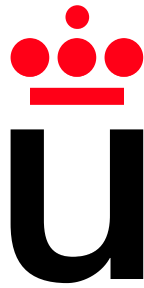

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

IMPLEMENTACIÓN DE CLIENTES WEB MEDIANTE PROGRAMACIÓN FUNCIONAL CON SCALA Y LOS FRAMEWORKS AKKA-HTTP Y AKKA-STREAM

Autor: Raúl Velasco Rubio
Tutor: Juan Manuel Serrano Hidalgo

10/10/2024

Agradecimientos

A mamá y Álvaro.

Resumen

Este proyecto consiste en la programación de clientes que accedan a APIs públicas de servicios Web para la descarga de datos, todo ello mediante el lenguaje Scala y su ecosistema de librerías de programación funcional. Para ello se han utilizado la librería de akka-http para acceder a endpoints simples y la librería de akka-stream para acceder a endpoints paginados, ambos de la API pública que pertenece al videojuego World of Warcraft.

Palabras clave:

- Programación funcional
- API
- Scala
- akka-http
- akka-stream
- World of Warcraft
- Bigdata
- Dataset

Índice de contenidos

Índice de figuras IX

Índice de códigos XII

1. Introducción 1

2. Objetivos 3

2.1. Metodología . . . 4

2.1.1. Librería y Dependencias . . . 5

3. Descripción informática 7

3.1. Web API World of Warcraft . . . 7

3.2. Acceso a endpoints simples . . . 12

3.2.1. Patrón de diseño . . . 12

3.2.2. Endpoint Character . . . 16

3.2.3. Endpoint Mount . . . 20

3.3. Acceso a endpoints paginados . . . 22

3.3.1. Patrón de diseño . . . 22

3.3.2. Clase Search . . . 25

3.4. Construcción del dataset . . . 27

4. Experimentos / validación 31

4.1. Descarga de datos . . . 31

4.2. Rendimiento . . . 33

5. Conclusiones 37

5.1. Líneas futuras . . . 38

Bibliografía 41

Índice de figuras

3.1. Petición a la API de WoW con el entorno integrado. 8
3.2. Respuesta de la API de WoW desde el entorno integrado. 8
3.3. Diagrama de clases de la clase **HttpEndpoint**. 13
3.4. Diagrama de clases de las clases del acceso al endpoint **Character**. 16
3.5. Diagrama de clases de las clases del acceso al endpoint **Mount**. . 20
3.6. Diagrama de clases de la clase **PaginationEndpoint**. 22
3.7. Diagrama de clases de la clase **Search**. 26

4.1. Construcción de los objetos **Request** y **Response** para descargar de un endpoint simple. 33
4.2. Construcción de los objetos **Request** y **Response** para descargar de un endpoint paginado. 34
4.3. Rendimiento del ordenador durante la ejecución del script. 35

Índice de códigos

3.1. Respuesta del recurso pedido en 3.2. 11
3.2. Petición de recurso a la API de WoW. 11
3.3. Clase **HttpEndpoint** y definición de sus instancias. 14
3.4. Estructura de paquetes y clases. 15
3.5. Definición de la clase **Character**. 17
3.6. Definición de la clase **Request** de **Character**. 17
3.7. Definición de la clase **Response** de **Character**. 18
3.8. Lógica de la función de cálculo de límites de peticiones. 18
3.9. Método **to** de la clase **Character**, donde se construye la petición HTTP. 19
3.10. Método **from** de la clase **Character**, donde se procesan las respuestas de la API. 20
3.11. Definición de la clase **MountSingle**. 21
3.12. Clase **PaginationEndpoint** y definición de sus instancias. 23
3.13. Método **foldResponse** del trait **PaginationEndpoint**. 24
3.14. Método **updateNextRequest** del trait **PaginationEndpoint**. . 24
3.15. Método **nextState** del trait **PaginationEndpoint**. 24
3.16. Método **stream** del trait **PaginationEndpoint**. 25
3.17. Definición de la clase **Search**. 26
3.18. Definición de la clase lista de **Mount**. 26
3.19. Implementación de los métodos abstractos **foldResponse** y **updateNextRequest**. 27
3.20. Clase **Main**, núcleo de ejecución de la aplicación. 28
3.21. Implementación del comando de acceso al endpoint **Mount** dentro de la clase **Command**. 29
3.22. Fragmento de la lógica de iteración sobre la lista de nombres de jugadores para la creación del dataset. 30

4.1. Generación del **bearer token** en línea de comandos. 31
4.2. Ejecución genérica de la aplicación. 31
4.3. Descarga de datos del personaje **Gordenor**. 32
4.4. Descarga de datos paginados de los objetos que incluyan la palabra **sword**. 32
4.5. Preparación del entorno del Notebook con los imports necesarios. 32
4.6. Definición de las variables necesarias para el entorno. 33

4.7. Ejemplo de entradas de tipo **Character** del dataset generado. . . 35

1
Introducción

Este proyecto representa la primera parte de un proyecto más amplio que abarca la descarga y manejo de datos a través de la programación funcional. Esta primera parte se trata de un cliente que permite obtener recursos de una API pública y, la segunda parte, es la creación de un dataset con esos recursos y la realización de consultas sobre el mismo.

Este proyecto se centra en la programación funcional, que se caracteriza por su énfasis en la inmutabilidad y las operaciones sin efectos secundarios, lo que facilita el razonamiento sobre el código y reduce los errores en sistemas complejos. Esto es especialmente relevante en el procesamiento de datos a gran escala, donde las operaciones deben ser eficientes y fáciles de paralelizar. Scala, el lenguaje de programación elegido para este proyecto, ofrece características como funciones de alto orden, tipado fuerte y patrones de diseño concisos, lo que la convierte en una elección ideal para manejar las complejidades y escalabilidad requeridas en el manejo de grandes volúmenes de datos.

Scala[1] es un lenguaje de programación moderno diseñado para abordar las necesidades de la era del Bigdata. Combina la programación funcional y orientada a objetos, permitiendo a los desarrolladores construir sistemas robustos y escalables. En el paradigma de la programación funcional, las funciones son ciudadanos de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como argumentos y retornadas desde otras funciones, permitiendo una mayor flexibilidad y reutilización del código.

Uno de los elementos clave para este proyecto son las APIs (o interfaces de programación de aplicaciones). Éstas suponen un elemento fundamental en el desarrollo de software moderno, permitiendo la comunicación entre diferentes

sistemas y servicios de manera estandarizada. Facilitan la integración y el manejo de funcionalidades ajenas al propio software, ampliando así sus capacidades sin necesidad de desarrollar complejas soluciones desde cero. Este proyecto se centra en la creación de un cliente que utilice la API pública del videojuego World of Warcraft[2], lo que permite la realización de consultas de información sobre distintos elementos del juego como los personajes, los objetos y las monturas. Este cliente permitirá la descarga de datos de la API pública para su posterior manipulación y análisis.

World of Warcraft (WoW) es uno de los juegos de rol multijugador masivos en línea (MMORPG por sus siglas en inglés) más populares y longevos del mundo. Desarrollado por Blizzard Entertainment, sumerge a los jugadores en un vasto y detallado universo de fantasía donde pueden explorar reinos extensos, luchar contra criaturas, completar misiones, y socializar con otros jugadores de todo el mundo. World of Warcraft se destaca por su rica narrativa, sistema de clases y razas, economía interna compleja, y eventos en constante evolución que mantienen el juego fresco y atractivo. En lo personal, llevo jugando a este videojuego desde 2005 y puedo decir que es mi favorito, me apasiona sumergirme en su mundo y disfrutar de las historias que año a año nos van revelando y entusiasmando.

Dentro de este contexto, World of Warcraft no es solo un juego, sino una fuente dinámica y rica de datos. La API de World of Warcraft permite acceder a una amplia gama de información, reflejando las acciones e interacciones de millones de jugadores. Estos datos ofrecen una oportunidad única para analizar patrones de comportamiento, tendencias en la economía virtual, y mucho más, convirtiendo a World of Warcraft en opción ideal para la exploración de datos en un entorno de entretenimiento y juego.

2
Objetivos

El objetivo principal de esta primera parte es desarrollar una aplicación o cliente con el que obtener datos de la API pública del videojuego World of Warcraft a través de la programación funcional, ya que este paradigma permite un manejo más eficiente y versátil de los datos. Esto nos permitirá después construir un dataset con la información y los datos obtenidos que podrá ser usado en otras aplicaciones. La descarga de datos desde una fuente tan rica y compleja como World of Warcraft ofrece una oportunidad única para investigar y desarrollar habilidades en la manipulación de grandes conjuntos de datos, lo que es fundamental en el campo de la ingeniería de datos.

1. **Análisis del funcionamiento de la Web API**

   Para lograr todo lo mencionado anteriormente, primero deberemos estudiar el funcionamiento de la API pública de World of Warcraft. Habrá que comprender los sistemas de autenticación que utiliza para poder acceder a sus endpoints. También habrá que estudiar los diferentes endpoints que la componen y cuáles de ellos son simples y cuáles paginados, para luego poder elegir los que incorporaremos en nuestra aplicación y así centrarnos en su estructura.

2. **Implementación del acceso a Endpoints Simples**

   Una vez analizada la API de World of Warcraft podremos diseñar la implementación del acceso a los endpoints simples. Este desarrollo debe ser sencillo y reutilizable, de manera que podamos optimizar el código lo máximo posible para abarcar diferentes endpoints bajo la misma estructura. Además, no solo deberemos centrarnos en la gestión efectiva de las respuestas si no también de los posibles errores.

