

Documentación Técnica: Sistema Estudiantil de Alto Rendimiento con Redis

Introducción

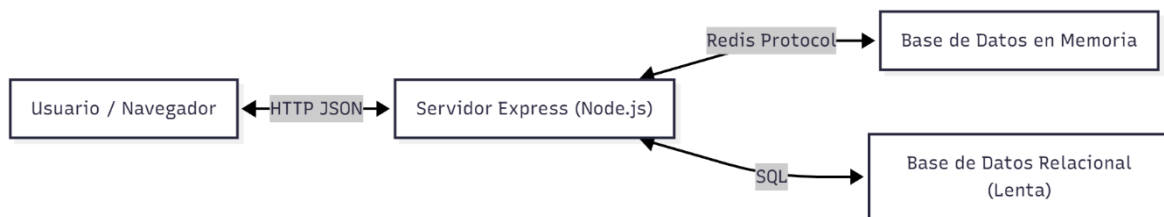
A continuación, se detalla la estructura modular del código base, explicando la interacción entre las distintas capas de la aplicación. Se profundizará en el diseño interno de los servicios y en los patrones de diseño aplicados para integrar Redis de manera eficiente. Este análisis técnico busca demostrar cómo dicha integración es crítica para escalar el sistema y mantener un rendimiento óptimo bajo carga.

Además de describir la jerarquía de directorios y la separación de responsabilidades en el código, este documento justifica la selección de estructuras de datos específicas de Redis (como Sorted Set o Hashes) para casos de uso concretos. Se examinará cómo el manejo eficiente del pool de conexiones y las estrategias de caché reducen la latencia de la base de datos principal, asegurando una experiencia de usuario fluida.

1. Visión General del proyecto:

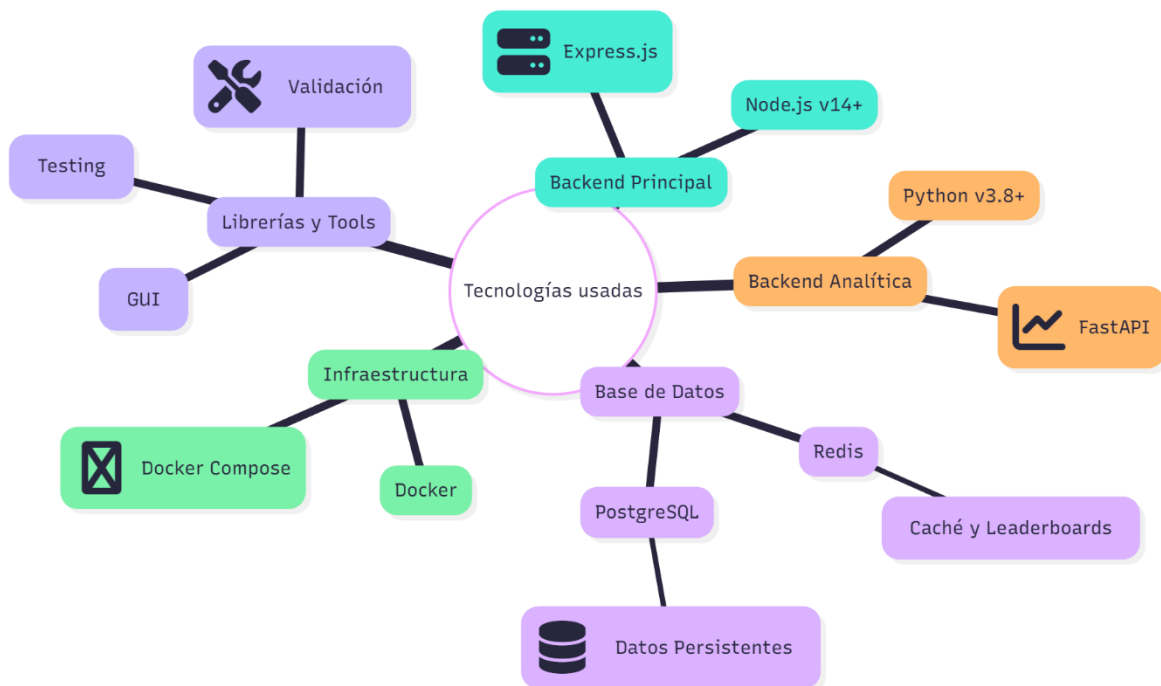
El sistema se fundamenta en una arquitectura de alto desempeño diseñada para manejar miles de operaciones simultáneas con tiempos de respuesta sub-milisegundo. Este objetivo se logra posicionando a Redis como el componente central de la infraestructura: no se limita a almacenar copias temporales (caché), sino que funciona como la base de datos principal para las estructuras críticas del negocio, asegurando que los datos más importantes estén siempre disponibles en la memoria RAM para un acceso inmediato.

1.1 Diagrama de Arquitectura Lógica:



Tecnologías usadas

A continuación, se detalla las tecnologías empleadas en el desarrollo y despliegue del sistema:



2. Análisis Profundo del Modelo de Datos (Redis Patterns)

A diferencia del modelo relacional, donde se prioriza la reducción de redundancia (normalización), el modelado en Redis es práctico y se centra en la velocidad de recuperación. Las estructuras de datos se diseñan como 'vistas materializadas' de lo que la aplicación necesita consumir. Aceptamos e incluso fomentamos la desnormalización para que los datos estén pre-organizados en el formato exacto de lectura, evitando así la necesidad de realizar cruces (joins) o transformaciones complejas en tiempo de ejecución.

