

Spring Data:

Persistencia en Sistemas Relacionales

Daniel García Costa

2026

¿Que vamos a ver?

- ¿Que es Spring Data?
- Modelo de dominio
- Consultas en Spring Data
 - Métodos de abstracción
 - JPQL
 - SQL nativo
- Rendimiento
- Optimizaciones
- Sistema de ejemplo
- Mucho código

¿Qué es Spring Data?

Spring Data es el paquete del ecosistema Spring para **acceder a datos de forma consistente.**

- Define un modelo común de acceso a datos
- Orquesta todo lo relacionado con el acceso a datos
- Reduce código repetitivo
- Se integra con distintos motores

Qué NO es Spring Data

Spring Data:

- ✗ No elimina el modelo relacional
- ✗ No es ni sustituye al ORM
- ✗ No optimiza consultas automáticamente
- ✗ No toma decisiones por nosotros

Facilita el acceso a datos, pero **las decisiones siguen siendo nuestras.**

Capas de Spring Data

Aplicación



Spring Data — Abstracción de repositorios

Abstracción para definir repositorios y consultas



JPA — API estándar de persistencia

Especificación estándar (interfaces y anotaciones)



Hibernate — ORM (Object–Relational Mapping)

Traducción y mapeo de objetos a SQL



JDBC Driver — Comunicación con la base de datos

Ejecuta SQL contra la base de datos



Base de datos

Idea clave

Spring Data facilita el acceso a datos pero **no elimina el modelo relacional.**

- Las queries siguen existiendo
- El SQL sigue ejecutándose
- Las decisiones siguen importando

Modelo relacional

La base de datos trabaja con:

- Tablas
- Filas
- Claves primarias y ajenas
- Tipos simples
- Operaciones SQL

La lógica es **estructural y declarativa**.

Modelo de dominio

La aplicación trabaja con:

- Objetos
- Relaciones entre entidades
- Comportamiento
- Identidad lógica
- Reglas de negocio

La lógica es **orientada a objetos**.

El problema central

- El modelo relacional \neq modelo de dominio
- Los objetos no son filas
- Las relaciones no son joins directos
- El rendimiento no es transparente

**¡Necesitamos una capa
intermedia!.**

¿Qué consultamos en Spring Data?

En Spring Data **no consultamos tablas.**

Consultamos:

- Entidades de dominio
- Gestionadas por JPA
- Mapeadas a tablas relacionales

Entidad como contrato

Una entidad define:

- Qué datos existen
- Cómo se relacionan
- Cómo se identifican
- Cómo se persisten

La base de datos queda detrás de ese contrato.

Usamos anotaciones para definir las propiedades y relaciones de nuestras entidades

Categoría	Anotación	Para qué sirve
Entidad	@Entity	Marca la clase como entidad persistente
	@Table	Define la tabla asociada
Identidad	@Id	Define la clave primaria
	@GeneratedValue	Estrategia de generación del ID
Columnas	@Column	Configura nombre, nulabilidad, longitud
	@Basic	Atributo persistente simple
Relaciones	@ManyToOne	Relación muchos-a-uno
	@OneToMany	Relación uno-a-muchos
	@OneToOne	Relación uno-a-uno
	@ManyToMany	Relación muchos-a-muchos
Relaciones	@JoinColumn	Define la columna FK
	@JoinTable	Tabla intermedia
Carga	fetch = LAZY / EAGER	Controla cuándo se cargan relaciones
Ciclo de vida	@PrePersist	Callback antes de insertar
	@PreUpdate	Callback antes de actualizar
Temporal	@Temporal	Mapea fechas a tipos SQL
Herencia	@Inheritance	Estrategia de herencia
Control ORM	@Transient	Campo no persistente

Anotaciones de control (Spring Data / JPA)

Categoría	Anotación	Para qué sirve
Transacciones	@Transactional	Delimita una transacción
	@Modifying	Indica que una query modifica datos (DML)
Ciclo de vida	@PrePersist	Antes de insertar
	@PostPersist	Después de insertar
	@PreUpdate	Antes de actualizar
	@PostUpdate	Después de actualizar
	@PreRemove	Antes de borrar
	@PostRemove	Después de borrar