3. **Implementación del acceso a Endpoints Paginados**

   En cuanto a los endpoints paginados, el objetivo es manejar adecuadamente grandes conjuntos de datos, lo que requiere una fuerte robustez para adaptarse a la diversidad de los diferentes tipos de recursos de la API. También hay que tener en cuenta la posible relación que pueda haber entre este tipo de endpoints y los simples que previamente habremos implementado, de nuevo, con la intención de optimizar nuestro código.

4. **Construcción del dataset**

   Una vez llegados a este punto podremos finalizar con la construcción del dataset. El objetivo a la hora de la descarga de datos es conseguir un dataset lo suficientemente grande y completo, lo que permitirá un análisis posterior más exhaustivo. Además, debe ser accesible para que las operaciones sobre él sean rápidas y mantengan la integridad del dataset al completo.

2.1. Metodología

El lenguaje principal utilizado en este proyecto es Scala[1], un lenguaje de programación de alto nivel que fusiona los paradigmas de programación orientada a objetos y funcional. Scala es conocido por su capacidad para manejar alta concurrencia y escalabilidad, lo que resulta especialmente útil en este proyecto, dado que implica hacer múltiples solicitudes a la API de World of Warcraft.

Todo el proceso de desarrollo de este proyecto se ha llevado a cabo en diferentes contextos. Para comenzar, se utilizaron dos IDEs (o entornos de desarrollo): **IntelliJ IDEA**[3] y **Visual Studio Code** (VS Code)[4]. Ambos IDEs son compatibles con Scala y ofrecen diversas funcionalidades y plugins para mejorar la

eficiencia del desarrollo, como el resaltado de sintaxis, el autocompletado de código y la integración con **Git**[5].

Por otro lado, como hemos mencionado que el desarrollo tuvo lugar en diferentes contextos, también hay que mencionar que tuvo lugar en diferentes máquinas con diferentes sistemas operativos. En concreto, este proyecto se realizó en dos sistemas operativos: **Windows**[6] y **MacOS**[7]. Esto demuestra que el proyecto es multiplataforma y puede ser desarrollado y desplegado en diversos entornos.

Para la ejecución del proyecto se utilizó la línea de comandos y un **Jupyter Notebook**[8]. El **Jupyter Notebook** se ejecutó a través de un contenedor **Docker**[9] con el kernel de Scala de **Almond**[10], proporcionando un entorno interactivo para ejecutar y manejar la aplicación. Además, se incluyen instrucciones para la ejecución del programa en ambos entornos dentro del **README.md** que se encuentra en el repositorio con el código.

El proyecto utiliza **sbt** (Scala Build Tool)[11] como herramienta de construcción y compilación. **sbt** se encarga de la compilación del proyecto, gestión de dependencias y empaquetado del proyecto para su despliegue. La configuración de **sbt** se encuentra en el archivo **build.sbt**, que contiene las instrucciones necesarias para compilar y ejecutar el proyecto.

2.1.1. Librería y Dependencias

Este proyecto depende de varias librerías y módulos, entre ellos:

- **Akka Stream[12] y Akka HTTP[13]:** Ambas forman parte del toolkit de Akka, que sirve para construir aplicaciones altamente concurrentes, distribuidas y resilientes en Scala. Akka HTTP es un conjunto de librerías que se usa para construir aplicaciones tanto de servidores con servicios RESTful como de aplicaciones de clientes HTTP. Por otro lado, Akka Stream es otro módulo de Akka que se centra en procesar y transformar streams de datos de manera que no se sature el sistema, asegurando que no se pierden datos. Es muy útil en aplicaciones que necesitan procesar largos streams de datos, como es nuestro caso, y se integra perfectamente con Akka HTTP para procesar streaming de peticiones y respuestas HTTP.

- **Spray JSON[14]:** Esta librería se utiliza para el procesamiento de JSON en Scala ya que permite serializar y deserializar JSON de manera muy sencilla, algo que es esencial para manipular las peticiones y respuestas de la API de World of Warcraft.

- **Case-app[15]:** Esta librería ayuda a construir aplicaciones de línea de comandos en Scala. Se encarga de procesar los argumentos que introducimos

en la línea de comandos facilitando la implementación de aplicaciones en CLI.

- **Cats**[16]**:** Es una librería que proporciona abstracciones para programación funcional en Scala. Ofrece diversas herramientas muy útiles en el desarrollo con programación funcional, como pueda ser su amplia variedad de **Type Classes** e **Implicits**.
- **Logback**[17]**:** Esta librería se utiliza para registrar eventos de la aplicación, lo cual es esencial para el seguimiento de problemas y el análisis de comportamientos. Permite de manera sencilla una configuración muy completa para poder manipular y formatear todos los mensajes de logs.

3

Descripción informática

Todo el código desarrollado ha sido almacenado en un repositorio personal de GitHub llamado **wow-api**[18] siguiendo las prácticas habituales de control de versiones. En este repositorio se pueden encontrar todos los ficheros que se mencionan a lo largo de este documento y además incluye un archivo **README.md** con información relevante del propio proyecto, su estructura, sus funcionalidades y su uso.

3.1. Web API World of Warcraft

La Web API de **World of Warcraft**[2] es un conjunto de servicios web proporcionados por **Blizzard Entertainment** que permiten a los desarrolladores interactuar con los datos del juego. La API proporciona acceso a una variedad de datos del juego, como información sobre personajes, ítems, monturas, y más. Además, la API incluye un entorno de pruebas integrado en la propia web desde la cual se pueden hacer peticiones como se muestra en las figuras 3.1 y 3.2. Las solicitudes a la API se hacen a través del protocolo **HTTP**, y las respuestas se proporcionan en formato **JSON**. En cuanto al **throttling**, los clientes de la API tienen un límite de 36000 peticiones por hora a un ritmo de 100 peticiones por segundo, ralentizando el servicio si se supera el primero de los límites y devolviendo un error 429 si se supera el segundo.

Si queremos hacer uso de esta API primero debemos autenticarnos. El proceso comienza creando una cuenta de usuario en la web de **Battle.net Developers**.

Mount API Expand All

GET **Mounts Index** /data/wow/mount/index

GET **Mount** /data/wow/mount/{mountId}

Returns a mount by ID.

Parameter	Type	Value	Description
:region	string REQUIRED	eu	The region of the data to retrieve.
{mountId}	integer REQUIRED	125	The ID of the mount.
namespace	string REQUIRED	static-eu	The namespace to use to locate this document.
locale	string	es_ES	The locale to reflect in localized data.

Try It | Clear Results | Clear OAuth Token

Request URL

```
https://eu.api.blizzard.com/data/wow/mount/125?namespace=static-eu&locale=es_ES&access_token=EUriFxPqrQf6WqQTQ6T23TLyA3XY
```

Figura 3.1: Petición a la API de WoW con el entorno integrado.

Response Status

```
200 OK
```

Response Headers

```
cache-control: public, max-age=86400
content-type: application/json;charset=UTF-8
last-modified: Wed, 16 Aug 2023 01:24:16 GMT
```

Response Body Copy


```
{
  "_links": {
    "self": {
      "href": "https://eu.api.blizzard.com/data/wow/mount/125?namespace=static-10.1.5_50232-eu"
    }
  },
  "id": 125,
  "name": "Tortuga de montar",
  "creature_displays": [
    {
      "key": {
        "href": "https://eu.api.blizzard.com/data/wow/media/creature-display/17158?namespace=static-10.1.5_50232-eu"
      },
      "id": 17158
    }
  ],
  "description": "Puede que el \"sin prisa, pero sin pausa\" no te haga ganar siempre la carrera, pero llegarás a la meta
```


Figura 3.2: Respuesta de la API de WoW desde el entorno integrado.

Aquí podremos generar un **Client ID** y un **Secret ID**, ambos necesarios para hacer la petición de generación de credenciales a la propia API en `https://oauth.battle.net/token`, lo que nos devolverá un **Bearer Token**, un elemento muy habitual en términos de autenticación a la hora de acceder a APIs públicas.

Una vez autenticados podremos comenzar las peticiones a la API. La estructura básica de estas peticiones incluye el token que hemos mencionado anteriormente, la dirección base para cualquier API de Blizzard (`https://eu.api.blizzard.com/`), la dirección específica de cada endpoint, el **namespace** y la región con el idioma o **locale**. La región es la separación de información que hace el videojuego en sus servidores para un óptimo funcionamiento del mismo, que suele coincidir con continentes (Europa, Asia, Estados Unidos...). El **locale** es un parametro que modifica el idioma en el que serán solicitados y devueltos los datos, independientemente de la región, lo que nos permite solicitar información y datos de la región que deseemos en nuestro idioma. Los namespaces, por otro lado, son la distinción que hace Blizzard de sus endpoints en función del tipo de información que contienen. Podemos distinguir entre tres tipos de namespaces: **estáticos (static)**, que recopilan información constante y permantente en el juego como objetos y monturas; **dinámicos (dynamic)**, que albergan información cambiante en tiempo real como la economía y el valor de la moneda del juego; y de **perfiles (profile)**, que contienen información específica de los personajes de los jugadores.

Dentro de la API de World of Warcraft encontramos una variedad muy amplia de endpoints. La primera distinción que nos ofrecen es entre **Profile APIs**, donde encontramos todos los endpoints con namespace de tipo **profile**; y **Game Data APIs**, que recopila todos los endpoints de los tipos **static** y **dynamic**.

Si repasamos la sección de **Profile APIs** encontramos casi veinte endpoints diferentes:

Character Achievements Información referente a los logros obtenidos dentro del juego por un personaje.

Character Appearance Donde podemos recuperar archivos en formato de imagen sobre la apariencia de los personajes.

Character Collections Recopilación de los diferentes coleccionables del juego, como objetos, monturas, mascotas..

Character Encounters Información de los enfrentamientos de un jugador en las diferentes mazmorras y bandas del juego.

Character Mythic Keystone Profile Estadísticas del progreso de mazmorras míticas en cada temporada.