2.1 Gestión de Sesiones: SessionService

- i) Patrón: Hash Object Storage con Rolling Expiry.
- ii) Estructura Redis: `HASH`
- iii) Clave: `session:{token}` (donde `{token}` es un UUID v4).
- iv) Justificación Técnica:

Un `HASH` en Redis es eficiente en memoria; codifica campos pequeños como un "ziplist" compacto.

Permite leer/escribir campos individuales (`HGET`, `HSET`) sin serializar todo el objeto (a diferencia de almacenar un JSON en lectura (Middleware))

```
const session = await redis.hgetall(`session:${token}`);
```

```
if (session) await redis.expire(`session:${token}`, 1800); // Renovación de TTL, ahorrando CPU en el servidor Node.js.
```

- v) Complejidad Ciclomática: $O(1)$ para crear, leer y validar.
- vi) Implementación en Código:

```
javascript
// src/services/sessionService.js
// Escritura (Login)
await redis.hset(`session:${token}`, userObj);
await redis.expire(`session:${token}`, 1800); // TTL 30 min
```

3. Instrucciones de uso e instalación:

Requisitos Previos:

- a) Node.js: v14 o superior.
- b) Redis: Debe tener un servidor Redis ejecutándose en localhost: 6379.

Instalación:

1. Clonar el repositorio o descargar el código.
2. Instalar dependencias: **`npm instal`**.

Configuración:

El proyecto espera que Redis esté ejecutándose en el puerto por defecto. Si necesitas cambiar la configuración, puedes crear un archivo .env en la raíz (aunque no es estrictamente necesario para la prueba local por defecto): **`REDIS_URL=redis://localhost:6379`**.

Ejecución:

El proyecto consta de dos scripts principales para probar la funcionalidad:

1. Poblar la Base de Datos (seed)
Este script genera 1000 estudiantes aleatorios, sus puntuaciones y algunas sesiones activas. Ejecutar esto primero: **`node src/scripts/seed.js`**.
Esto limpiará la base de datos actual (FLUSHDB) y generará nuevos datos.
2. Iniciar el Servidor (start)
Esto levantará el servidor Express en <http://localhost:3000>: **`npm start`**.
3. Usar el Dashboard
Abre tu navegador y visita <http://localhost:3000>. Desde ahí podrás visualizar la demostración interactiva:
 1. Gestión de Sesiones: Login/Logout con tokens.
 2. Leaderboard en Vivo: Tabla de posiciones que se actualiza al instante.
 3. Benchmark Comparativo: ¿SQL vs Redis? Ejecuta una carrera en tiempo real y ve la diferencia gráfica.
 4. Prueba de Estrés: Lanza 1,000 usuarios concurrentes (3,000 operaciones) y mide el throughput real de tu máquina.
 - Tip: Abre la consola del navegador (F12) para ver los logs detallados de cada operación.

(Opcional) Ejecutar Pruebas de Consola (queries)

Si prefieres ver los logs en la terminal: **`node src/scripts/queries.js`**.

Diseño de la base de datos

```
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

```
CREATE TABLE students (  
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),  
    name VARCHAR(100) NOT NULL,  
    email VARCHAR(150) UNIQUE NOT NULL,  
    career VARCHAR(100) NOT NULL,  
    semester INTEGER CHECK (semester > 0 AND semester <= 14),  
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,  
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP  
);
```

```
CREATE TABLE scores (  
    id SERIAL PRIMARY KEY,  
    student_id UUID REFERENCES students(id) ON DELETE CASCADE,  
    score DECIMAL(5, 2) NOT NULL CHECK (score >= 0 AND score <= 100),  
    recorded_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP  
);
```

```
CREATE INDEX idx_students_email ON students(email);  
CREATE INDEX idx_scores_student_id ON scores(student_id);  
CREATE INDEX idx_scores_value ON scores(score DESC);
```

Diccionario de datos:

Tabla: Students

Columna	Tipo de Dato	Restricciones	Descripción
id	UUID	PK, Default uuid_generate_v4()	Identificador único universal del estudiante.
id	VARCHAR(100)	NOT NULL	Nombre completo del estudiante.
email	VARCHAR(150)	UNIQUE, NOT NULL	Correo electrónico institucional.
career	VARCHAR(100)	NOT NULL	Carrera que cursa el estudiante.
semester	INTEGER	CHECK (>0 AND <=14)	Semestre actual (1-14).
created_at	TIMESTAMP	Default 'CURRENT_TIMESTAMP'	Fecha de registro.
updated_at	TIMESTAMP	Default 'CURRENT_TIMESTAMP'	Fecha de última actualización.

Tabla: Scores

Columna	Tipo de Dato	Restricciones	Descripción
id	SERIAL	PK	Identificador autoincremental del registro de nota.
student_id	UUID	FK -> 'students(id)'	Referencia al estudiante. Borrado en cascada.
score	DECIMAL(5, 2)	NOT NULL, CHECK (0-100)	Calificación numérica (ej. 95.50).
recorded_at	TIMESTAMP	Default 'CURRENT_TIMESTAMP'	Fecha y hora en que se registró la nota.

Consultas del Caso de Uso (DML)

A continuación, se presentan 15 consultas SQL representativas para la gestión del portal de estudiantes.

1. Registrar un nuevo estudiante

