**Logotipo

El contenido generado por IA puede ser incorrecto.**

**Informe Un API de otro mundo**

**Asignatura:** Computación Paralela y Distribuida  
**Profesor:** Sebastián Salazar Molina  
**Fecha de entrega:** 15 de julio de 2025

**Integrantes del equipo:**

* Javier Nanco Becerra jnanco@utem.cl
* Aranza Sue Díaz adiazt@utem.cl
* Ignacio Baeza Villarroel ibaeza@utem.cl

Contenido

[1. Introducción 2](#_Toc203311954)

[2. Objetivos del proyecto 2](#_Toc203311955)

[3. Descripción general de la API 3](#_Toc203311956)

[4. Tecnologías y herramientas empleadas 4](#_Toc203311957)

[5. Requisitos para la ejecución local 5](#_Toc203311958)

[6. Configuración del entorno 5](#_Toc203311959)

[7. Ejecución del servidor 6](#_Toc203311960)

[8. Aplicación del paralelismo 7](#_Toc203311961)

[9. Documentación y pruebas 7](#_Toc203311962)

[10. Autenticación y seguridad 8](#_Toc203311963)

[11. Manejo de errores 8](#_Toc203311964)

[12. Resultados esperados 9](#_Toc203311965)

[13. Arquitectura del sistema 10](#_Toc203311966)

[13.1. Flujo general de ejecución 10](#_Toc203311967)

[13.2. Resolución de claves foráneas con caché 11](#_Toc203311968)

[13.3. Paralelización de consultas 11](#_Toc203311969)

[13.4. Capa de estadísticas 12](#_Toc203311970)

[13.5. Observaciones de diseño 12](#_Toc203311971)

[14. Diseño y estructura del código 13](#_Toc203311972)

[14.1 utils/: Conexión a base de datos 13](#_Toc203311973)

[14.2 services/: Lógica de negocio 14](#_Toc203311974)

[14.3 routers/: Definición de endpoints 14](#_Toc203311975)

[14.4 models/: Validación de datos con Pydantic 15](#_Toc203311976)

[15. Conclusión 15](#_Toc203311977)

# 1. Introducción

El presente informe describe el desarrollo de una API REST denominada **TrabajoRest**, elaborada como parte del segundo trabajo práctico del curso de Computación Paralela y Distribuida. Esta API replica la funcionalidad de una interfaz previamente publicada en línea (<https://api.sebastian.cl/isekai/swagger-ui/index.html>), incorporando técnicas de paralelismo para mejorar el rendimiento en operaciones de consulta sobre una base de datos PostgreSQL.

# 2. Objetivos del proyecto

* Replicar la funcionalidad de la API original en un entorno local controlado.
* Aplicar técnicas de paralelismo en la consulta y procesamiento de datos para aumentar la eficiencia del servicio.
* Desarrollar una solución modular, documentada y extensible, utilizando buenas prácticas de programación.
* Integrar herramientas de prueba y desarrollo para facilitar la depuración y validación de resultados.

# 3. Descripción general de la API

La API desarrollada permite realizar consultas relacionadas con un mundo ficticio (isekai), ofreciendo acceso a información estructurada en distintas entidades:

* Estratos sociales
* Especies
* Géneros
* Estadísticas de conteo
* Estadísticas de edad

Las respuestas se ofrecen en formato JSON, y los endpoints están organizados de manera intuitiva, facilitando su consumo por parte de usuarios o aplicaciones cliente.

# 4. Tecnologías y herramientas empleadas

A continuación, se listan las tecnologías y dependencias utilizadas para implementar y ejecutar la solución:

Lenguaje y frameworks

|  |  |  |
| --- | --- | --- |
| Tecnología | Versión | Descripción |
| Python | 3.11.9 | Lenguaje principal de desarrollo |
| FastAPI | 0.116.1 | Framework para construcción de APIs web |
| Uvicorn | 0.35.0 | Servidor ASGI para correr FastAPI |
| psycopg2 | 2.9.10 | Driver para conexión a PostgreSQL |
| python-dotenv | 1.1.1 | Manejo de variables de entorno |
| PostgreSQL | 16.9 | Sistema de gestión de bases de datos |

Herramientas complementarias

|  |  |
| --- | --- |
| Herramienta | Función |
| pgAdmin | Gestión visual de base de datos PostgreSQL |
| DBeaver | Cliente universal para bases de datos |
| Git | Control de versiones y colaboración |

# 5. Requisitos para la ejecución local

Para ejecutar el proyecto de manera local es necesario contar con los siguientes elementos instalados:

* Python 3.11.9
* PostgreSQL 16.9 (puede estar alojado en servidor remoto)
* Las dependencias del proyecto, instalables mediante:

Pantalla negra con letras blancas

El contenido generado por IA puede ser incorrecto.

