, preciso y completo de requisitos, categorizados de manera adecuada, que sirvan de base sólida para las siguientes fases del desarrollo, como el diseño y la implementación.

Durante el análisis de requisitos se realiza una división crucial entre los requisitos funcionales y no funcionales. Los requisitos funcionales especifican claramente qué tareas o funciones concretas debe realizar el sistema; se centran en el comportamiento del sistema bajo condiciones específicas. Por otro lado, los requisitos no funcionales definen cómo el sistema debe comportarse en términos de rendimiento, usabilidad, seguridad, mantenibilidad, escalabilidad y otras características operativas que no son directamente funcionales, pero sí críticas para el éxito del proyecto.

Requisitos funcionales

Los requisitos funcionales de este proyecto son aquellas características que definen lo que la plataforma web debe ser capaz de hacer, centradas en las funciones que los usuarios podrán realizar.

En primer lugar, la plataforma debe permitir la gestión y visualización de rankings para jugadores y guilds, obteniendo los datos necesarios mediante consultas realizadas a una API REST externa. Estos rankings se almacenarán en una base de datos interna para garantizar una disponibilidad más eficiente y mejorar el rendimiento general del sistema.

Asimismo, la plataforma ofrecerá una interfaz web intuitiva que facilite a los usuarios realizar comparaciones entre jugadores individuales y guilds. Esto incluirá funcionalidades para filtrar y ordenar los datos según diversos criterios. Además, será posible visualizar tendencias históricas, proporcionando un registro detallado de los cambios ocurridos en los rankings durante un periodo específico, a través de representaciones gráficas y tablas.

Desde una perspectiva técnica, se adoptará una arquitectura modular basada en el patrón Modelo-Vista-Controlador (MVC). Esta decisión responde a la necesidad de mantener un sistema escalable, organizado y que facilite futuras actualizaciones y mantenimiento.

Adicionalmente, si los recursos y el tiempo del proyecto lo permiten, se incluirán funcionalidades secundarias tales como una comparación avanzada de guilds, permitiendo análisis detallados sobre la evolución completa de las mismas a lo largo del tiempo. También se contempla la posibilidad de exportar datos en formatos como CSV o JSON, así como optimizar aún más las consultas hacia la API REST y la base de datos utilizando sistemas de caché como Redis.

Por último, existen funcionalidades claramente delimitadas y excluidas de esta primera versión del producto, entre las que se encuentran la vinculación de cuentas de usuario con la aplicación web, la integración directa con el juego BDO, la monetización o publicidad inicial, así como cualquier interacción directa con los servidores de BDO más allá del consumo de su API REST.

Requisitos no funcionales

En cuanto a los requisitos no funcionales, éstos incluyen aspectos esenciales relacionados con el rendimiento, la escalabilidad, la usabilidad y la eficiencia general del sistema.

Por ejemplo, se prioriza la optimización del consumo de la API REST externa para minimizar latencias. La interfaz web debe ser altamente intuitiva para garantizar una experiencia de usuario satisfactoria, facilitando la navegación y comprensión de la información presentada.

Otro aspecto no funcional crítico es la escalabilidad, asegurando que la plataforma pueda crecer y adaptarse fácilmente a cambios futuros en términos de volumen de datos y número de usuarios, gracias al enfoque modular MVC adoptado. La robustez y eficiencia del almacenamiento de datos en la base interna también es un requisito importante, garantizando tiempos de respuesta rápidos y consistentes para los usuarios finales.

Diseño

Para garantizar un sistema eficiente, escalable y fácil de mantener, se han evaluado diversas tecnologías en cada una de las capas del proyecto, así como qué estrategia seguir para organizar el proyecto. A continuación, se presentan las opciones consideradas, junto con un análisis comparativo de sus ventajas y desventajas, y la justificación de la selección final.

Lenguajes y herramientas

Frontend

El frontend representa la capa de presentación en una aplicación web. Su función principal es gestionar la interacción entre el usuario y el sistema, proporcionando una experiencia visual e interactiva que permita la navegación eficiente y el acceso a los datos.