```
INSERT INTO students (name, email, career, semester)
VALUES ('Juan Perez', 'juan.perez@example.com', 'Ingeniería de Sistemas', 3);
```

2. Obtener perfil de estudiante por ID

```
SELECT * FROM students WHERE id = 'uuid-del-estudiante';
```

3. Buscar estudiante por correo electrónico

```
SELECT * FROM students WHERE email = 'juan.perez@example.com';
```

4. Actualizar semestre del estudiante

```
UPDATE students SET semester = 4, updated_at = CURRENT_TIMESTAMP
WHERE email = 'juan.perez@example.com';
```

5. Eliminar un estudiante (y sus notas en cascada)

```
DELETE FROM students WHERE id = 'uuid-del-estudiante';
```

6. Buscar estudiantes por nombre (búsqueda parcial)

```
SELECT * FROM students WHERE name ILIKE '%Juan%';
```

Gestión de Notas y Leaderboard

7. Registrar una nueva nota

```
INSERT INTO scores (student_id, score) VALUES ('uuid-del-estudiante', 95.5);
```

8. Obtener el historial de notas de un estudiante

```
SELECT score, recorded_at FROM scores
WHERE student_id = 'uuid-del-estudiante'
ORDER BY recorded_at DESC;
```

9. Actualizar una nota específica (corrección)

```
UPDATE scores SET score = 98.0 WHERE id = 123;
```

10. Obtener el Top 10 de mejores notas globales

```
SELECT s.name, sc.score, s.career
FROM scores sc
JOIN students s ON sc.student_id = s.id
ORDER BY sc.score DESC
LIMIT 10;
```

Analítica y Reportes

11. Promedio de notas por carrera

```
SELECT s.career, AVG(sc.score) as average_score
FROM students s
JOIN scores sc ON s.id = sc.student_id
GROUP BY s.career
ORDER BY average_score DESC;
```

12. Contar estudiantes por semestre

```
SELECT semester, COUNT(*) as total_students
FROM students
GROUP BY semester
ORDER BY semester ASC;
```

13. Obtener estudiantes con notas superiores a 90

```
SELECT DISTINCT s.name, s.email
FROM students s
JOIN scores sc ON s.id = sc.student_id
WHERE sc.score > 90;
```

14. Encontrar estudiantes inactivos (sin notas registradas)

```
SELECT * FROM students s
WHERE NOT EXISTS (SELECT 1 FROM scores sc WHERE sc.student_id = s.id);
```

15. Obtener la nota más reciente de cada estudiante