# 6. Configuración del entorno

El proyecto emplea un archivo *.env* para definir los parámetros de conexión a la base de datos. Entre las variables necesarias se encuentran:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Es posible generar este archivo a partir de un archivo de ejemplo incluido en el repositorio.

# 7. Ejecución del servidor

Para iniciar el servidor en modo de desarrollo, se deben seguir los siguientes pasos:

1. Clonar el repositorio y acceder al directorio de la API:

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

1. Iniciar el servidor con recarga automática de código:

***Pantalla de computadora con fondo negro

El contenido generado por IA puede ser incorrecto.Nota:*** *El modo --reload está pensado únicamente para entornos de desarrollo.*

# 8. Aplicación del paralelismo

Uno de los aspectos centrales del trabajo es la aplicación de **paralelismo en las consultas a la base de datos**. Esta optimización se implementa en los endpoints que requieren múltiples consultas para obtener claves foráneas (FK) de las tablas strata, species y genders.

Mediante el uso del módulo concurrent.futures.ThreadPoolExecutor, las consultas se realizan en paralelo, reduciendo significativamente el tiempo de espera. Esta estrategia se encuentra en el archivo stats\_service.py, específicamente en la función get\_fks\_parallel.

Además, se implementa un sistema de **caché de claves foráneas** (FK\_CACHE) para evitar consultas repetidas y acelerar aún más las respuestas.

# 9. Documentación y pruebas

Durante la ejecución local, la API cuenta con una interfaz de documentación interactiva proporcionada automáticamente por FastAPI y Swagger. Esta interfaz puede accederse desde:

<http://localhost:8000/docs>

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Desde dicha página es posible realizar pruebas a todos los endpoints implementados, revisar los parámetros esperados y las posibles respuestas.

# 10. Autenticación y seguridad

Actualmente, la API **no implementa mecanismos de autenticación** ni control de acceso. Todos los endpoints están disponibles de manera pública para facilitar las pruebas durante el desarrollo. Sin embargo, se contempla la futura inclusión de medidas de seguridad para su despliegue en producción.

# 11. Manejo de errores

La API proporciona respuestas estructuradas para situaciones de error. A continuación, se ejemplifican dos casos frecuentes:

* **Error 404 (No encontrado)**

Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

* **Error 422 (Error de validación de parámetros):**

Texto

El contenido generado por IA puede ser incorrecto.

# 12. Resultados esperados

Al completar la configuración y ejecutar el servidor, el usuario podrá consultar los datos definidos en la base de datos mediante endpoints claros y eficientes, con tiempos de respuesta optimizados gracias al paralelismo y uso de caché implementados.

En condiciones ideales, con una base de datos cargada con **aproximadamente 100 millones de filas**, se espera que:

* El endpoint de estadísticas de **conteo** (/count) responda en **menos de 20 segundos**.
* El endpoint de estadísticas de **edad** (/age-stats) lo haga en **menos de 40 segundos**.

