

Arquitectura de la app

Idea de la app

(PRINCIPAL)

Algo de cartas sobre pájaros(tipo magic o algo parecido)

-funcionalidad marketplace

- funcionalidad social comunidades, chat privados, equipos... (Eventos importante)

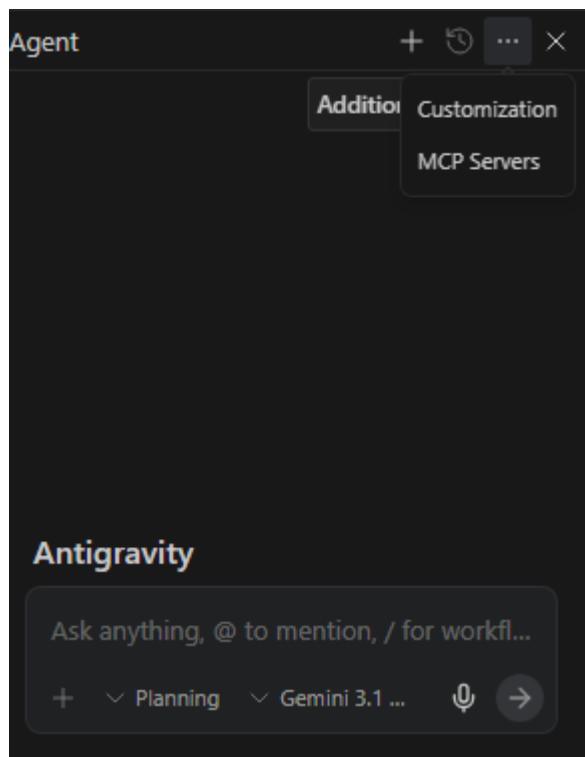
- funcionalidad de encontrar pájaros tipo pokemon go y de crear nuevos pájaros que no tengas, jugar entre cada uno con batallas y que tambien adquieras recompensas

- datos detallados de cada pájaro con un sistema mmo balanceado

- premiar la socialización entre usuarios y hacer una buena experiencia sobre los pájaros

Cómo lo vamos a estar trabajando

Usaremos el editor Antigravity. En este en los 3 puntos de la zona del chat (arriba) le damos a habilitar los MCPs.

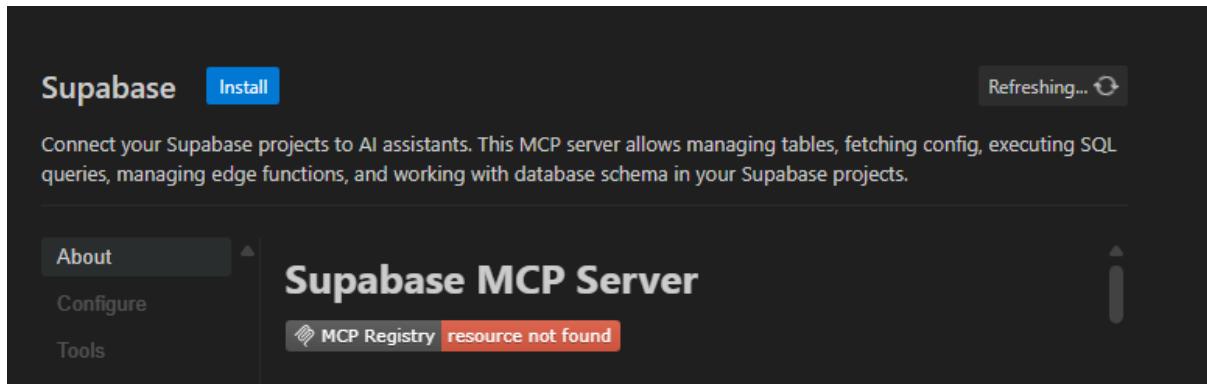


Un Mcp es una forma de darle herramientas externas y configuradas a la IA de internet (hecho por ingenieros de IA) pero en local, de manera que le permite a la IA ser más eficiente a la hora de trabajar (menos tokens). También vamos a usar unas