Desde el punto de vista arquitectónico, el frontend pertenece a la Vista (View) dentro del MVC.

Algunas de las características fundamentales del frontend son:

* Renderización de contenido. Muestra información al usuario de forma dinámica.
* Interactividad. Captura eventos por parte del usuario y los transmite al modelo. Luego se encargará de responder a dichas acciones, modificando el contenido mostrado en la web.
* Consumo de datos desde el backend.
* Optimización de rendimiento. Minimizar tiempos de carga, mejorar la accesibilidad y el rendimiento de la web.   
    
  Cuando hablemos de rendimiento hemos de diferenciar entre rendimiento de arranque y rendimiento de ejecución *(startup performance / runtime performance)*.  
    
  El rendimiento de arranque indica cuán rápida es una aplicación a la hora de iniciarse y cargar los componentes visuales necesarios hasta que el usuario pueda ver el contenido e interactuar con él. Este rendimiento se ve principalmente afectado al descargar el “bundle”. El bundle es la compilación del código del desarrollador. Las tres opciones listadas a continuación poseen un rendimiento de arranque similar.  
  Algunos factores que influyen: bundle size, compresión y minificación del código (Terser), carga diferida (lazy loading; cargar sólo lo necesario al principio) y optimización de imágenes y recursos (WebP, SVG).  
    
  Por otro lado, tenemos el rendimiento de ejecución. En este caso se analiza qué tan rápido responde la aplicación cuando el usuario interactúa con ella. Esta parte es influenciada tanto por el código del desarrollador, como por el framework en sí. Por lo que, a diferencia del rendimiento de arranque, aquí sí es importante analizar qué herramienta usar.  
  Algunos factores que influyen: eficiencia en la manipulación del DOM (React ya usa por ejemplo un Virtual DOM para esto), gestión del estado (datos que afectan a la UI), renderización condicional y re-renderizado de componentes.

Hoy en día existen un gran número de tecnologías capaz de llevar a cabo este cometido. Entre ellas:

* React.js.
* Vue.js.
* Angular.

React.js

React es una biblioteca de JavaScript para la construcción de interfaces de usuario declarativas y eficientes. Es decir, permite construir estructuras visuales de manera modular y reutilizable.

El enfoque clásico de usar HTML, CSS y JavaScript requiere de actualizar la página por completo. Con React, podemos actualizar únicamente el componente objetivo, optimizando mucho más las actualizaciones.

Se basa en un modelo o forma de trabajo de componentes reutilizables.

En comparación a las otras opciones, es la que menos funcionalidades aporta al desarrollador, pero a su vez es la que más ligera y más especializada en su trabajo: proporcionar una interfaz de usuario. Posee un enfoque minimalista.

Sin embargo, como consecuencia, al no tener muchas funcionalidades, es necesario importar muchos paquetes o librerías externas que no están implementadas en React.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Arquitectura basada en componentes, mejorando el mantenimiento y reutilización de código | Requiere de un gran número de librerías externas |
| Renderizado eficiente con Virtual DOM. Permite que el rendimiento de la web no se vea afectado por actualizaciones, actuando como memoria intermedia entre el estado de la página web y el DOM real | Mayor curva de aprendizaje con respecto a tecnologías más básicas como Vue.js |
| Extensa comunidad. Mantenido por Facebook |  |
| Fácil de integrar con otras librerías |  |

Angular

Angular se trata de un framework completo, desarrollado por Google basado en TypeScript. Está diseñado para la construcción de aplicaciones escalables y estructuradas. Por tanto, suele estar más orientado a aplicaciones grandes y completas que requieran de una arquitectura sólida.

Uno de los objetivos principales de Angular es aumentar las aplicaciones con capacidad MVC, con el fin de hacer que el desarrollo y las pruebas sean más fáciles.