Character Profile Conjunto de datos de todo tipo sobre un personaje en concreto, desde nivel, equipo, clase y raza... Aquí también podemos encontrar referencia a los otros endpoints con la información específica que recoge ese endpoint.

Guild Información sobre hermandades, miembros, estadísticas...

...

Por otro lado tenemos la sección de **Game Data APIs**, donde tenemos más de treinta endpoints disponibles:

Achievement A diferencia del homólogo anterior, este recopila información de todos los logros posibles, sus requisitos y sus recompensas.

Auction House Este, junto al de **WoW Token** son los dos únicos endpoints con namespace de tipo **dynamic**. El primero alberga el estado actual de la casa de subastas en cada región y su disponibilidad, mientras que el segundo contiene información en tiempo real de la evolución del valor de la moneda dentro del juego.

Creature Todos los datos posibles sobre cada una de las criaturas y monstruos del juego.

Item Toda la información sobre cada uno de los objetos obtenibles dentro del juego, desde los consumibles de un solo uso hasta los equipables como las armaduras.

Mount La colección completa de todas las monturas disponibles en el juego.

Profession Información del progreso, recompensas y características de todas las profesiones del juego.

Quest Detalles, requisitos y recompensas de las misiones que podemos completar.

Spell Todos los hechizos de todas las clases con su información y su evolución a lo largo de los niveles.

...

Todos estos endpoints son de tipo simple, lo que quiere decir que devuelven una única instancia de un recurso de la API. Por ejemplo, si hablamos del endpoint simple de **Mount**, en cada petición podríamos recuperar la información de una única montura en concreto.

Por otro lado estarían los endpoints de tipo paginado, en los que obtenemos una lista o listas de recursos o, lo que es lo mismo, varios recursos simultáneamente. En el caso del endpoint de tipo paginado de **Mount** podríamos recuperar una lista de la información de varias monturas en una sola petición. De los endpoints que hemos mencionado anteriormente solo algunos están adaptados y tienen una versión de tipo paginado. **Spell, Mount e Item** son ejemplos de esto y mantienen una estructura similar respecto a los de tipo simple, aunque con diferencias a la hora de realizar la solicitud que explicaré en la sección 3.3. En el caso de la sección de **Profile APIs** no vamos a encontrar ningún endpoint que sea de tipo paginado.

He incluido un par de ejemplos tanto de peticiones como de respuestas a la API de World of Warcraft. El primero de ellos es el que aparece en las figuras 3.1 y 3.2, que ha tenido lugar a través de la propia interfaz web que proporciona la propia API que ya hemos mencionado. El segundo ejemplo es el que se puede observar en los códigos 3.2 y 3.1, que se ha realizado a través de la línea de comandos utilizando la función **curl**.

```
{"_links":{"self":{"href":"https://eu.api.blizzard.com/data/wow/mount
↪  /101?namespace=static-10.1.5_50232-eu"}}, "id":101, "name":"Gran
↪  kodo blanco", "creature_displays":[{"key":{"href":"https://eu.api
↪  .blizzard.com/data/wow/media/creature-display/14349?namespace=sta
↪  tic-10.1.5_50232-eu"}, "id":14349}],"description":"Un extraño
↪  ejemplar albino de la especie que se descubrió por primera vez en
↪  Desolace. Los centauros creen que los kodos blancos son heraldos
↪  de la destrucción.",
↪  "source":{"type":"VENDOR","name":"Vendedor"},
↪  "faction":{"type":"HORDE","name":"Horda"},
↪  "requirements":{"faction":{"type":"HORDE","name":"Horda"}}}
```

Código 3.1: Respuesta del recurso pedido en 3.2.

```
curl -H "Authorization: Bearer ${BEARER_TOKEN}" https://eu.api.blizza
↪  rd.com/data/wow/mount/101?namespace=static-eu&locale=es_ES
```

Código 3.2: Petición de recurso a la API de WoW.

3.2. Acceso a endpoints simples

Para la implementación tanto de los clientes de acceso a endpoints simples como de los clientes de acceso a endpoints paginados he utilizado como referencia el proyecto de **hablatraining** de **TwitterV2**[19]. De este proyecto he extraído la estructura principal del proyecto y los patrones de diseño base para su desarrollo, especialmente para las clases **HttpEndpoint** y **PaginationEndpoint** que definiremos más adelante.

A la hora de seleccionar endpoints simples de la API de World of Warcraft a los que acceder he decidido escoger tres: dos con un namespace de tipo estático: **Mount** e **Item**, y otro con un namespace de tipo perfil: **Character**. El endpoint **Mount** almacena datos referentes a las monturas coleccionables del juego; el endpoint **Item** almacena información de todos los objetos disponibles para equipar a tus personajes; y el endpoint **Character** recoge datos de cada uno de los personajes que han creado los jugadores, incluyendo nombre, nivel, tiempo jugado... La elección de estos tres endpoints se debe a que considero que son bastante completos, almacenan muchos datos y de diferentes tipos y además son fáciles de comprender para alguien que no conozca el contexto del videojuego, por lo que me parecen bastante interesantes y versátiles.

3.2.1. Patrón de diseño

La implementación del acceso a endpoints simples se ha basado en el uso de las **Type classes**, un patrón de diseño esencial al estar dentro del paradigma de la programación funcional. Las **Type classes** permiten añadir nuevas funcionalidades a un tipo de datos cerrado sin modificar este último y sin necesidad de utilizar herencia. Este tipo de desacoplamiento ofrece mayor seguridad en tiempo de compilación sobre los propios tipos y mayor reusabilidad de las clases como tal. Estas **Type classes** están formadas por la **Type class** como tal, que se define como un **trait** con al menos un parámetro genérico; y las instancias de la **Type class**, donde se definen los tipos específicos que adoptará y las implemenctaciones concretas de los métodos de la **Type class**. Además, gracias a la librería **Cats** podemos aprovecharnos de la implementación de una **Interface syntax**, que simplifica enormemente el uso de los métodos de la **Type class**.

La clase **HttpEndpoint** sería un claro ejemplo de una **Type class**, como podemos ver en el diagrama 3.3 y en el código 3.3. Es un **trait** definido con el tipo genérico **Request** y con el tipo interno **Response**, donde los métodos que incluye se abstraen de estos tipos. Nos encontramos con dos métodos abstactos (**from** y **to**), de los que se encargarán de definir las instancias de **HttpEndpoint**; y un método ya definido que nos ofrece la **Type class**: **apply**. Más adelante definimos las instancias como tal del trait **HttpEndpoint** donde si que expresaremos el

tipo específico que adoptará la clase. En nuestro caso, este tipo es **Request**. También podemos observar en esta clase la mencionada sintaxis de la librería **Cats**, que simplifica el uso del método **apply** a través de una clase implícita y su método **single**.

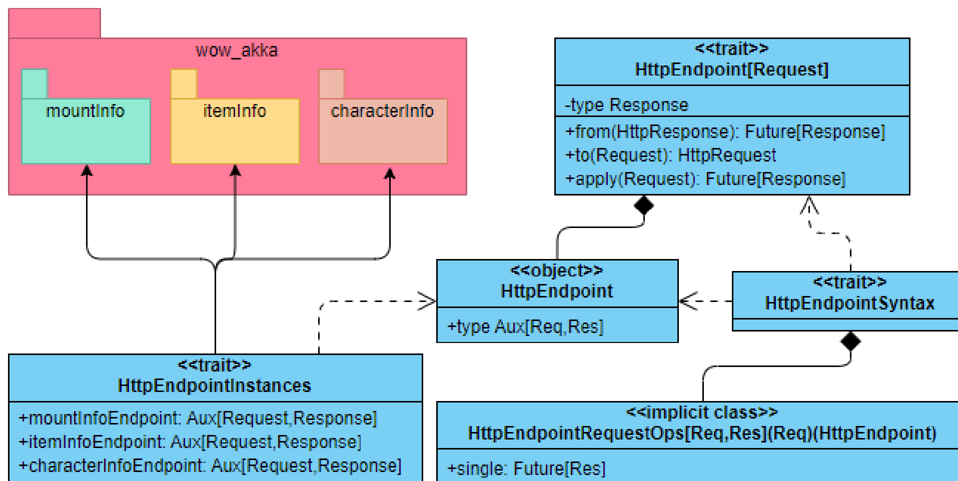

*Figura 3.3: Diagrama de clases de la clase **HttpEndpoint**.*

```
1  trait HttpEndpoint[Request] {
2    type Response
3    def from(response: HttpResponse)(implicit mat: Materializer, as:
     ↪  ExecutionContext): Future[Response]
4    def to(request: Request): HttpRequest
5    def apply(request: Request)(implicit system: ActorSystem[_], ec:
     ↪  ExecutionContext): Future[Response] =
6      Http().singleRequest(to(request))
7        .flatMap(from)
8  }
9
10 object HttpEndpoint {
11   type Aux[Req, Res] = HttpEndpoint[Req] {type Response = Res}
12 }
13
14 trait HttpEndpointSyntax {
15   implicit class HttpEndpointRequestOps[Req, Res](request:
     ↪  Req)(implicit ep: HttpEndpoint.Aux[Req, Res]) {
16     def single(implicit system: ActorSystem[_], ec:
       ↪  ExecutionContext): Future[Res] =
17       ep.apply(request)
18   }
19 }
20
21 trait HttpEndpointInstances {
```

```
22    implicit val mountInfoEndpoint:
      ↪  HttpEndpoint.Aux[wow.mountInfo.Request, wow.mountInfo.Response]
      ↪  =
23      wow_akka.mountInfo.Run
24    implicit val characterInfoEndpoint:
      ↪  HttpEndpoint.Aux[wow.characterInfo.Request,
      ↪  wow.characterInfo.Response] =
25      wow_akka.characterInfo.Run
26    implicit val itemInfoEndpoint:
      ↪  HttpEndpoint.Aux[wow.itemInfo.Request, wow.itemInfo.Response] =
27      wow_akka.itemInfo.Run
28  }
```

*Código 3.3: Clase **HttpEndpoint** y definición de sus instancias.*

Viendo lo anterior, también podríamos decir que la clase **HttpEndpoint** se podría ajustar al patrón de diseño de programación orientada a objetos **Strategy (Estrategia)**, pues esta clase supone el **Contexto** al ser una interfaz abstracta y delega la definición de los algoritmos específicos (los métodos **from** y **to**) a cada una de las estrategias concretas, es decir, a la definición de cada uno de los diferentes objetos de los endpoints escogidos. El uso de este patrón nos permite encapsular la lógica en cada una de las clases concretas, aumentando muchísimo la flexibilidad y la escalabilidad del proyecto, así como la limpieza del código.