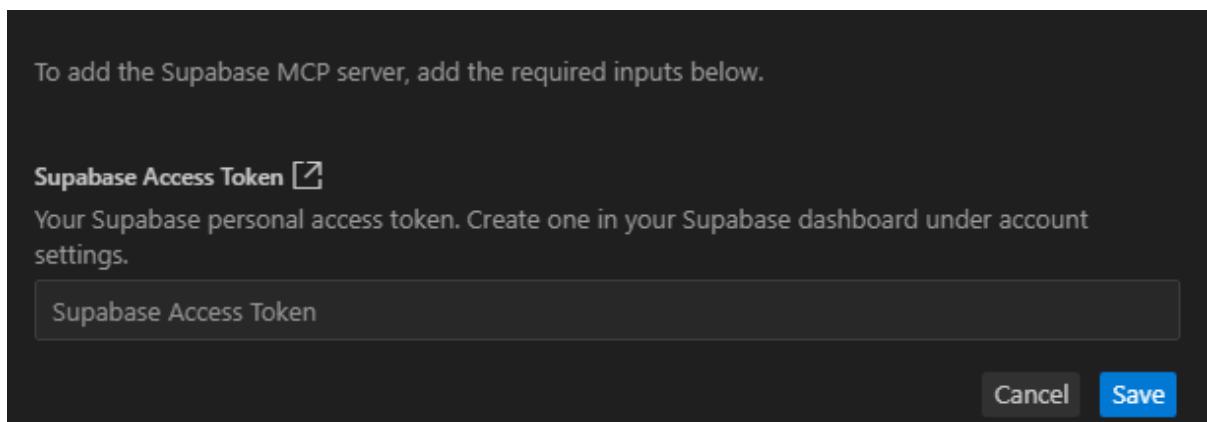
customizaciones, que es como una configuración dedicada para el proyecto con un contexto con el que va a estar trabajando el agente puesto. Previamente necesitaremos tener instalado npx <https://nodejs.org/en/download/>.

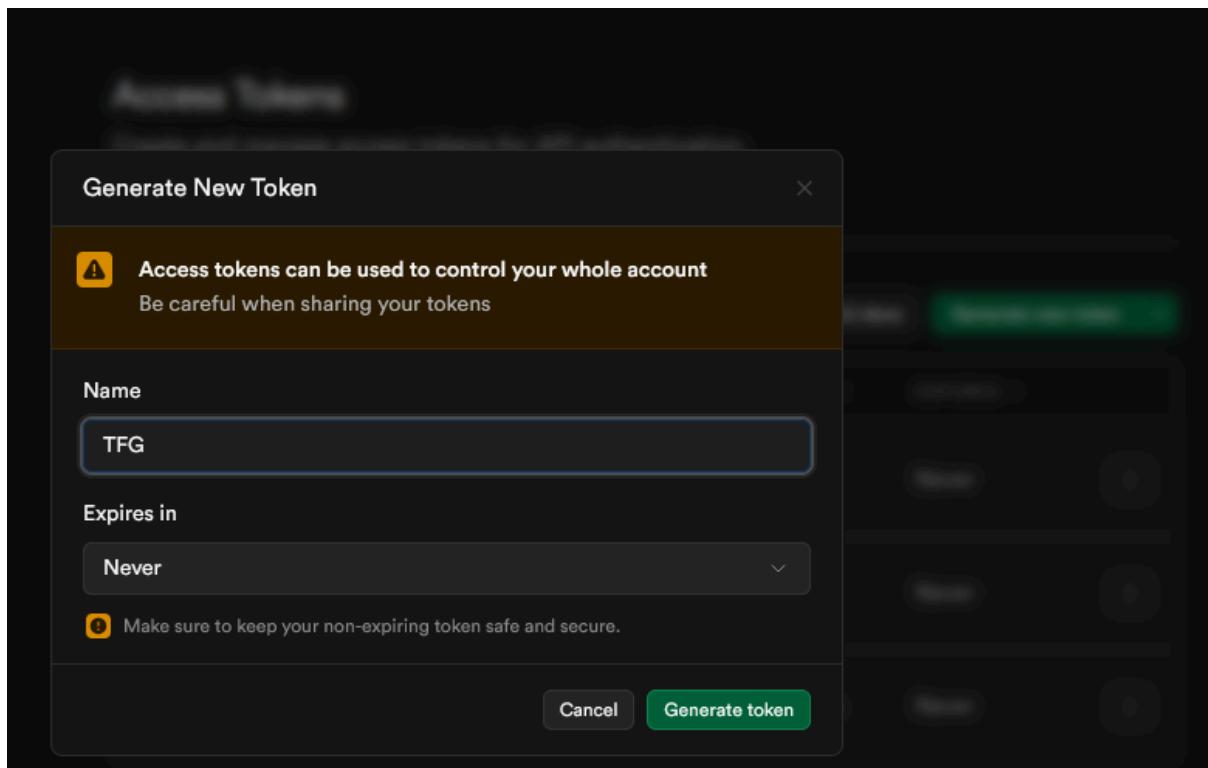
MCPs:

- Supabase
- Github



Le damos a instalar; nos pide en algunas el token de supabase, donde se encuentra en el perfil de nuestro usuario (suele proporcionarse mediante un link que te sale).





Customizaciones: están en el drive. Se pone dándole a “Global” y metiendo ahí el rol de “cerebro.txt”.

The screenshot shows the Agent application interface. At the top, there's a "Customizations" section with a sub-section for "Rules". It includes a "Rules" tab, a "+ Global" button, and a "+ Workspace" button. Below this, a note says "Rules help guide the behavior of Agent." In the main workspace, there's a file named "GEMINI.md" and a connection to "MCP: Supabase". A text editor window is open, showing the beginning of a "Global Antigravity Rules File" with the text "Write your global rules for Agent here".

Frontend

- funcionalidad simple para que cualquier persona entienda bien el funcionamiento de la aplicación da igual la edad y el nivel técnico

Backend

APIS RECOMENDADAS POR EL PROFE

APIs útiles para el proyecto de DAM

Unsplash API y Pexels API son opciones populares para descargar fotos gratuitas de alta calidad mediante solicitudes programáticas.

Unsplash API

Proporciona acceso a millones de imágenes gratuitas para uso comercial sin atribución obligatoria. Regístrate en unsplash.com/developers para obtener una clave API gratuita, con límites iniciales de 50 solicitudes por hora (ampliables). Usa endpoints como /photos/random para fotos aleatorias o /search/photos para búsquedas específicas.

Pexels API

Ofrece fotos y videos gratuitos con licencia CC0, ideal para proyectos web. Crea una cuenta en pexels.com/api para tu clave API gratuita (200 solicitudes por hora, 20,000 mensuales). Endpoints clave incluyen /v1/photos/popular y /v1/search para curación y búsquedas.

Otras alternativas simples

LoREM Picsum: API sin registro para imágenes placeholder aleatorias (ej.

<https://picsum.photos/800/600>), perfecta para pruebas rápidas sin límites estrictos.

Pixabay API: Miles de fotos gratuitas con clave gratuita, soporta búsquedas por categoría y alta resolución.

Nuthatch API — Es una API gratuita orientada a aves; su catálogo incluye fotos (compiladas con imágenes de fuentes libres como Unsplash + contribuciones privadas).

Vecteezy API — Permite acceder a una biblioteca enorme de fotos, vectores y recursos gráficos; útil si no necesitas fotos específicamente ornitológicas, pero sí imágenes "libres" de aves o naturaleza.

Alojamiento de Datos

Supabase con JSON

SQLite en local

Manejo de Datos

Se centrará en usar Java + Spring para el manejo de los datos y la información.

Aquí tienes el diagrama conceptual de lo que vamos a construir, seguido de la tabla definitiva que resume tu **Stack Reactivo y Asíncrono**.