Angular ya proporciona por su parte una división automática del código a través de un enrutador. Esto permite que sólo se cargue el código necesario para procesar las vistas que se solicitna.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Arquitectura basada en módulos, lo cual lo hace ideal para proyectos grandes | Mayor curva de aprendizaje en comparación al resto de opciones. Requiere además de conocer sobre TypeScript y la arquitectura MVC |
| Integración nativa con TypeScript. Esto le aporta una mayor robustez y escalabilidad | Es un framework pesado. Por lo que habrá un mayor consumo de recursos |
| Directivas y binding bidireccional para la sincronización eficiente de datos |  |
| Soporte por parte de Google |  |

Vue.js

Vue es un framework progresivo (gradual y flexible; no debemos de adoptar un framework sí o sí) de JavaScript diseñado para la construcción de interfaces. Se caracteriza por una sintaxis muy sencilla y una curva de aprendizaje baja, siendo así un punto intermedio entre React y Angular.

Intenta seleccionar lo mejor de React y de Angular. Esto lo hace, por ejemplo, utilizando directivas y CLI como Angular para extender el comportamiento de los elementos HTML y un mejor control del proyecto (respectivamente) y a través de Vue Router. Vue Router permite la navegación en aplicaciones de una sola página (SWA) sin la necesidad de recargar la página completa, entre otras cosas.

Cabe mencionar que características avanzadas como enrutamiento y gestión de estados se ofrecen a través de librerías, pues la biblioteca central se centra sólo en la capa vista del modelo MVC.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Fácil de aprender y usar | Comunidad pequeña en comparación a React y Angular |
| Renderización eficiente con Virtual DOM | No es una buena opción para proyectos grandes |
| Pocas dependencias externas para funcionalidades básicas |  |
| Sistemas de directivas que facilita la manipulación del DOM |  |

Backend

El backend es el encargado de gestionar la lógica de negocios, persistencia de datos y la comunicación con el frontend a través de la API REST.

En nuestro caso a la hora de elegir una tecnología se han de tener en cuenta: rendimiento, escalabilidad, facilidad de mantenimiento y facilidad de uso.

A nivel personal, si bien con el frontend se deseaban explorar nuevas herramientas, con el backend se ha decidido tomar un acercamiento más conservador, acercándonos a tecnologías ya conocidas por el desarrollador.

Aún así, procederemos a analizar algunas de las opciones disponibles para un proyecto de este tipo. Y, al igual que con el frontend, tomar una decisión informada sobre qué herramienta utilizar para esta capa.

* Java.
* Node.js.
* Python.

Java (Spring Boot)

Spring Boot es un framework basado en Spring, diseñado para acelerar y simplificar el desarrollo de microservicios y aplicaciones web escalables. Su propósito es eliminar la configuración manual que normalmente se requiere en aplicaciones Java, permitiendo construir APIs REST y sistemas empresariales de forma eficiente. Se basa en tres características clave:

* Configuración automática. Además, posee configuraciones predeterminadas para reducir la complejidad inicial. Esto evita a los desarrolladores la tarea de configurar manualmente múltiples archivos o beans.
* Enfoque de configuración obstinado. En lugar de obligar al desarrollador a definir cada detalle cero, ofrece valores predeterminados óptimos para la gran mayoría de escenarios.
* Capacidad de crear aplicaciones autónomas. Es posible crear aplicaciones que puede ejecutarse sin necesidad de un servidor externo.

Esto permite desarrollar una aplicación con un mínimo de instalación y configuración.

Lo hace ideal para el desarrollo de APIs REST, microservicios y aplicaciones empresariales escalables.

Spring Boot, de forma similar a Angular, está diseñado a construir entornos concurrentes con un alto rendimiento. Por lo que en comparación a frameworks más ligeros como Go o Node.js, puede resultar menos óptimo en cuanto a rendimiento.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Poca configuración manual, facilitando la integración con bases de datos | Complejidad a la hora de aprenderlo |
| Ideal para grandes aplicaciones con muchas conexiones simultáneas | Si se usa para aplicaciones pequeñas, puede no ser lo más óptimo en cuanto a consumo de memoria |
| Integración con servicios de seguridad como JWT | Configuración avanzada compleja |
| Excelente para aplicaciones con alta concurrencia gracias a su rendimiento |  |
| Amplia documentación y soporte empresarial |  |

Por tanto, Spring Boot, al igual que Angular, está más bien dirigido a proyectos complejos y escalables.