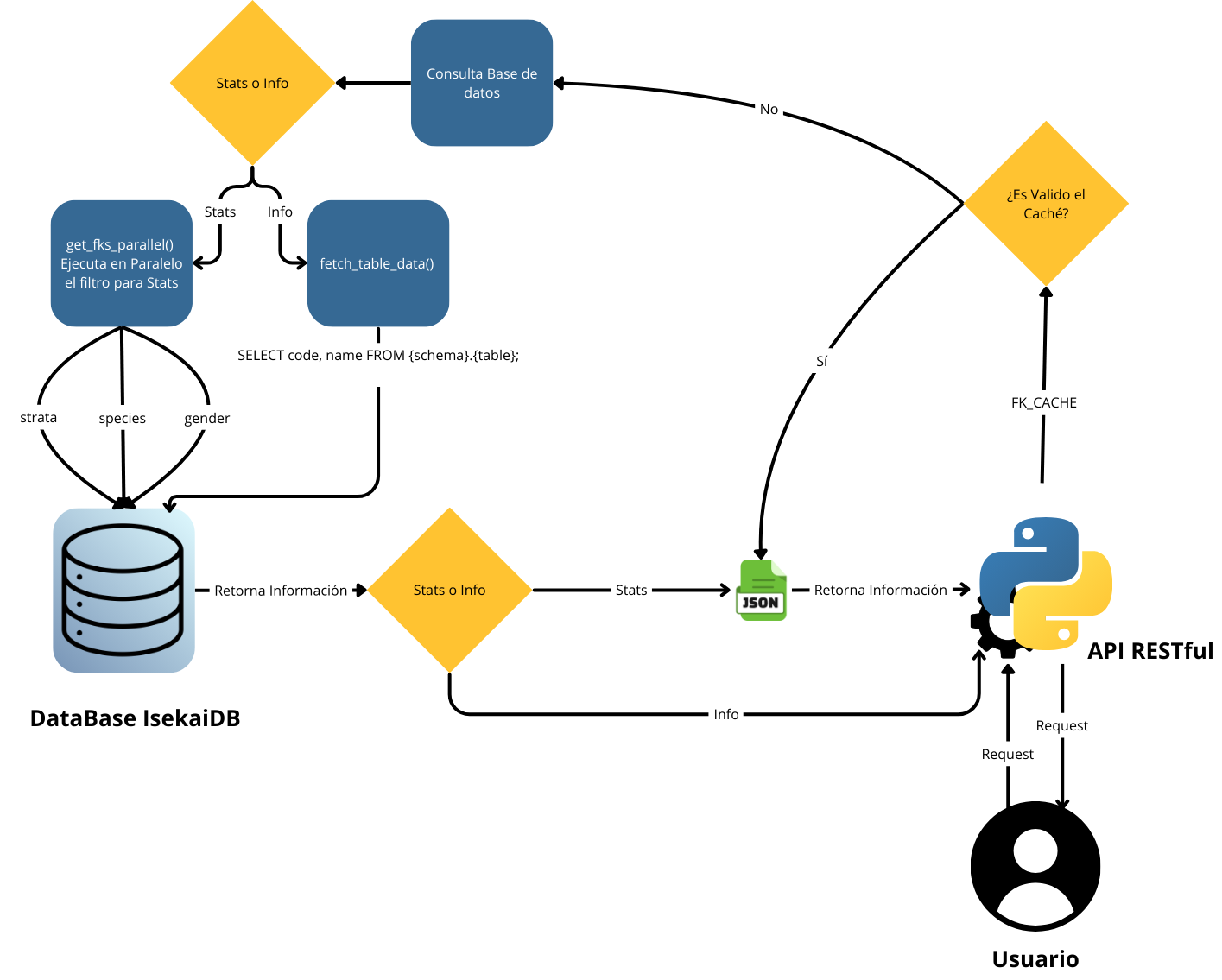
Estas cifras representan los **objetivos de rendimiento** del sistema en escenarios de alta carga.

***Importante:*** *Actualmente, la base de datos utilizada* ***no contiene la cantidad total de registros requerida (100 millones)****, por lo que* ***no es posible realizar una evaluación empírica completa*** *de dichos tiempos de respuesta. Sin embargo, la arquitectura propuesta y las técnicas aplicadas están diseñadas para escalar adecuadamente bajo dichas condiciones.*

Se espera que este enfoque facilite el consumo de datos por aplicaciones cliente, manteniendo una **arquitectura escalable, modular y eficiente**, apta para extenderse o adaptarse a escenarios de producción reales.

# 13. Arquitectura del sistema

La arquitectura de la aplicación se basa en una estructura modular que separa claramente las responsabilidades entre capas de servicios, acceso a datos y lógica de optimización. La API se desarrolla sobre el framework **FastAPI**, utilizando consultas SQL dirigidas a una base de datos **PostgreSQL**, con un enfoque orientado al rendimiento gracias a paralelización y caching.



## 13.1. Flujo general de ejecución

Cuando un usuario accede a un endpoint de estadísticas (por ejemplo, /count o /age-stats), se inicia un flujo de ejecución que traduce los códigos de entrada (como species\_code, gender\_code, strata\_code) a sus respectivas **claves primarias (FK)**. Estas claves son necesarias para filtrar correctamente la información en las consultas SQL.

## 13.2. Resolución de claves foráneas con caché

La función get\_fk\_from\_code(...) encapsula la lógica de resolución de códigos a claves foráneas. Para evitar realizar las mismas consultas múltiples veces, se implementa una **caché en memoria ()**, con una duración temporal de **300 segundos por entrada**.

Este patrón responde al modelo **cache-aside**, en el que la función:

* Primero verifica si el resultado existe y está vigente en caché.
* Si no existe o ha expirado, ejecuta una consulta a la base de datos.
* Almacena el resultado junto con un timestamp.

Este enfoque reduce significativamente el tiempo de acceso en casos de repetición de solicitudes, y permite mantener la sincronización con la base de datos sin requerir herramientas externas como Redis.

## 13.3. Paralelización de consultas

Una mejora clave se encuentra en la función get\_fks\_parallel(...), donde se aprovecha la biblioteca concurrent.futures para ejecutar las consultas de FK en **paralelo**. Esto es especialmente útil porque las tres consultas (a las tablas strata, species, y genders) son independientes entre sí.