```
SELECT s.name, sc.score, sc.recorded_at
FROM students s
JOIN scores sc ON s.id = sc.student_id
WHERE sc.recorded_at = (
    SELECT MAX(recorded_at) FROM scores WHERE student_id = s.id
);
```

4. Justificación del Modelo de Datos NoSQL (Redis)

La elección de Redis como motor principal para las estructuras de datos "calientes" (Hot Data) se fundamenta en necesidades específicas de rendimiento que una base de datos relacional tradicional (RDBMS) no puede satisfacer eficientemente bajo alta carga.

A. Baja Latencia (< 1ms)

El sistema requiere respuestas en tiempo real para sesiones y rankings. Redis, al operar completamente en memoria RAM, elimina la latencia de I/O de disco, permitiendo tiempos de lectura/escritura en el orden de microsegundos.

B. Estructuras de Datos Especializadas

A diferencia de un RDBMS que solo ofrece tablas, Redis ofrece estructuras optimizadas algorítmicamente para nuestros casos de uso:

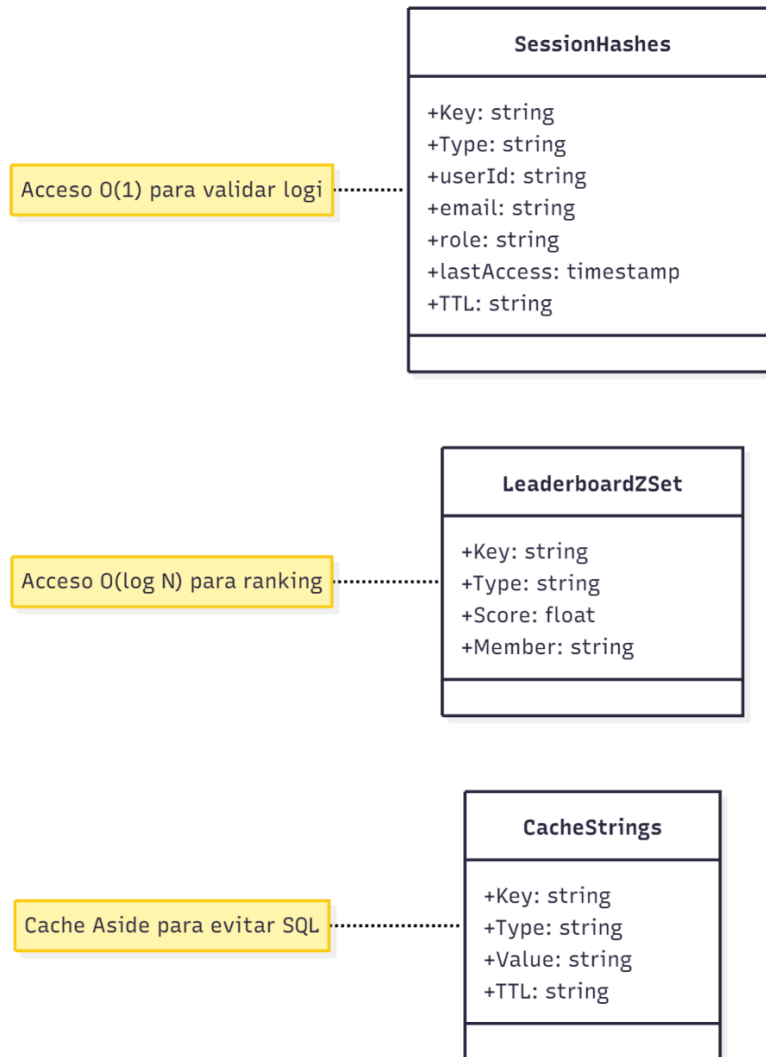