El Stack 100% Asíncrono (WebFlux / Reactor)

Módulo / Tecnología	¿Por qué lo usamos? (La Ventaja)	¿Cómo se implementa? (La Práctica)
---------------------	----------------------------------	------------------------------------

Spring WebFlux <i>(Servidor Netty)</i>	Es el motor central no bloqueante. Permite que un solo servidor maneje miles de partidas a la vez sin quedarse sin memoria RAM.	Se usa en lugar de Spring MVC (Tomcat). En tu código Java, en vez de devolver Carta , devuelves Mono<Carta> o Flux<Carta> .
RSocket <i>(Comunicación)</i>	Protocolo bidireccional más rápido que WebSockets. Soporta <i>Backpressure</i> : si el móvil va lento, ajusta el envío de datos para no colgar la app React.	Se habilita con spring-boot-starter-rsocket . Usas la anotación @MessageMapping para escuchar los movimientos (ej: "Atacar") del cliente.
Spring Data R2DBC <i>(Conexión DB)</i>	Es el driver asíncrono para PostgreSQL (Supabase). Evita que el servidor se quede "congelado" esperando a que la base de datos lea el inventario.	Reemplaza a Hibernate/JPA. Creas repositorios reactivos (ReactiveCrudRepository) que devuelven Mono / Flux al interactuar con las tablas.
Jackson + R2DBC Converters <i>(Mapeo JSON)</i>	R2DBC es "crudo" y no entiende el tipo JSONB de Postgres mágicamente. Jackson traduce ese JSON a objetos Java al vuelo.	Creas una clase que implemente Converter<Json, MiCartaRecord> y usas ObjectMapper de Jackson para deserializar el string JSONB de Supabase.
Spring Data Redis Reactive <i>(Caché Marketplace)</i>	Para el Marketplace, consultar Supabase por cada búsqueda saturaría la red. Redis mantiene el catálogo en RAM para respuestas en submilisegundos.	Usas ReactiveRedisTemplate para guardar y leer las ofertas activas sin bloquear el hilo principal.
Redisson <i>(Distributed Locks)</i>	Evita el "doble gasto" o clonación de cartas en el Marketplace. Bloquea una carta a nivel de red entera si dos personas la compran a la vez.	Cuando un usuario le da a "Comprar", pides un lock (RLockReactive) a Redisson por el ID de la carta. Nadie más puede tocarla hasta que se libere.
Spring Security Reactive <i>(Auth)</i>	Valida que el usuario sea quien dice ser sin frenar el flujo de datos. Se integra directamente con el sistema Auth de Supabase.	Configuras un ReactiveJwtDecoder que verifica la firma del token enviado desde la WebView y guarda el usuario en el ReactiveSecurityContext .
Spring Kafka o RabbitMQ <i>(Event Broker)</i>	Saca el trabajo pesado fuera de la partida. (Ej: Repartir experiencia, guardar logs, dar recompensas al acabar un duelo).	Al acabar la partida, emites un evento PartidaTerminada . El hilo del jugador queda libre al instante, y otro microservicio procesa las recompensas.

Las Alternativas (El "Plan B")

Si durante el desarrollo veis que el modelo reactivo ([Mono](#) / [Flux](#)) hace que el código sea muy difícil de leer o testear, estas son las alternativas que sacrifican un poco de rendimiento puro por **facilidad de desarrollo**:

- **Alternativa a WebFlux (Motor): *Spring Web MVC + Virtual Threads (Java 21+)*.**

- *Por qué:* Te permite escribir código tradicional paso a paso (síncrono y fácil de leer) pero con casi el mismo rendimiento masivo que WebFlux gracias a los hilos virtuales de Java.
- **Alternativa a RSocket (Comunicación): *WebSockets Clásicos (STOMP)*.**
 - *Por qué:* Es mucho más fácil de integrar con React y la WebView. Hay decenas de librerías listas para usar (como `sockjs-client`), mientras que RSocket requiere un poco más de configuración en el frontend.
- **Alternativa a R2DBC (Base de datos): *Spring Data JPA (Hibernate) + Hypersistence Utils*.**
 - *Por qué:* R2DBC requiere que hagas tú mismo el mapeo de relaciones y JSONB. JPA/Hibernate te hace toda la "magia" de convertir las tablas y el JSON de Supabase en objetos Java con simples anotaciones, a costa de usar operaciones bloqueantes.
- **Alternativa a Kafka/Redis (Mercado simple): *Optimistic Locking en PostgreSQL*.**
 - *Por qué:* Si no quieres pagar/mantener servidores extra de Redis o Kafka, puedes usar un simple campo `@Version` en Supabase. Si hay un conflicto de compra, la base de datos lanza una excepción, se cancela todo, y le muestras un error de "Carta ya vendida" al usuario.