Se utiliza un **ThreadPoolExecutor**, que lanza tres tareas simultáneamente y espera los resultados. Esto permite reducir el tiempo total de espera, aprovechando la naturaleza I/O-bound de las operaciones con bases de datos.

## 13.4. Capa de estadísticas

Una vez obtenidas las claves foráneas, se procede a realizar consultas SQL a la tabla persons, filtrando por dichas claves. Estas consultas generan:

* Conteo y proporción de coincidencias (count\_and\_percentage\_by\_codes)
* Estadísticas de edad (age\_stats\_by\_codes): mínimo, máximo, promedio, desviación estándar

Cada función está encapsulada en el módulo stats\_service.py, el cual actúa como una **capa de servicios lógica** dentro de la API, siguiendo principios de separación de responsabilidades y reutilización de código.

## 13.5. Observaciones de diseño

* El uso de caché TTL balancea eficiencia y frescura de datos.
* La implementación no requiere herramientas externas, facilitando su despliegue local y en entornos de pruebas.
* La arquitectura permite futuras mejoras, como reemplazar el caché local por una solución distribuida o aplicar estrategias similares en otras partes del sistema.

# 14. Diseño y estructura del código

La API ha sido desarrollada con una estructura modular que sigue buenas prácticas de arquitectura, separando la lógica en capas según su responsabilidad. Esta organización permite mantener un código limpio, reutilizable y fácil de extender o depurar. El proyecto se divide principalmente en las siguientes carpetas:

## 14.1 utils/: Conexión a base de datos

Contiene utilidades de bajo nivel, en particular el módulo db\_utils.py, que centraliza la gestión de la conexión a la base de datos PostgreSQL.

* Las credenciales se cargan desde un archivo .env usando python-dotenv.
* La función get\_connection() reutiliza una conexión global (conexion), evitando reconexiones innecesarias.
* Se define la variable PERSONS\_TABLE, configurable mediante entorno, para adaptar la tabla principal sin modificar el código.

Este diseño desacopla la configuración de la base de datos del resto del sistema.

## 14.2 services/: Lógica de negocio

Esta carpeta implementa la funcionalidad central que interactúa con la base de datos. Se encuentra dividida por dominio:

* **info\_service.py**

Proporciona funciones para consultar tablas base como strata, species y genders, devolviendo listas de objetos InfoItem. Toda la lógica se abstrae en fetch\_table\_data(), que acepta dinámicamente el nombre de la tabla.

* **stats\_service.py**

Contiene la lógica estadística más avanzada, incluyendo:

* get\_fk\_from\_code(...): Traduce códigos a claves foráneas con un sistema de caché con TTL (300 segundos).
* get\_fks\_parallel(...): Utiliza ThreadPoolExecutor para resolver múltiples claves en paralelo.
* count\_and\_percentage\_by\_codes(...) y age\_stats\_by\_codes(...): Ejecutan consultas SQL optimizadas para obtener estadísticas agregadas.

Estas funciones están diseñadas para ser reutilizadas desde los endpoints expuestos en los routers.

## 14.3 routers/: Definición de endpoints

Los routers representan la capa de presentación de la API, encargados de exponer las rutas que consumen los servicios.

* **info.py**

Define rutas como /strata, /species y /genders, que retornan datos base para poblar interfaces o formularios. Se apoyan en el servicio info\_service.

* **stats.py**

Expone rutas como /count y /age-stats, que requieren parámetros y retornan estadísticas filtradas. Se apoyan en stats\_service y utilizan modelos de entrada como CountFilterInput.

La separación por archivo mejora la claridad y el mantenimiento de las rutas según su propósito.

## 14.4 models/: Validación de datos con Pydantic

La carpeta models contiene esquemas Pydantic definidos en schemas.py, utilizados para validar tanto las entradas como las respuestas.

Incluye:

* InfoItem: Representa elementos base como especie, género o estrato, con atributos code y name.
* ProblemDetails: Modelo estandarizado de errores, inspirado en el RFC 7807.
* CountFilterInput: Esquema de entrada para filtros usados en endpoints estadísticos (strata\_code, species\_code, gender\_code).

Esto asegura consistencia en los datos procesados y mejora la generación automática de documentación vía Swagger.

# 15. Conclusión

El desarrollo de la Aplicación “**Un API de otro mundo**” ha permitido no solo replicar la funcionalidad esperada, sino también aplicar conceptos de programación paralela para mejorar su rendimiento. Este enfoque demuestra cómo es posible escalar servicios web simples mediante técnicas de optimización concurrente, y representa un ejercicio práctico y valioso en el contexto de la asignatura.