Dicho todo lo anterior conviene hacer una mención a la estructura de paquetes y clases que presenta el proyecto y que podemos apreciar en la estructura 3.4. En ella podemos diferenciar tres paquetes principales: **main, wow y wow-akka**. En el paquete **main** nos encontramos la clase principal ejecutable **Main.scala** y la clase con las definiciones de la línea de comandos **Command.scala**, ambas las explicaremos más adelante. El paquete **wow** modela la API de World of Warcraft en términos de clases del framework **Akka HTTP**. Por último, el paquete **wow-akka** se encarga de la implementación particular del acceso a cada uno de los endpoints escogidos, además de incluir un conjunto de clases con funcionalidades adicionales para el funcionamiento del proyecto.

```
src/main
|-- resources
\-- scala
    |-- main
    |   |-- Command.scala
    |   \-- Main.scala
    |-- wow
    |   |-- Character.scala
    |   |-- Item.scala
    |   |-- Mount.scala
```

```
|   |-- Search.scala
|   |-- characterInfo
|   |   |-- Request.scala
|   |   \-- Response.scala
|   |-- itemInfo
|   |   |-- Request.scala
|   |   \-- Response.scala
|   |-- mountInfo
|   |   |-- Request.scala
|   |   \-- Response.scala
|   \-- searchData
|       |-- Request.scala
|       \-- Response.scala
\-- wow_akka
    |-- HttpBody.scala
    |-- HttpEndpoint.scala
    |-- package.scala
    |-- PaginationEndpoint.scala
    |-- QueryParams.scala
    |-- RateLimitHeaders.scala
    |-- characterInfo
    |   |-- From.scala
    |   |-- Run.scala
    |   \-- To.scala
    |-- itemInfo
    |   |-- From.scala
    |   |-- Run.scala
    |   \-- To.scala
    |-- mountInfo
    |   |-- From.scala
    |   |-- Run.scala
    |   \-- To.scala
    \-- searchData
        |-- From.scala
        |-- Run.scala
        |-- RunPagination.scala
        \-- To.scala
```

Código 3.4: Estructura de paquetes y clases.

3.2.2. Endpoint Character

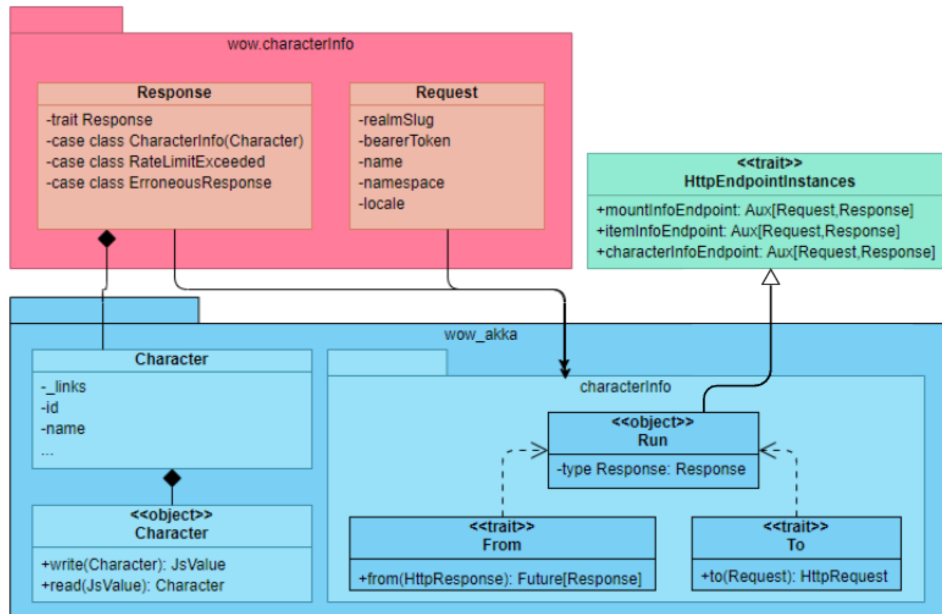

Figura 3.4: Diagrama de clases de las clases del acceso al endpoint ***Character****.*

La implementación del acceso a los endpoints simples comienza con la definición de las clases de las que queremos obtener información. Esta clase deberá incluir todos los campos del objeto que solicitamos al endpoint, además de los métodos necesarios de serialización/deserialización para su correcto procesamiento como podemos ver en 3.5. Este último proceso lo he implementado gracias a la librería **Spray JSON**[14].

```
1  case class Character(
2        _links: Option[JsValue],
3        id: Int,
4        name: String,
5        faction: Option[JsValue],
6        ... // Más definiciones de campos
7  )
8
9  object Character {
10   implicit val characterJsonFormat: RootJsonFormat[Character] = new
   ↪   RootJsonFormat[Character] {
11     def write(obj: Character): JsValue = {
12       JsObject(
13         "_links" -> obj._links.toJson,
14         "id" -> JsNumber(obj.id),
15         "name" -> JsString(obj.name),
16         "faction" -> obj.faction.toJson,
```

```
        ... // Más serializaciones
    }
    def read(json: JsValue): Character = {
      val fields = json.asJsObject.fields
      Character(
        _links = fields.get("_links").map(_.convertTo[JsValue]),
        id = fields("id").convertTo[Int],
        name = fields("name").convertTo[String],
        faction = fields.get("faction").map(_.convertTo[JsValue]),
        ... // Más deserializaciones
    }
  }
}
```

*Código 3.5: Definición de la clase **Character**.*

Posteriormente deberemos definir, para cada objeto, una clase que se encargue de definir las peticiones: **Request** (3.6); y una clase que se encargue de procesar las respuestas: **Response** (3.7). Es importante que esta última sea capaz de procesar tanto las respuestas correctas como las erróneas. Además, la respuesta debería ser capaz de almacenar los límites de peticiones que permite esta API, que generalmente se almacenan en los **headers** de respuesta.

```
case class Request(
    realmSlug: String,
    name: String,
    bearerToken: String,
    namespace: String,
    locale: String)
```

*Código 3.6: Definición de la clase **Request** de **Character**.*

```
sealed trait Response

case class CharacterInfo(body: Character, rateRemaining: Int,
↪  rateReset: Long) extends Response
case class RateLimitExceeded(rateResetTime: Long) extends Response

sealed trait ErroneousResponse extends Throwable with Response

case class ErroneousJsonResponse(jsonError: JsValue) extends
↪  ErroneousResponse {
  override def toString: String = if (jsonError != null)
  ↪  jsonError.toString else "NULL"
```

```
10 }
11
12 case class ErroneousTextResponse(textError: String) extends
   ↪ ErroneousResponse {
13   override def toString: String = textError
14 }
```

*Código 3.7: Definición de la clase **Response** de **Character**.*

En mi caso, estos límites de throttling los he implementado de manera manual basándome en la información que ofrece la documentación de la API pública de WoW[2], ya que los headers de respuesta no los devuelven. Para ello almacenamos en un fichero local el momento en el que tendrá lugar el reinicio del límite (una hora después de la primera solicitud) y el número de peticiones restantes hasta que ocurra ese momento, de manera que podremos activar el control de solicitudes gracias a este cálculo manual de peticiones restantes. Se puede observar la lógica de esta función en el código 3.8.

```
1  def parseRateLimitHeaders(response: HttpResponse): Option[(Int,
   ↪ Long)] = {
2      val headerOption = response.header[Date]
3      headerOption.flatMap { date =>
4        val currentTime = parseHttpDate(date.value)
5        readFile() match {
6          case (r, i) =>
7            rateRemaining = r
8            rateReset = i
9          case _ => rateReset = currentTime.plusSeconds(window);
           ↪ rateRemaining = 36000
10       }
11       timeRemaining = rateReset.getEpochSecond -
         ↪ currentTime.getEpochSecond
12       if (timeRemaining <= 0L || rateRemaining <= 0) {
13         rateRemaining = 36000
14         rateReset = currentTime.plusSeconds(window)
15         timeRemaining = window
16       }
17       rateRemaining -= 1
18       writeFile(rateRemaining, rateReset)
19       Some((rateRemaining, timeRemaining))
20     }}
```

Código 3.8: Lógica de la función de cálculo de límites de peticiones.

El núcleo de la implementación de endpoints reside en el trait **HttpEndpoint**, el cual mencionamos y explicamos anteriormente y se puede observar en el código 3.3. Este trait define una interfaz genérica para representar un endpoint HTTP y establece dos métodos principales que facilitan la interacción entre la lógica de la aplicación y las solicitudes HTTP: El método **from** se encarga de transformar una respuesta HTTP en un tipo de respuesta específico para la aplicación, mientras que el método **to** realiza la función opuesta, convirtiendo un objeto de solicitud específico de la aplicación en una solicitud HTTP.