Node

Node.js es un entorno de ejecución de JavaScript en el lado del servidor. Permite ejecutar código JavaScript fuera del propio navegador, convirtiéndolo en una opción popular para el desarrollo de API’s, microservicios y aplicaciones en tiempo real.

A diferencia de lenguajes tradicionales como Java o Python, Node.js usa un modelo basado en eventos y bucle de eventos *(Event Loop)*. Esto le permite manejar múltiples solicitudes sin necesidad de crear nuevos hilos para cada una.

Cada vez que un usuario hace una nueva solicitud HTTP, lenguajes como Java crean un nuevo hilo para manejarla. Esto quiere decir que si hay miles de solicitudes concurrentes, es posible que el sistema se sature, al haber un límite de hilos disponibles. Node.js, sin embargo, no hace esto. Usa un único hilo. Esto significa que hay menos consumo de memoria, menos sobrecarga y una mayor eficiencia de lectura y escritura de datos de forma asíncrona.

Node.js maneja múltiples solicitudes gracias a tres componentes clave:

1. Event Loop. Es un bucle que constantemente revisa si hay tareas pendientes por hacer.
2. Non-Blocking I/O. Se encarga de leer archivos, bases de datos o APIs sin detener el proceso principal.
3. Callbacks, Promises y async/await. Permiten manejar tareas asíncronas sin crear nuevos hilos. Esto lo hace porque en lugar de esperar a que una tarea acabe, pasa a la siguiente y ya volverá a la anterior cuando esté lista.

Node se caracteriza por ser rápido y eficiente. Además, al poseer el Event Loop tiene una arquitectura no bloqueando, dependiendo de un único hilo. Es posible reutilizar mucho código al usar JavaScript en el front y backend.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Código simple y fácil de desarrollar | No ideal para tareas que consuman mucha memoria |
| Posee un buen rendimiento para microservicios y APIs REST | No escala bien en aplicaciones monolíticas |
| Maneja eficientemente múltiples conexiones simultáneas | Código asíncrono difícil de testear |
| Gran soporte y muchas librerías disponibles | Poco eficiente para cálculos pesados |

Una aplicación monolítica es aquella que tiene todo su código (frontend, backend, lógica de negocio y base de datos) en un único bloque, ejecutándose en un solo servidor.

Node.js tiene problemas con aplicaciones de este tipo porque si bien es capaz de manejar muchas conexiones, no es ideal para tareas pesadas, lo cual puede ocurrir al tener todo el código centralizado. Además, al ejecutar todo en un único hilo, si algo falla puede afectar a toda la aplicación. Una forma de subsanar esto en alguna medida es separando la aplicación en microservicios.

Node.js es útil para APIs REST con alta concurrencia porque cada solicitud sólo necesita leer o escribir datos (I/O), sin hacer cálculos pesados. Un único hilo se puede encargar de esto sin problemas de memoria.

Otro gran punto a favor es que funciona bien con MongoDB y Redis la ser también asíncronas no bloqueantes.

Python (Flask)

Flask es un microframework para Python para crear aplicaciones web y APIs REST de manera rápida y flexible. Se enfoca en gran medida en la simplicidad y modularidad, permitiendo a los desarrolladores agregar sólo lo que se necesiten, sin sobrecargar como lo hacen otros frameworks como Django.

Flask utiliza librerías como Werkzeug para manejar solicitudes y respuestas HTTP y Jinja2 para el renderizado dinámico de plantillas HTML.

Su principal característica y punto atractivo es lo ligero que es.

Se suele usar cuando se quiere desarrollar APIs REST rápidamente, sirviendo datos al frontend. Ideal para aplicaciones web pequeñas o incluso medianas que no requieran de la complejidad aportada por Django (competidor directo). Además, al no imponer una estructura de organización, permite un alto grado de personalización.

Sin embargo, si la aplicación tendrá una alta concurrencia y se busca rendimiento, no se suele recomendar.