Sorted Sets (ZSETS) para Leaderboards: Mantener un ranking actualizado de miles de estudiantes en SQL requiere índices costosos y consultas `ORDER BY` pesadas. Los ZSETS de Redis mantienen los datos ordenados en tiempo real tras cada inserción ($O(\log(N))$), permitiendo consultas de rango instantáneas.

Hashes para Sesiones: Permiten almacenar objetos planos con acceso directo a campos individuales sin necesidad de serializar/deserializar todo el objeto JSON, optimizando el ancho de banda y CPU.

TTL (Time-To-Live): El manejo de expiración de sesiones y caché es nativo en Redis. En SQL, esto requeriría "Garbage Collection" jobs periódicos que consumen recursos.

5. Diagramas del Modelo Implementado

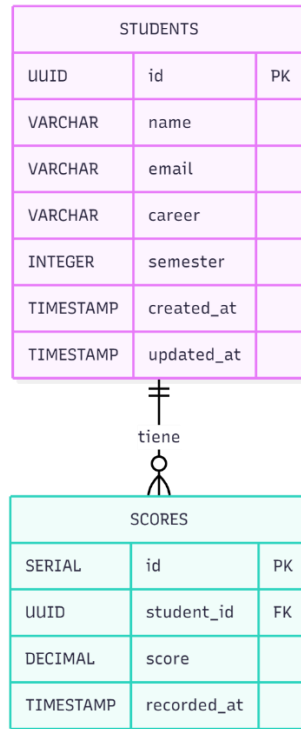
Aunque Redis es "schema-less", diseñamos un modelo lógico estricto para mantener la consistencia.



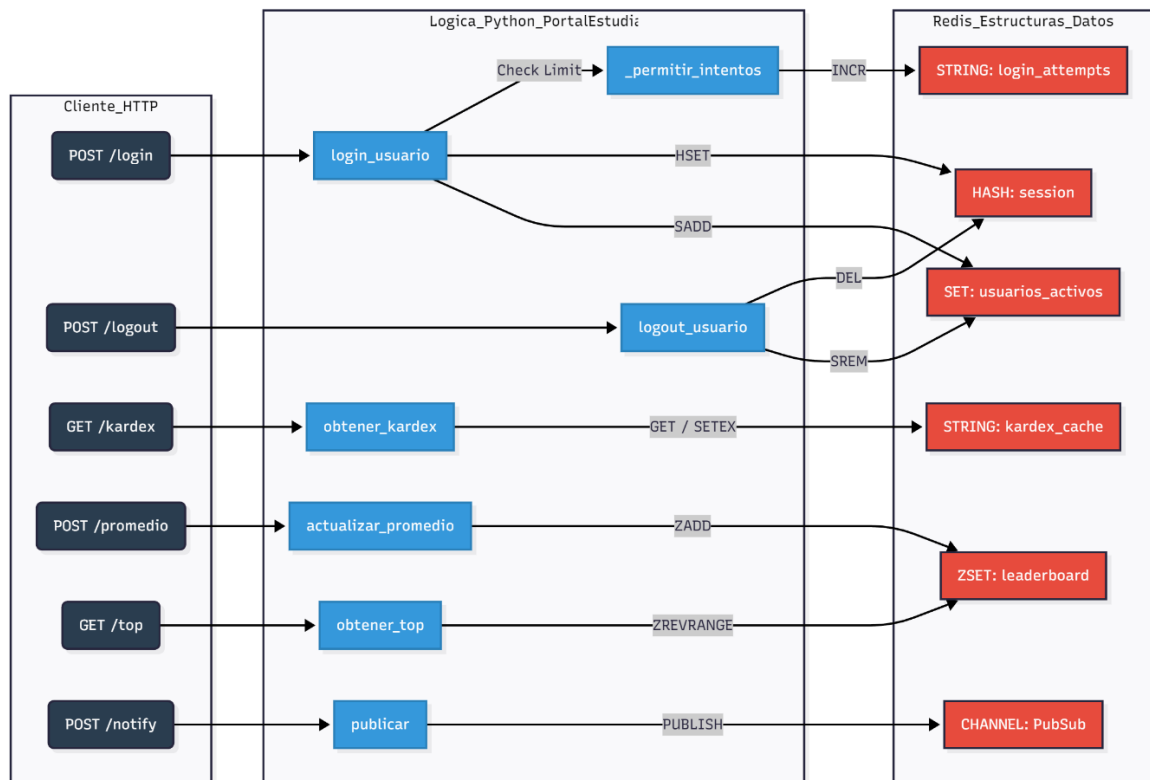
Detalles de Claves y Valores

Patrón de Clave	Tipo de Dato	Propósito	Expiración
session:{uuid}	Hash	Almacena estado de autenticación del usuario.	30 min (Rolling)
leaderboard	Sorted Set	Ranking global ordenado por calificación.	Persistente
user:{id}	String	Caché del perfil completo del estudiante.	5 min