Inicialización del esquema (DDL)

Opción	Comportamiento
<code>none</code>	No hace nada con el esquema
<code>validate</code>	Verifica que el esquema exista
<code>update</code>	Sincroniza cambios automáticamente
<code>create</code>	Borra y crea el esquema
<code>create-drop</code>	Crea al arrancar, borra al parar

Se configura en el `application.properties`
(`spring.jpa.hibernate.ddl-auto`)

Impacto real del DDL automático

- Útil en desarrollo
- Peligroso en producción
- No sustituye migraciones
- Puede generar cambios costosos

Decisión de arquitectura, no de código

¿Dónde queda el SQL?

Aunque trabajemos con entidades:

- SQL sigue existiendo
- Hibernate lo genera o lo ejecuta
- La base de datos decide el coste (planificador del SGBD)

Spring Data **te abstraee de las consultas pero no las elimina.**

Repository

Un Repository es:

- Una abstracción
- Orientada al dominio
- Tipada con entidades
- Independiente del motor SQL

Es la **frontera** entre la aplicación y la persistencia.

Tres formas de consultar datos

En Spring Data podemos consultar usando:

1. Métodos de abstracción del Repository
(<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>)
2. JPQL (@Query)
3. SQL nativo (nativeQuery)

A mayor abstracción:

- Menos control
- Más conveniencia

A menor abstracción:

- Más control
- Más responsabilidad

Rendimiento

Cuando trabajamos con bases de datos:

- Todas las consultas tienen coste
- El volumen de datos importa
- La abstracción no es gratuita

El rendimiento depende de **cómo accedemos a los datos**

¿La abstracción es más lenta?

Pues... no siempre...

Depende de:

- La consulta
- El volumen de datos
- El uso de entidades o proyecciones
- El trabajo extra del ORM

El problema no es la abstracción, es lo **qué se pide a la base de datos.**

Costes ocultos habituales

- Cargar entidades completas
- Inicializar relaciones
- Consultas innecesarias
- N+1 SELECT
- Transferencia de datos excesiva

Muchos problemas no vienen del SQL, sino del **modelo de acceso**.

Ejemplo N+1 SELECT

```
SELECT t1.*, t2.*, ((t1.actual/t2.total)*100) DIV 1 AS progreso,
      CASE WHEN t1.actual = t2.total AND t1.nbpar = t2.total AND t1.nbcomp = 1 THEN 'bg-success'
            WHEN actual > 0 AND t1.fact <> factorial(actual) AND t1.nbcomp = 1 THEN 'bg-success'
            WHEN t1.nbpar = t2.total AND t1.nbcomp = 0 THEN 'bg-danger'
            WHEN actual > 0 AND t1.fact = factorial(actual) THEN 'bg-info'
            WHEN actual > 0 AND t1.fact <> factorial(actual) AND t1.nbcomp = 0 THEN 'bg-warning'
            WHEN t1.discard = 1 THEN 'bg-secondary'
            ELSE 'text-black'
      END AS class,
      CASE
        WHEN t1.actual = t2.total AND t1.nbpar < t2.total AND t1.nbcomp = 1 THEN '*'
        ELSE ''
      END AS aux
FROM
  (SELECT al.usuario, al.id_alumno, al.observaciones AS obs, al.muerto AS discard, MAX(n_problema) AS actual,
        CAST(SUM(CASE WHEN completa = 0 THEN 1 ELSE 0 END) AS SIGNED) AS nbpar,
        CAST(SUM(CASE WHEN completa = 1 THEN 1 ELSE 0 END)AS SIGNED) AS nbcomp,
        SUM(n_problema) AS fact
   FROM awpsolver.resultados_json rj
  LEFT JOIN awpsolver.alumnos al ON al.id_alumno = rj.id_alumno
  WHERE rj.id_grupo_experimento = :v
  GROUP BY al.usuario, al.id_alumno
  UNION
  SELECT al.usuario, al.id_alumno, al.observaciones AS obs, al.muerto AS discard, 0 AS actual,
        0 AS nbpar, 0 AS nbcomp, 0 AS self_fact
   FROM awpsolver.alumnos al
  LEFT JOIN grupos_experimentos ge ON al.id_grupo = ge.id_grupo
  WHERE ge.id_grupo_experimento = :v AND al.id_alumno NOT IN(
        SELECT id_alumno FROM awpsolver.resultados_json WHERE id_grupo_experimento = :v)
  ORDER BY 2) AS t1,
  (SELECT COUNT(*) AS total, factorial(COUNT(*)) AS fact FROM awpsolver.problemas WHERE id_experimento
  IN(SELECT id_experimento FROM awpsolver.grupos_experimentos WHERE id_grupo_experimento = :v)) AS t2;
```