BORRADOR::

Este es un planteamiento de **Matrícula de Honor**. La combinación de un stack tecnológico puntero (Reactive Spring) con una gamificación compleja (Geolocalización + TCG) es extremadamente ambiciosa.

Para un TFG de DAM, el tribunal valora dos cosas: **la complejidad técnica** (que ya la tienes de sobra con WebFlux/RSocket) y **la viabilidad**.

Aquí tienes el **Guion Técnico y Funcional Definitivo** estructurado para tu TFG. Úsalo como índice de tu memoria y guía de desarrollo.

Título del Proyecto: AVIS: Chronicles of Flight

(Nombre provisional: Avis TCG)

1. Resumen Ejecutivo (El "Pitch")

Aplicación móvil híbrida de colección y estrategia en tiempo real. Combina la **exploración física** (Geolocalización tipo Pokémon GO para avistamiento y recolección de recursos) con **batallas estratégicas de cartas** (tipo Magic/Hearthstone) basadas en ornitología real. El sistema incluye un **Marketplace en tiempo real** y funciones sociales avanzadas, soportado por una arquitectura backend **Reactiva y No Bloqueante**.

2. Stack Tecnológico (Definido por tus requisitos)

Para el TFG, justificaremos este stack como la solución a la "Alta Concurrencia y Baja Latencia".

Backend (Java Ecosystem)

- **Framework Core:** Spring Boot 3 (WebFlux).
- **Motor Asíncrono:** Project Reactor (Mono/Flux).
- **Comunicación Cliente-Servidor:** **RSocket** (para el juego en tiempo real) + REST (para configuración inicial).
- **Persistencia:** Spring Data R2DBC (Reactiva).
- **Base de Datos:** **Supabase (PostgreSQL)**. Uso intensivo de tipos **JSONB** para las cartas y **PostGIS** (si Supabase lo permite, o coordenadas simples) para la geolocalización.
- **Caché & Locks:** **Redis Reactive** (con Redisson para bloqueos distribuidos en el Marketplace).
- **Event Broker:** RabbitMQ o Kafka (para desacoplar la lógica de recompensas post-partida).
- **Seguridad:** Spring Security Reactive + JWT (validando contra Supabase Auth).

Frontend (Cliente Móvil)

- **Tecnología:** React Native o Flutter (Recomendado Flutter si buscas rendimiento nativo en mapas, pero React Native se integra fácil con RSocket-js).
- **Base de Datos Local:** **SQLite** (para guardar mazos y caché de imágenes offline).

APIs Externas (Gestión de Recursos)

- **Nuthatch API:** Fuente de verdad para datos taxonómicos (nombre científico, familia).
 - **Unsplash / Pexels API:** Fondos de cartas (hàbitats) y eventos.
 - **Vecteezy:** Iconografía vectorial para la UI (plumas, nidos, ataque).
-

3. Arquitectura del Sistema: El "Reactive Flow"

Este diagrama es vital para tu memoria. El flujo no es lineal, es un flujo de streams.

1. **La Capa de Entrada (RSocket/Netty):**
 - Mantiene una conexión persistente con el móvil.
 - Ejemplo: El usuario entra a una "Batalla". Se abre un canal bidireccional.
 2. **El "Game Loop" (Reactor):**
 - No hay `Thread.sleep()`. Todo son eventos.
 - *Input*: Jugador tira carta -> *Process*: Validar maná -> *Output*: Actualizar estado del tablero para ambos jugadores.
 3. **Gestión de Datos (R2DBC + JSONB):**
 - Las cartas no son filas rígidas de SQL. Son objetos JSON dentro de Postgres. Esto permite que una carta tenga atributos únicos (ej: "Vuelo", "Migración") sin alterar la tabla principal.
 4. **Marketplace (Redis + Redisson):**
 - Cuando alguien compra una carta, Redisson aplica un "Lock" distribuido. Si dos usuarios pulsan comprar al milisegundo, solo uno pasa, el otro recibe error, garantizando la integridad transaccional.
-

4. Módulos Funcionales (El Core del Juego)

A. Módulo "Exploración" (Tipo Pokémon GO)

- **Mecánica:** El usuario camina por el mundo real.
- **El mapa:** Muestra "Nidos" o "Avistamientos".
- **La acción:** Al llegar a un punto, no "capturas" al pájaro directamente. Haces "Avistamiento".
 - Usas la cámara (AR opcional) o una interfaz simple.
 - **Reward:** Obtiienes "Plumas" (moneda de crafteo) o "Esencia de [Especie]" (necesaria para crear la carta).
 - *Integración API:* Aquí llamas a **Nuthatch API** para mostrar datos reales curiosos del pájaro avistado.