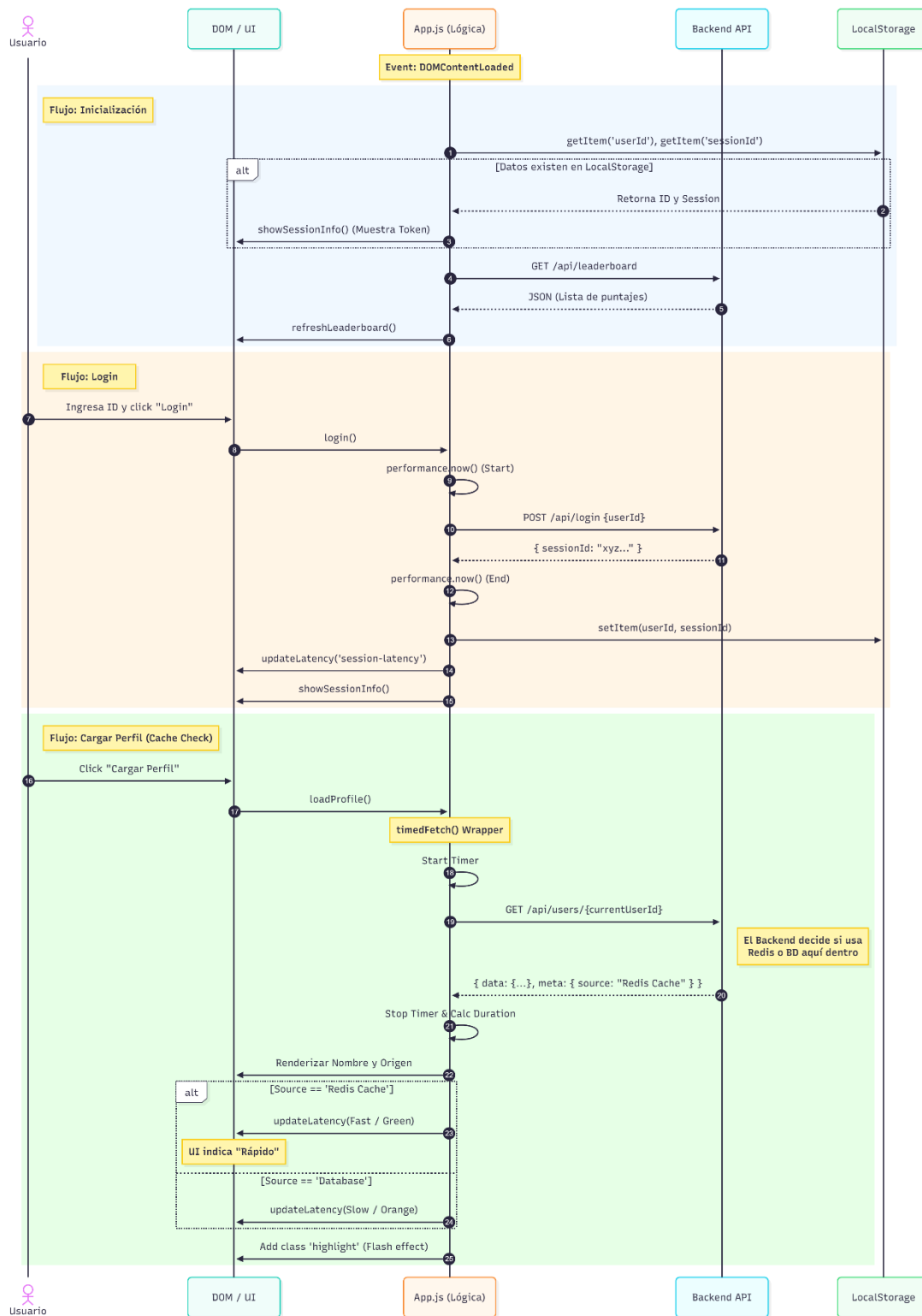
BD - ER



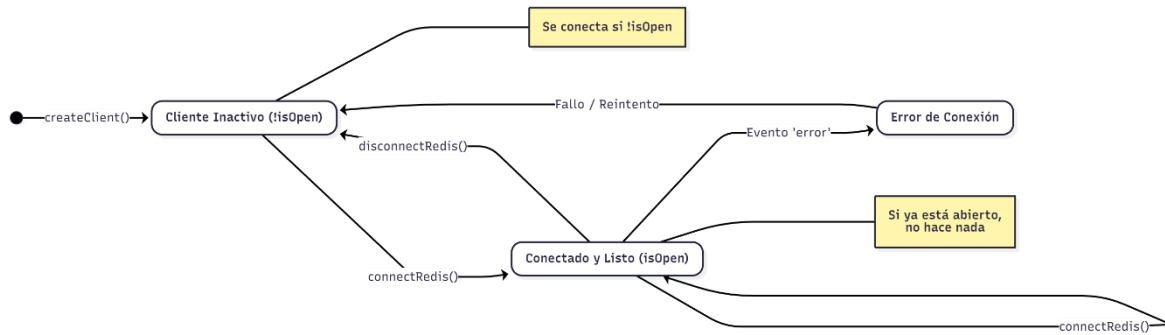
Backend Redis



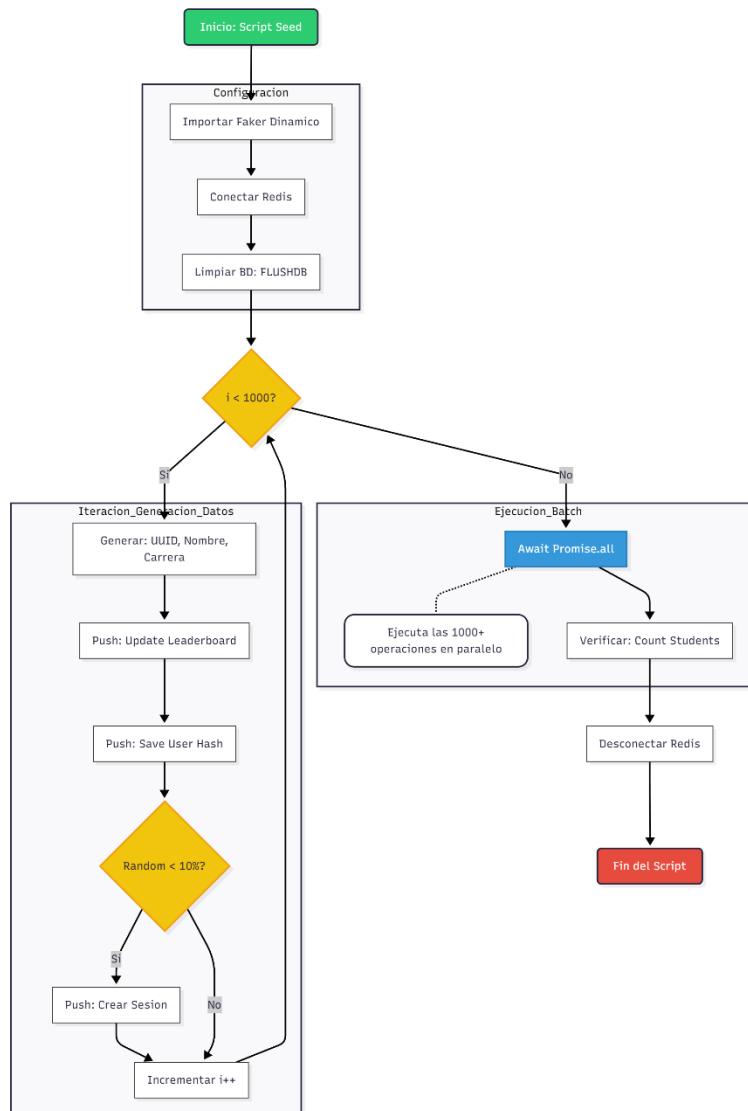
App.js



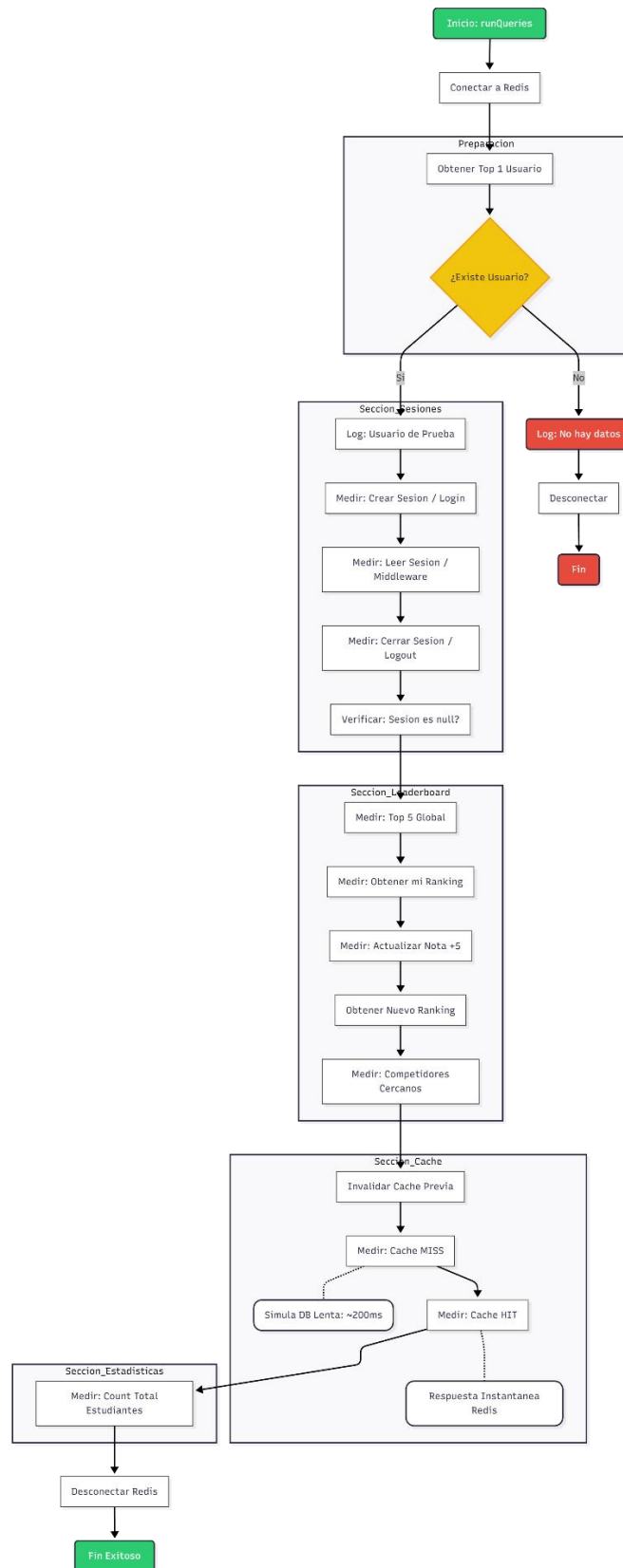
Redis.js



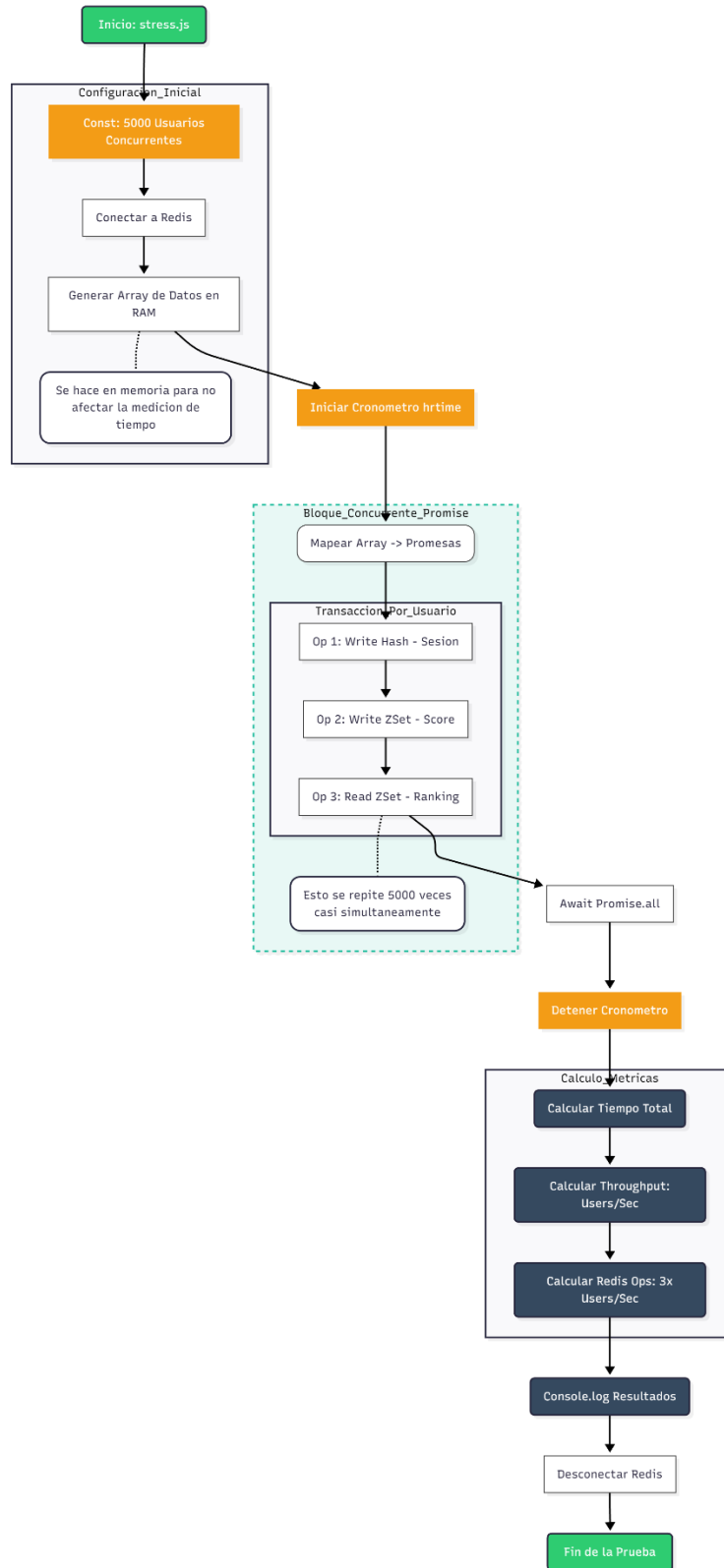
Seed.js



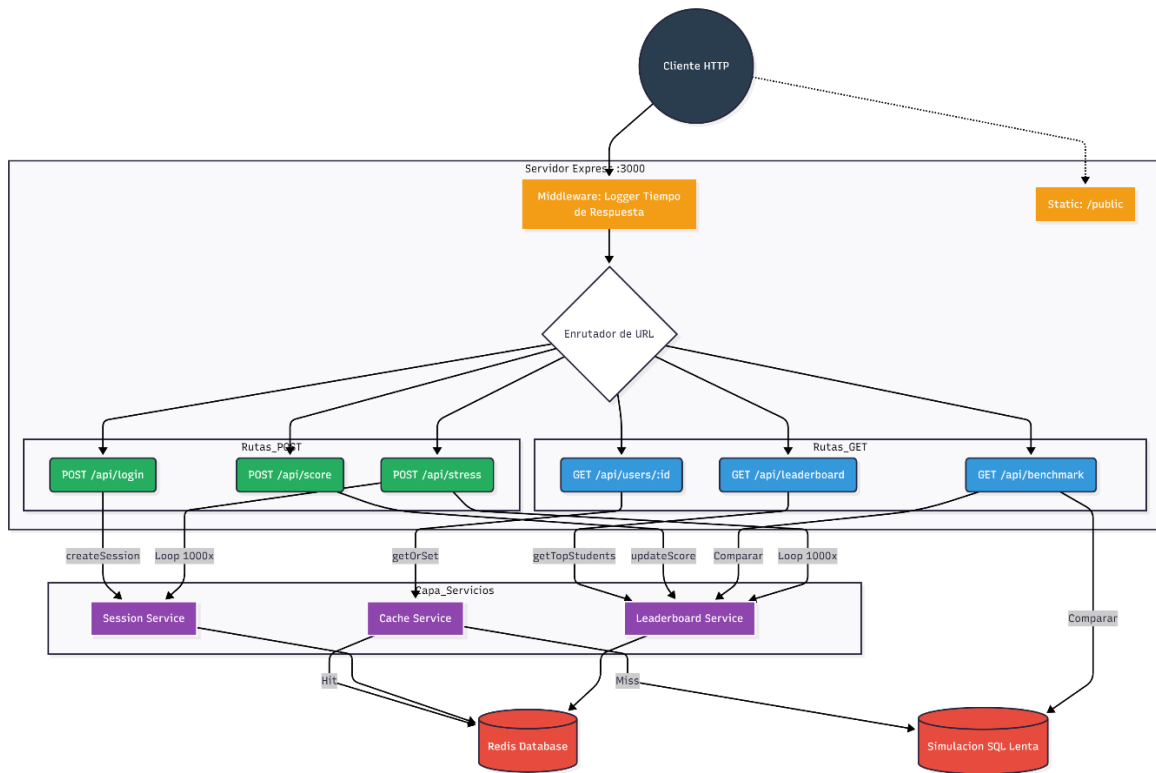
Queries.js



Stress.js



Server.js



6. Comparativa: Relacional Tradicional vs Enfoque Redis

A continuación, se contrastan las operaciones críticas del sistema bajo ambos paradigmas.

A. Ranking y Leaderboards

Característica	Enfoque Relacional (SQL)	Enfoque Redis (Sorted Sets)
Operación	SELECT count(*) FROM users WHERE score > my_score	ZREVRANK leaderboard my_id
Complejidad	O(N) (Sin índice) o O(log N) (Con índice B-Tree)	O(log N) (Skip List)
Escalabilidad	Se degrada linealmente con el volumen de datos. Costoso en updates frecuentes.	Constante y extremadamente rápido incluso con millones de registros.
Actualización	Bloqueos de fila/tabla al actualizar scores.	Atómico y sin bloqueos (Single-threaded event loop).

B. Gestión de Sesiones

Característica	Enfoque Relacional (SQL)	Enfoque Redis (Hashes)