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Código minimalista y fácil de leer | No posee una estructura definida, lo que la flexibilidad que aporta a su vez puede ocasionar caos en proyectos grandes |
| Permite crear APIs rápidamente | Necesidad de agregar manualmente herramientas de autenticación y seguridad |
| Es ligero y rápido para aplicaciones pequeñas | No es óptimo para el manejo masivo de solicitudes |
| Fácil de integrar con bases de datos | No es ideal para aplicaciones monolíticas grandes |
| Compatible con un gran número de bibliotecas | Inferior en cuanto al manejo de I/O |

Base de datos

A la hora de elegir la base de datos más adecuada miraremos factores como:

1. Rendimiento.
2. Escalabilidad.
3. Estructura de datos.
4. Tipos de consultas que podemos realizar.

La base de datos manejará rankings, almacenamiento de históricos y comparaciones de datos.

Las bases de datos se dividen en dos tipos principales: (1) relacionales o SQL y (2) no relacionales o NoSQL.

Las bases de datos SQL se caracterizan por almacenar sus datos en estructuras en forma de tablas, mientras que las NoSQL lo hacen en formatos más flexibles, como pueden serlo documentos, claves – valor o grafos. Cada tipo tiene sus ventajas y desventajas y se ha de analizar minuciosamente qué se quiere que una base de datos haga en un proyecto a la hora de elegir una. Algunas cosas a tener en cuenta son:

* Volumen de datos.
* Requisitos de rendimiento.
* Escalabilidad.

**SQL**

Las bases de datos relacionales organizan la información en tablas con filas y columnas (tuplas de información), siguiendo un modelo muy estructurado. Los datos se relacionan entre sí a través de claves primarias y claves foráneas.

Cada entidad o modelo posee su propia tabla y se pueden relacionar entre ellas a través de la clave foránea. La clave primaria sirve como identificador único de cada tupla de información.

El lenguaje de consultas es SQL *(structured query language)*. Es posible relacionar tablas entre ellas con JOINs.

Una característica clave de las bases de datos relacionales es su propiedad ACID. Esto garantiza que las transacciones u operaciones sean:

* Atómicas. Garantiza que cada transacción sea una unidad indivisible, por lo que o bien se ejecuta en su totalidad o no se ejecuta.
* Consistentes. Los datos han de pasar de un estado válido a otro estado válido después de realizarse alguna transacción. Se han de mantener las restricciones y reglas definidas. En caso de que no sea así, se revertirá.
* Aisladas. Las transacciones concurrentes no deben de intervenir entre ellas. Cada transacción ha de ejecutarse como si fuera una única operación.
* Duraderas. Una vez que una transacción se confirme *(commit)*, sus efectos han de persistir, incluso ante fallos en el sistema.

Si bien esta propiedad garantiza una alta fiabilidad y consistencia, impacta en el rendimiento de escritura de datos. Mismo principio se aplica para la indexación y verificación de integridad de datos.

**NoSQL**

Las bases de datos no relacionales por otro lado almacenan los datos en formatos más flexibles que no requieren de una estructura tabular. Esto puede resultar muy útil cuando los datos, su estructura, puede cambiar con frecuencia o cuando el rendimiento es más importante que la integridad de los datos.

Hay diversos tipos de bases de datos no relacionales, entre los cuales:

* Bases de datos documentales (MongoDB). Almacenan datos en formato JSON/BSON.
* Claves – Valor (Redis). Datos organizados como *key:value*.
* Grafos (Neo4j). Especializadas en datos altamente conectados.

En las bases de datos NoSQL no existen relaciones explícitas entre los datos, sino que todo está en un único documento JSON. Son implícitas en el propio documento.

Las bases de datos NoSQL no poseen características ACID. Sin embargo, aún poseen una cierta consistencia: eventual. Este tipo de consistencia si bien no garantiza que los nodos tengan todos la misma información de forma inmediata, se asegura que, eventualmente, todos se encuentren en un mismo estado final; convergen al mismo estado. Esto tiene como implicaciones que las operaciones de lectura puedan devolver datos desactualizados. Este *trade-off* es aceptable cuando una pequeña ventana de inconsistencia es tolerable a cambio de un mayor rendimiento y escalabilidad.