B. Módulo "Laboratorio" (Crafteo de Cartas)

- **Concepto:** No compras sobres aleatorios (loot boxes), creas las cartas con lo recolectado.
- **Lógica:** 10 Plumas + 1 Esencia de Gorrión = Carta "Gorrión Común".
- **Personalización:** El usuario puede subir una foto (si la tomó él) o usar la generada por la API para su carta.

C. Módulo "Arena" (Batallas TCG - Tipo Magic)

- **Estructura de la Carta (JSON):**
 - *Coste*: Comida (Gusanos, Semillas).
 - *Stats*: Envergadura (Vida), Pico (Ataque).
 - *Habilidad*: "Canto Territorial" (Buff a aliados), "Vuelo Picado" (Daño rápido).
- **Flujo de Batalla (RSocket):**

- Turnos dinámicos.
- El servidor valida reglas: "¿Tiene suficiente comida para jugar el Águila Real?".

D. Módulo "Marketplace" (Trading)

- **Compra/Venta:** Usuarios venden cartas repetidas o crafteadas.
- **Subastas:** Sistema de pujas en tiempo real (aquí WebFlux brilla).

E. Módulo Social

- **Bandadas (Clanes):** Grupos de usuarios.
 - **Chat:** Privado y de Clan (usando el mismo socket de RSocket).
-

5. Modelo de Datos (Híbrido SQL/NoSQL)

En Supabase (PostgreSQL), usaremos un esquema inteligente:

Tabla: `users` (Sincronizada con Auth)

- `id, username, geo_home` (lat/long base), `wallet_balance`.

Tabla: `card_definitions` (El "Pokedex")

- `id`
- `scientific_name` (de Nuthatch API)
- `rarity`
- `base_stats` (JSONB): `{"attack": 5, "defense": 3, "diet": "insectivore"}`
- `image_url` (Link a Pexels/Unsplash cacheado)

Tabla: `inventory_cards` (Las cartas reales de los usuarios)

- `id` (UUID único de ESA carta)
- `user_id`
- `definition_id`
- `level` (Sistema MMO, la carta sube de nivel al usarse)
- `custom_name` (Nombre que le pone el usuario)

Tabla: `market_listings`

- `id, seller_id, card_uuid, price, status` (ACTIVE, SOLD, LOCKED).
-

6. Integración de Imágenes (El reto de las APIs)

Dado que las APIs tienen límites, no podemos llamar a Unsplash cada vez que se carga una carta.

Estrategia "Lazy Loading & Caching":

1. **Primer avistamiento:** El servidor busca en Nuthatch/Unsplash "Sparrow".
 2. **Descarga y Optimización:** El servidor descarga la imagen, la redimensiona (para móvil) y la sube al **Storage de Supabase**.
 3. **Persistencia:** Se guarda la URL de Supabase en la tabla `card_definitions`.
 4. **Futuro:** Las siguientes veces, se sirve desde Supabase (rápido y sin gastar cuota de API externa).
-

7. Plan de Contingencia (El "Plan B")

Si WebFlux se vuelve un infierno (la curva de aprendizaje es muy alta), el plan de repliegue debe estar documentado para el TFG:

1. **De RSocket a WebSocket (STOMP):** Más documentación disponible y más fácil de depurar.
2. **De R2DBC a JPA + Hibernate:** Si las consultas SQL complejas fallan en reactivo, volver a JPA bloqueante pero usar **Virtual Threads (Java 21)**. Esto permite mantener la "alta concurrencia" sin la complejidad del código reactivo (`.flatMap`, `.subscribe`). **Menciona esto a tu profesor, le encantará que conozcas los Virtual Threads.**