Podemos ver ejemplos de estos métodos en los fragmentos de codigo 3.10 y 3.9, implementados gracias a la librería **Akka HTTP**[13].

```
1   def to(request: Request): HttpRequest = {
2     ... // Validaciones de los campos de la solicitud
3
4     // Use a StringBuilder to construct the URI string
5     val uri = new mutable.StringBuilder(s"https://eu.api.blizzard.com
      ↪  /profile/wow/character/")
6     uri.append(request.realmSlug)
7     uri.append("/")
8     uri.append(request.name)
9
10    val queryParams = Map[String, String]()
11      .add("namespace", request.namespace)
12      .add("locale", request.locale)
13    val query = Uri.Query(queryParams)
14    if (query.nonEmpty) {
15      uri.append(s"?$query")
16    }
17
18    // Construct and return the HttpRequest
19    HttpRequest(
20      uri = Uri(uri.toString()),
21      headers =
        ↪  Seq(Authorization(OAuth2BearerToken(request.bearerToken)))
22    )
23  }
```

*Código 3.9: Método **to** de la clase **Character**, donde se construye la petición HTTP.*

```scala
def from(response: HttpResponse)(implicit mat: Materializer, ec:
    ExecutionContext): Future[Response] = {
  parseBody(response).map { bodyE =>
    parseCharacterInfo(response, bodyE)
      .orElse(...) // Manejo de errores
  }
}

private def parseCharacterInfo(response: HttpResponse, bodyE:
    Either[String, JsValue]): Option[CharacterInfo] = {
  for {
    body <- bodyE.toOption if response.status == StatusCodes.OK
    character <- Try(body.convertTo[Character]).toOption
    (rateRemaining, rateReset) <- parseRateLimitHeaders(response)
  } yield CharacterInfo(character, rateRemaining, rateReset)
}
```

*Código 3.10: Método **from** de la clase **Character**, donde se procesan las respuestas de la API.*

3.2.3. Endpoint Mount

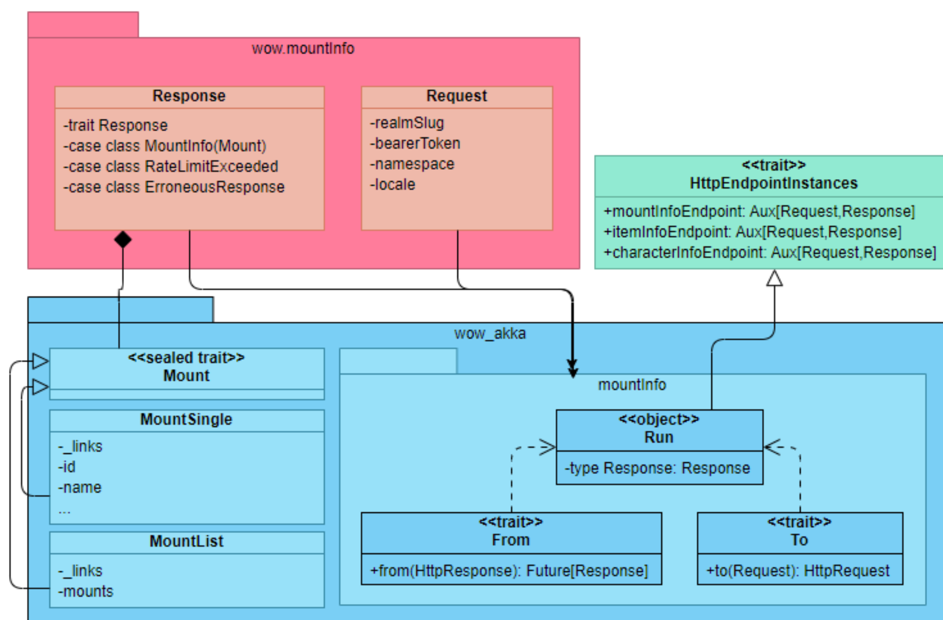

*Figura 3.5: Diagrama de clases de las clases del acceso al endpoint **Mount**.*

La implementación del acceso al endpoint **Mount** la hemos apoyado en el mismo patrón de las **Type classes**, como se puede observar en el diagrama 3.5. En este caso la instancia de **HttpEndpoint** que corresponde a este endpoint (**mountInfoEndpoint**) hace uso de las definiciones de **from y to** que están dentro del paquete **wow-akka**; además utilizar las clases **Response y Request** que están dentro del paquete **wow.mountInfo**.

La diferencia principal la encontramos en la propia definición de la clase **Mount**, que corresponde a la equivalencia del endpoint de la API de World of Warcraft homónimo a una clase de **Scala**. Aquí es donde podemos ver que la clase **Mount** de hecho se define como un **trait**, del que luego heredan las clases **MountSingle** y **MountList** que corresponden a las definiciones para el acceso a endpoints simples y paginados respectivamente. En la definición para el endpoint paginado nos centraremos más adelante, pero por ahora mencionaremos que la clase **MountSingle** es equivalente a la clase **Character** que vimos anteriormente, la cual incluye un objeto con los métodos necesarios para la serialización y deserialización de instancias de esta clase, como se observa en el código .

```
1   sealed trait Mount
2
3   case class MountSingle(
4                           _links: Option[JsValue],
5                           id: Int,
6                           name: String,
7                           creature_displays: Option[List[JsValue]],
8                           description: Option[String],
9                           source: Option[JsValue],
10                          faction: Option[JsValue],
11                          requirements: Option[JsValue]
12                        ) extends Mount
13
14  object MountSingle {
15    implicit val mountSingleJsonFormat: RootJsonFormat[MountSingle] =
      ↪  jsonFormat8(MountSingle.apply)
16  }
```

*Código 3.11: Definición de la clase **MountSingle**.*

3.3. Acceso a endpoints paginados

La estructura de los endpoints paginados en la API de World of Warcraft es muy similar entre todos ellos. Todos mantienen el mismo comienzo de la URL (`/data/wow/search/`) y finalizan con el nombre del recurso que va a ser devuelto. Además, en todos los endpoints paginados se puede añadir un filtro de palabras, para mostrar solo los recursos que contengan esa palabra en alguna parte de sus datos. Además de esto la respuesta de la API también mantiene una estructura similar, siendo uno de los parámetros de respuesta una lista con los objetos deseados devueltos.

Dada la naturaleza de este tipo de endpoints decidí adaptar tanto la clase **Mount** como la clase **Item** para que pudiesen formar esas listas de objetos que he mencionado antes. Esto, junto con una nueva clase **Search** suficientemente abstracta como para poder abarcar cualquier tipo de objeto que solicitemos a endpoints paginados, fueron las bases para la implementación de este cliente.

3.3.1. Patrón de diseño

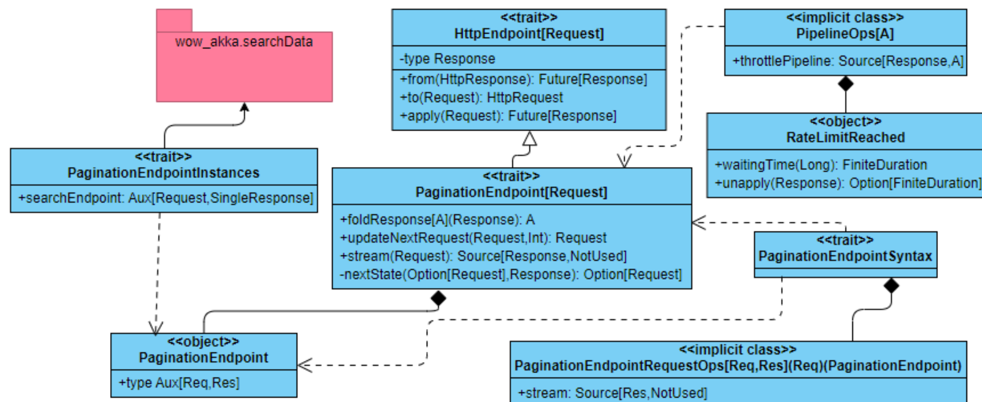

*Figura 3.6: Diagrama de clases de la clase **PaginationEndpoint**.*

Como mencionamos en el punto anterior y como se puede ver en el código 3.12 y en la figura 3.6, aquí también estamos usando las **Type classes**. La estructura es exactamente la misma que en la clase **HttpEndpoint** (y además hereda de esta última), donde estamos definiendo un trait con el tipo genérico **Request** y más adelante definimos la instancia específica de estos tipos. En este caso, únicamente lo instanciaremos con el tipo de la clase **Search** porque, como ya explicamos anteriormente, esta clase es capaz de interpretar los diferentes objetos que sean devueltos de la API.

```scala
trait PaginationEndpoint[Request] extends HttpEndpoint[Request] {
  ...
}

object PaginationEndpoint {
  type Aux[Req, Res] = PaginationEndpoint[Req] {type Response = Res}
}

trait PaginationEndpointInstances {
  implicit val searchEndpoint:
  ↪  PaginationEndpoint.Aux[wow.searchData.Request,
  ↪  wow.searchData.SingleResponse] = wow_akka.searchData.Run
}
```

*Código 3.12: Clase **PaginationEndpoint** y definición de sus instancias.*

Si entramos más en profundidad, los endpoints paginados en el proyecto se definen utilizando el trait **PaginationEndpoint**, que hereda del trait **HttpEndpoint**. Esta abstracción proporciona una interfaz para gestionar peticiones HTTP que devuelven resultados paginados a través de cuatro métodos clave, dos de ellos abstractos que delegan su definición a las instancias concretas de la **Type class** (**foldResponse y updateNextRequest**, y dos de ellos específicos que ofrece directamente **PaginationEndpoint** (**stream y nextState**. También nos encontramos con una clase implícita llamada **PipelineOps**, que incluye una serie de métodos necesarios para el manejo del throttling y los límites de peticiones.