**Diferencias claves**

|  |  |
| --- | --- |
| SQL | NoSQL |
| Tablas con filas y columnas | Documentos, clave – valor, columnas o grafos |
| Relaciones fuertes a través de claves foráneas | Relaciones implícitas dentro de los documentos |
| Escalabilidad vertical (mejora a través de hardware) | Escalabilidad horizontal (añadir más nodos o servidores para distribuir la carga) |
| Alto rendimiento en consultas complejas a través de JOINs | Mejor rendimiento en consultas simples |
| Rendimiento medio en escritura | Alto rendimiento en escritura |
| ACID, transacciones | Consistencia eventual |
| Ideal para datos muy bien estructurados y relaciones complejas entre ellos | Datos flexibles y de rápido crecimiento |

* Escalabilidad vertical. Es fácil de administrar y posee una menor latencia al encontrarse todos los datos en un único nodo. Sin embargo, está limitado por el hardware y el costo incrementa a medida que se adquiere hardware más potente.
* Escalabilidad horizontal. Es capaz de manejar grandes volúmenes de datos y soportar una alta tasa de solicitudes. Gracias a que su crecimiento es a través de más nodos o servidores, puede crecer según la demanda. Esto tiene como contraposición un incremento en la dificultad de gestionar y sincronizar los datos entre nodos y lleva consigo posibles problemas de latencia y consistencia de datos.

Si bien tradicionalmente las bases de datos relacionales escalan verticalmente, existen soluciones para implementar una escalabilidad horizontal.

Indexación

La indexación es un mecanismo utilizado en bases de datos, tanto SQL como NoSQL, para mejorar la eficiencia de búsquedas y otras operaciones de acceso a datos. Existen sutiles diferencias en la forma en que ambos tipos lo hacen.

Los índices permiten acceder a datos en Ologn o incluso O1. Sin ellos, se tendría que recorrer toda la tabla / estructura, siendo On su rendimiento. ***¿Apartado para explicar Big O notation?***

Para este proyecto en concreto estamos considerando las siguientes opciones:

* PostgreSQL.
* MongoDB.
* DB4O.
* Redis.

PostgreSQL

PostgreSQL es una base de datos relacional robusto, escalable y optimizado para la integridad de datos y consultas avanzadas.

Las estructuras de datos que se almacenan en PostgreSQL están claramente ordenadas y bien relacionadas entre ellas. Esto permite la realización de consultas complejas.

Para el proyecto en cuestión, sería una buena opción si se realizan muchas operaciones de comparación entre jugadores y guilds, así como cálculos matemáticos.

MongoDB

MongoDB es una base de datos no relacional que basa su estructura de datos en documentos. Está diseñada para tener una alta escalabilidad en volumen de datos, flexibilidad y buen rendimiento, en especial en consultas simples.

Además, MongoDB almacena su información en formato JSON/BSON, ideal para aplicaciones que atacan API REST que devuelven también JSON.

MongoDB se ve como una potencial opción gracias al gran rendimiento para almacenar información de jugadores, puntuaciones y guilds sin restricciones de esquema, pues se almacenan los JSON que devuelve la propia API. Además, las consultas simples por jugadores o por ranking, si se realiza una indexación, resultarían bastante eficientes.

La existencia de índices TTL *(Time-To-Live)* de MongoDB también cumplirán una función clave en el proyecto. Pues cierta información sólo se almacenará durante un período de tiempo determinado antes de su borrado sin necesidad de eventos o triggers propios de las bases de datos relacionales.

Por contraparte, se debe de considerar la posible repetición de información en los documentos al no tener una estructura rígida clásica de las bases de datos relacionales. Tampoco existen transacciones ACID como tal.

DB4O

Se ha considerado DB4O gracias a la gran integración que tiene con aplicaciones escritas en Java. Su principal ventaja es la capacidad de almacenar y recuperar objetos directamente, sin necesidad de pasar por un proceso de conversión a un modelo SQL o NoSQL.