JPQL vs SQL nativo

Ambos ejecutan SQL.

Diferencias:

- JPQL prioriza el modelo de dominio
- SQL nativo prioriza el modelo relacional
- El control es distinto

El rendimiento depende más de la query que del lenguaje.

Algunas técnicas para reducir coste

Queries específicas

Aprovecha el SGBD, pierde el miedo a escribir 100 líneas de SQL

Proyecciones

Traer solo lo necesario (tablas intermedias, vistas materializadas, ...).

Agregados

(COUNT , SUM , AVG) sobre el SGBD y no en dominio

Paginación real

Traer solo lo necesario, no todo + filtro posterior

- Menos columnas = Menos memoria
- Menos trabajo del ORM
- Menos transferencia de datos

**Son la primera herramienta
antes de optimizar consultas.**

Y todo eso... ¿como lo vemos?

Simulación de un juego de estrategia MMO (multijugador masivo en línea) multimundo.

- **Servidor Shard (mundo)**
 - Aloja jugadores
 - Contiene la dinámica del juego
 - Reenvía los eventos al Master
- **Servidor Master (coordinador)**
 - Recibe eventos de los Shard, consulta estados, jugadores, rankings, etc.

Arquitectura

- **Servidor Master**

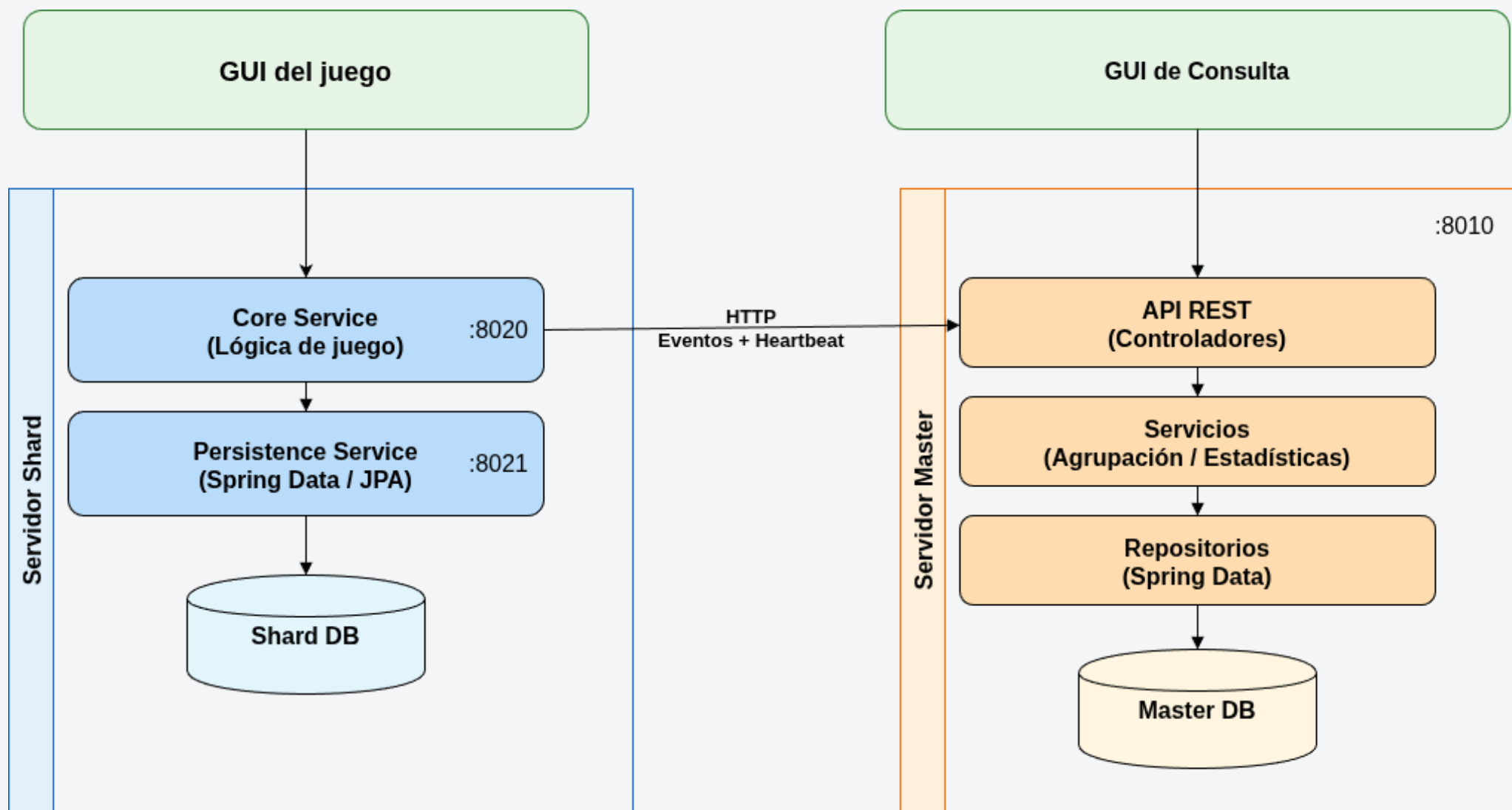
- Monolítico (un único servicio)
- División lógica en controladores -> servicios -> repositorios
- Solo recibe y agrega información
- GUI de consulta