- **foldResponse** (3.13): Un método abstracto que procesa la respuesta HTTP. Permite manejar respuestas exitosas, respuestas que indican un límite excedido y otros tipos de respuestas.
- **updateNextRequest** (3.14): Otro método abstracto que se encarga de, dada la petición actual, generar la próxima petición para continuar con el flujo.
- **stream** (3.16): Un método que crea un flujo (stream) de respuestas HTTP. Utiliza el método **unfoldAsync** para iterar a través de las páginas de resultados.
- **nextState** (3.15): Un método que se encarga de comprobar si hay que seguir solicitando páginas al endpoint y, en caso afirmativo, llama al método **updateNextRequest** para actualizar la solicitud y obtener la siguiente página de resultados.

El método **stream** es particularmente interesante, ya que permite procesar

múltiples páginas de resultados de forma asincrónica, aprovechando las capacidades de Akka Stream[12].

```
def foldResponse[A](response: Response)(
    ok: (Int, Long, Search) => A,
    limit: Long => A,
    other: => A): A
```

*Código 3.13: Método **foldResponse** del trait **PaginationEndpoint**.*

```
def updateNextRequest(request: Request, next: Int): Request
```

*Código 3.14: Método **updateNextRequest** del trait **PaginationEndpoint**.*

```
private def nextState(state: Option[Request], response: Response):
↪  Option[Request] =
state.fold(Option.empty[Request]) { request =>
  foldResponse(response)(
    (_, _, search) => {
      if (search.page >= search.pageCount) {
        None
      } else {
        val nextRequest = updateNextRequest(request, search.page + 1)
        Some(nextRequest)
      }
    },
    _ => {
      Some(request)
    },
    {
      None
    }
  )
}
```

*Código 3.15: Método **nextState** del trait **PaginationEndpoint**.*

```scala
def stream(search: Request)(implicit system: ActorSystem[_], ec:
    ExecutionContext): Source[Response, NotUsed] =
    Source.unfoldAsync(Option(search)) {
      case state@Some(request) =>
        apply(request).map(response => {
          Some((nextState(state, response), response))
        })
      case None =>
        Future.successful(None)
    }.throttlePipeline
```

*Código 3.16: Método **stream** del trait **PaginationEndpoint**.*

En la clase **PaginationEndpoint** también podríamos interpretar que se utiliza de la misma manera el patrón **Strategy**, sirviendo de **Contexto** o interfaz abstracta y delegando la implementación de la estrategia concreta de los métodos **stream** y **apply** a las clases específicas.

También podemos interpretar que la estructura de la interfaz **PaginationEndpoint** se basa en el patrón de diseño **Decorator (Decorador)**, ya que añade de manera dinámica nuevas funcionalidades a la interfaz **HttpEndpoint**, sobre la que se apoya y a la que recurre en tiempo de ejecución. Este patrón, además de promover el **Principio de abierto/cerrado**, proporciona mayor flexibilidad que la herencia al uso ya que no tenemos que modificar el código existente.

3.3.2. Clase Search

Siguiendo con la misma estructura que en la implementación del acceso a los endpoints simples, comenzamos por la definición de las clases **Search** (3.17, que es el tipo de objeto que obtendremos al realizar peticiones a endpoints paginados), **Request** y **Response**, y con ello procesaremos las peticiones y las respuestas de tipo paginado prácticamente igual que con los endpoints simples. La diferencia principal es que la clase **Search** recibe una lista de objetos, y ésta ha de abstraerse del tipo de objeto para poder procesar cualquiera de los ya definidos en Scala para los endpoints simples. Simplemente, como ya se mencionó, hemos tenido que adaptar esos objetos añadiendo una clase que herede del trait principal y que sea capaz de procesar una lista de los mismos, como en 3.18.

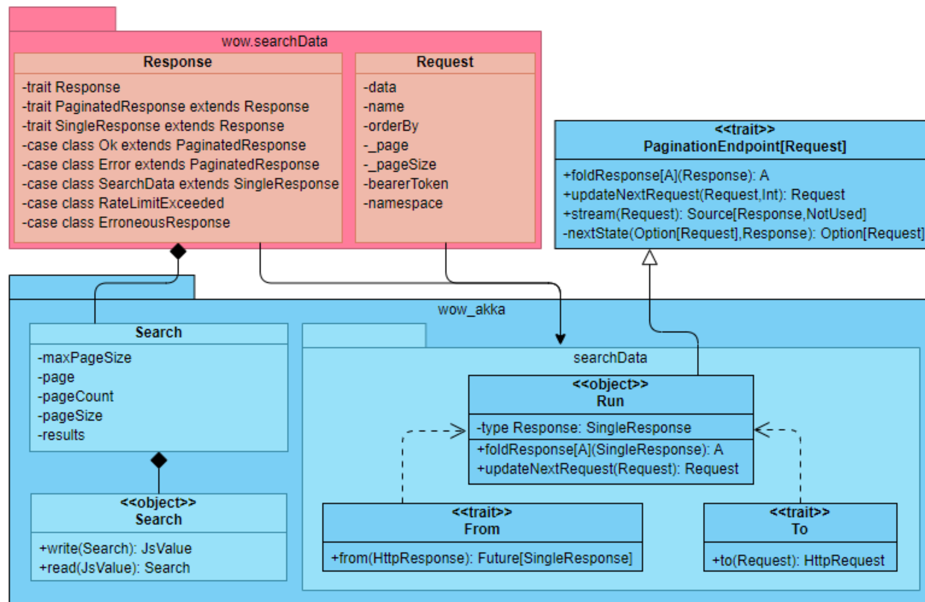

*Figura 3.7: Diagrama de clases de la clase **Search**.*

```scala
case class Search(
   maxPageSize: Int,
   page: Int,
   pageCount: Int,
   pageSize: Int,
   results: List[JsValue])

object Search {
  implicit val searchJsonFormat: RootJsonFormat[Search] =
   ↪ jsonFormat5(Search.apply)
}
```

*Código 3.17: Definición de la clase **Search**.*

```scala
case class MountList(
      _links: Option[JsValue],
      mounts: Option[List[JsValue]]
    ) extends Mount

object MountList {
  implicit val mountListJsonFormat: RootJsonFormat[MountList] =
   ↪ jsonFormat2(MountList.apply)
}
```

*Código 3.18: Definición de la clase lista de **Mount**.*

Otra de las diferencias respecto al acceso a endpoints simples es que ahora, debido a la nueva **Type class PaginationEndpoint**, se delega también la implementación de los métodos **foldResponse y updateNextRequest** a las instancias de ésta. En nuestro caso estas implementaciones las asume la clase **Search** en su objeto **Run**, como se puede ver en el código 3.19.

```
object Run extends PaginationEndpoint[Request]
  with From
  with To {
  type Response = SingleResponse

  // use pattern matching to handle different response types
  def foldResponse[A](response: SingleResponse)(ok: (Int, Long,
  ↪ Search) => A, limit: Long => A, other: => A): A = {
    response match {
      case SearchData(search, remaining, resetTime) => ok(remaining,
      ↪ resetTime, search)
      case RateLimitExceeded(resetTime) => limit(resetTime)
      case _ => other
    }
  }

  // update the page number for the next request
  def updateNextRequest(request: Request, next: Int): Request =
    request.copy(_page = next)
}
```

*Código 3.19: Implementación de los métodos abstractos **foldResponse** y **updateNextRequest**.*

3.4. Construcción del dataset

La clase **Main** sirve como núcleo de ejecución. Esta clase extiende la clase **CommandApp**, propia de la librería **Case-app**[15] que permite la gestión de los diferentes comandos que hemos implementado para acceder a cada uno de los endpoints escogidos para este proyecto, como se puede ver en el código 3.20 junto con el procesamiento del comando que hace las peticiones al endpoint de **Mount**.

```
@AppName("wow-api")
@AppVersion("1.0")
@ProgName("wow-api")
object Main extends CommandApp[Command] {

```

```
6    def run(command: Command, rargs: RemainingArgs): Unit =
7      command match {
8        case cmd: MountInfo => runMountInfo(cmd)
9        case cmd: CharacterInfo => runCharacterInfo(cmd)
10       case cmd: ItemInfo => runItemInfo(cmd)
11       case cmd: SearchData => runSearchData(cmd)
12       // More commands here
13     }
14
15   private def runMountInfo(cmd: MountInfo): Unit =
     ↪  withExecutionContext {
16     implicit system =>
17       implicit ec =>
18         try {
19           cmd.toMountInfo.fold(
20             error => Future.failed(new Exception(error)),
21             r => wow_akka.mountInfo.Run(r)
22           )
23         } catch {
24           case e: Exception => handleError(e, "MountInfo command")
25         }
26   }(println, _.printStackTrace)
```

*Código 3.20: Clase **Main**, núcleo de ejecución de la aplicación.*

Cada uno de estos comandos está definido dentro de la clase **Command**, la cual se apoya de nuevo en la librería **Case-app**[15] para implementar mensajes de ayuda y abreviaciones útiles en la ejecución por línea de comandos. También vemos en esta clase el uso de la librería **Cats**[16], pues la usamos para validar que la solicitud de datos se ha formulado de la manera correcta y, en base a los parámetros introducidos, nos encargaremos de añadir más o menos información cuando procesemos esa entrada con nuestro cliente. En el código podemos ver la implementación del comando **Mount**.

```
1  case class MountInfo(@HelpMessage("Mount ID") @ExtraName("id") id:
   ↪  Option[Int],
2                       @HelpMessage("Index") @ExtraName("in") index:
                        ↪  Boolean = false,
3                       @HelpMessage("Bearer Token") @ExtraName("bt")
                        ↪  bearerToken: String,
4                       @HelpMessage("Namespace (static-eu)")
                        ↪  @ExtraName("ns") namespace: String,
5                       @HelpMessage("Locale (es_ES)") @ExtraName("lo")
                        ↪  locale: String)
6
```

```
7      extends Command {
8
9      def toMountInfo: Validated[String, mountInfo.Request] =
10       (id, index) match {
11         case (Some(id), _) => Valid(mountInfo.Request(Left(id),
           ↪  bearerToken, namespace, locale))
12         case (_, true) => Valid(mountInfo.Request(Right(index),
           ↪  bearerToken, namespace, locale))
13         case _ => Invalid("You can only specify either index or id.")
14       }
15   }
```

*Código 3.21: Implementación del comando de acceso al endpoint **Mount** dentro de la clase **Command**.*

Para la construcción del dataset escogí utilizar el tercer endpoint simple: **Character**. Esta decisión tuvo varias motivaciones, siendo la primera de ellas exprimir más este endpoint ya que los otros dos habían tenido mayor presencia en los accesos a endpoints paginados. También decidí utilizar este endpoint porque es el más robusto de los tres, ya que los **Item** difieren mucho entre si en sus atributos y las **Mount** también. Por último elegí este endpoint porque considero que la información de cada personaje es muy interesante al haber sido elaborada por jugadores diferentes.