Esto le proporciona un gran rendimiento si la aplicación se decide desarrollar en Java. Además, posee las consultas embebidas con *Query By Example (QBE)* o *Native Queries (NQ)* las cuales poseen un alto rendimiento y además son fáciles de implementar. Al tener un consumo muy bajo de recursos al no requerir una base de daos como tal, lo hace ideal para aplicaciones pequeñas o dispositivos embebidos.

Las limitaciones que presenta DB4O es que no es ideal para el almacenamiento de grandes volúmenes de datos o una alta concurrencia. Además, no posee soporte oficial desde 2014.

Redis

Redis *(Remote Dictionary Server)* es una base de datos no relacional en memoria, basada en un modelo clave – valor. Es una base de datos extremadamente rápida y se usa principalmente para el almacenamiento de datos en caché, sesiones de usuario, colas de mensajes y ranking de datos temporales.

No se trata de una base de datos de almacenamiento persistente. Redis, en este proyecto, no se considera para ese fin. La estamos considerando para mejorar en gran medida el rendimiento de la aplicación al almacenar datos que se pueden consultar recurrentemente en memoria, evitando consultas repetidas a la API REST o incluso a la propia base de datos persistente. Se usaría como herramienta complementaria para mejorar el rendimiento.

Los datos en Redis se almacenan con un tiempo de borrado. Similar al índice TTS de MongoDB.

En este proyecto Redis se podría utilizar para consultar frecuentes a la API REST externa, para guardar ranking o incluso la expiración automática de históricos de corto tiempo.

Despliegue

Decisión final

A continuación se proporcionan las tecnologías elegidas junto con las razones principales para ello.

React

* Opción más ligera entre las presentadas. Sólo se requiere una herramienta para presentar los datos y recoger los eventos por parte del usuario. Es decir, algo que cumpla la función de Vista en el MVC.
* Arquitectura basada en componentes es compatible con la estrategia de desarrollo que se quiere adoptar.
* Si se hace un uso correcto del DOM virtual, el rendimiento web se verá positivamente afectado.
* Supone una puerta para aprender JavaScript y JavaScript + XML, el lenguaje de React (JSX).
* Vue y Angular son opciones destinadas al desarrollo de aplicaciones más completas y complejas que la que se quiere tratar en este desarrollo.

Spring Boot

* Requiere de poca configuración para conectar con la base de datos y la vista.
* Ideal para aplicaciones que tendrán conexiones simultáneas.
* Buena opción para aplicaciones con alta concurrencia gracias al buen rendimiento que aporta.
* Ya se poseen conocimientos previos de Java, por lo que la curva de aprendizaje es mucho menor en comparación a las otras opciones.
* Un punto importante en contra es que Node.js es ideal para una aplicación que potencialmente hará un gran número de llamadas o solicitudes simultáneas a la API REST. Spring Boot palidece en comparación.

MongoDB

* Los datos recibidos por la API REST no presentan una estructura compleja ni son diversos. Al presentar una estructura simple, una base de datos relacional que requiere de una estructura rígida no es la mejor opción.
* Los datos se recibirán en formato JSON. MongoDB almacena los datos directamente en formato JSON/BSON, por lo que es ideal. Se ahorra un paso de conversión de estructura de datos, mejorando así el rendimiento de la aplicación.
* Si se indexan bien los documentos, presentará un mejor rendimiento de lectura.
* Los índices TTL son enormemente atractivos para el proyecto. No sólo porque satisfagan una necesidad crítica, sino también porque son una mejor opción en cuanto a rendimiento que triggers o eventos propios de bases de datos relacionales.
* Posee un gran rendimiento en escritura de datos. SQL ha de mantener la consistencia ACID a la hora de escribir datos, además de optimizar sus índices y estructuras de datos. Si bien esto otorga ciertas ventajas, lo hace a coste de rendimiento.
* No se tiene pensado realizar consultas o queries complejas, por lo que la facilidad para ello aportado por los JOINs de las bases de datos relacionadas no se ve aprovechada.