Almacenamiento	Tabla 'Sessions' en disco.	Memoria RAM.
Limpieza	Jobs programados ('DELETE FROM sessions WHERE exp < NOW()').	Evicción pasiva automática (TTL nativo).
Latencia	~10-100ms (Depende de carga e I/O).	< 1ms (Constante).

C. Rendimiento y Latencia

Métrica	SQL (PostgreSQL/MySQL)	Redis (In-Memory)	Mejora
Tiempo de Acceso	~0.1 ms (si está en buffer pool) a 10ms (disco SSD)	005 ms (5 microsegundos)	~20x - 2000x
Tiempo de CPU	Alto (requiere sorting de filas)	Bajo (estructura pre-ordenada)	Drástica
Concurrencia	Bloqueos de fila, overhead transaccional (ACID)	Single-threaded Event Loop (lock-free lógico)	Alta escalabilidad

D. Modelo de Concurrency

SQL Tradicional: Utiliza un modelo multi-hilo o multi-proceso. Bajo carga extrema (ej. 10k req/seg), el Context Switching de la CPU y la gestión de bloqueos (Locks) se convierten en el cuello de botella antes que el disco.

Redis: Utiliza un modelo Single-Threaded con I/O Multiplexing (epoll/kqueue).

- a) Ventaja: Elimina condiciones de carrera (Race Conditions) sin necesidad de bloqueos complejos en la aplicación. Las operaciones son atómicas por definición.
- b) Limitación: Comandos muy lentos (ej. `KEYS` o operaciones $O(N)$ grandes) pueden bloquear a todos los clientes. Por eso usamos estrictamente comandos $O(1)$ u $O(\log N)$.

E. Arquitectura General

Relacional: Arquitectura monolítica donde la BD es a menudo el cuello de botella (SPoF) para lecturas intensivas.

Redis (Cache-Aside Pattern): Arquitectura resiliente. Redis absorbe el 90-99% de las lecturas, protegiendo a la base de datos persistente (SQL) para que solo maneje escrituras críticas y datos fríos.

Manual técnico

Estructura del proyecto:

__src/server.js	#Servidor Express y API REST.
__public/	#Frontend de demostración (HTML/JS).
__src/config/redis.js	#Configuración y cliente de conexión Redis.
__src/services/	
____sessionService.js	#Manejo de sesiones (Hash) con TTL.
____leaderboardService.js	#Gestión de rankings (Sorted Sets).
____cacheService.js	#Utilidad para caché genérico (Strings).
__src/scripts/	
____seed.js	#Script de generación de datos fake.
____queries.js	#Script de validación por consola.