Primeramente quise obtener una lista de jugadores que perteneciesen al mismo servidor de juego en el que yo juego (**Sanguino**) y obtuve esa lista a través de la página **WoWProgress**[20]. Seguidamente desarrollé un script en **Bash** que iteraba sobre esa lista y llamaba a la aplicación de Scala para recolectar la información correspondiente al perfil de cada uno de estos jugadores. Como resultado se generaba un archivo en formato **JSON** que servirá para futuras aplicaciones de procesamiento y consulta de datos. Se puede ver un fragmento de este script en el código 3.22.

```
while IFS=$'\r\n' read -r character_name; do
    # Convert character name to lowercase and encode it
    ...

    # Fetch the character data
    result=$(java -jar "$JAR_PATH" character-info --rs=sanguino
    ↪  --cn="$character_name_encoded" --bt=xxxxx --ns=profile-eu
    ↪  --locale=es_ES)

    # Check if the result matches the undesired pattern, and if so,
    ↪  skip this iteration
    ...
```

```
    # If this is not the first entry, prepend a comma
    ...

    # Append the result to the output file
    echo "$result" >> $OUTPUT_FILE

done < sanguino-players.txt
```

Código 3.22: Fragmento de la lógica de iteración sobre la lista de nombres de jugadores para la creación del dataset.

4

Experimentos / validación

4.1. Descarga de datos

La aplicación desarrollada puede ser ejecutada a través de la línea de comandos, tal y como viene explicado en el **README.md** del repositorio de código. La descarga de datos a través de la línea de comandos comienza generando el correspondiente **bearer token** (4.1) a partir de las credenciales que podremos obtener en la web de desarrolladores de Blizzard[2], tal y como ya se explicó en el apartado 3.1.

```
curl -u {client_id}:{client_secret} -d grant_type=client_credentials
↪  https://oauth.battle.net/token
```

Código 4.1: Generación del ***bearer token*** *en línea de comandos.*

Para montar el fichero ejecutable podemos hacerlo directamente con el comando `sbt assembly` y debería de realizarse desde la carpeta **target/scala-2.13/** (`cd target/scala-2.13/`) y debe hacerse tal y como se indica en el código 4.2, ajustando la versión actual de la aplicación.

```
java -jar wow-api-1.0.jar [command] [options]
```

Código 4.2: Ejecución genérica de la aplicación.

Un ejemplo para la descarga de datos de mi personaje podría ser el que se realiza en el código 4.3, mientras que un ejemplo de descarga en un endpoint paginado podría ser el del código 4.4. El control y la depuración de estas ejecuciones pudo ser posible gracias a la librería **Logback**, pues se utilizaba para revisar cada una de las ejecuciones y devolver información relevante en caso de producirse cualquier tipo de error.

```
java -jar wow-api-1.0.jar character-info --rs=sanguino --cn=gordenor
↪  --bt=${BEARER_TOKEN} --ns=profile-eu --locale=es_ES
```

*Código 4.3: Descarga de datos del personaje **Gordenor**.*

```
java -jar wow-api-1.0.jar search-data --da=item --na=sword --ob=id
↪  --pa=1 --ps=100 --bt=${BEARER_TOKEN} --ns=static-eu
```

*Código 4.4: Descarga de datos paginados de los objetos que incluyan la palabra **sword**.*

También se puede probar la aplicación a través de un **Jupyter Notebook**[8]. Este Notebook se puede montar tal y como se explica en el fichero **README.ipynb** del repositorio de código y como se ha comentado previamente en el apartado 2.1.

Primeramente debemos importar las librerías y paquetes de la aplicación necesarios, como se muestra en el código 4.5.

```
1 import $ivy.`com.typesafe.akka::akka-http-spray-json:10.4.0`
2 import $ivy.`com.typesafe.akka::akka-slf4j:2.7.0`
3 ...
4 import $cp.target.`scala-2.13`.`wow-api_2.13-1.0.jar`
5 ...
6 import _root_.akka.actor.typed.ActorSystem
7 import akka.stream.scaladsl._
8 ...
9 import wow_akka._
```

Código 4.5: Preparación del entorno del Notebook con los imports necesarios.

Después definiremos tres variables necesarias para el entorno:

- **ActorSystem** : Variable implícita necesaria para las comunicaciones HTTP.
- **ExecutionContext** : Variable implícita necesaria para manejar los **Future**.
- **bearer token** : Variable que pasaremos por entorno y necesaria para las solicitudes a la API de WoW, como hemos explicado en el código 4.1.

```scala
implicit val system = ActorSystem(Behaviors.empty, "wow-api")
implicit val ec = system.executionContext
def bearerToken = scala.util.Properties.envOrElse("BEARER_TOKEN",
↪  "undefined")
```

Código 4.6: Definición de las variables necesarias para el entorno.

Finalmente definiremos los objetos de tipo **Request** y **Response**. Este último lo manejaremos a través de un **Future** en el caso de los endpoints simples (figura 4.1), y de un **Source** (stream) para el caso de los endpoints paginados (figura 4.2). La ventaja de la implementación de los endpoints paginados que he llevado a cabo es que se le pueden aplicar una serie de filtros al resultado, como limitar a un número concreto de respuestas.

```
[6]: val request = wow.mountInfo.Request(Left(363), bearerToken, "static-eu", "es_ES")

     val response: Future[wow.mountInfo.Response] = request.single

request: wow.mountInfo.Request = Request(
  idOrIndex = Left(value = 363),
  bearerToken = "EUaWnUirQxY7O9Z78TPjMYsMVo6L50NAyg",
  namespace = "static-eu",
  locale = "es_ES"
)
response: Future[wow.mountInfo.Response] = Success(
  value = MountInfo(
    body = MountSingle(
      _links = Some(
        value = JsObject(
          fields = TreeMap(
            "self" -> JsObject(
              fields = TreeMap(
                "href" -> JsString(
                  value = "https://eu.api.blizzard.com/data/wow/mount/363?namespace=static-10.1.5_50232-eu"
                )
              )
            )
          )
        )
      ),
      id = 363,
      name = "Invencible",
      creature_displays = Some(
        value = List(
          JsObject(
            fields = TreeMap(
              "id" -> JsNumber(value = 31007),
              "key" -> JsObject(
                fields = TreeMap(
                  "href" -> JsString(
                    value = "https://eu.api.blizzard.com/data/wow/media/creature-display/31007?namespace=static-10.1.5_50232-eu"
                  )
```

*Figura 4.1: Construcción de los objetos **Request** y **Response** para descargar de un endpoint simple.*

4.2. Rendimiento

Para la creación del dataset se ha utilizado una lista de **1518** entradas, cada una de ellas el nombre de un personaje diferente tal y como se explica en el apartado 3.4.

El script tarda aproximadamente unos **75 minutos** en procesar todas las entradas, dando error en **487**, lo que supone un **32 %** del total. Se genera un

```
[8]: val pagRequest = wow.searchData.Request("item", "sword", "id", 1, None, bearerToken, "static-eu")

     val pagResponse: Source[wow.searchData.SingleResponse, akka.NotUsed] = pagRequest.stream

[8]: pagRequest: wow.searchData.Request = Request(
       data = "item",
       name = "sword",
       orderBy = "id",
       _page = 1,
       _pageSize = None,
       bearerToken = "EUaWnUirQxY709Z78TPjMYsMVo6L50NAyg",
       namespace = "static-eu"
     )
     pagResponse: Source[wow.searchData.SingleResponse, akka.NotUsed] = Source(SourceShape(flatten.out(2070609079)))

[9]: pagResponse.take(2)
             .toMat(Sink.seq)(Keep.right) // RunnableGraph
             .run                         // Future

     res8: Future[Seq[wow.searchData.SingleResponse]] = Success(
       value = Vector(
         SearchData(
           body = Search(
             maxPageSize = 100,
             page = 1,
             pageCount = 4,
             pageSize = 100,
             results = List(
               JsObject(
                 fields = TreeMap(
                   "data" -> JsObject(
                     fields = TreeMap(
                       "id" -> JsNumber(value = 1008),
                       "inventory_type" -> JsObject(
                         fields = TreeMap(
                           "name" -> JsObject(
                             fields = TreeMap(
                               "de_DE" -> JsString(value = "Waffenhand"),
                               "en_GB" -> JsString(value = "Main Hand"),
                               "en_US" -> JsString(value = "Main Hand"),
```

*Figura 4.2: Construcción de los objetos **Request** y **Response** para descargar de un endpoint paginado.*

archivo **.json** que ocupa unos **4,4MB**. El dataset resultante contiene un total de **1031** entradas de tipo **Character** y un ejemplo de estas entradas sería el que tenemos en el fragmento 4.7, que está recortado porque cada entrada tiene una gran cantidad de atributos y estos últimos varios subatributos. Es interesante destacar como algunos de estos atributos directamente hacen referencia a otros endpoints de la API.


```
{
  "body": {
    "_links": {
    ...
    },
    "achievement_points": 21570,
    "achievements": {
      "href": "https://eu.api.blizzard.com/profile/wow/character/sang
      ↪  uino/noraym/achievements?namespace=profile-eu"
    },
    "achievements_statistics": null,
    "active_spec": {
      "id": 1467,
      "key": {
      ...
      },
```

```
      "name": "Devastación"
    },
    "appearance": {
      "href": "https://eu.api.blizzard.com/profile/wow/character/sang
      ↪  uino/noraym/appearance?namespace=profile-eu"
    },
    "average_item_level": 448,
    "character_class": {
      "id": 13,
      "key": {
      ...
      },
      "name": "Evocador"
    },
    "name": "Noraym",
     ...
  }
}
```

*Código 4.7: Ejemplo de entradas de tipo **Character** del dataset generado.*

Como podemos ver en la figura 4.3, el consumo de recursos que tuvo lugar durante la ejecución del script no fue excesivamente alto.