Mención especial a Redis. Redis, como se ha comentado, no es una base de datos persistente. Si bien no se ha elegido, sí que se tiene en mente integrarla en el proyecto si el tiempo de desarrollo lo permite. Es una de las características cruciales que se quieren abordar una vez terminado el core del proyecto gracias a la gran mejora de rendimiento que Redis puede llegar a proporcionar.

Descripción de la arquitectura

La arquitectura de software es la estructura fundamental de un sistema de software, compuesta por toda una serie de componentes, relaciones y reglas que gobiernan su diseño y evolución.

Es decir, es la forma en la que se organiza el código, módulos y servicios de forma que trabajen juntos, definiendo cómo se comunica uno con otro y cómo se gestionan sus responsabilidades.

Las arquitecturas de software surgieron ante una necesidad nacida del incremento en complejidad de los proyectos. A medida que las aplicaciones se hacían más complejas y grandes, se tuvo que establecer una normativa o conjunto de reglas para organizar cómo se crean. Sin una arquitectura, el software se vuelve difícil de:

* Mantener.
* Escalar.
* Probar.
* Comprender.

Pero para aplicar correctamente una arquitectura, se ha de planificar, diseñar y estudiar el proyecto en sí previamente a elegir una o varias arquitecturas. Entonces, ¿cómo se elige?

* Se ha de ajustar a los requisitos funcionales y no funcionales.
* Comprensible por los desarrolladores a cargo.
* Escalable.
* Ha de poder facilitar el testeo y la integración.

Hoy en día, es común aplicar múltiples arquitecturas. Se suelen combinar varios patrones arquitectónicos compatibles entre sí, aportando cada uno toda una serie de fortalezas en partes específicas del sistema.

Para este proyecto, se ha decidido realizar una combinación de tres arquitecturas complementarias, respondiendo a las necesidades de escalabilidad, mantenibilidad y separación de responsabilidades.

Arquitectura cliente-servidor

Es una arquitectura distribuida donde el sistema se divide en dos partes muy bien diferenciadas:

* El cliente, o frontend, responsable de solicitar servicios y de interactuar con el usuario. Se utilizará React.js (interfaz gráfica, navegación e interacción).
* El servidor, o backend, que response a las solicitudes con lógica de negocio y acceso a datos. A través de Spring Boot (procesamiento de datos, comunicación con la API REST, validación, acceso a base de datos y caché con Redis).

Esta arquitectura permite una clara separación de responsabilidades (interfaz y lógica de negocio). Permite escalar ambas partes por separado y, si se desea, desarrollar cada una de forma paralela. Además, es posible la reutilización de código. Pues el backend al sólo presentar datos es posible usarlo para otras interfaces, por ejemplo.

Arquitectura en capas

Este modelo permite organizar el sistema en capas lógicas, donde cada capa tiene una responsabilidad clara y se comunica sólo con la capa inmediatamente superior o inferior.

En este proyecto se diferenciarán las siguientes capas:

1. Capa de presentación. Controlador, que expondrá la API REST y manejará las peticiones del cliente.
2. Capa de servicios. Contendrá la lógica de negocio.
3. Capa de acceso a datos. Se comunicará con MongoDB y/o Redis.
4. Base de datos. Persistencia de los datos. Utilizando MongoDB de como persistencia semipermanente y Redis como caché.

Al aplicarla, se busca mejorar la modularidad, mantenimiento y las pruebas.

Modelo – Vista – Controlador (MVC)

Finalmente, a través del MVC se separa el software en tres componentes:

* Modelo. Representación y lógica de los datos.
* Vista. Interfaz de usuario.
* Controlador. Gestiona la comunicación entre vista y modelo, actuando de puente.

Este modelo reforzará la claridad del flujo de la información y desacopla el sistema.

Diagramas y modelos

Planificación

Desarrollo

Metodología de trabajo

Prototipo de la solución

Conclusiones

Bibliografía y fuentes

* <https://steamdb.info/app/582660/charts/> - steamcharts BDO population
* <https://github.com/man90es/BDO-REST-API> - BDO REST API GitHub
* <https://man90es.github.io/BDO-REST-API/> - API Documentation
* <https://bdo.hemlo.cc/leaderboards/> - hemlo