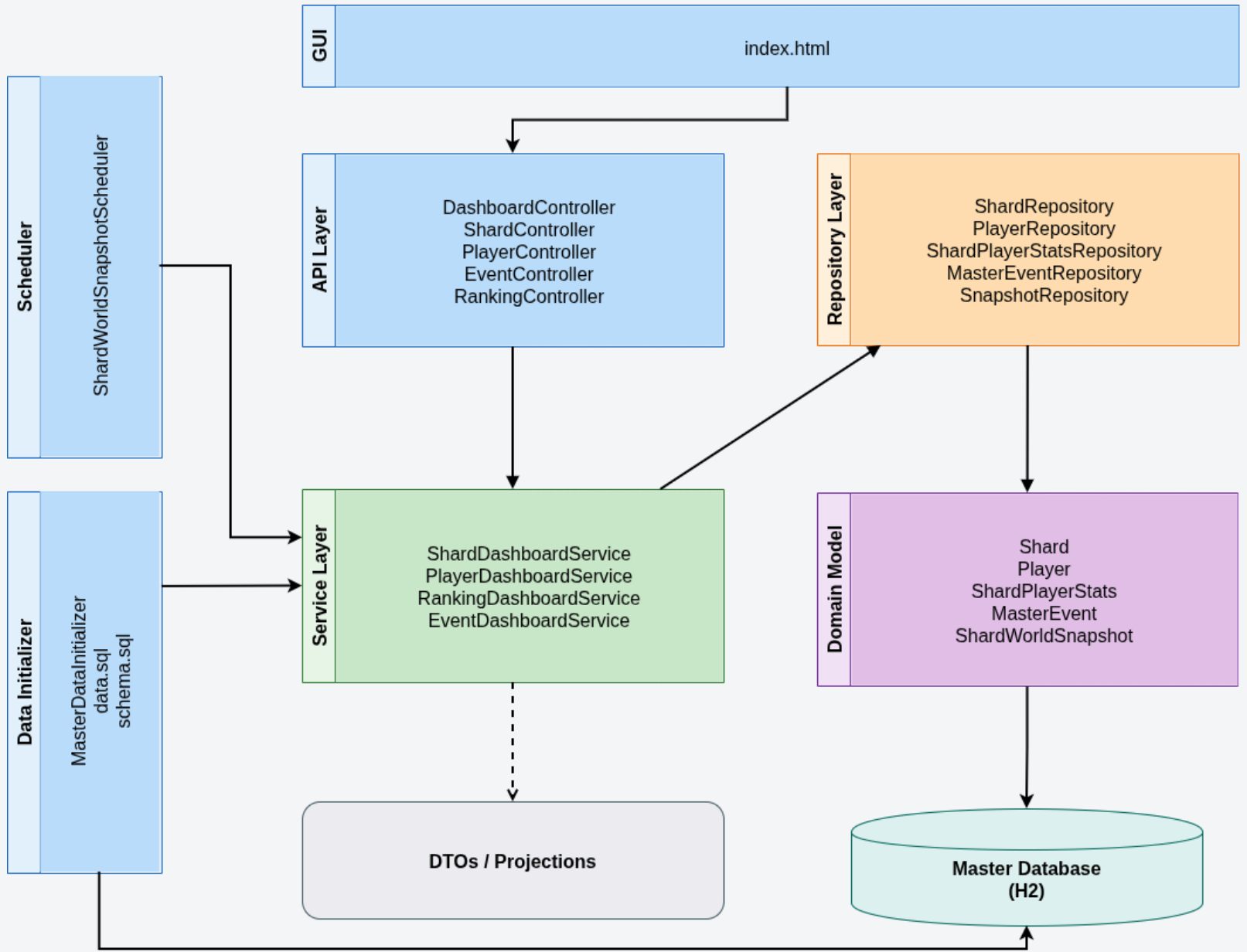
- **Servidor Shard**

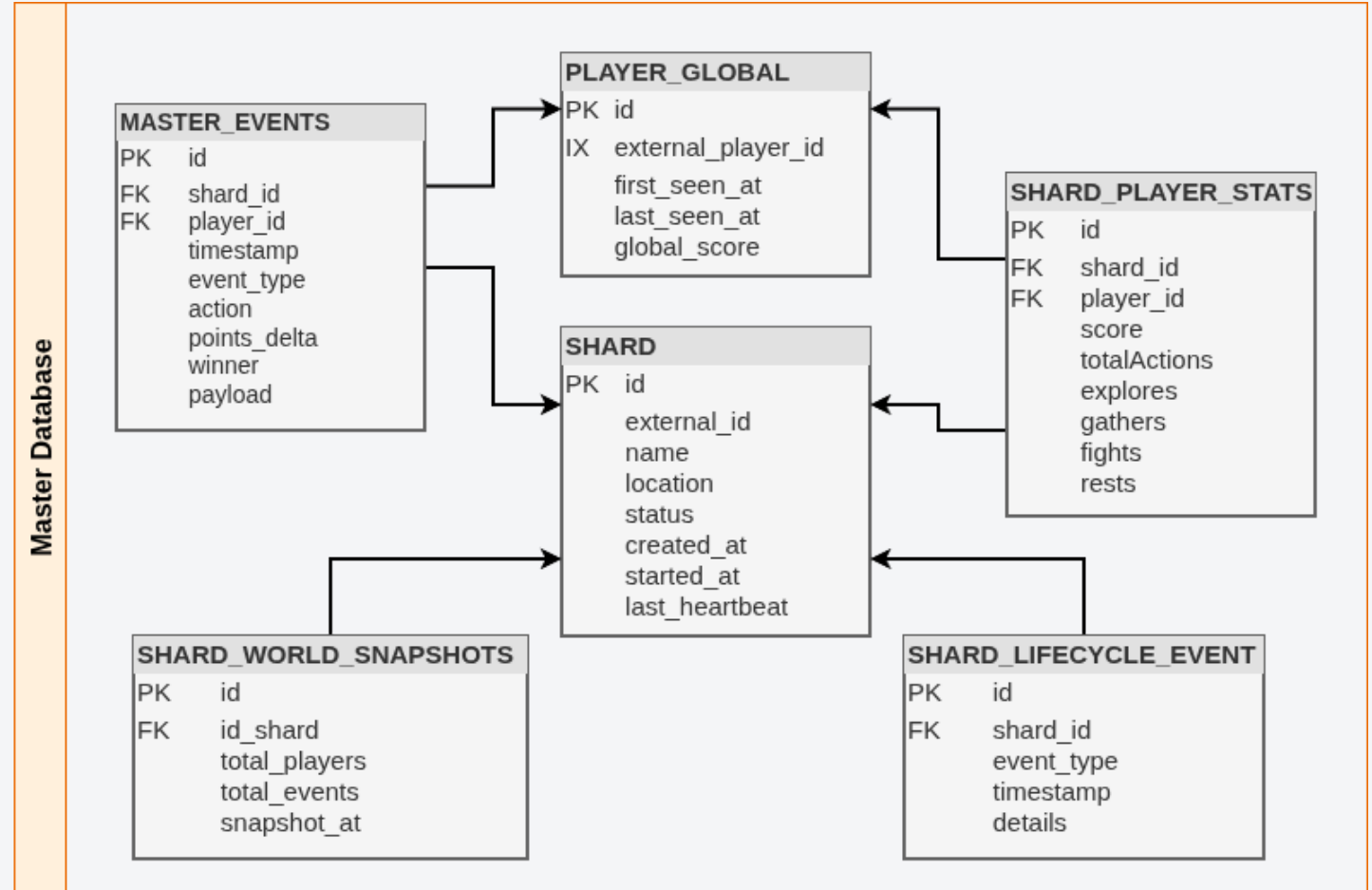
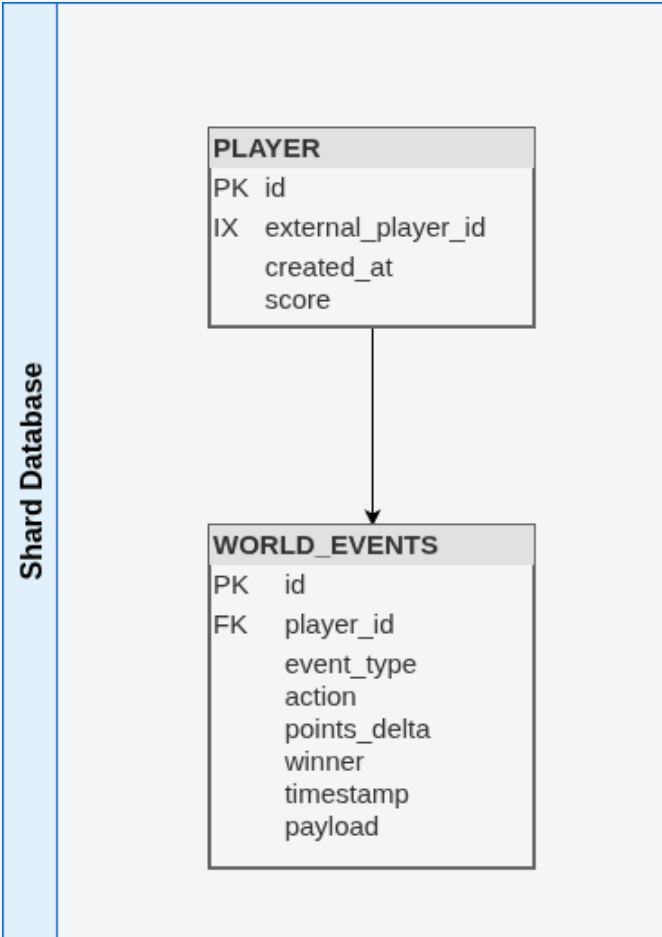
- Dos servicios (Persistence + Core)
- Gestiona los eventos de los jugadores
- GUI para simular jugadores
- Reenvío de eventos a Master

¿Por qué esta arquitectura?

- Separación clara de responsabilidades
- Escalado horizontal de shards
- Aislamiento del dominio de juego
- Centralización de analítica
- Escenario realista para estudiar el comportamiento de las capas de persistencia







Clona los siguientes proyectos

- **Master**
 - <https://inmaculados.uv.es/dagarcos/PSIM-master>
- **Shard(s)**
 - <https://inmaculados.uv.es/dagarcos/PSIM-shard-core>
 - <https://inmaculados.uv.es/dagarcos/PSIM-shard-persistence>

Y ahora... ¡a ver código!