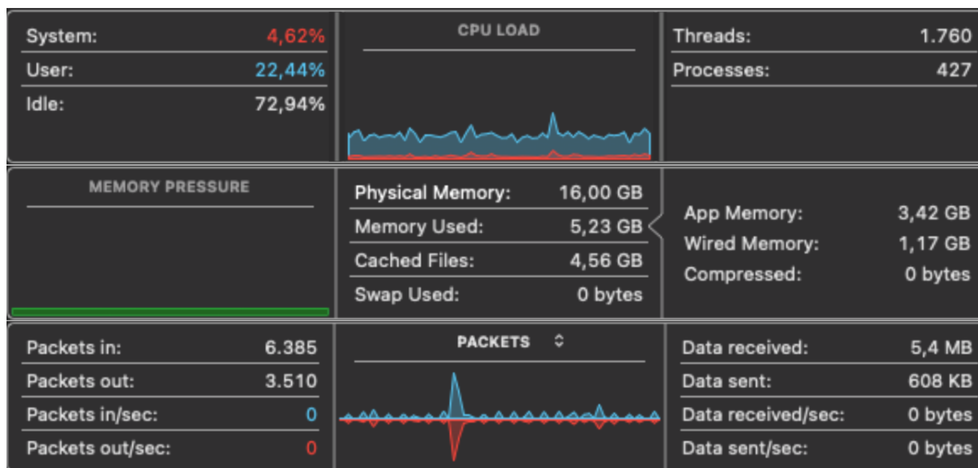

Figura 4.3: Rendimiento del ordenador durante la ejecución del script.

5
Conclusiones

En lo referente al proyecto en **Scala**, estoy satisfecho con el resultado alcanzado. Ha quedado un proyecto limpio, documentado y correctamente estructurado, de manera que es fácilmente escalable y mantenible. Los patrones y las librerías utilizadas se ajustan perfectamente a las necesidades del proyecto y ayudan a que su uso y su implementación sean sencillos e intuitivos. Sin embargo, hay varias cosas que se han quedado pendientes y que pueden perfeccionar el proyecto que explico en el apartado 5.1.

Si nos centramos a los objetivos específicos que mencionamos al comienzo podemos hacer una serie de apuntes respecto a ellos:

- **Análisis del funcionamiento de la Web API**: La API que ofrece Blizzard sobre World of Warcraft es bastante extensa y completa y eso ha ayudado muchísimo en su proceso de análisis. El hecho de que tuviese integrada en la propia web una interfaz para hacer las peticiones agilizaba el proceso y ayudaba a la hora de elegir los endpoints para los que finalmente implementaríamos los clientes de acceso. Sin embargo, el hecho de que la API no devolviese de ninguna manera el estado actual del **throttling** supuso uno de los problemas principales en el desarrollo del proyecto y también una de las principales mejoras para el mismo, que explico más adelante en 5.1.

- **Implementación del acceso a Endpoints Simples**: Este sin duda fue el objetivo más sencillo de implementar y el que menos tiempo de desarrollo consumió. Sirvió como punto principal de acercamiento a la implementación

de los clientes de acceso, las **Type Classes** y la estructura principal del proyecto. Además, ya desde un principio se podían palpar los resultados de las primeras implementaciones y era muy satisfactorio ver como extraía datos del videojuego que tanto me gusta.

- **Implementación del acceso a Endpoints Paginados**: Esta parte me sorprendió en cuanto a la complejidad que acabó suponiendo. La adaptación de los clientes de acceso a endpoints simples para que admitiesen endpoints paginados parecía trivial al principio, pero llegué a encontrarme un obstáculo que supuso con diferencia el bloqueo más largo durante el desarrollo. Esto fue porque a la hora de recuperar los datos de los endpoints **Mount** y **Character** todo funcionaba perfectamente, pero para el endpoint **Item** se rompía la estructura y devolvía error. Tarde bastante tiempo en debuggear este error porque no parecía tener un origen claro y fue bastante frustrante. Finalmente el error se producía al intentar capturar el nombre de cada **Item** en diferentes idiomas, ya que había algunos que no estaban traducidos a todos los idiomas y el parseo no lo estaba haciendo correctamente.

- **Construcción del dataset**: El último de los objetivos fue el que permitió ver el resultado de todo el trabajo anterior. Quizá donde más tiempo invertí en el desarrollo de esta última parte fue en intentar que la ejecución del programa desde la línea de comandos fuese lo más amigable posible, intentando que se pudiesen mostrar mensajes de ayuda relevantes y la versión actual, pero tuve varios problemas con la versión de los diferentes frameworks y no resultó como me habría gustado, podría ser una mejora interesante para el futuro también. Finalmente conseguí generar un dataset con datos bastante interesantes que permitirá un análisis posterior también muy interesante, pero siempre se puede enriquecer añadiendo más entradas al script de descarga y generando otros datasets complementarios.

Para terminar, la programación funcional, aunque presenta una curva de aprendizaje inicial, ofrece ventajas significativas en términos de eficiencia, seguridad y claridad del código. Su aplicación, como se ha visto en este trabajo con el framework Scala, es una muestra palpable de cómo estas ventajas pueden traducirse en soluciones robustas y escalables en el mundo real.

5.1. Líneas futuras

- **Tests unitarios**: La implementación de tests unitarios sería desde luego una mejora al proyecto en general, agilizando la depuración de errores y facilitando la implementación de posibles nuevas funcionalidades.

- **Endpoints**: Para aumentar funcionalidad de manera directa lo ideal sería aumentar la cobertura de endpoints de la API de WoW incluyendo nuevos como Logros, Mascotas, Profesiones...
- **Throttling**: Una de las mayores dificultades que me he encontrado durante el desarrollo del proyecto ha sido la implementación de **Ratelimits** personalizados, ya que la API de WoW no devuelve **Headers** con esta información. Tuve que implementar una solución que almacenase de manera local en un **.txt** las peticiones restantes basándome en la información que proporcionan desde la documentación. Al no ser una solución demasiado elegante, la mejora directa sería una correcta implementación de esta funcionalidad levantando un **Redis** o cualquier sistema de gestión de Bases de Datos que evite tener que almacenar y gestionar la información de límite de peticiones en ficheros de texto locales.

Bibliografía

[1] Scala, "Scala," 2023. [Online]. Available: https://www.scala-lang.org/

[2] Blizzard Entertainment, "World of warcraft api documentation," 2023. [Online]. Available: https://develop.battle.net/documentation/

[3] JetBrains, "Intellij idea," 2023. [Online]. Available: https://www.jetbrains.com/idea/

[4] Microsoft, "Visual studio code," 2023. [Online]. Available: https://code.visualstudio.com/

[5] Software Freedom Conservancy, "Git," 2023. [Online]. Available: https://git-scm.com/

[6] Microsoft, "Windows," 2023. [Online]. Available: https://support.microsoft.com/

[7] Apple, "Macos," 2023. [Online]. Available: https://support.apple.com/macos

[8] Project Jupyter, "Jupyter," 2023. [Online]. Available: https://jupyter.org/documentation

[9] Docker Inc., "Docker," 2023. [Online]. Available: https://docs.docker.com/

[10] Almond, "Almond," 2023. [Online]. Available: https://almond.sh/

[11] Scala, "sbt," 2023. [Online]. Available: https://www.scala-sbt.org/

[12] Lightbend, "Akka stream," 2023. [Online]. Available: https://doc.akka.io/docs/akka/current/stream/index.html

[13] ——, "Akka http," 2023. [Online]. Available: https://doc.akka.io/docs/akka-http/current/

[14] Spray, "Spray json," 2023. [Online]. Available: https://github.com/spray/spray-json

[15] Alexandre Archambault, "Case-app," 2023. [Online]. Available: https://github.com/alexarchambault/case-app

[16] Typelevel, "Cats," 2023. [Online]. Available: https://typelevel.org/cats/

[17] QOS.ch, "Logback," 2023. [Online]. Available: https://logback.qos.ch/

[18] Raúl Velasco Rubio, "wow-api," 2023. [Online]. Available: https://github.com/raulvr2/wow-api

[19] Juan Manuel Serrano, "TwitterV2," 2022. [Online]. Available: https://github.com/hablatraining/twitterv2

[20] WOWPROGRESS LLC, "Wowprogress," 2023. [Online]. Available: https://www.wowprogress.com/gearscore/eu/connected-sanguino